J Supercomput (2018) 74:1836–1862
https://doi.org/10.1007/s11227-017-2193-5

Design of an accurate and high-speed binocular pupil tracking system based on GPGPUs

Juan Mompeán[1,2] · Juan L. Aragón[1] · Pedro M. Prieto[2] · Pablo Artal[2]

Published online: 17 November 2017

Abstract An efficient and robust pupil tracking system is an important tool in visual optics and ophthalmology. It is also central to techniques for gaze tracking, of use in psychological and medical research, marketing, human–computer interaction, virtual reality and other areas. A typical setup for pupil tracking includes a camera linked to infrared LED illumination. In this work, we evaluate and parallelize several pupil tracking algorithms with the aim of accurately estimating the pupil position and size in both eyes simultaneously, to be applied in a high-speed binocular pupil tracking system. To achieve high processing speed, the original non-parallel algorithms have been parallelized by using CUDA and OpenMP. Modern graphics processors are designed to process images at high temporal frequencies and spatial resolution, and CUDA enables them to be used for general-purpose computing. Our implementation allows for efficient binocular pupil tracking at high speeds for high-resolution images (up to 988 fps with images of 1280×1024 pixels) using a state-of-the-art GPU.

Keywords High-speed eye pupil tracking · GPU · GPGPU · Image processing · Real-time systems · Gaze tracking

1 Introduction

Eye and gaze tracking has been a relevant topic for decades, not only providing key information about eye physiology in vision sciences and ophthalmology but also enabling a wide variety of applications on both the research and commercial fields.

✉ Juan Mompeán
juan.mompean@um.es

1 Dept. Ingeniería y Tecnología de Computadores, Universidad de Murcia, Murcia, Spain

2 Laboratorio de Óptica, IUiOyN, Universidad de Murcia, Murcia, Spain

Examples of such applications include novel human–computer interfaces, virtual reality immersion, remote control of devices, marketing/advertising research or in-vehicle driving assistance. Although there is a range of radically different approaches to eye and gaze tracking, optical methods are widely used nowadays because they are non-invasive and can be inexpensive. Some of the simplest and cheapest techniques rely on pupil tracking, i.e., on determining pupil position and size in real time.

Pupil tracking is of special interest in physiological optics. Ocular wavefront aberration measurement and manipulation require precise knowledge of the pupil position since its center is taken as origin of coordinates for expressing the eye's aberrations. Early aberrometers required careful alignment of the subject's pupil and uncomfortable methods for head fixation. Instead, current devices typically allow some degree of free movement and involve a tracking algorithm to determine the instantaneous pupil position [9]. Even more demanding are the requirements for aberration correction/manipulation using *adaptive optics*, since slight misalignment can affect the coupling between subject's and induced wavefronts. In this context, a fast and accurate pupil tracking system would be a requirement for a free floating system where pupil movements are detected and dynamically compensated on the adaptive element or using additional components such as galvanometric mirrors. Adaptive optics visual simulation [34], consisting of the manipulation of ocular aberrations to perform visual testing through a modified optics, is a practical application that requires a high-performance pupil tracking algorithm.

Pupil tracking is typically performed by using infrared illumination. This invisible light has advantages in terms of subject's comfort, who would probably be dazzled by a visible beam of similar intensity. In addition, pupil size does not change with infrared illumination, which is useful in many experiments. Furthermore, the usage of infrared light allows to use a filter to reject all the visible light, which is useful to avoid corneal reflections from the stimulus display. In real-time experiments or measurements, pupil tracking must be accomplished at a high speed and with high accuracy. While gaze tracking routines do not typically require high-accuracy pupil determination [14], a low-precision approach can lead to important errors in the final results in some visual optics applications.

Several algorithms for calculating the pupil position and size can be found in the literature and in some cases tailored for a particular optical setup and illumination. We have selected three of them: (1) the Starburst algorithm [19] which traces a number of rays that search for large gradient changes (in order to detect the edge of the pupil) and provides information not only about the pupil size and position but also about its shape and orientation; (2) an algorithm that thresholds the image and selects a circle, iteratively searching for the biggest circle [16]; and (3) an algorithm that uses the resulting edges from a Canny edge detector [7] and then applies the Hough transform [3] to finally calculate the pupil's position and size.

These three algorithms, which are further described in Sect. 2, have been parallelized and evaluated in different parallel computing systems, from general-purpose multicore CPUs to state-of-the-art GPUs, by using OpenMP [29] and CUDA [27] programming environments. It is important to note that although pupil tracking is a widely used tool, it is still hard to achieve a high-speed tracking with high-quality images. This is a challenging problem since accurately tracking the pupil at high speed is a computationally intensive task and high-definition images require some preprocessing treatment which is also power demanding. Furthermore, a binocular system is required to segregate and process two pupils simultaneously. As we will see, the three evaluated algorithms offer a different pupil detection accuracy depending on the input image quality and they also present big differences in their computational requirements. An exhaustive evaluation of the algorithms has been carried out by covering a wide range of tuning parameters for each one. Experimental results have shown a speedup of $57.3\times$ compared to their corresponding sequential implementation, which enables a high-speed processing (up to 1260 fps), while not sacrificing detection accuracy. (More than 90% of the pupils in the dataset were detected with an error lower than 6%.)

A preliminary version of a *monocular* pupil tracking approach was published in [25]. Compared with that conference version, this paper includes several extensions and improvements. The pupil size estimation accuracy has been improved by fitting to an ellipse rather than to a circle (analyzed in Sect. 5.1.1). We have improved the parallel version of the Starburst algorithm in [25] by introducing a number of additional optimizations, which are described in Sect. 4.1 and evaluated in Sect. 5. In particular, we have made use of CUDA streams, have reduced the memory bandwidth usage, accelerated corneal reflection removal and *pipelined* different stages of the pupil detection algorithm, in addition to other improvements such as removing synchronizations, reducing GPU-to-host communication, increasing thread re-usage, or using the CUDA *ballot* instruction. Another new contribution has been the extension of the tracking mechanism to process two eyes simultaneously, developing a *binocular pupil tracking system* (as described in Sect. 2.5 and evaluated in Sects. 4 and 5), which achieves a throughput of 988 fps, despite doubling the computational requirements.

The remainder of this paper is organized as follows: Sect. 2 describes the three algorithms that have been parallelized and evaluated. Section 3 details the experimental setup. In Sect. 4, we evaluate the optimizations implemented to accelerate the GPU code which are later evaluated and analyzed in Sect. 5. Section 6 reviews the most relevant literature in the field. Finally, Sect. 7 summarizes the main conclusions of the work.

2 Pupil tracking algorithms: a GPU-based implementation

The first goal of this work is to parallelize several state-of-the-art pupil detection algorithms. In order to correctly evaluate these algorithms, an exhaustive evaluation of the parameters of each of these algorithms must be performed. This will allow us to characterize their performance and achieved accuracy with different settings.

2.1 Preprocessing

The same preprocessing flow has been applied to all of the images in order to perform a fair comparison of the three algorithms. A set of good quality images, which are in focus and do not contain significant noise, has been used for the experiments (refer to Sect. 3 for additional details). However, the images contain corneal reflections added

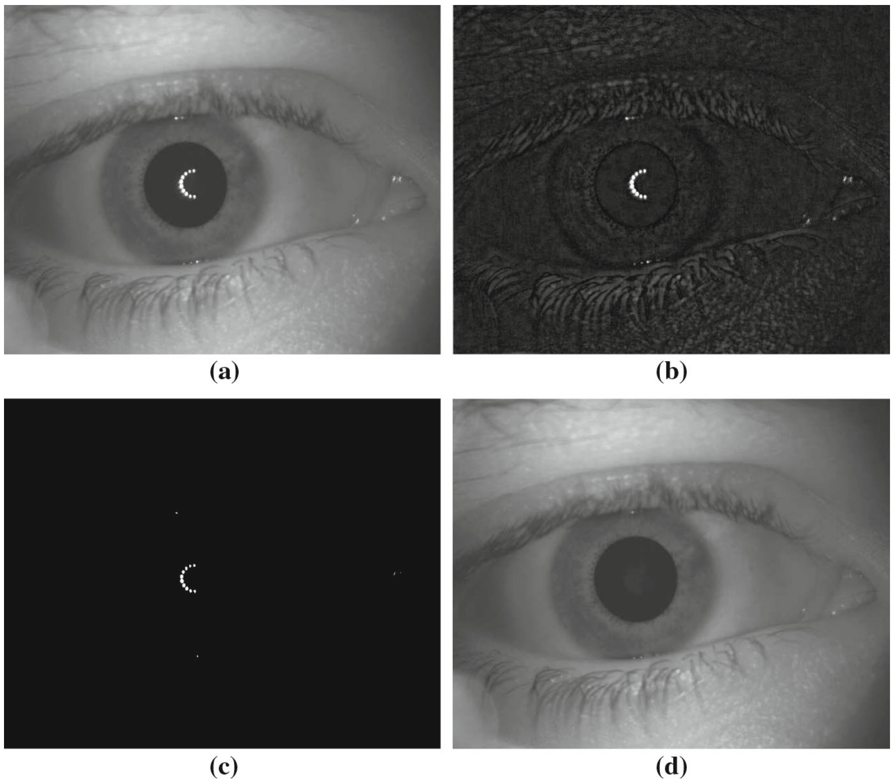

Fig. 1 **a** Eye image of a subject with the described illumination setup. **b** Result of the top-hat filter. **c** Threshold and dilation filters applied. **d** Final eye image with the corneal reflections removed

by the infrared LEDs used to illuminate the eye (as it will be explained in Sect. 3). These reflections must be removed to avoid problems in the pupil detection process. The chosen algorithm for this purpose was proposed by Aydi et al. [1]. It is a common approach 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 corneal reflection removal process is shown in Fig. 1.

The mentioned dilation and erosion filters are computationally intensive, and a *naïve* implementation in CUDA, processing n^2 elements per output pixel, might not perform as required. For our implementation, we have used squared kernels, which are separable filters that can be expressed as the outer product of two vectors. On the other hand, the thresholding filter is a straightforward operation in CUDA: Pixels with a value over a threshold are interpolated using 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.

Finally, although the image is clean, there are still some sharp edges (specially in the eyelashes) which could lead to wrongly detected points. To remove those sharp edges, a Gaussian filter is applied (with a kernel size of 5 and a sigma of 2). An implementation of the Gaussian filter, with a two-dimensional space domain convolution processing n^2 elements per output pixel, is straightforward in CUDA but not very efficient. To improve its performance, two separable kernels have been used, similarly to the approach used with the erosion and dilation filters. These CUDA optimizations are extensively detailed in Sect. 4. A median filter has also been tested as an alternative for the Gaussian filter. The median filter is usually used to remove noise while not destroying real information from the image. In our implementation, it selects the median value of the pixels around each pixel with a chosen radius. In the parallel median filter implementation, an histogram is used. One thread is used to calculate the histogram of the pixels around its center and to finally scan the histogram to find the median value.

2.2 Case 1: thresholding and labeling algorithm

For the first evaluated pupil tracking algorithm, a similar approach to the one proposed by Rankin et al. [31] has been used.[1] We have, however, modified the original algorithm to select the biggest blob instead of the central one. Our implementation uses several threshold values to improve the result. This is parameterized by defining a minimum and a maximum threshold and a step size. Depending on the chosen step size, the processing time of the algorithm will vary and also its accuracy. Therefore, the performance and accuracy achieved by this algorithm are a trade-off: The faster the less accurate, and vice versa. In our experiments, we have used 20 and 100 for the minimum and maximum threshold values. Step values ranging from 5 to 40 have been tested. After applying the threshold to the image, the labeling process is carried out. A state-of-the-art labeling algorithm proposed by Chen [8] has been used which scans the image to find unconnected areas that are assigned a different label. Figure 2a, b shows both a thresholded image and a labeled image, respectively.

When the image has been labeled, the histogram of the image is calculated to easily find the biggest labeled area (the one with the maximum value within the histogram) which corresponds to the pupil. Once the biggest area is found, the image is processed to remove the rest of areas, as it is shown in Fig. 2c. The final step detects the border of the labeled area. A simple border location algorithm has been used, which searches 0-value pixels in the image and checks whether the value of any of its eight neighbors is different than 0. In this case, its position is stored in a list of border pixels.[2] After this step, we have located the boundary of the biggest thresholded area of the image, as shown in Fig. 2d.

The final step of this pupil tracking algorithm consists of performing a RANSAC (random sample consensus) circle fit to find the pupil position. The points found after the previous processing of the thresholding and labeling algorithm usually are in the

[1] Their approach thresholds the image and applies a series of morphological operations to finally select the central blob.

[2] One CUDA thread is launched per each pixel in the image in order to detect the border pixels. Once the border pixels are detected, they are added in parallel to the list, by increasing a counter with an *atomic* operation.

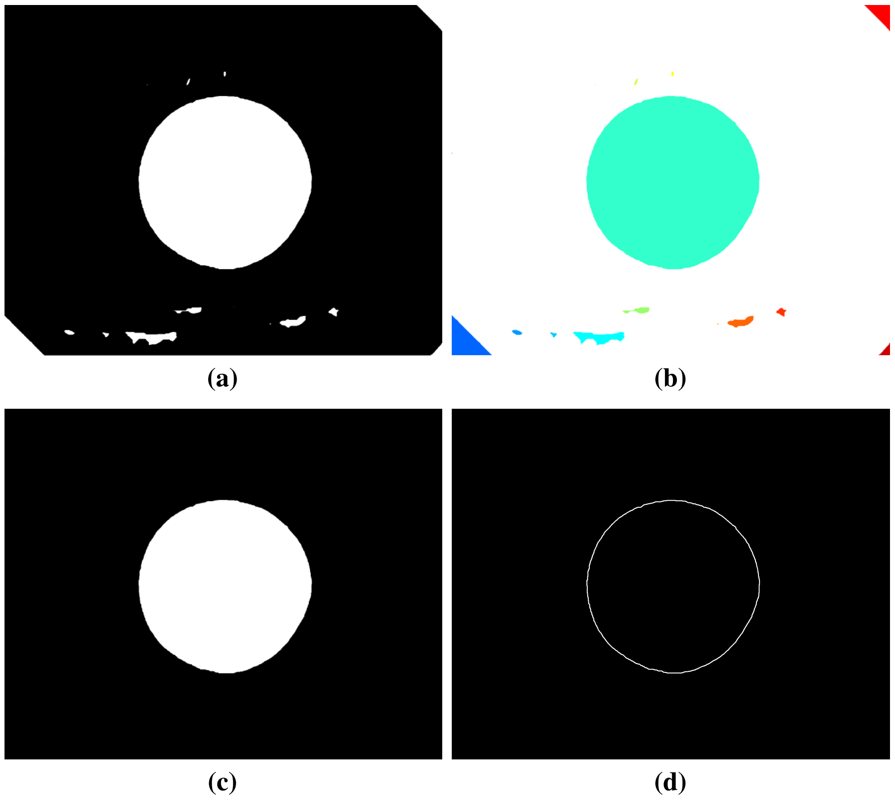
Fig. 2 **a** Thresholded image. **b** Labeled areas. **c** Biggest labeled area. **d** Boundary of the selected area

border of the pupil. However, some of them might be outside or inside the pupil. Therefore, if all of the points were used to perform a circle fit, the outliers would distort the result. Alternatively, the RANSAC algorithm uses only a few points to fit a circle and then checks how many of the points of the whole ensemble are closer than a given distance to the fitted circle (number of votes). We have used 5 points for the circle fit and a maximum distance of 2 pixels to accept the vote for 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 reused for all of the analyzed images, as described in Sect. 4. Every time that RANSAC is performed, 1024 fits are calculated using 1024 sets of 5 points. The sets are generated by multiplying the random numbers (in the range from 0 to 1) by the total amount of points, generating indexes in the array of points. Then a circle fit is performed for each set of points using the algorithm developed by Taubin [36]. Afterward, the distances are computed and a reduction operation is performed to sum up the votes for each circle. Finally, the circle with the maximum number of votes is selected as the best fit. The process is repeated for all of the thresholds, and the circle with the largest number of votes is selected as the circle that represents the pupil.

2.3 Case 2: Starburst algorithm

The second pupil detection algorithm is called Starburst and was proposed by Dongheng and Parkhurst [19]. 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 (as already explained in Sect. 2.1) is crucial.

After removing reflections, the Starburst algorithm is used to iteratively search for the pupil. The algorithm needs a starting central point for the search. For example, in a live capture, the previous center is used, which significantly speeds up the search and sometimes increases the accuracy. Otherwise, the center of the image is typically selected. 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, and using the pixels along its direction to calculate a gradient. This gradient is later inspected to search for the first value higher than a threshold (i.e., a high gradient change) which is interpreted as a potential pupil border. The result of each iteration is the set of potential border points found along the rays.

Initially, the rays' directions need to be obtained, and each direction is calculated by a CUDA thread. Then, for each ray one thread is issued per pixel along the ray to calculate the gradient for that pixel. If the pixel gradient is over a threshold, its position in the ray is stored using the *atomicMin* CUDA operation. Therefore, the closest point to the origin of the ray is saved (Fig. 3a). In a second stage of the Starburst algorithm, the set of border points are used to create new outgoing rays within an arc of $\pm 50°$, instead of the 360° used in the first stage, toward the center (Fig. 3b, c). The purpose of limiting the angle to $\pm 50°$ is to force finding new border points on the opposite side of the pupil. For the second stage, the parallelization is performed by using different CUDA threads in a similar way as for the first stage.

Finally, the border points found in both stages are joined (Fig. 3d) and the same RANSAC circle fit is performed, as described for the previous thresholding and labeling algorithm. Note, however, that in the original Starburst algorithm, the average location of the points was used as the convergence criteria, while our implementation utilizes a RANSAC circle fit because of its faster convergence. To this end, the distance between the new center and the old one is calculated, stopping if closer than 10 pixels. Otherwise, another iteration is executed (Fig. 3e) until the algorithm converges or a pre-defined number of iterations are performed.

2.4 Case 3: Canny edge detector and Hough transform algorithm

The last evaluated algorithm uses the Canny edge detector [7] and the Hough transform [3]. First, the Canny edge detector is applied to the image to obtain the edges (Fig. 4a). We are particularly interested in the edges of the pupil, which should be quite similar to a circumference. To that end, the Hough transform is applied to the resulted image and the best fitting circle is obtained (Fig. 4b). To perform the Canny edge detection, the image is first smoothed horizontally and vertically with a 1-D Gauss kernel, and

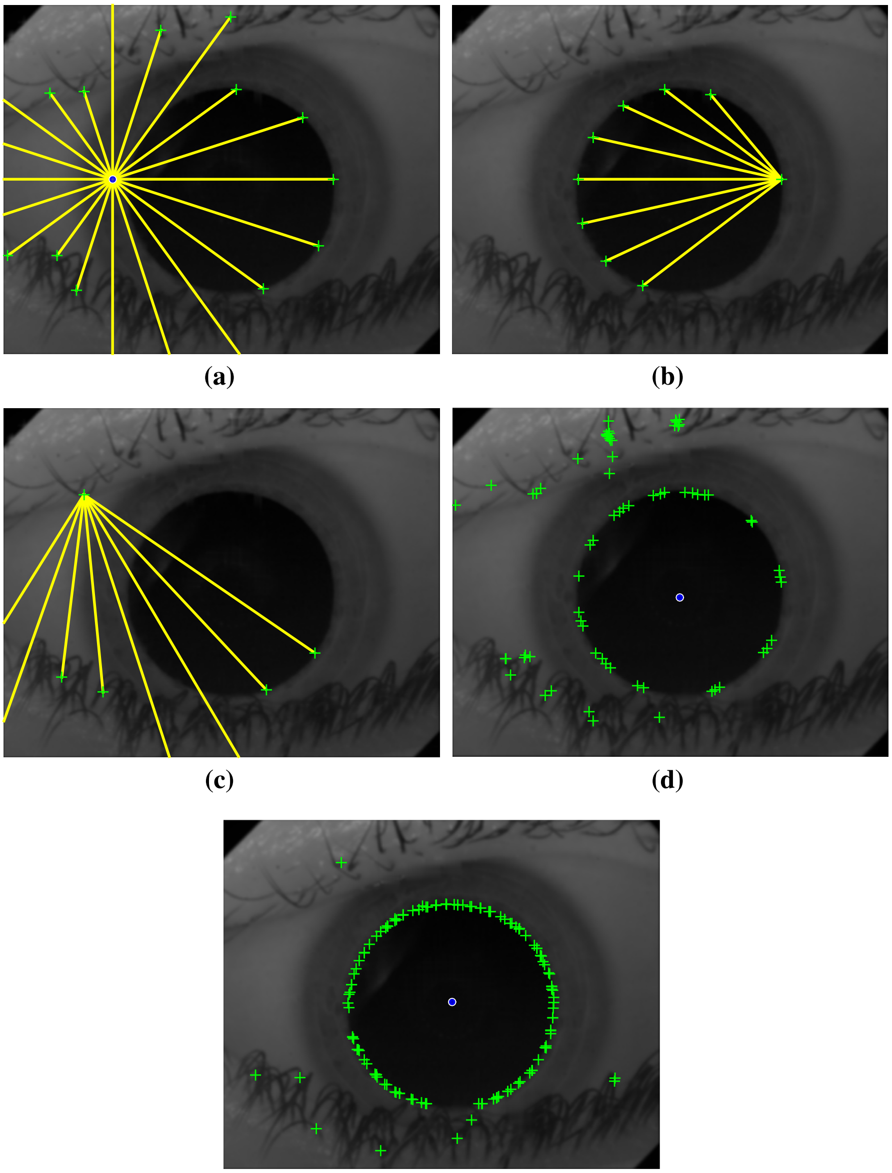

Fig. 3 **a** Rays are launched from a starting point (blue point). **b** Second stage's rays from one of the points found in the pupil border. **c** Second stage's rays from a point outside the pupil border. **d** Border points after the first iteration and the center of the best circle fitted with RANSAC (blue point). **e** Border points after the second iteration and the center of the best circle fitted with RANSAC (blue point) (color figure online)

then convoluted with the first derivative of a Gauss kernel; all of them are run in the GPU. Later, the gradient value and direction are calculated using one CUDA thread per processed pixel in the image.

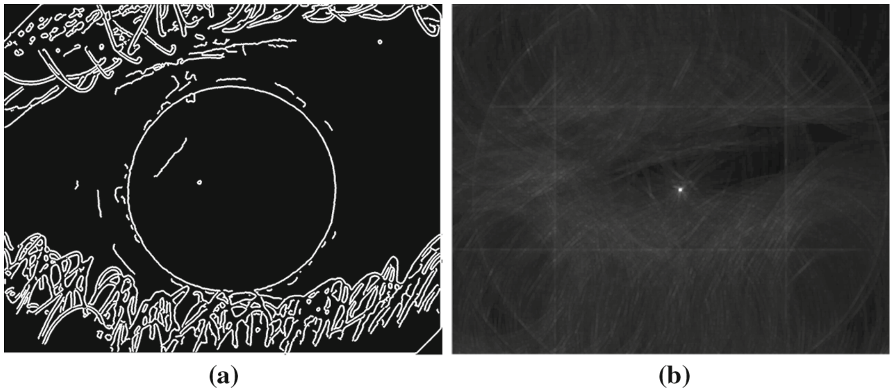

Fig. 4 **a** Canny edge detector result. **b** Hough transform result

The Canny edge detector uses two threshold values to process the initial image. The *high threshold* value is calculated as the minimum value which is higher than the 80% of the pixels' values. To that end, the histogram of the image is needed. The *low threshold* value is established at 40% of the *high threshold*. Then the maximum gradient value is calculated to normalize the gradients: The maximum reduction is performed first in shared memory for each block and then in global memory for all of the blocks. The double thresholding is applied followed by a hysteresis filter to remove the low values and keep the higher ones. All of these operations are performed in parallel in the GPU. An example of the output of the Canny edge detector is shown in Fig. 4a.

The next stage consists of applying the Hough transform. Since only a small fraction of the pixels in the resulting image are edges, it isn't necessary to test all of them. To reduce the amount of computation, an array of edges is initially built. One CUDA thread is launched per pixel of the image; if it is part of an edge, it is added to the array. Then the edge pixels vote for the different radii. The amount of memory allocated for voting depends on the number of radii tested and the size of the image. Each pixel in an edge votes by increasing the value at the locations which are at a distance r (current radius). While voting for all the pixels at a distance r would be the best approach, it is a very expensive operation. To reduce the cost of the voting process, a *step* parameter is used to define the distance between each of the pixels sampled. Several threads work on parallel on each edge pixel, voting for different radii. After the voting is finished the best center and radius is found by searching for the maximum in the voting data structure.[3]

2.5 Binocular pupil tracking

For some visual optics applications, both pupils need to be tracked. To that end, we have developed a binocular system capable of tracking both pupils simultane-

[3] A parallel maximum search has been performed using the Thrust library [4].

ously. Our approach assumes that both pupils are in the same image. Therefore, for evaluation purposes we have created a dataset of binocular images by duplicating each pupil from the monocular dataset. Note that since we are using one single image containing two pupils, the preprocessing stage is shared by both pupils and so not incurring in additional computational needs provided that the image size is not changed.

In our evaluation, only the Starburst algorithm has been tested with the binocular approach, as we will explain in Sect. 5. The algorithm has been modified accordingly so that each pupil is searched independently, assuming there is one pupil on the left half of the image and the other is on the right half. The algorithm takes this new scenario into account when it is launching the rays for searching the pupil borders. In particular, we need to provide the starting points for the left and right halves of the image, and use the correct image width for each pupil.

3 Experimental framework

There are many lighting configurations which might be used for a pupil tracking system; our particular approach consists of a set of infrared LEDs and a camera focusing at the eye (Fig. 5a, b). The optical setup in the figure appears more complicated because there are extra optical elements, not related to our pupil tracking system, because that setup was used for another optics experiment for locating the achromatizing pupil position and first Purkinje reflection in a normal population [22]. The high-speed camera can be seen at the end of the setup, whereas the ring of LEDs is closer to the eye of the subject. The subject sits in front of the camera and stays there, while the pupil is tracked. The tracking process can be done in real time or off-line if a video is captured. An image of an eye captured with this setup is already shown in Fig. 1a.

Two lighting modes are available; either half circle or the whole circle of LEDs can be turned on. The semicircular set of LEDs generates corneal reflections that appear as bright spots inside the pupil; however, using the full circle of LEDs would have added an artificial illuminated circle to the images that could lead to an increased number of false positives. We have opted for the semicircular ring of LEDs which produces a more homogeneous illumination to later eliminate the corneal reflections in the preprocessing stage, as explained in Sect. 2.1 and shown in Fig. 1.

The GPU used for testing the CUDA programs is a high-end NVIDIA GeForce GTX 980, with 2048 CUDA cores and 4GB of main memory. The CUDA version used to compile the code is CUDA 8.0. For the OpenMP experiments in a traditional CPU, we have used an Intel i5-4690 (up to 3.9 GHz) processor with 4 physical cores and 16 GB of 1600 MHz DDR3 RAM (dual channel). The compiler used is gcc-7 with the flags -O3 and -fopenmp.

Pupil images dataset The evaluation was performed using 964 pictures of 1280×1024 pixels taken with the described optical setup from 51 different subjects. The dataset contains pupils with a wide variety of radii, ranging from 1.66 to 4.28 mm, with a mean of 3.21 mm. Furthermore, the images in our dataset have different illumination

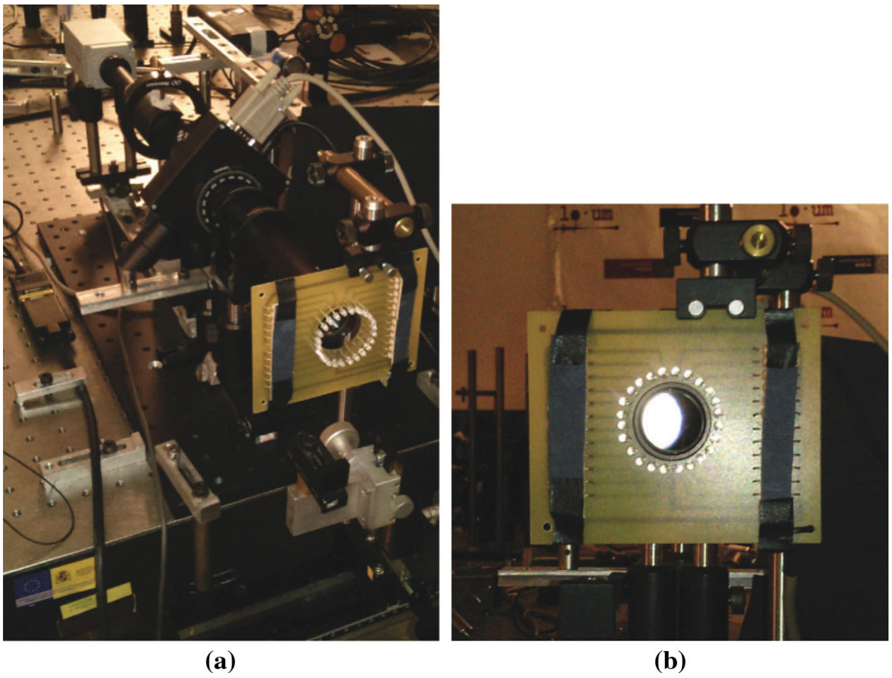

Fig. 5 **a** Optical setup for pupil tracking. **b** Infrared illumination subsystem

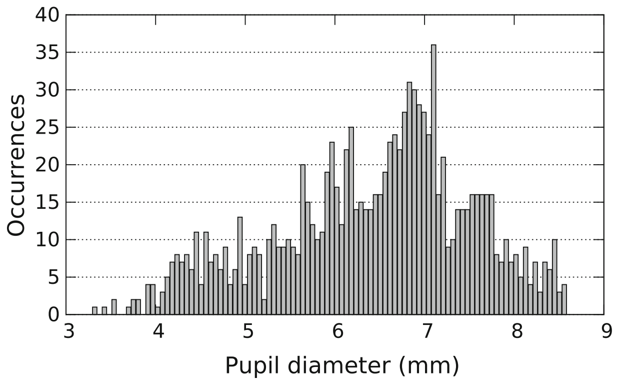

Fig. 6 Histogram of the different pupil sizes in the evaluated set of images

intensities and include pupils with a high variety of shapes since they belong to different subjects. In order to have a reference of the correct pupil position and size for each image in the dataset, they have been manually processed. Five points are selected in the border of the pupil, both a circle fitting and an ellipse fitting are performed and the result is saved as the reference. Figure 6 shows a histogram of the different pupil sizes found in the dataset.

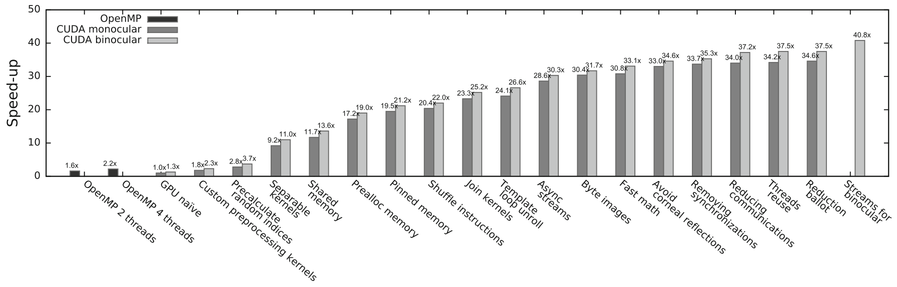

Fig. 7 Speedup of the parallelized Starburst algorithm by using OpenMP, CUDA monocular and CUDA binocular approaches (over sequential version)

4 Optimizing for high-speed pupil tracking

In order to use the huge computing power available in modern GPUs, a powerful programming language is required. CUDA was created to enable programmers to use all the hardware capabilities of the GPUs. However, while a *naïve* implementation of an algorithm in CUDA might provide a reasonable performance gain, additional and careful algorithm optimization must be carried out in order to achieve a much better and significant speedup. The following paragraphs explain the different CUDA optimizations that have been applied to the pupil tracking algorithms. As a quick reference, in Fig. 7 we show the speedup obtained by the Starburst algorithm for each CUDA optimization with respect to the sequential version in a high-end CPU for both the monocular and binocular datasets. The sequential version uses the OpenCV library (version 3.3.1) to perform the preprocessing tasks. OpenCV is a high-performance computer vision library which uses vector instructions in some of its functions [6]. For comparison purposes, we have also included numbers for an OpenMP implementation running in a multicore CPU (details shown in the previous section). Note that CUDA optimizations have been applied in a cumulative way since while each of them independently would not produce a big performance impact, however, altogether significantly reduce the global execution time. Furthermore, the reported speedups include the communication time required to copy the images and the results between the host and the GPU.

OpenMP implementation There are several loops where OpenMP has been used. The pragma *omp parallel for* has been used in the top-hat transform, the dilation, the thresholding, the corneal reflection removal and in the RANSAC circle fitting. Those are the loops where most of the execution time is spent and benefit the most from parallelization. The achieved speedup for a varying number of threads is shown in Fig. 7.

Naïve CUDA implementation A *naïve* CUDA implementation directly maps the C code into CUDA. While the code is not optimized, however, some good practices are still followed: Unnecessary memory copies are avoided, intrinsic functions are preferred (high precision is not necessary) and optimal block sizes are selected. This implementation uses the NPP (NVIDIA Performance Primitives) for performing the preprocessing operations: erosion, subtraction, dilation, thresholding and Gaussian filter. The naïve implementation for the monocular setup achieves a tracking speed of 21 fps and a speedup of 1.0 $\times$ (1.3$\times$ for the binocular setup) over the sequential version.

Custom preprocessing kernels Even though the NPP are highly optimized, still it is possible to improve their results with a custom implementation. This is due to the abstractions used in the NPP: We can avoid memory accesses and implement some optimizations by developing a tailored implementation. We have replaced all the functions used from the NPP (erosion, subtraction, dilation, thresholding and Gaussian filter) with our own kernels obtaining a significant improvement in the processing speed. The implementation for the monocular setup achieves a tracking speed of 38.6 fps and a speedup of 1.8 $\times$ (2.3$\times$ for the binocular setup) over the sequential version.

Pre-calculating random numbers Random numbers are used in the RANSAC method. They are generated with the *cuRAND* library [10] and scaled to match the range of the number of points previously found. However, generating random numbers is a computationally expensive task that should be avoided whenever possible. In the naïve implementation, new sets of random number are calculated every time that the RANSAC is performed, leading to a poor performance. However, due to the changes in the points found in the border of the pupil between iterations the same random points can be pre-calculated and reused through the whole execution. Since the amount of points found is unknown before running each iteration, the random numbers are generated between 0 and 1 and later scaled between 0 and $number\_of\_points$. 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 over the sequential version raises to 2.8 $\times$ (3.7$\times$ for the binocular setup).

Separable preprocessing kernels The erosion, dilation and gaussian filter operations are performed with a two-dimensional $k \times k$ filter in the naïve implementation. As a result, they generate a large number of load operations that saturate the memory system while not fully utilizing the CUDA cores. Although newer GPUs implement bigger caches that partially mitigate this problem, it is still possible to highly reduce the required memory bandwidth by dividing the kernel into two one-dimensional kernels. These three preprocessing operations can be divided into two kernels reducing the memory loads of each thread from $k \times k$ to $2 \times k$. And so only an intermediate buffer, which can be pre-allocated, is needed. After applying this optimization, the frames per second are dramatically increased to 203.2 and a huge speedup of 9.2 $\times$ is obtained (11.0 $\times$ for the binocular setup).

Preprocessing kernels with shared memory The previous optimization (*separable kernels*) has greatly reduced the memory bandwidth used by the preprocessing operations. However, there is still room for improvement in those kernels. 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 [30] 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 single CPU raises to 11.7 × (13.6 × for the binocular setup).

Pre-allocate memory The allocation of the memory in the GPU is a very slow operation, so whenever possible memory should be reused. 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. However, the size of those structures is known, so they can easily be pre-allocated when the execution is started and reused while processing the images. After applying this optimization, the obtained number of frames per second is 378.8 and the speedup increases to 17.2 × (19.0 × for the binocular setup).

Pinned memory The images must be copied from host memory to device memory; however, memory copies between them are relatively slow. Furthermore, the amount of time spent processing an image after applying the previous optimizations has decreased a lot, and the overhead due to the memory copies has increased its weight. More than 25% of the processing time is spent copying the images to the GPU. Each image is only copied once, so this cannot be reduced. However, using pinned memory instead of virtual memory the time spent copying the images could be reduced. When some data are copied to the GPU using virtual memory, it must be copied first to a pinned memory buffer. Therefore, if pinned memory is used from the beginning one copy is avoided. After switching to pinned memory, the number of frames per second obtained is 428.6 and the speedup raises to 19.5 × (21.2 × for the binocular setup).

Reduction with shuffle instructions To find the best fitting circle from all of the fitted circles using the randomly selected points, a reduction is performed to sum up the votes from every point. This reduction is performed in parallel for over 100 points for the 1024 circles. The naïve approach, which uses an `atomicAdd` operation, is very slow. (Note that in the naïve implementation we were using a typical approach using shared memory [13].) However, *shuffle* instructions provide a significant improvement for this stage, although as the time spent in the reduction operation is low, the global speedup is modest. *Shuffle* instructions enable CUDA threads within the same *warp* to send data to each other directly; therefore, the communication overhead of other approaches such as atomic operations or the use of shared memory and synchronization is highly reduced. The number of frames per second raises to 449.4, while the achieved speedup increases to 20.4 × (22.0 × for the binocular setup).

Joining kernels Programming abstractions are useful to simplify the code and improve legibility; however, they can affect the performance. In this case, there are three operations where separating the functionality has resulted in repeated accesses to memory and, as a consequence, wasted execution time. These three functions are: the last part of the top-hat function (second step dilation and subtraction) and the thresholding filter; generating gradients and searching the border points in Starburst; and the RANSAC with the reduction operation. These three functions share the same pattern: They are accessing the data generated from the previous one. After joining them into a single kernel, they avoid the storage of data in the (slow) global memory to be later re-loaded and reused by the following step. After this optimization, we achieve 512.6 fps and a speedup of 23.3 × (25.2 × for the binocular setup).

Template unrolling Templates can be used to implement generic programming, but they are also very useful to enable compile time optimizations. They are supported by CUDA, and we have used them to parameterize some loops. After the use of templates for the gaussian, the erosion and the dilation filters, the radius of these convolutions can be parameterized enabling the compiler to apply loop unrolling for these functions. However, templates reduce the flexibility of the code, since only the values specified at compile time can be used. To increase the number of allowed values, a `switch` with different radius sizes has been added to the kernels to keep a small amount of flexibility. Unrolling the loops avoids checking for each iteration the loop's termination condition, thereby reducing some time. The number of frames per second raises to 530.3 and the achieved speedup to 24.1 × (26.6 × for the binocular setup).

4.1 Additional CUDA optimizations for Starburst

Asynchronous computing with CUDA streams Usually, the biggest advantage when using CUDA streams is the ability to run several small kernels in parallel. However, this is not always possible due to inter-dependencies between kernels. This is the particular case of the Starburst algorithm, although it is still possible to run in parallel some stages such as the initializing portion of the algorithm. For example, we have used CUDA streams for calls to the *cudaMemsetAsync* function in order to initialize the memory used for storing the located pupil border points. But there are other advantages when using CUDA streams. In particular, the overhead of launching the kernels is significantly reduced due to the fact that they can be launched, while the previous kernels are still running, and so almost hiding the launching time overhead. After applying this optimization, the number of frames per second raises to 629.4 and the speedup to 28.6 × (30.3 × for the binocular setup).

Reducing memory bandwidth usage We are working with 8-bit grayscale images; however, we were using 32-bit integers for storing each pixel. Since the memory bandwidth for the GPU is a scarce resource, we should try to reduce its usage as much as possible. Simply by changing the data type of the image pixels from integer (as we were using in [25]) to byte, the bandwidth is reduced to one-fourth for many memory operations. The resulting frames per second raises to 667.9 and the speedup increases to 30.4 × (31.7 × for the binocular setup).

Enabling fast math Even though modern GPUs implement high-performance floating point units, they still perform worse than integer operations and might be avoided at the expense of losing some accuracy—if we seek a high-performance implementation and the loss of accuracy can be afforded. Although our Starburst implementation uses single-precision floats for a few operations (calculating rays' directions, circle fitting, ellipse fitting and the Gaussian filter), there is still room for improvement. In CUDA, some single-precision mathematical functions can be translated into *transcendental functions* [20] (e.g., sine, logarithm and exponential). These functions are faster but less accurate than the standard ones, and they can easily be switched on or off using the *-use-fast-math* compilation flag. There is another math optimization that can be used. In particular, calculating the second power of a number using the *pow* function is less efficient than simply multiplying a number by itself, and this is a common operation in our algorithm. The number of frames processed per second raises to 677.9 and the speedup obtained is 30.8 $\times$ (33.1 $\times$ for the binocular setup).

Avoiding corneal reflection interpolation As explained before, the corneal reflection removal stage is crucial for the Starburst algorithm to accurately locate the pupil. But it is a relatively expensive task in terms of processing time. Fortunately, it is possible to significantly speed it up. While the straightforward approach simply replaces the pixels in the detected reflections with interpolated values and continues searching for a high gradient change along a ray, using the interpolated values, a more effective approach consists of ignoring the whole corneal reflection area during the searches of the gradient, avoiding a lot of useless computation. In order to implement this approach, a new data structure for storing the affected areas is created. This data structure is used to check whether the current pixel along a ray is inside or outside a corneal reflection. The resulting amount of frames per second is 725.7 and the new speedup is 33.0 $\times$ (34.6 $\times$ for the binocular setup).

Removing synchronizations While the implementation of the preprocessing filters we used for the monocular version in [25] was highly optimized, some additional performance gain can be obtained by reducing the synchronization required to perform each operation. In [25], the shared memory was initialized to a given value which was later overwritten with the values loaded from the image. The purpose of initializing the shared memory was to fill with valid values positions that could be outside of the image. However, the initialization can be replaced with a test to determine whether a given position is inside or outside of the image. With this test, a call to the *__syncthreads()* function can be eliminated and some time is additionally saved due to the more efficient usage of the already limited GPU shared memory. The new amount of frames per second is 740.4 and the speedup obtained is 33.7 $\times$ (35.3 $\times$ for the binocular setup).

Reducing device-to-host communications In [25], the selection of the best circle fit was divided in two steps, requiring two memory copies and two synchronizations in the host: Firstly, the index of the best circle fitting was found and copied to the host, and secondly, the best circle fitting itself was copied to the host. This operation can be optimized if the kernel in charge of selecting the best fitting copies it to a given memory address that is known by the host. Therefore, only one copy and one synchronization are required. After removing this extra communication, the amount of frames processed per second raises to 747.4, whereas the new speedup to 34.0 $\times$ (37.2 $\times$ for the binocular setup).

Increasing thread re-usage Depending on the amount of work assigned to each thread sometimes, it is useful to reuse the same thread to process several elements instead of launching one thread per element. We have opted for this approach when the rays are

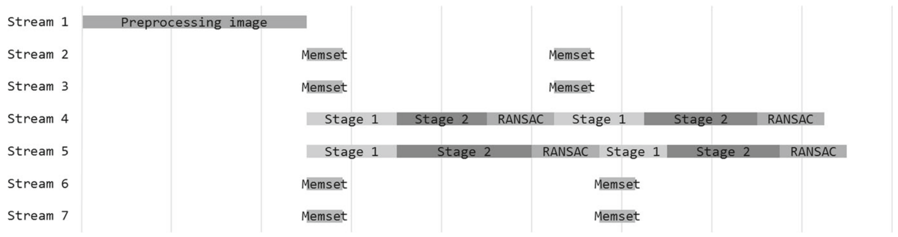

Fig. 8 Kernel execution overlapping when searching for two pupils in parallel

launched and inspected for locating the pupil border. In the previous approach, each thread was restricted to process only one pixel, while in the current implementation each thread is able to process several pixels. Only a small increase in the amount of frames per second is achieved with a final number of 753.0, and a speedup of 34.2 × (37.5 × for the binocular setup).

Reduction with the ballot instruction While performing a reduction with the CUDA *shuffle* instruction is a very fast approach (as seen in the previous subsection), there is a more efficient method for counting votes (which is the last step of the RANSAC operation). The CUDA *ballot* instruction sets the ith bit of a given integer to 1 when a thread is active and the value provided to *ballot* is different than zero. An integer mask is returned per each warp. Therefore, counting the bits set to 1 in that mask (with the *__popc* instruction) will return the number of votes obtained in the warp. Still the votes from every warp need to be added up, but using an *atomicAdd* at this level is not a bottleneck. After applying this optimization the number of frames per second raises to 760.2 and the speedup to 34.6 × (37.5 × for the binocular setup).

Using CUDA streams for binocular images In the binocular version, the search of the left and right pupils is completely independent and can be done in parallel. To increase the parallelism, we have made use of CUDA streams since they allow to concurrently run several kernels. Obviously, these kernels cannot have any dependency among themselves or they will have to be executed sequentially. Additionally, some time is saved because some *cudaMemset* calls that were previously performed sequentially have been replaced with the *cudaMemsetAsync* function, and so they can be executed in parallel with other computation or among themselves. Furthermore, some extra time is additionally saved due to the *cudaLaunch* calls overhead being completely hidden by computation. To better illustrate this effect, a simplified execution timeline is shown in Fig. 8. All the kernels (shown as different stages) run concurrently for both pupils, and only two explicit synchronizations on the host code are mandatory. The first synchronization is for testing that some points were found in the image. And the second synchronization is for copying the best circle of each iteration to the host to test if the termination condition is met. After applying this optimization to the binocular setup, the number of frames per second raises to 689.8 and the speedup to 40.8 ×.

Pipelining pupil tracking A similar approach as before can be done at a higher level, when a flow of images is considered for pupil tracking. Some systems are designed

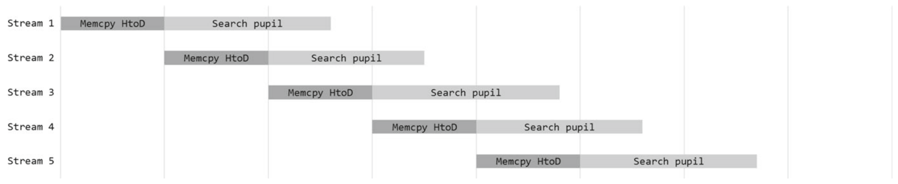

Fig. 9 Pipelining pupil tracking: overlapping computing stages with memory/communication phases

for achieving a low latency, while others are designed for obtaining a high throughput. So far we have optimized our pupil tracking system for a reduced latency, taking into account one single image. A different approach can directly focus on increasing the overall throughput of the real-time system even though the latency of processing an individual image may slightly increase as a side effect. This can be achieved by *pipelining* the whole algorithm and taking into account that our pupil tracking system is intended not for processing a single image but a real-time flow of images. Figure 9 illustrates how a classical pipelining approach has been applied to our process, efficiently overlapping different and independent computing stages and also overlapping computing with communication phases.

As the images are completely independent from each other, they can be processed in parallel. Thus, while one image is copied to the GPU, the previous one is processed, efficiently overlapping computation with memory operations. Note that we have also overlapped the different computation stages of several images. As a result, the usage of the GPU has increased and the throughput is maximized, although the latency of processing each image has increased. To enable this approach, it is necessary to use either dynamic parallelism in the GPU or several threads in the CPU. We have decided to implement it using dynamic parallelism in the GPU. The key is being able to launch several memcpys and several kernels for searching the pupils without waiting for the previous one to finish. As we have explained before, in the standard implementation there are a couple of mandatory synchronizations on the host side. Thus, it was not possible to launch several kernels for processing different pupils without waiting for the previous ones to finish. However, if the processing loop is encapsulated within a single kernel that copies the result to a given memory address, several kernels can be launched in parallel. Dynamic parallelism enables this approach because it allows new kernels to be launched from kernels that are already running in the GPU, so we only need to add a wrapper kernel to the processing loop. After applying this optimization the number of frames per second raises to 1260 and the speedup to 57.3 $\times$ (988 fps and 58.5 $\times$ for the binocular setup). Since the achieved performance depends on the number of streams used, its effect will be further analyzed and discussed in Sect. 5.1.2.

5 Accuracy and performance evaluation

5.1 Starburst algorithm evaluation

The accuracy of the Starburst algorithm for several image preprocessing configurations is shown in Fig. 10a. A deep exploration of the parameter space has been carried out,

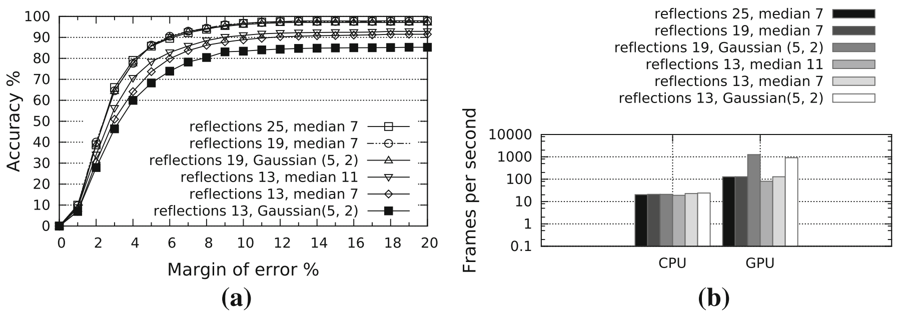

Fig. 10 **a** Starburst's accuracy for different configurations. **b** Starburst's performance for the same configurations

but for the sake of visibility we only report here six representative configurations. Two of the selected configurations 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 reflection removal kernel is also specified, with values between 13 and 25.

The accuracy of a pupil detection algorithm is measured as the percentage of images where the center and radius have been found within a given margin of error with respect to the correct position and size. For example, if the error in the calculated radius is 0.1 mm, whereas the error in the center is 0.05 mm, and assuming a pupil radius of 2 mm, then the relative margins of error are 5 and 2.5% for the pupil radius and the center, respectively. For our evaluation report, the biggest error is always used (5% in the previous example).

In Fig. 10a, the six configuration legends are ordered from more to less accurate. It can be observed that for the three first configurations more than 85% of the images detected with an error under 5% and also more than 95% of images detected with an error smaller than 10%.[4] To complement the accuracy evaluation, a performance comparison is shown in Fig. 10b for the same configurations. Clearly, using the median filter is slower than using the Gaussian filter due to the different performance of both methods, while a similar accuracy is achieved. Therefore, considering both accuracy and performance metrics, the best configuration was [reflections 19, Gaussian(5,2)] reporting an average of 1260 frames per second.

5.1.1 Ellipse versus circle fitting

As cited earlier, the form of the pupil varies significantly from subject to subject. Although it can be effectively approximated by a circle, it is more precisely approximated by an ellipse [38]. As we are developing a high accurate pupil tracking system, we have evaluated both cases.

In order to measure the accuracy of both approaches, all the images in the dataset have been manually fitted to an ellipse in the same way they were manually fitted

[4] Note that for the average pupil in the dataset (the average radius is 3.21 mm) a relative error of 5% will translate into an absolute radius error of 0.16 mm.

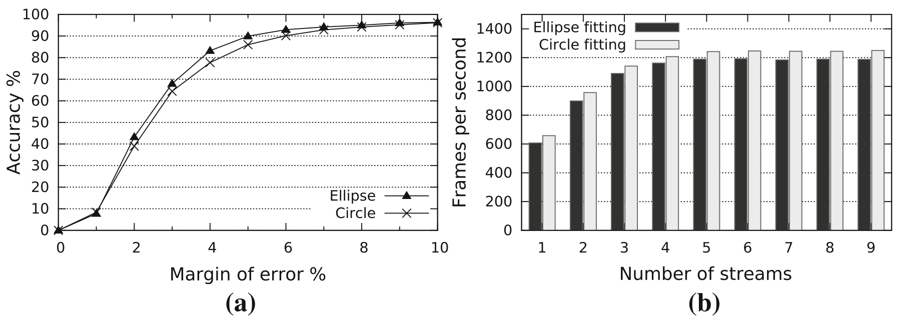

Fig. 11 **a** Accuracy of circle fitting versus ellipse fitting. **b** Performance of circle fitting versus ellipse fitting with varying number of streams

to a circle previously. Figure 11a shows a comparison of the accuracy of both circle and ellipse fittings. The overall accuracy is similar, and as we relax (increase) the margin of error, more pupils are successfully detected within a lower margin of error for the *ellipse fitting* than for the *circle fitting*, which makes the former a more accurate approach at the cost of some performance degradation.

As expected fitting to an ellipse is more computing intensive than fitting to a circle. As a result, the performance of pupil tracking is reduced when the ellipse fitting is used, as it is shown in Fig. 11b. We also evaluate here the impact of using a different number of CUDA streams (as explained in 4.1). In general, it can be seen that using the *ellipse fitting* degrades performance in an absolute number of around 60 fps (on average) regardless of the number of CUDA streams used. This represents less than 5% of performance degradation (in terms of *fps*) that could be tolerated in case the extra accuracy is required.

5.1.2 Pipelining evaluation

As illustrated in Fig. 9, the pipelined implementation for processing the pupil images can be configured with a varying number of CUDA streams. In the *non-pipelined* implementation, where a single image was processed at a time, the kernels were launched from the host (the CPU). Alternatively, in the new implementation several images are processed concurrently and the kernels are launched from the device (the GPU). Since CUDA version 5.0, it is possible to launch new kernels from the device itself using what NVIDIA refers to as *dynamic parallelism*. While this feature is intended to dynamically change the amount of threads working in some data, it has another interesting use case. In particular, it can be used for wrapping several calls to kernels, memcpys and/or synchronizations into a single kernel to be run individually.

In the previous *non-pipelined* Starburst implementation, we couldn't start processing one image until the result of the previous one was copied to the host and the convergence criterion was met. However, the new implementation using *dynamic parallelism* can process in parallel as many images as needed without waiting for the convergence criterion because this is now tested by a CUDA thread, so no synchro-

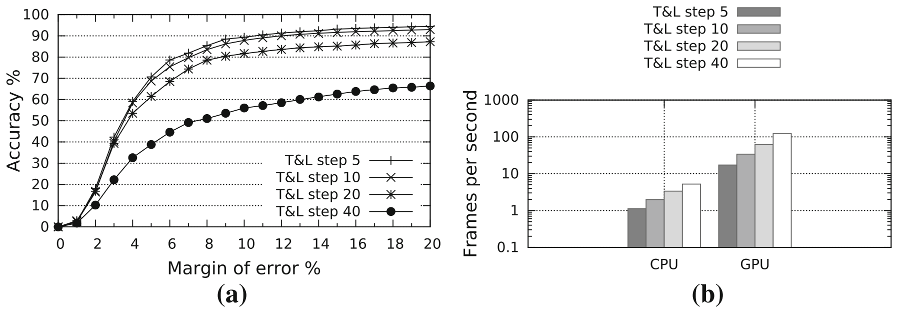

Fig. 12 **a** Thresholding and labeling accuracy for different step sizes. **b** Thresholding and labeling performance for the same step sizes

nization is required on the host side. Still, the new *pipelined implementation* adds some overhead, and therefore, the speed is slightly reduced if only one stream is used.

Figure 11b shows the results of running the *pipelined* implementation with different amounts of CUDA streams, ranging from 1 to 9. As it can be observed, the maximum rate achieved by the circle fitting version is 1260 fps (a speedup of 57.3 ×), while the ellipse fitting version achieves 1180 fps (a speedup of 53.6 ×). Note also that in the non-pipelined GPU version 33% of the time is used for transferring the images from the host to the device; therefore, the pipelined version is expected to achieve at least 50% more frames per second due to overlapping memory copies with computation. Furthermore, the throughput has been increased by an additional 17% due to the benefit of overlapping the computation of different stages for different images, in addition to hiding the overhead of launching kernels and memory copies, as illustrated in Fig. 9.

5.2 Thresholding and labeling algorithm evaluation

The accuracy of the thresholding and labeling algorithm (referred to as T&L in Fig. 12a) is quite good taking into account that this algorithm is very sensitive to overlapping of the eyelids and eyelashes. In particular, it is able to achieve an accuracy over 90% for small steps within a reasonable margin of error. However, its accuracy decreases as the *step size* parameter is increased. As expected, the performance (Fig. 12b) is directly related to the step size parameter: An increase in the step size results in a reduced execution time because it decreases the number of iterations of the algorithm. Moderately high speeds are achieved with a step size of 20 and 40 (61 and 121 fps, respectively). After examining the accuracy and the performance of the T&L algorithm, it is clear that there is a trade-off between them both. Increasing one will decrease the other, so a balance must be found.

5.3 Hough transform algorithm evaluation

The Hough transform algorithm turns out to obtain very accurate results (Fig. 13a). Different configurations have been tested changing the amount of pixels voting for each

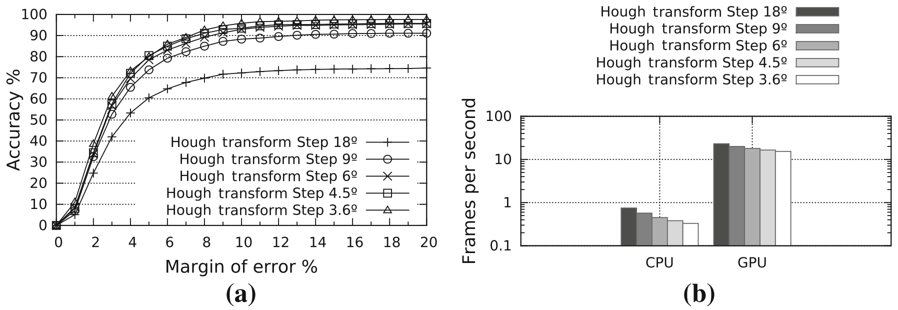

Fig. 13 **a** Hough transform accuracy for different step sizes. **b** Hough transform performance for the same step sizes

position and radius. In particular, we have tested the following five configurations: One pixel tested every 18°, every 9°, every 6°, every 4.5° and every 3°. Depending on the *step size* parameter, the accuracy and performance are either increased or decreased. As expected, the best accuracy is achieved by the configuration with the smallest step and vice versa. Despite its accuracy, it is a very slow algorithm, as it is shown in Fig. 13b. The Hough transform requires a huge computing power because it is checking all the possible radii for all the pixels in the image (refer to Sect. 2.4 for the algorithm details). Again, the performance depends on the step size used, and as expected the configurations with bigger step sizes are faster than those with smaller step sizes (with a performance ranging from 15 to 23 fps).

5.4 Side-by-side comparison of the three algorithms

A side-by-side comparison of the accuracy (Fig. 14a) and the performance (Fig. 14b) helps to understand how well the three algorithms behave. For the sake of simplicity, only the best configuration for each algorithm is shown. For Starburst, a corneal reflection removal kernel of size 19 and a Gaussian filter of size 5 and sigma 2 are used. For thresholding and labeling, a step size of 20 is chosen. And for Hough transform, a step size of 3.6° has been used.

If both metrics, performance and accuracy, are used to compare the three algorithms, 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 (96.4% of the pupils detected within a margin of error of 10%), it exhibits a extremely low performance (15.3 fps) making it unusable for real-time applications. Contrarily, the Starburst algorithm achieves 1260 fps for the mentioned accuracy. On the other hand, the threshold and labeling 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 fps. Summarizing, the GPU-accelerated version of

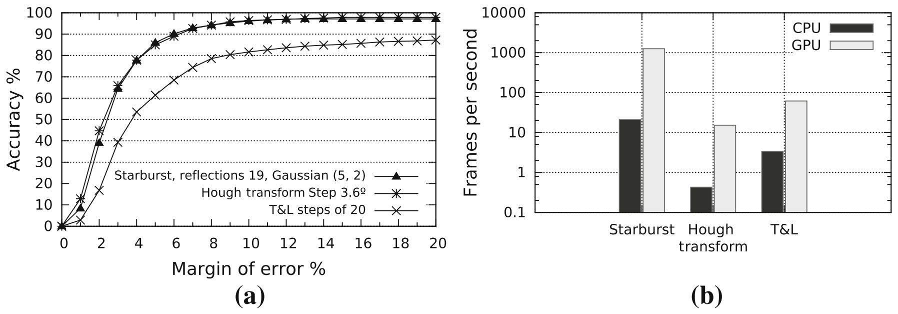

Fig. 14 **a** Accuracy comparison of the 3 algorithms. **b** Performance of the 3 algorithms using a resolution of 1280 × 1024

Starburst, after applying all the CUDA optimizations discussed in this work, is by far the fastest approach while exhibiting a high accuracy.

6 Related work

GPU processors are used nowadays for an increasingly number of applications. As expected, some works are related to pupil tracking on GPUs. Mulligan was the first to develop a GPU-based implementation of a pupil tracking algorithm [26]. It was a simple algorithm using a threshold for determining whether a pixel belongs to the pupil or not. Then, the center of mass of the pixels believed to be inside the pupil is calculated. Furthermore, only a relatively small region of interest of 192 × 192 is used (the total size is 640 × 480) for calculating the center of mass. Finally, a test is performed to accept or discard the ROI as an acceptable pupil. However, this approach may have problems to find the pupil when the eyelashes are partially hiding it or when there are big corneal reflections inside the pupil. The speed achieved by this implementation was quite outstanding when the paper was published, and it managed to process up to 250 fps for low-resolution images.

Another approach was proposed by Borovikov, a GPU-based implementation of an algorithm designed for color images [5]. A custom blob detector is used for searching the pupil. It is an iterative method, which assumes that the pupil is a dark blob close to the image center. A weighted center of mass is calculated during each iteration. In order to calculate the center of mass, only the pixels inside of a given radius around the previous center of mass are used, and the darker the pixels, the more the weight they are assigned. The termination condition evaluates the distance between the current center of mass and that from the previous iteration, checking whether the difference is smaller than a given threshold. Another pupil tracking implementation using GPUs was published by Du Plessis and Blignau [11]. They present a GPU-assisted eye tracker which is capable of attaining a sample frequency of 200 Hz in a mid-range laptop. Due to the limitations of the hardware used, they obtained a low throughput even though they used a small ROI for performing the tracking.

However, the pupil tracking literature related to GPGPU is relatively small if it is compared with all the available papers about GPGPU. There is a huge number of papers related to medical image processing [33,37] and image processing [39]. Some of the problems those papers try to solve are closely related to the ones we try to solve in this paper. We perform some image processing operations and the final purpose is to find the pupil. Comparing with most of the literature regarding medical image processing, the idea is the same. Some operations are performed in the images to find something (organs, tumors, bones, etc.).

More pupil tracking algorithms without using GPUs can be found in the literature. A well-known approach is the Starburst algorithm [19], one of the evaluated in this paper. The original algorithm only processes a small percentage of the pixels of the image; therefore, it achieves a higher throughput than others with higher processing requirements. Still it has a high computing cost when high-resolution images are used. As explained in Sect. 2, to track the pupil a few lanes of pixels going from an initial position to the edges of the image are processed. The algorithm expects to find a sharp change within the gradient of these lanes, and that pixel is considered to be in the border of the pupil. In a second stage, the process is repeated, but now the lanes go backwards from the found points toward the previous initial position within a search angle. The goal of the second stage is to find points in the opposite border of the pupil. After finding a set of border pixels, its center of mass is calculated and the process is repeated until it converges. Finally, a RANSAC operation is performed to calculate the best fitting circle to the pupil border points that have been found.

Another approach to find the border of the pupil proposes the use of an algorithm called Graph Cuts [24]. The algorithm searches the minimum *cut* between the pupil and the rest of the eye. In order to do this, the pixels are encoded as nodes of a graph, while the edges are their relationships. Since the algorithm expects the pixels inside the pupil to be dark, a previous step for locating and avoiding the eyelashes is required. Otherwise, they could be wrongly labeled as pupil border points. Then a pixel from inside the pupil and a pixel from outside are selected as initial values for starting the Graph Cuts search.

Another interesting approach has been proposed using the algorithm Gradient Vector Flow (GVF) snake [15]. This algorithm is capable of finding the borders of a given object with a very high precision. An initial estimation of the pupil position is calculated using a region of interest around the center of the image. This might be a problem if the pupil was outside of that area, but this is a strange case in most setups. An histogram of the selected area is calculated in order to select a good threshold, and later, a morphological processing is applied to remove the eyelashes and others dark areas of the eye. This algorithm results in a highly accurate location of the pupil border. However, it relies on very high contrast images and it may not perform as well with images with a lower contrast.

While the above are somehow complex algorithms for tracking the pupil, simpler approaches which offer reasonable good results can be also found in the literature. The pupil usually has a quite well-defined edge around it; therefore, a combination of the Hough transform and the Canny edge detector algorithm [35] has also been used for detecting the pupil [23]. Although the initial goal was doing iris segmentation, still it can be used for pupil tracking. As it is typical in other algorithms, this one requires a preprocessing stage for removing the eyelids. Then the Canny algorithm is used for processing the image and the Hough transform searches for a circle within a range of possible radii. The method is quite accurate (as shown in Sect. 5), although it is sensitive to all the features present in the image, either from the eye itself or from the corneal reflections. However, it is a extremely slow approach given the huge computing power required by the Hough transform.

Another technique consists in using blobs as applied in [31]. In this approach, the central blob of the image is assumed to be the pupil. This is a rather simple method. Using a threshold the pupil is selected from the whole image. However, there are other parts of the eye that could make this method to fail, such as the eyelids and the eyelashes, which frequently overlap with the pupil and could result in a bad detection. Note that this method was also intended to perform iris segmentation, and therefore, the iris is expected to be visible and so the pupil.

Gaze tracking is based on tracking the pupil and it is also a very interesting topic in many areas. There are multiple approaches in the literature about how to do it [21]. The typical setup used for gaze tracking is a bit different than the one used just for pupil tracking. For gaze tracking, the subject usually has a big freedom of movement and a wide-field angle camera is used. Still infrared illumination is preferred. As a result, the resolution available for tracking the pupil is highly reduced comparing with setups where the camera is much closer to the eye. Therefore, the precision for detecting the pupil is reduced as well [12,32]. It would be also possible to have a camera tracking the movements of the subject which would be closer to the subject but keeping a high freedom of movement. But it would be an expensive, complex and most probably an error-prone system. However, having a head-mounted system allows head movements while increasing the resolution available for performing pupil tracking [2,18].

It is also possible to use a different illumination setup to track the pupil. By using two LEDs placed at different distances of the camera, the dark and bright pupil effect can be generated turning on one of the LEDs at a time. This system was proposed by Ebisawa [28] resulting in an increase in the complexity of the illumination setup and its synchronization with the camera but simplifying the pupil tracking. Finally, GPUs have also been used in a related field to pupil tracking as it is *blink detection* [17]. Although for this application the whole head is included in the images, it again demonstrates how well GPUs fit real-time image processing.

7 Conclusions

This work demonstrates that highly accurate and fast pupil tracking can be achieved. This has been possible with the Starburst algorithm that has been parallelized by using CUDA. The speedup achieved by a high-end GPU is 57.3 $\times$ with respect to the non-parallelized version, highly surpassing the 2.2 $\times$ speedup obtained by using OpenMP in a high-end multicore CPU. Indeed, the performance of the parallelized binocular pupil tracking algorithm is so high (988 fps) 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 GPU-accelerated algorithm to perform accurate and real-time (25 Hz) pupil tracking even at much higher 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. Summarizing, the proposed parallel implementation is a powerful tool that will help to improve many visual optics applications which rely on pupil tracking, enabling processing speeds and a pupil tracking accuracy that were not possible previously.

Acknowledgements This research has been supported by the European Research Council Advanced Grant ERC-2013-AdG-339228 (SEECAT), "Fundación Séneca," Murcia, Spain (Grant 19897/GERM/15), and the Spanish SEIDI under Grants FIS2013-41237-R, and TIN2015-66972-C5-3-R, as well as European Commission FEDER funds.

References

1. Aydi W, Masmoudi N, Kamoun L (2011) New corneal reflection removal method used in iris recognition system. World Acad Sci Eng Technol 5(5):898–902
2. Babcock JS, Pelz JB (2004) Building a lightweight eyetracking headgear. In: Proceedings of the ACM Symposium on Eye Tracking Research and Applications, pp 109–114
3. Ballard DH (1981) Generalizing the hough transform to detect arbitrary shapes. Pattern Recognit 13(2):111–122
4. Bell N, Hoberock J (2011) Thrust: a productivity-oriented library for cuda. GPU Comput Gems Jade Ed 2:359–371
5. Borovikov I (2009) Gpu-acceleration for surgical eye imaging. In Proceedings of the 4th SIAM Conference on Mathematics for Industry (MI09), San Francisco, CA, USA
6. Bradski G (2000) The opencv library. Dr. Dobb's J Softw Tools Prof Program 25(11):120–123
7. Canny J (1986) A computational approach to edge detection. IEEE Trans Pattern Anal Mach Intell 8(6):679–698
8. Chen P, Zhao HL, Tao C, Sang HS (2011) Block-run-based connected component labelling algorithm for gpgpu using shared memory. IET Electron Lett 47(24):1309–1311
9. Chirre E, Prieto PM, Artal P (2014) Binocular open-view instrument to measure aberrations and pupillary dynamics. Opt Lett 39(16):4773–4775
10. cuRAND library, CUDA 7 (2010) NVIDIA Corporation, Santa Clara
11. Du Plessis J-P, Blignaut P (2016) Performance of a simple remote video-based eye tracker with gpu acceleration. J Eye Mov Res 9(4):1–11
12. Hansen DW, Majaranta P (2011) Basics of camera-based gaze tracking. In: Majaranta P (ed) Gaze interaction and applications of eye tracking: advances in assistive technologies. IGI Global, Hershey, pp 21–26
13. Harris M et al (2007) Optimizing parallel reduction in CUDA. NVIDIA Dev Technol 2(4). http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf
14. Hennessey C, Noureddin B, Lawrence P (2008) Fixation precision in high-speed noncontact eye-gaze tracking. IEEE Trans Syst Man Cybern Part B Cybern 38(2):289–298
15. Jarjes AA, Wang K, Mohammed GJ (2010) GVF snake-based method for accurate pupil contour detection. Inf Technol J 9(8):1653–1658
16. Koprowski R, Szmigiel M, Kasprzak H, Wróbel Z, Wilczyński S (2015) Quantitative assessment of the impact of blood pulsation on images of the pupil in infrared light. JOSA A 32(8):1446–1453

17. Lalonde M, Byrns D, Gagnon L, Teasdale N, Laurendeau D (2007) Real-time eye blink detection with gpu-based sift tracking. In: Proceedings of the IEEE 4th Canadian Conference on Computer and Robot Vision, Montreal, Canada, pp 481–487
18. Li D, Babcock J, Parkhurst DJ (2006) Openeyes: a low-cost head-mounted eye-tracking solution. In: Proceedings of the ACM Symposium on Eye Tracking Research and Applications, San Diego, California, pp 95–100
19. Li D, Winfield D, Parkhurst DJ (2005) Starburst: a hybrid algorithm for video-based eye tracking combining feature-based and model-based approaches. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)—Workshops, San Diego, CA, USA
20. Lindholm E, Nickolls J, Oberman S, Montrym J (2008) Nvidia tesla: a unified graphics and computing architecture. IEEE Micro 28(2):39–55
21. Majaranta P, Bulling A (2014) Eye tracking and eye-based human–computer interaction. In: Fairclough SH, Gilleade K (eds) Advances in physiological computing. Springer, London, pp 39–65
22. Manzanera S, Prieto PM, Benito A, Tabernero J, Artal P (2015) Location of achromatizing pupil position and first purkinje reflection in a normal population. Invest Ophthalmol Vis Sci 56(2):962–966
23. Masek L et al (2003) Recognition of human iris patterns for biometric identification. Master's Thesis, University of Western Australia
24. Mehrabian H, Hashemi-Tari P (2007) Pupil boundary detection for iris recognition using graph cuts. In: Proceedings of the International Conference on Image and Vision Computing New Zealand (IVCNZ), pp 77–82
25. Mompeán J, Aragón JL, Pedro P, Pablo A (2015) Gpu-accelerated high-speed eye pupil tracking system. In: 2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Florianópolis, Brazil, IEEE, pp 17–24
26. Mulligan JB (2012) A GPU-accelerated software eye tracking system. In: Proceedings of the ACM Symposium on Eye Tracking Research and Applications, Santa Barbara, CA, USA, pp 265–268
27. Nvidia Corporation (2015) CUDA C Programming guide
28. Ohtani M, Ebisawa Y (1995) Eye-gaze detection based on the pupil detection technique using two light sources and the image difference method. In: IEEE 17th Annual Conference on Engineering in Medicine and Biology Society, 1995, vol 2. IEEE, pp 1623–1624
29. OpenMP Architecture Review Board (2011) OpenMP application program interface version 3.1
30. Podlozhnyuk V (2007) Image convolution with CUDA. NVIDIA Corporation White Paper, vol 2097, no 3
31. Rankin DM, Scotney BW, Morrow PJ, McDowell DR, Pierscionek BK (2010) Dynamic iris biometry: a technique for enhanced identification. BMC Res Notes 3(1):182
32. San Agustin J, Skovsgaard H, Mollenbach E, Barret M, Tall M, Hansen DW, Hansen JP (2010) Evaluation of a low-cost open-source gaze tracker. In: Proceedings of the ACM Symposium on Eye-Tracking Research and Applications, Austin, Texas, pp 77–80
33. Schellmann M, Gorlatch S, Meiländer D, Kösters T, Schäfers K, Wübbeling F, Burger M (2011) Parallel medical image reconstruction: from graphics processing units (gpu) to grids. J Supercomput 57(2):151–160
34. Schwarz C, Prieto PM, Fernández EJ, Artal P (2011) Binocular adaptive optics vision analyzer with full control over the complex pupil functions. OSA Opt Lett 36(24):4779–4781
35. Soltany M, Zadeh ST, Pourreza H-R (2011) Fast and accurate pupil positioning algorithm using circular Hough transform and gray projection. In: Proceedings of the International Conference on Computer Communication and Management (CSIT), Sydney, Australia, vol 5, pp 556–561
36. Taubin G (1991) Estimation of planar curves, surfaces, and nonplanar space curves defined by implicit equations with applications to edge and range image segmentation. IEEE Trans Pattern Anal Mach Intell 13(11):1115–1138
37. Valero P, Sánchez JL, Cazorla D, Arias E (2011) A gpu-based implementation of the mrf algorithm in itk package. J Supercomput 58(3):403–410
38. Wyatt HJ (1995) The form of the human pupil. Vis Res 35(14):2021–2036
39. Yam-Uicab R, Lopez-Martinez JL, Trejo-Sanchez JA, Hidalgo-Silva H, Gonzalez-Segura S (2017) A fast Hough transform algorithm for straight lines detection in an image using gpu parallel computing with cuda-c. J Supercomput 73(11):4823–4842