

# GPU-acceleration for Surgical Eye Imaging

Igor Borovikov \*

## Abstract

This paper discusses GPU-related aspects of developing computer vision algorithms for detection of the pupil and limbus (the boundary between the iris and the white of the eye). As an example for our case study of GPU-enabled algorithms, we introduce and explore a custom version of blob detector used for finding the pupil in pre-operational images. This work was completed as part of an exploratory project by Kerner Graphics, Inc. to identify possible performance and resolution improvements for 3D eye surgery imaging products being developed by TrueVision Systems, Inc.

## 1 Pupil and Limbus Detection.

Our goal was to build robust, real-time algorithms for tracking the main eye features during pre-operation and in-operation procedures. Having those features identified in the pre-operational images is important for later tracking of the eye during surgery and allows matching it to the pre-operational state. While the limbus is the main feature of interest, its direct detection can be challenging. We chose an approach relying on finding the pupil first and then using its center as a reference point for subsequent limbus detection, as in number of previous works (i.e. [1], [2] and [3]).

## 2 Previous Work and Pupil Detection.

There are numerous works in the area of detecting the pupil and iris of an eye under different conditions and for different purposes. Several relatively recent papers offer comprehensive reviews of methods developed for the task of detecting the iris and pupil ([1], [2], [3], [4] and [5]).

We found that the success of the methods described in the existing publications often critically depends on the assumptions made about data acquisition, variety of handled cases and the required accuracy and robustness of the method. For example in [4] the method has to work with a database where the images were normalized during “enrollment process” and hence the acquisition and filtering of the input images has to be consistent with those of the images stored in the database. Other works (e.g. [1]) assume highly controlled envi-

ronment for data acquisition. Such assumptions and prerequisites make it difficult, if possible at all, for direct comparison of different methods. That proved to be true when we attempted to reproduce basic results of the methods based on the general approach of applying Hough transform to the edge image.

Methods based around Hough transform are among the most popular for iris and pupil detection. Normally we can expect that the pupil boundary is fully visible on pre-operational images - unlike the limbus boundary that can be partially covered by eyelids. Having the pupil completely visible suggests that we may attempt direct detection of the pupil as a circular feature using general techniques such as circular Hough transform. However, such a direct approach proved to be hindered with a number of problems in the case of our data which were quite noisy and uncalibrated.

One of the problems is that standard Hough transform implementations (like that one in OpenCV) are sensitive to noise in the image. A fair amount of pre-processing was required to make the boundary of the pupil more suitable for meaningful Hough transform application. The filtering techniques we applied were ranging from aggressive noise removal to thresholding and Mean Shift Filtering that would convert the initial image to its denoised segmented version.

Even with a relatively clean input image, tweaking the parameters of the Hough transform proved to be challenging over the entire range of samples as the method was still sensitive to the strength of the pupil boundary and was also easily selecting other circular features like boundaries of highlights. To avoid this obstacle, highlights were isolated by histogram-based thresholding and then masked out.

Further improvement was achieved by collecting a large enough number of circles detected with Hough transform, next filtering the collected set and finally clustering it to detect the most prominent circular feature. This worked well for most of the input samples but was still failing on pupils with low contrast boundary.

A literature search showed that, indeed, approaches based on Hough transform combined with various edge detection and thresholding techniques are not necessarily well suited for the task (see related discussion in [1]).

The need to overcome the challenges of low-contrast

---

\*Kerner Graphics, San Rafael, CA.

cases led us to looking for alternative methods like the custom blob detector described next.

### 3 Custom Blob Detector for the Pupil.

Our main objective was to provide a robust method of pupil detection in the case of uncalibrated images with large variety of contrast, in particular with low contrast of the pupil-iris boundary. Different approaches we tried were evaluated on the number of outliers that were not handled by the algorithm with acceptable results. We resisted a temptation to declare a number of underexposed (hence - quite noisy) images as outliers so we still used them in the tuning process. The motivation behind this was to reduce the required human operator interference to a minimum. This allowed the method to converge on a robust custom blob detector described in this section.

To find the pupil, we were looking for a dark blob near the center of the image. While standard blob detection techniques can be used, we opted for a custom one that would also return a good approximation of the pupil radius. This method is the focus of our case study of converting CPU-based OpenCV implementation to GPU-accelerated solution.

The method is based on an iterative procedure where each iteration  $i$  would find the center of mass  $p_i$  of weighted  $RGB$  pixels  $c(x, y)$  in the circular area  $a_i$  centered around current approximation to the pupil center  $p_{i-1}$ :

$$(3.1) \quad p_i = \frac{\sum_{(x,y) \in a_i} I(x, y)W(x, y)M(c(x, y))}{\sum_{(x,y) \in a_i} M(c(x, y))}$$

Here  $I(x, y) = (x, y)$  is simply the identity mapping  $I : Z^2 \rightarrow Z^2$  of pixel coordinates providing for this equation to be vector-valued. Note that the integer 2D point ( $Z^2$ -valued) coordinates of the pixels result in 2D coordinates for  $p_i$  (though  $p_i \in R^2$ ).

The pixel color function  $c : Z^2 \rightarrow RGB$ , maps pixel coordinates at  $(x, y)$  into the pixel color in the  $RGB$  domain, which is the red-green-blue color space  $[0, 1]^3 \subseteq R^3$ .

The function  $M : RGB \rightarrow D$  where  $D \subseteq [0, 1]$  is defined as measure of a pixel relative darkness and was calculated to provide higher value to darker pixels with a desired hue, i.e. to the pixels of the pupil.

The weight function  $W : Z^2 \rightarrow [0, 1]$  has an inverted bell-shape around the current center  $p_{i-1}$  (discussed below) and also penalizes proximity to the image edge.

The first approximation for  $p_1$  is picked in the center of the image. When each next  $p_i$  is calculated, the method checks for the distance  $|p_i - p_{i-1}| > T_i$  to determine when to stop iterations. Here  $T_i$  is the

threshold.

Starting with a relatively large radius  $r_i$  of the area  $a_i$  we are decreasing it when successive iterations converge. When the next change of the radius doesn't change the location of the center, we declare the entire method converged.

The particular dependence of the weight function  $W(x, y)$  on the normalized distance  $r$  from the current pupil center was picked to emphasize boundary pixels versus those close to the center:

$$W(r) = \begin{cases} r(2-r), & 0 \leq r \leq 1 \\ 0, & r > 1. \end{cases}$$

where

$$r = \frac{R(x, y)}{R_c}$$

is the distance  $R(x, y)$  from the current pupil center normalized by its current radius  $R_c$ . This choice was made after extensive experimentation and is purely empirical. Apparently, many functions with similar shape of the graph emphasizing the disk edge will work, so we used this freedom to simplify software implementation.

In our tests, a bell-shaped weight function (say,  $1-r^2$ ) was tending to end iterations too early, as soon as uniform color area was reached. Also the low weight of boundary pixels was making it not sensitive enough to detect the actual boundary of the pupil. The "inverted" weight  $W(r)$  was providing both good convergence and robust estimation for the pupil radius.

A typical result of method application is shown on Figure 1. The processed background image on this figure is the result of applying aggressive low-pass filtering to the original image. Due to such pre-processing it was possible to avoid masking out highlights in most cases.

The initial circle is drawn with red color. The radius of the circles is determined using an adaptive algorithm ensuring that the disk enclosed by the circle contains the boundary of the pupil. This can be done by analysing the histogram of the image inside the circle.

The histogram for the first step is drawn in green in the left bottom corner of the image. It shows two distinct peaks corresponding to the two image segments intersected by the disk: the pupil and the iris. While decreasing the radius on each subsequent iteration we would stop at the smallest radius that still exhibits "two-peaks" quality of the histogram. Of course, the initial location and radius of the circle must be picked to satisfy this condition too. The final histogram inside the

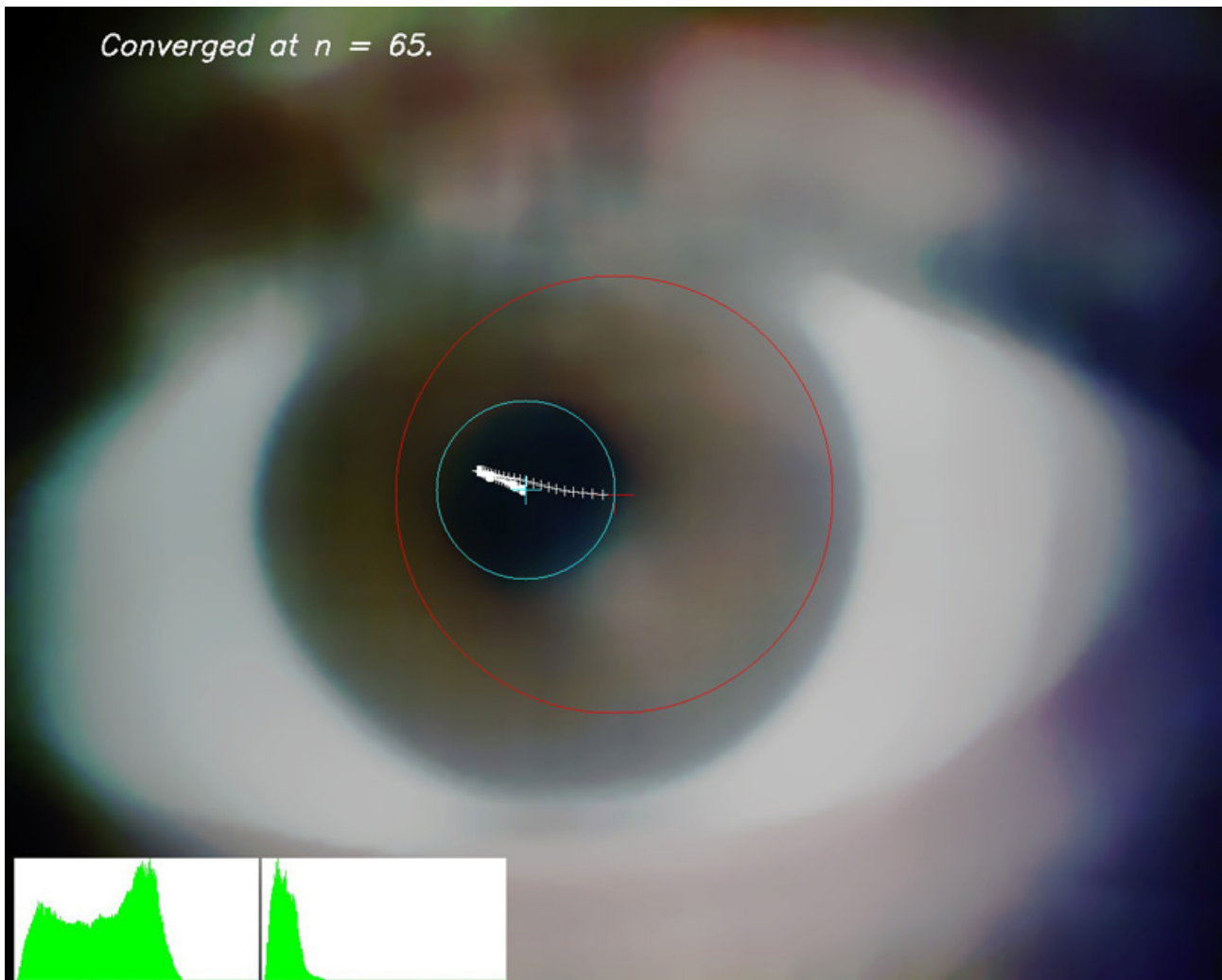


Figure 1: Convergence of Custom Blob Detector in  $n = 65$  iterations. (Sample data courtesy TrueVision.)

final circle (drawn in cyan) of the converged algorithm is drawn next to the initial. Intermediate centers are drawn as white crosses.

The backtracking of the method is clearly visible on Figure 1 and can be easily explained. Initially it was happening due to skewed directional lighting: on the early iterations the larger circular areas were attracted to the darker part of the iris in the left part in the image space. However, with the radius reduced during later iterations, the method successfully converged to the pupil.

The described method for detecting the pupil succeeded on almost all samples and provides a good estimate for the pupil radius with the error comparable to the deviation of the pupil shape from a circle. The method performed well also for low contrast cases where even a human observer would have difficulty to out-

line the pupil. Unfortunately direct comparison of this method with the existing ones was not immediately possible as none of the techniques described in the cited publications were not immediately applicable to our input data.

#### 4 Implementing Pupil Detector with CUDA.

While the described algorithm was giving good accuracy in approximating the pupil and its center, its performance was not particularly fast, far below practical applications requirements. It was possible to consider a trade-off between accuracy and speed, but a port to GPU would provide a better opportunity to speed up calculations.

NVidia's CUDA (Compute Unified Device Architecture) is a framework for parallel computing that uses graphics processing units (GPUs) for numeric compu-

tations [6]. The API provided by NVidia allows easy integration of CUDA into C/C++ applications. Availability of CUDA for consumer level hardware makes it particularly attractive from the point of view of numerical performance/cost ratio.

A direct port from OpenCV to CUDA would give sub-optimal performance, though, so we had to take into account CUDA massively parallel computing model specifics and ensure coalescent memory access [6]. Namely coalescent memory access was the main focus of our next phase of the project.

The implementaton strategy was chosen based on the availability of off-shelf CUDA samples and the complexity of the required custom code. One approach would be to directly implement parallel calculation of both numerator and denominator of (3.1) in corresponding kernels (pieces of CUDA code that run in highly parallel manner on GPU) but that would require large custom kernels and hence non-trivial optimization efforts to ensure balanced execution.

One of the alternatives to such a straightforward approach was to allow allocation of extra GPU memory for caching per-pixel  $x$ - and  $y$ -components of numerator and per-pixel weight from the denominator of (3.1) in order to minimize custom GPU code and reuse a very efficient SDK sample to perform arrays reduction (i.e. summation in our case).

The “Reduction” [7] sample picked for the adaption was providing a highly efficient algorithm for calculating a “reduced” value (e.g. max, min or sum) for an array of input values by applying an hierarchical approach by breaking the original data into optimal size sub-arrays and by handling them in a parallel manner until all levels of the hierarchy are considered.

Initially each of our arrays (i.e. numerator and denotminator values) was calculated using a dedicated kernel. But since calculations for all three arrays of weights are very similar, it was beneficial to combine them into single custom kernel (see optimization results table). Besides other arguments, the kernel takes three arrays of per-pixel “masses” to accumulate results. The amount of additionally required GPU memory is obviously in the order of  $O(N)$  where  $N$  is number of pixels in the original image with a small caveat explained below.

Finally, the summation for all three cached arrays of weights was done using modified CUDA SDK sample “Reduction” which provides a highly optimized algorithm and code tailored for CUDA architecture. The minor caveat was that the sample code requires input arrays to be of size of power of two so the linear growth of additional memory is actually happening in steps: when image data size exceeds the current upper lim-

	CPU	A	B	C	D	E
time ms	90.6	49.1	5.02	3.25	2.42	2.22

Table 1: Comparison of CPU and GPU performances.

iting  $2^n$  we need to allocate twice more memory  $2^{n+1}$  to accomodate the arrays for weights. This was not a problem as the memory allocation and de-allocation was well within capabilities of modern graphics hardware for all samples we had at our disposal. Additionally, this memory allocation was cycle-invariant with respect to the iterations cycle so it was possible to move it outside of the cycle during optimization. Thus the arrays for weights and GPU image data would be allocated once and then reused during iterative process making each iteration much more efficient.

Tests of the CUDA-enabled application were done on GTX260 hardware and compared to non-parallelized OpenCV CPU implementation. The results are presented in the Table 1.

We were interested in average time per iteration. Most of the iteration complexity is in the calculation of the new center of mass in (3.1). We compared average iteration time using CPU and CUDA code in different stages of optimization on typical input images of resolution 1280x1024. The CPU code was executed on Intel ®Xeon 3Ghz with 800Mhz front bus system. The graphics card was deployed on the same system in PCI Express slot (which throttled by half from the maximum possible exchange rate of the PCI Express 2 interface of the video board). However throttled bandwidth was not much of a concern after all cycle-invariant memory operations were factored out.

The table legend is the following. Stage A corresponds to “naive” CUDA version with no additional optimization applied. Stage B shows significant improvement due to moving invariant VRAM and RAM allocations outside of the iterations cycle (which is a standard optimization technique). On the Stage C three separate kernels for (3.1) calculations were combined into a single one. Next (stage D), conditional logic in the kernel was optimized. Finally, on stage E, calculations of the color weights function and penalty for proximity to the image borders was moved out to the dedicated preprocessing kernel. That simplified the main weight kernel called each iteration.

The overall port to CUDA resulted in about 40x speed up with respect to the original CPU version. Further optimization of CUDA-based implementation is very likely possible but may not be practical as the bottlenecks of the overall performance moved to the other parts of the algorithm.

It is also worth noting that the lower floating

point precision normally used for GPU calculations caused no discernible numerical degradation in the output, as the results from CPU and GPU versions were nearly identical. This can be explained by a stably convergent iterative nature of the algorithm (errors of earlier iterations would be corrected on later iterations) and also by rather high threshold (around 1 pixel) that would stop iterations.

## 5 Conclusion.

The developed method showed promise especially in its CUDA-based implementation. Even with up to a hundred of iterations of custom pupil detector per image the resulting code was running at interactive or near-interactive speed. We are looking into implementing further steps of the algorithm (limbus detection and tracking during surgery), with CUDA in mind as well.

## 6 Acknowledgements.

TrueVision provided sample images for this work. Together with Norman Noble and Chris Smith at Kerner Graphics, TrueVision also provided feedback and valuable discussions which the author greatly appreciates.

## References

- [1] D. R. Iskander, M. J. Collins, S. Mioschek, and M. Trunk *Automatic Pupillometry From Digital Images* IEEE Transactions on Biomedical Engineering, Vol. 51, No. 9, (September, 2004) pp. 1619-1627.
- [2] E. M. Arvacheh, H. R. Tizhoosh *Iris Segmentation: Detecting Pupil, Limbus and Eyelids* Image Processing, 2006 IEEE International Conference on 8-11 Oct. 2006, pp. 2453 - 2456
- [3] W. J. Ryan, A. T. Duchowski, S. T. Birchfield *Limbus/pupil switching for wearable eye tracking under variable lighting conditions* March 2008, ACM ETRA '08: Proceedings of the 2008 Symposium on Eye Tracking Research and Applications pp. 61-64.
- [4] S. Dey, D. Samanta *An Efficient Approach for Pupil Detection in Iris Images* International Conference on Advanced Computing and Communications, 2007. AD-COM 2007. pp. 382-389
- [5] Li Peihua, Liu Xiaomin *An incremental method for accurate iris segmentation* 19th International Conference on Pattern Recognition, 2008. ICPR 2008. pp. 1-4
- [6] NVIDIA CUDA: *Compute Unified Device Architecture*, Programming Guide, NVidia, 2009
- [7] M. Harris *Optimizing Parallel Reduction in CUDA* NVidia CUDA SDK Reduction Sample, 2009