A GPU Accelerated Evolutionary Computer Vision System

Eberhard Karls Universitat Tubingen
Wilhelm-Schickard-Institut fur Informatik
Abt. Rechnerarchitektur, Sand 1, 72076 Tubingen
marc.ebner@wsii.uni-tuebingen.de
http://www.ra.cs.uni-tuebingen.de/mitarb/ebner/welcome.html

We have used the graphics processing unit (GPU) of the graphics card to create an evolutionary image processing system which is able to learn how to detect a user-specified object in an image. The system receives an image sequence as input. The user only has to tell the system where this object is located. This is done by using the mouse pointer. The user simply moves the mouse over the desired object and then presses the mouse button as long as the object is located under the mouse pointer. The user follows this object over several frames while keeping the mouse button pressed. As this is being done, the system evolves a population of image processing algorithms by exploiting the power of the GPU at interactive rates. Our system is the first GPU accelerated evolutionary image processing system (Figure 1) which allows the automatic creation of object detection algorithms [2]. This is the first step towards building fully adaptive evolutionary vision systems [1].

Consumer graphics cards are specifically optimized to render images at high speeds. A three-dimensional scene consists of numerous triangles which are fed to the graphics card. In order to obtain photo-realistic images, small programs can be sent to the graphics card to specify computations which should be carried out per vertex (vertex shaders) or per pixel (pixel shaders). The OpenGL shading language (OpenGLSL) has been developed as a standard to program vertex and pixel shaders. This shading language as well as the computations which are carried out on the graphics card are highly optimized for rendering three-dimensional scenes consisting of thousands of triangles. We use this programming paradigm to perform image processing on the graphics card efficiently.

To fully exploit the power of the GPU, we use exactly the same paradigm which is used when rendering images. Only a single polygon is rendered. This polygon represents the output image of the image processing algorithm. The image processing algorithm (generated by simulated evolution) is fed to the pixel shader. This pixel shader is then used to compute the correct output color for each pixel. The original input image is supplied to the pixel shader as a texture.

GECCO'09, July 8–12, 2009, Montreal, Canada.

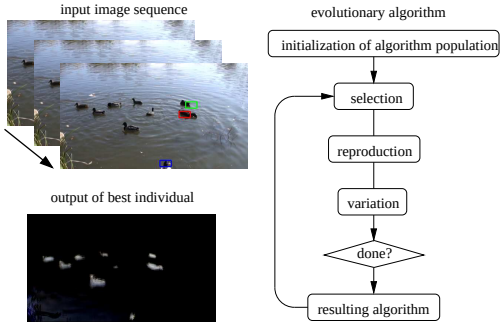


Figure 1: Evolutionary Object Detection System.

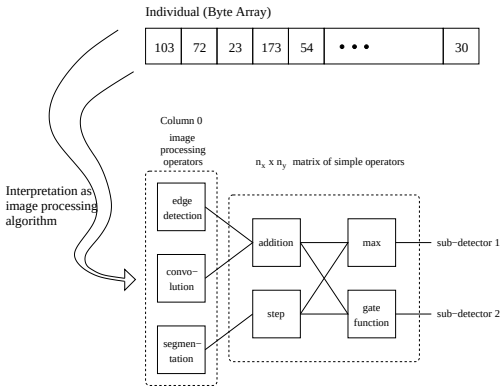


Figure 2: The genotype of an individual is simply a byte array which is modified through simulated evolution.

The pixel shader we use is actually a universal pixel shader which is able to interpret the genetic material of an individual of the population as an image processing algorithm. Each individual consists of an array of integers and represents an image processing algorithm (Figure 2). This a variant of the Cartesian Genetic Programming approach. For each input image, all of the individuals of the population are evaluated by sending the array of integers to the GPU. The input image is loaded into the GPU as a texture only once. In order to fully exploit the power of the GPU we first compute modified output data by accessing the original texture of the image. Operators include convolution, edge detection, Laplacian or image segmentation.

The GPU is used to compute the output image and also to display the input image and the output of the three best im-

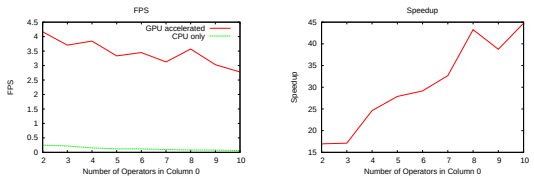
age processing algorithms. The evolutionary algorithm itself is run on the main CPU. The evaluation of the individuals and the display of the images take approximately 94% of the total time. We have used GLUT such that the system can be compiled easily for Mac OS X, Linux or the Microsoft Windows Operating system. Since the OpenGLSL is used to compute the output image, the code is highly portable. The system can be run on any graphics card as long as the graphics card supports vertex and pixel shaders, e.g. OpenGLSL 2.0 and up.

Each operator can be applied to the original image at a slight offset or scale. This is readily possibly by using the texture processing operations of the OpenGLSL. The mip map mechanism allows us to read out the texture at any scale. Once the major operators such as convolution, edge detection or segmentation have been applied, the results of these operations is recombined using arithmetic or threshold operations. These operations are arranged in a $n_x \times n_y$ matrix. The input is fed from left to right through this matrix. On the right hand side we read of the output of a total of n_y sub-detectors. Each row of the right hand side of the matrix can be viewed as a sub-detector which is designed by evolution to extract the desired object.

Our system adheres to the image rendering paradigm as close as possible in order to use the GPU as efficiently as possible. That is why image processing operators such as convolution and edge detection are applied first and then the output of these operators is recombined to detect the object. Allowing full image processing operators at every position of the matrix would also have been possible. However, in this case, we would have to read out the result computed by an operator from the graphics card, and then again send this result to the graphics card as a texture since textures are read only. We have reduced the texture transfer between the CPU and the GPU to a minimum (only the input image is transfered). Thus, the GPU architecture with read only texture and fast rasterization of triangles is fully exploited by our system.

The output of all sub-detectors is averaged to obtain the overall output for the input image. The object is said to be located at the position which has the highest response. Let $\vec{p}_d$ be the detected position and let $\vec{p}_m$ be the position specified by the user through the mouse pointer. The quality of the detector is measured by computing the distance between the detected position and the position of the mouse pointer. This distance is used as the fitness $f = (\vec{p}_d - \vec{p}_m)$ of an individual. A perfect individual would always respond with the position directly beneath the mouse pointer. It would have a fitness of zero.

Individuals from the current population are modified slightly using mutation and crossover operators. The crossover operator selects two individuals and exchanges parts of the genetic material of one individual with the other individual. The mutation operator changes some of the genetic material at random. Starting with μ parent individuals λ offspring are generated. Fitness is computed for the current input image for both, parent and offspring. All individuals, parent and offspring, are sorted with respect to fitness. Individuals with the same fitness are considered to be identical. Only the best μ individuals are selected as parents of the next generation. This is the Darwinian mechanism "survival of the fittest". In the course of several generations, the individuals adapt to the given problem, locating the detecting



(a) (b)

Figure 3: (a) FPS obtained with and without GPU acceleration. (b) Speedup.

the desired object in the image.

The system was tested on several different video sequences. The task was to locate a sample object (bird, fish, duck, or street sign) within the image. In each case, the system was able to evolve a reasonably good detector within relatively short amount of time. A reasonably good detector in this case is a detector which is located somewhere on the object but does not have to be positioned exactly on the position of the mouse pointer. What was particularly surprising is that the system evolved quite robust detectors which continued to extract the object after evolution was turned off even though the object was distorted or its scale changed. Writing similarly robust detectors would have taken weeks to develop manually.

Figure 3(a) shows the frame rate which is achieved when each individual consists of a $n+2\times2$ matrix of operators, i.e. n high level operators and then a 2×2 matrix of arithmetic operations to recombine the output and 23 individuals are evaluated for each input image. Each image had a size of 320×240 pixels. The speedup (depending on the number of high level operators used) is shown in Figure 3(b). This data was measured on a Linux system (Intel Core 2 CPU running at 2.13GHz) equipped with a GeForce 9600GT/PCI/SEE2. Given a more powerful graphics card, the system can easily be scaled up. One can simply increase the number of algorithms which are evaluated for each image or one can increase the size of the images which are processed. Approximately 92% of the total computation time are used to perform image processing on the GPU, 2% are used for rendering and the remaining 6% is used for all other computations on the CPU including the operations of the evolutionary algorithm.

A real-time evolutionary object recognition system working at interactive rates is of considerable interest to everyone working in the field of evolutionary image processing. Until now, algorithms were usually evolved offline. Without the power of the GPU interactive rates would not be possible. With our evolutionary system we are able to evolve object detectors within minutes.

1. REFERENCES

[1] M. Ebner. An adaptive on-line evolutionary visual system. In E. Hart, B. Paechter, and J. Willies, editors, *Workshop on Pervasive Adaptation, Venice, Italy*, pages 84–89. IEEE, 2008.

[2] M. Ebner. A real-time evolutionary object recognition system. In L. Vanneschi, S. Gustafson, A. Moraglio, I. D. Falco, and M. Ebner, editors, *Genetic Programming: Proceedings of the 12th European Conference, EuroGP 2009, Tübingen, Germany, April*, pages 268–279, Berlin, 2009. Springer.