In *Proceedings of the 2012 Symposium on Eye Tracking Research and Applications* (ETRA), ACM, New York, pp. 265-268.

A GPU-Accelerated Software Eye Tracking System

Jeffrey B. Mulligan*
NASA Ames Research Center

Abstract

Current microcomputers are powerful enough to implement a real-time eye tracking system, but the computational throughput still limits the types of algorithms that can be implemented in real time. Many of the image processing algorithms that are typically used in eye tracking applications can be significantly accelerated when the processing is delegated to a graphics processing unit (GPU). This paper describes a real-time gaze tracking system developed using the CUDA programming environment distributed by nVidia. The current implementation of the system is capable of processing a 640 by 480 image in less than 4 milliseconds, and achieves an average accuracy close to 0.5 degrees of visual angle.

Keywords: image processing, GPU, CUDA, eye tracking

1 Introduction

In the last decade or so, increases microcomputer power have transformed video-based eye trackers from specialized hardware devices to software systems running on generic PC platforms. Nevertheless, real-time system accuracy is still less than what can be obtained when more sophisticated processing is applied off-line to recorded sequences of eye images. Dedicated hardware solutions have been demonstrated [Amir et al. 2005] that provide big speedups and are desirable for commodity embedded applications, but the complexity of FPGA programming and circuit design put this approach beyond the reach of most eye movement researchers and small laboratories. Here we consider software-based approaches based on commodity computer hardware, with the ultimate goal of matching or beating the performance of current commercial offerings that advertise accuracies of a fraction of a degree at frame rates from 250 to 1250 Hz.

The computational power of personal computers has been greatly increased by the introduction of powerful graphics cards containing a specialized graphics processing unit (GPU), with hundreds of processors that all execute in parallel. The architecture is known as *Single Instruction / Multiple Data*, or SIMD; all processors execute the same instruction simultaneously, each on a different part of the dataset. Because many of the operations required to implement a basic eye tracker are inherently parallel, the GPU architecture seemed like a promising avenue for an improved real-time gaze tracking system. In this paper, we describe the results of our attempts to adapt a basic software eye tracker to the GPU environment.

GPU programming has been greatly facilitated by the introduction of high-level language support; there are two options in widespread use: OpenCL and CUDA. Obtaining the best performance from an nVidia GPU appears to be easier using CUDA than with OpenCL [Fang et al. 2011]. In this paper, we describe an implementation using CUDA, which consists of two parts: a set of *kernels* comprised of instructions which run on the GPU's SIMD architecture, and host subroutines which call the kernels and perform other housekeeping functions. When invoking a kernel, the programmer requests that it be executed by a given number of *threads*; when this number is greater than the number of processors, the CUDA runtime library handles scheduling, so the programmer does not need to be concerned with the details of the underlying hardware. The computation performed by each thread must be independent of all other threads, so that the order in which threads execute does not affect the result. For image processing, it is typical to have one thread per pixel. Algorithms in which each pixel is processed independently can be implemented very efficiently, making good utilization of the GPU's resources. The architecture is less well-suited to algorithms in which the operation performed on a pixel depends on the results of operations on neighboring pixels, such as flood-fill, and the "starburst" algorithm used for eye tracking [Li et al. 2005].

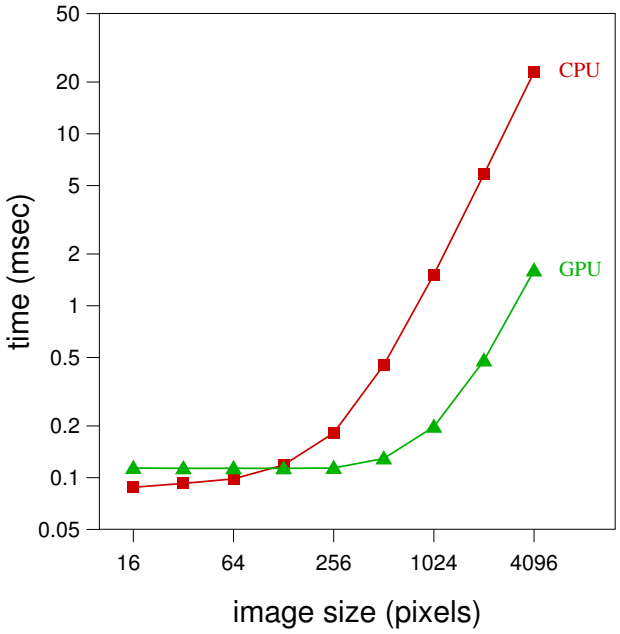

Figure 1: Execution times for floating-point addition of square images on the GPU (green triangles) and host CPU (red squares).

The individual processors on the GPU are not particularly fast, but high performance is achieved though massive parallelism. Figure 1 shows a plot of CPU and GPU execution times for floating point addition of square images of various sizes. The results for the GPU are roughly independent of image size up to about a quarter of a megapixel. In this example, the asymptotic speed advantage of the GPU is about a factor of 14.

In order to achieve this performance, the GPU must necessarily be able to move data to and from its memory efficiently. Figure 2 shows the time required to move 1.2 megabytes of data (a 640x480 image of single-precision floating point numbers), using instructions of different data widths. The native word size of the GPU is 32

*e-mail: jeffrey.b.mulligan@nasa.gov

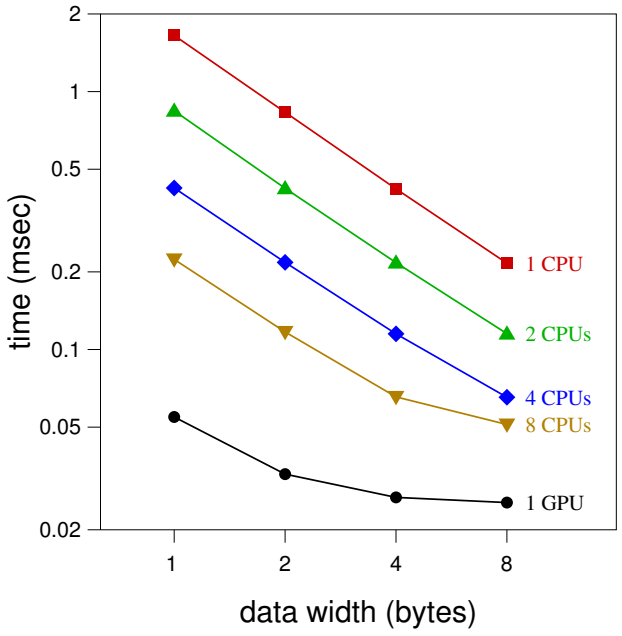

Figure 2: Relative speeds of GPU and host CPU for moving a 1.2 MB block of data.

bits, and consequently there is little difference in the time required to move the data using 4-byte moves versus 8-byte moves. On the CPU, on the other hand, the time drops in direct proportion to the amount of data moved per instruction. Figure 2 also illustrates that the time can be reduced when the CPU's task is divided between different processor cores on a multi-processor machine. The time is for the most part inversely proportional to the number of processors, but this relation breaks down when 8 processors are moving 64 bit words simultaneously, indicating saturation of the memory bus.

2 System Architecture

2.1 Hardware Platform

The host computer for our system is a dual processor workstation (Marquis C732T, ASL Workstations), with two Intel Xeon processors ("Gainestown" model E5520) running at 2.27 GHz, each with 4 cores. The motherboard (Supermicro X8DA3) provides 3 PCI-Express slots. One of these is populated with a CUDA-capable video card (nVidia GeForce GTX 285), while the second holds a four-channel analog video frame grabber (Sensoray model 811). Images are acquired by a SONY pan-tilt-zoom camera (model EVI-D70), which is controlled over a serial line using SONY's VISCA^{TM}protocol. Near-infrared illumination is provided by a group of light-emitting diodes (LEDs) powered by a custom-built LED controller, which receives commands over a second serial line. The output of the video card is sent to a flat panel monitor (Samsung model 204BW) with a native resolution of 1680 x 1050.

2.2 Software platform

The host computer runs the CentOS distribution of the Linux operating system (CentOS release 5.6), which is a freely-available version of Red Hat Enterprise Linux. The frame grabber is supported natively by the Video-For-Linux-2 (V4L2) driver. The nVidia driver for the video card was installed as part of the CUDA installation.

The approach we adopted was to integrate GPU acceleration into our existing in-house software environment, known as QuIP, which stands for **Qu**ick **I**mage **P**rocessing; the name is a pun on the fact that simple operations can be implemented by short scripts, or "quips." QuIP can be divided into two parts: a "back end" that consist of a set of subroutines which carry out a variety of useful functions; and a "front end" consisting of a text interpreter which parses the QuIP scripting language used to sequence the operations performed by the back end. QuIP also provides an interface to control V4L2 frame grabbers and VISCA^{TM}compatible cameras. The QuIP interpreter, and the scripts which implement the eye tracking functions described in this paper, are publicly available under the NASA Open Source Agreement (NOSA), and may be downloaded from our web site http://scanpath.arc.nasa.gov/quip/.

QuIP has two distinct input modes; *command mode* was developed with interactive use in mind: commands are selected from a menu (which can be displayed on screen by typing a question mark), and commands requiring arguments will prompt for them if they are not provided. In command mode, there is one command for every basic operation. For a complicated image processing algorithm, this can begin to resemble assembler code. To allow the user to write more natural and compact code, an *expression mode* is also provided, which can be invoked from one of QuIP's submenus. QuIP's expression language is similar to C, using the same set of operators, but the variables are *data objects*, which can be images, matrices, vectors, scalars, or tensors, with as many as 5 dimensions. When an expression is written involving objects containing more than one element, the operations are performed for each element.

Our goal in incorporating GPU acceleration into QuIP was to do so as transparently as possible. One reason why this cannot be done *completely* transparently is that the GPU operates in its own memory space, and, in general, does not "see" the same memory space as the host CPU program. Therefore, data objects that will be manipulated by the GPU must be explicitly created in the GPU's memory space. Subroutines to transfer data between the GPU and the host are provided by CUDA, and are bound to commands in QuIP. Once an object has been created, it can be used for the most part without regard to whether it resides on the device or the host, as all of the core functions have implementations in both contexts.

Memory on nVidia GPUs is divided into a number of categories: a) most of the on-board memory is "global" memory, which can be seen by each processing core; b) each core has a small amount of on-chip memory in the form of registers and cache; c) in addition, the CUDA library provides a special allocator function for host memory which can be accessed by the GPU. Host-mapped objects provide a convenient way to pass data between the host CPU and the GPU.

The frame buffer memory on the video card cannot be accessed directly from either the host or the GPU. To display an image from on-board memory, it is bound to an OpenGL texture and rendered into an on-screen rectangle.

QuIP is linked with the Motif library of graphical user interface (GUI) widgets; simple commands create widgets which execute a fragment of script when they are activated. Our prototype eye tracking system has a number of control panels which are used to enable and disable features, set thresholds, and so on. There are also several windows which display images: one window can display the live video, and smaller windows display the regions-of-interest corresponding the pupil and corneal reflex (CR). There is also a large plotting window capable of live display of measurements and intermediate results, and a full-screen stimulus window.

3 Algorithms

In this implementation, a set of standard algorithms have been used to demonstrate the potential of the software environment, which may provide a starting point for enhancement by ourselves or others. A number of threshold values are initialized manually by the user/experimenter. However, thresholds could be automatically determined through analysis of the image histogram.

3.1 Image validation

The first step is to compute the mean value of the image. When the subject is position, the variation in this value is relatively small (and can be reduced further by disabling automatic gain control on the camera). Blinks are readily apparent as transient positive blips. If the image mean is between an upper and lower threshold, then processing continues. A slider widget is used to manually adjust the thresholds while observing the signal and the threshold bounds in the plotting window.

3.2 Pupil finding

Candidate pupil pixels are found by applying a simple threshold. In the QuIP expression language, this operation can be invoked succinctly using C language conditional operator:

```
pup_mask = input > pup_threshold ? 0 : 1 ;
```

In this case, `pup_mask` and `input` are full-size images, while `pup_threshold` is a scalar value.

Further processing of the pupil is done on a 192 x 192 region-of-interest (ROI) to reduce the amount of data that must be processed and to exclude regions far from the pupil that may contain dark pixels below the pupil threshold. The pupil ROI is initially positioned at the center of the frame.

The pupil area is computed by taking the sum of the pixels in the mask. This is compared to an upper and lower threshold. if it falls within those values then it is assumed that the pupil falls within the current ROI. Otherwise, the ROI is scanned through the frame until a position that meets the criterion is found. If none is found, processing of this frame stops and the previous ROI location is retained.

When a ROI with an acceptable pupil area is found, the centroid is computed:

```
pup_area = sum(pup_roi);
tmp_img = pup_roi * pup_roi_x;
cx = sum(tmp_img)/pup_area;
tmp_img = pup_roi * pup_roi_y;
cy = sum(tmp_img)/pup_area;
```

The images `pup_roi_x` and `pup_roi_y` have the same size as the pupil ROI, and are initialized so that each pixel contains the value of its x (or y) coordinate relative to the image center.

If the pupil centroid is displaced from the ROI center by more than a small value (currently set to 2 pixels), the ROI is moved by an amount equal to the centroid position, and the centroid is recalculated relative to the new ROI. Under normal circumstances, recalculation of the centroid should be equivalent to subtracting the displacement values from the original values, but the displaced ROI may include new pupil pixels, or the full displacement may not be possible if the image boundary is reached. Note that the pupil centroid is calculated on the GPU but tested on the host, requiring synchronization of the two execution streams.

3.3 Corneal reflex finding

The corneal reflex (CR), or "glint," is located in much the same way as the pupil: candidate pixels are selected from the full image with a threshold and a search is performed within the pupil ROI for a smaller region containing an appropriate number of selected pixels. A 64 x 64 ROI is used for the CR; it is repositioned to keep the CR centroid near the center. The sizes chosen for the ROIs represents a compromise between rejection of spurious pixels from the centroid calculation (improved by a smaller ROI) and the ability to maintain track in the presence of large inter-frame movements. Using the sizes reported here, the system has no problem following large saccades or rapid head movements.

3.4 Point-of-gaze estimation

Our current implementation estimates the point-of-gaze via a simple linear transformation of the pupil-CR vector, with coefficients determined by regression on a calibration data set. (We expect that better performance might be obtained using a model-based approach [Beymer and Flickner 2003; Model and Eizenman 2011], although a recent report [Hennessey et al. 2008] claims superior results using the P-CR method.) Because point-of-gaze estimation generally involves calculations with scalar values (rather than computations on entire images), there is no particular advantage in using the GPU for this step, and modest increases in complexity are unlikely to have a large impact on system throughput.

4 Performance

System time performance of the can be monitored in two ways: execution times on the host can be assessed by reading the system clock (using the operating system call `gettimeofday`), while execution times on the GPU can be measured independently using event timing functions provided by the CUDA library. QuIP provides a simple set of commands to insert both types of checkpoints within scripts.

Figure 3 shows the execution profile for a single frame of video. The data from the frame grabber is provided by the driver in buffers residing in general-purpose host memory, encoded as YUV 422 (luminance samples at full resolution alternate with chroma samples at half resolution, i.e. each line consists groups of four samples: $y_1, u_{1,2}, y_2, v_{1,2}$). YUV to grayscale conversion is performed by a subroutine on the host, depositing the result in a buffer of host-mapped memory provided by CUDA. (This step would be unnecessary if a monochrome digital camera were used.) Next, the GPU is commanded to copy this buffer to an area of device global memory. The next step, calculation of the mean value, is done by first converting the entire image to single precision floating point, and then summing. Figure 3 illustrates the case in which the centroids of both the pupil and CR have moved relative to the previous frame, necessitating relocation of the ROIs and recalculation of the centroids. Figure 3 also shows the execution profile when the same computations are performed exclusively on the host.

System positional accuracy has been assessed using a simple 9-point calibration and linear regression of the fixation point screen coordinates on the pupil-CR position difference vector. Sixty samples were obtained at each fixation location; the RMS deviation of the reconstructed points from the fixation points was 0.56 degrees vertically and 0.28 degrees horizontally.

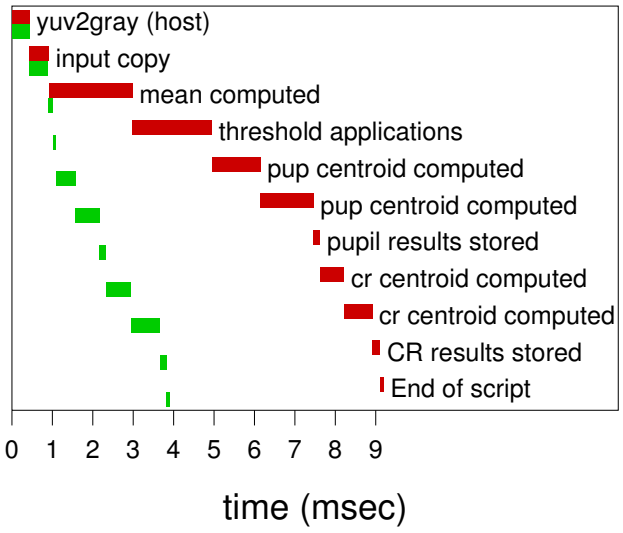

Figure 3: Timeline plot showing execution times for various components of the algorithm on the host (upper bars, in red), and on the GPU (lower bars, in green).

5 Challenges and future work

Although the current prototype system does not *yet* represent a significant advance in the state-of-the-art, we are nevertheless encouraged by these preliminary results, and see a number of areas where significant improvements may be made. Perhaps the most obvious is the method by which the pupil is located, namely thresholding followed by a centroid computation. Biases are introduced when the pupil region is occluded by eyelashes, or intersects with the CR. A superior method is to locate the pupil edge pixels, and fit a shape such as an ellipse, thus providing information not only about size and location but also shape and orientation [Zhu et al. 1999; Li et al. 2005].

While our prototype system has no trouble running at 30 or 60 Hz, the frame processing time of 4 milliseconds imposes a maximum frame rate of 250 Hz. To work with cameras capable of even higher frame rates (which is desirable for some applications), additional optimizations must be obtained. The development philosophy of QuIP was that the overhead of script interpretation was relatively insignificant compared to the computations being dispatched by the interpreter. But when smallish ROIs are processed by powerful GPUs, this is no longer the case. Current development efforts seek to enhance QuIP by adding the capability to create and store sequences or "chains" of operations in a list structure, akin to the idea of a "display list" in graphics programming. All error checking is done when the chain is created, so the chain can be executed with minimal overhead. This solution retains the advantages of coding in the scripting language, while getting most of the speed that a compiled program could attain.

Further speed increases could likely be obtained by applying a technique known as *pixel pipelining*. In the current design of QuIP, all basic operations are done on a per-image basis; when a series of operations is required, each basic operation is performed over the entire image, with the results stored in a temporary buffer. When pixel pipelining is applied, all of the operations needed at a pixel are performed together, eliminating the need for memory accesses to temporary storage. The disadvantage of this approach is that the interpreter cannot be built with all possible computations precompiled, and some form of just-in-time compilation would be required. This technique would likely be beneficial for both the GPU and the host CPU.

Inspection of figure 3 shows that much of the GPU's time is spent computing centroids. A dedicated centroid-computing function using pixel pipelining techniques would likely provide a significant speed increase, and would be much simpler to implement than a general-purpose QuIP compiler.

6 Conclusion

We have described a basic gaze tracking system implemented in a scripting language controlling GPU-accelerated image processing. Powerful GPUs are readily available and inexpensive: a new graphics card with the nVidia GeForce GTX 560 Ti, which has 384 cores (compared to 240 in the GTX 285 used in the system described in this paper) retails for around $250. These cards provide an easy way to super-charge a modest host computer. The QuIP interpreter is freely-available, and allows developers to exploit the power of the GPU using a high-level language, while ignoring the most of the details of CUDA programming. We hope that these developments may spawn a new generation of low-cost video-based gaze tracking systems.

7 Acknowledgments

Supported by the System-Wide Safety and Assurance Technologies (SSAT) project of NASA's Aviation Safety program (AvSP). Thanks to Tina Beard and four anonymous reviewers for comments on the manuscript.

References

AMIR, A., ZIMET, L., SANGIOVANNI-VICENTELLI, A., AND KAO, S. 2005. An embedded system for an eye-detection sensor. *Computer Vision and Image Understanding 98*, 104–123.

BEYMER, D., AND FLICKNER, M. 2003. Eye gaze gracking using an active stereo head. In *Proc. IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, vol. 2, IEEE, 451–458.

FANG, J., VARBANESCU, A. L., AND SIPS, H. 2011. A comprehensive performance comparison of CUDA and OpenCL. In *Proceedings of the International Conference on Parallel Processing*, IEEE, 216–225.

HENNESSEY, C., NOUREDDIN, B., AND LAWRENCE, P. 2008. Fixation precision in high-speed noncontact eye-gaze tracking. *IEEE Transactions on Systems, Man and Cybernetics, Part B: Cybernetics 38*, 289–298.

LI, D., WINFIELD, D., AND PARKHURST, D. 2005. Starburst: A hybrid algorithm for video-based eye tracking combining feature-based and model-based approaches. In *Proc. IEEE Computer Society Conference on Computer Vision and Pattern Recognition, Workshop on Vision for Computer-Human Interface*, IEEE, 79.

MODEL, D., AND EIZENMAN, M. 2011. User-calibration-free remote eye-gaze tracking system with extended tracking range. In *Proc. Canadian Conference Electrical and Computer Engineering (CCECE)*, IEEE, 1268–1271.

ZHU, D., MOORE, S. T., AND RAPHAN, T. 1999. Robust pupil center detection using a curvature algorithm. *Computer Methods and Programs in Biomedicine 59*, 145–157.