Evolving Object Detectors with a GPU Accelerated Vision System

Marc Ebner

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

Abstract. Using GPU processing, it is now possible to develop an evolutionary vision system working at interactive frame rates. Our system uses motion as an important cue to evolve detectors which are able to detect an object when this cue is not available. Object detectors consist of a series of high level operators which are applied to the input image. A matrix of low level point operators are used to recombine the output of the high level operators. With this contribution, we investigate, which image processing operators are most useful for object detection. It was found that the set of image processing operators could be considerably reduced without reducing recognition performance. Reducing the set of operators lead to an increase in speedup compared to a standard CPU implementation.

1 Motivation

In the field of evolutionary computer vision, evolutionary algorithms are used to search for optimal or approximately optimal solutions for computer vision problems [2]. A programmer developing a computer vision algorithm needs to decide in what sequence well known image processing operators have to be arranged to obtain a desired result. In evolutionary computer vision, Genetic Programming [1,13] is used to arrange different image processing operators to obtain a particular output or to perform a given task such as object recognition. This approach is particularly interesting for problems for which the solution is not readily apparent to those skilled in the art. Unfortunately, most experiments in evolutionary computer vision require enormous computational resources because multiple algorithms have to be evaluated over several generations to find an appropriate solution. However, the graphics processing unit (GPU) of a PC can be used for speeding up image processing tasks [7,8]. The GPU is ideal for speeding up image processing as the same operation usually needs to be computed for each image pixel.

We have created a GPU accelerated evolutionary image processing system which is able to learn how to detect a user-specified object in an image [3,4].

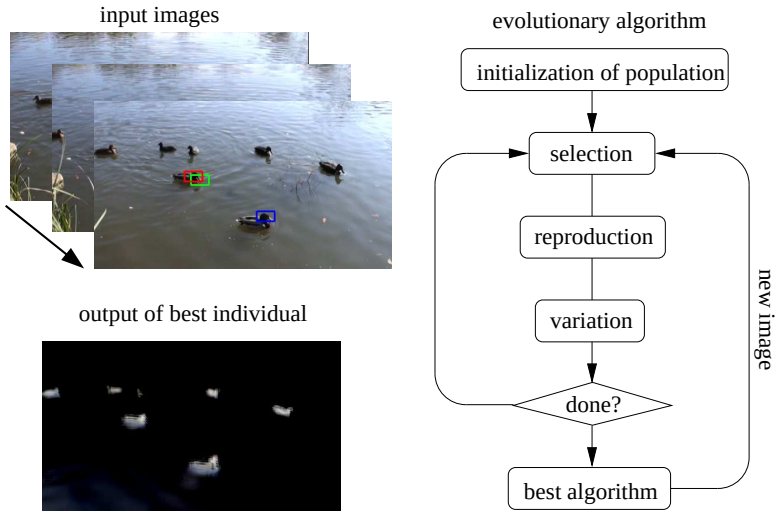


Fig. 1. Evolutionary object detection system.

This is the first system of this type [1]. The system receives an image sequence as input. The user has to tell the system where this object is located using the mouse pointer. The user simply moves the mouse over the object to be detected and then follows the object while pressing the mouse button. The system maintains a population of image processing algorithms. For each new image, all algorithms are run on the input image. The algorithm transforms the input image into another image, the output image. The largest pixel response of the output image is taken as the position of the detected object.

The power of the GPU is required to evaluate multiple algorithms for each incoming image at interactive rates. Figure 1 provides an overview. The evolutionary algorithm uses a generational based model. For each new image, a new generation of algorithms is created. Parents are re-evaluated on the new image. Both offspring and parents are considered, when selecting the parents of the next generation. Fitness is computed as the Euclidean distance between the desired object position (position of the mouse pointer) and the detected position of the algorithm. Over the course of several images, the population of algorithms adapts to the problem of detecting the desired object position.

Humans are able to locate and detect objects in single images. Similarly, the object detection algorithms only use a single image as input and not multiple images. However, motion is an important cue. That's why this system uses image sequences as input. If the desired object moves around in the scene, then the background constantly changes while there is little change in the object. The evolutionary algorithm discounts all aspects that are irrelevant but necessary in

[1] A video of this system is available for download from http://www.ra.cs.uni-tuebingen.de/mitarb/ebner/research/publications/uniTu2/EvoCV.m4v

detecting the object successfully. If there is a large change in the object because of a change in perspective, then the evolved detector may focus only on color or on color and texture if color is not sufficient by itself. We have recently extended this evolutionary vision system by removing the necessity of having the user interact with the system. Moving objects are automatically detected and used as teaching input [5]. Thus, motion is used as a cue but object detection can still be performed when this cue is not available as the object detectors only use single images.

Each object detector consists of several image processing operators known from the computer vision literature [10,21]. The complete list of operators is given by Ebner [4]. The representation of the individuals will be described below. With this contribution, we investigate which of the available operators are actually used and how important they are, i.e. the fraction of their usage. In addition, we rigorously evaluate the speedup obtained compared to standard processing on the CPU.

2 Evolutionary Computer Vision

Evolutionary algorithms are especially useful in computer vision when it is not at all clear what an optimal algorithm should look like. Evolutionary algorithms can be used to find optimal parameters for an existing algorithm but can also be used to evolve an algorithm from scratch. Evolutionary computer vision started in the early 1990s when Lohmann used an Evolution Strategy to find an algorithm which computes the Euler number of an image [15]. Initially, research focused on the automatic generation of low-level operators, e.g. edge detectors [9] or feature detectors [19]. Katz and Thrift [12] applied evolutionary algorithms for target recognition.

Using evolutionary algorithms, it is possible to evolve adaptive operators which are optimal or near optimal for a given task [6]. Poli [18] has shown that Genetic Programming is particularly useful for image processing. Johnson et al. [11] have used Genetic Programming to evolved visual routines. Current research focuses on the evolution of low-level detectors [22] and object recognition [14]. A taxonomic tutorial on the field of evolutionary computer vision is given by Cagnoni [2].

Most experiments in evolutionary computer vision are performed off-line because of the enormous computational resources which are required because each individual of the population has to be evaluated for several generations. In contrast to such off-line experiments, we try to build an adaptive, self-learning vision system which is always on. In our system, multiple alternative image processing algorithms are run on each new input image. This is done using GPU accelerated image processing.

3 GPU Accelerated Image Processing

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) [20] 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. Instead of using the OpenGLSL, we could also have used CUDA [17]. CUDA allows for more general image processing algorithms. However, it does not provide Mip Maps. Mip Maps are usually used for texturing polygons. In our case, Mip Maps allow for scale space processing [23].

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 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, thus Mip Mapping is available. 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 bytes and represents an image processing algorithm.

For each input image, all of the individuals of the population are evaluated by sending the array of bytes one after the other to the GPU. The input image is loaded into the GPU as a texture only once. Thus, the speedup depends on the number of individuals to evaluate. The more individuals are evaluated, the higher the speedup. The GPU renders the input image on the screen and also shows the output of the three best individuals of the population. The evolutionary algorithm itself is run on the main CPU. The evaluation of the individuals and the display of the images take approximately 81% of the total time. Since the OpenGLSL is used to compute the output image, the code is highly portable and 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.

4 Representation

A special data structure is used for all image processing algorithms in order to fully exploit the power of the GPU. The representation is shown in Figure 2. This is a variant of the Cartesian Genetic Programming approach [16]. First, n_1 high level operators are applied to the input image. Operators include convolution, edge detection, Laplacian or image segmentation. The high level operators can

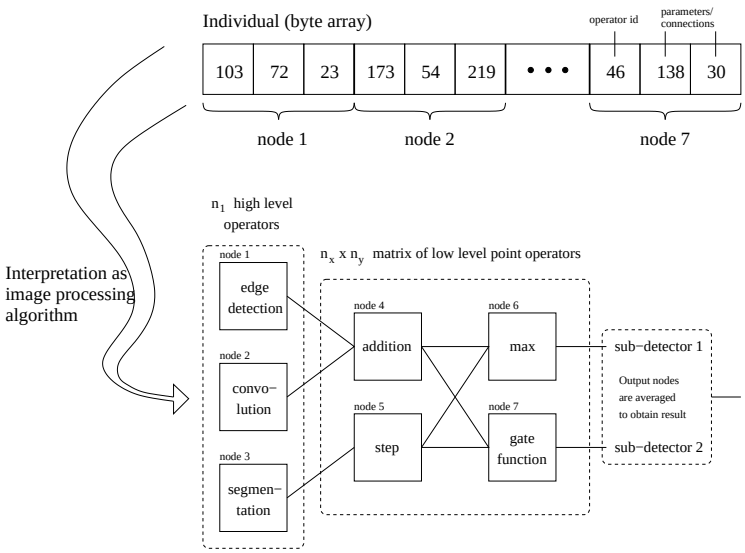


Fig. 2. The genotype of an individual is simply a byte array which is modified through simulated evolution. Each individual represents a computer vision algorithm. High level operators are located in column 0. Low level operators are located inside a $n_x \times n_y$ matrix. The low level operators are used to recombine the output of the high level operators.

access the pixels of the original image using a mask or they can simply return a constant value. Some of the operators use parameters which specify the scale level at which the operator is to be applied. 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. Some also allow for a shift of the entire image. The additional parameters required for these operations are all part of the genetic representation.

The output of these high level operators is then recombined using a $n_x \times n_y$ processing matrix. We call this the $n_1 - n_x \times n_y$ representation. It consists of $3(n_1 + n_x n_y)$ bytes. The mapping from genotype to phenotype is fully described in Ebner [4]. The operators used as well as the connectivity of the matrix is optimized by the evolutionary algorithm. The input is fed from left to right through this matrix. Arithmetic, threshold or channel selecting operations are used for this task. Gate functions which can be used to implement if-functions are also included. 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. Thus, on the right hand side of the $n_x \times n_y$ processing matrix, the output of n_y sub-detectors is available. The output of all sub-detectors is averaged to obtain the overall response to the input image. The object is said to be located at the position with the largest pixel value. If multiple pixels have the same maximum value, then the center of gravity is computed.

This representation 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. It would have also been possible to allow full image processing operators at every position of the matrix. However, in order to do this, one would have to first read out the results computed by an operator from the graphics card, and then again send this result to the graphics card as a texture because texture access is read only. The current representation reduces the texture transfer between the CPU and the GPU to a minimum because only the input image is transfered.

5 Experiments

The system was tested on two video sequences. The first sequence shows a radio controlled car, while the second sequence shows a toy train. These two sequences have also been used for previous experiments [5]. The radio controlled car is relatively easy to detect because of its distinct color. No other object in the sequence has the same purple color as shown on the car. The train was mostly yellow and red. However another stationary object also had the same two colors, albeit in a different arrangement. Thus, the toy train is more difficult to detect. For each sequence, the moving object, car respectively toy train was detected automatically. The center of this moving object was used to compute the fitness.

Let $\boldsymbol{p}_m$ be the position of the moving object and let $\boldsymbol{p}_d$ be the object position as detected by the algorithm. The quality or fitness f of the detector is measured by computing the Euclidean distance beween these two positions $f = |\boldsymbol{p}_m - \boldsymbol{p}_d|$. A perfect individual would always respond with the exact same position as the moving object. It would have a fitness of zero. Evolution was turned on whenever the fitness increased beyond 25 pixels. Evolution was turned off whenever the fitness decreased below 10 pixels for five consecutive images. Figure 3(a/b) shows the probability that evolution was turned on again for both image sequences. For each of the representations, 10 runs with different seeds were conducted. Figure 3(c/d) shows the percentage of image frames that evolution was turned on for both image sequences.

When evolution is turned on, offspring are generated using mutation and crossover. Starting with $\mu = 3$ parent individuals, $\lambda_o = 20$ offspring are generated. An additional number of $\lambda_r = 20$ offspring are generated completely at random. The mutation operator is used to generate 50% of the λ_o offspring. The mutation operator either increases or decreases one of the parameters by one or applies a per bit mutation with a probability of $p_{\text{mut}} = 2/l$ where l is the length of the genotype in bits, i.e. on average, we will have two bit changes per mutation. The crossover operator selects two individuals and exchanges parts of the genetic material of one individual with the other individual (two point crossover with a probability $p_{\text{cross}} = 0.5$). 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

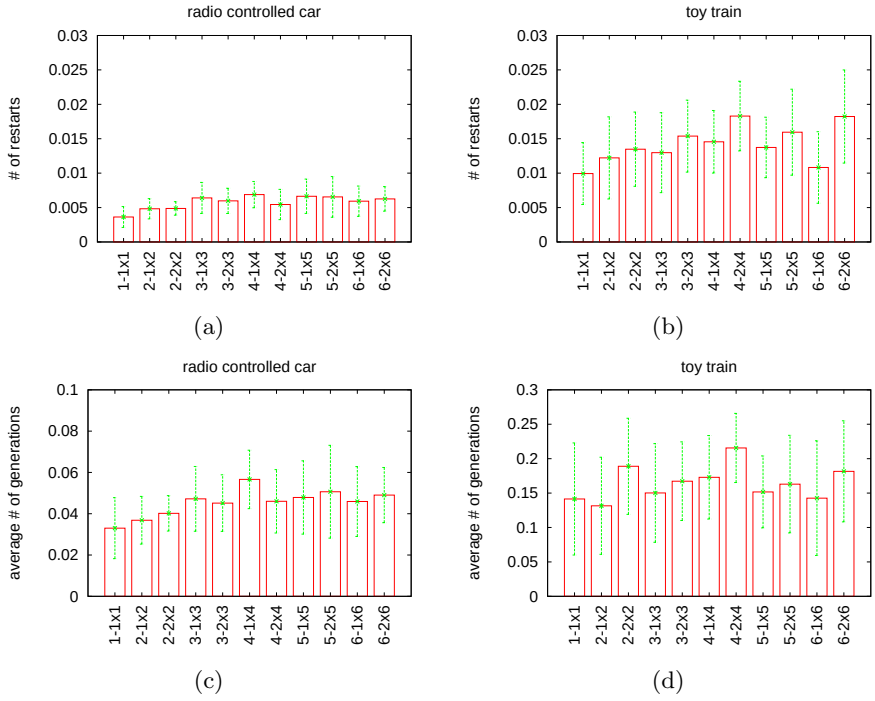


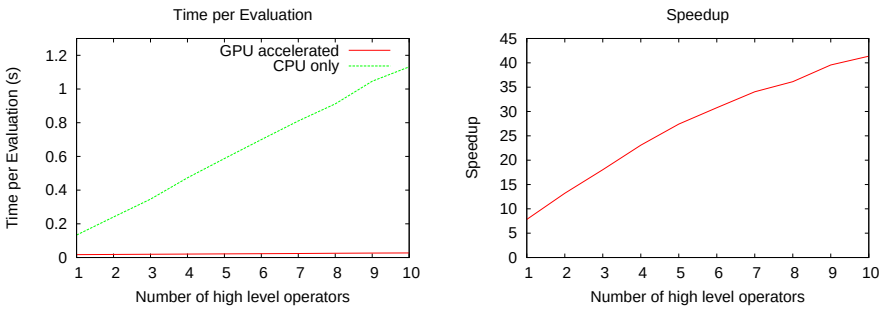
Fig. 3. (a/b) Probability that evolution had to be restarted. (c/d) Percentage of image frames that evolution was required (a/c) Radio controlled car (b/d) Toy train.

identical. Only the best μ individuals are selected as parents for the next generation. The set of image processing operators is shown in Figure 4. The operators have been fully described in [4].

High level image processing operators	`Image`, `DX`, `DY`, `Lap`, `Grad`, `ImageGray`, `ImageChrom`, `ImageLogDX`, `ImageConv1`, `ImageConv4`, `ImageConv16`, `ImageConvd`, `ImageSeg`, `0.0`, `0.5`, `1.0`
Low level point operations	`id`, `abs`, `dot`, `sqrt`, `norm`, `clamp(0,1)`, `step(0)`, `step(0.5)`, `smstep(0,1)`, `red`, `green`, `blue`, `avg`, `minChannel`, `maxChannel`, `equalMin`, `equalMax`, `gateR`, `gateG`, `gateB`, `gateRc`, `gateGc`, `gateBc`, `step`, `+`, `-`, `*`, `/`, `min`, `max`, `clamp0`, `clamp1`, `mix`, `step`, `lessThan`, `greaterThan`, `dot`, `cross`, `reflect`, `refract`

Fig. 4. Set of image processing operators.

The individuals usually adapt to the given problem within a relatively short amount of time, i.e. on average within 10 image frames. The evolved detectors



(a) (b)

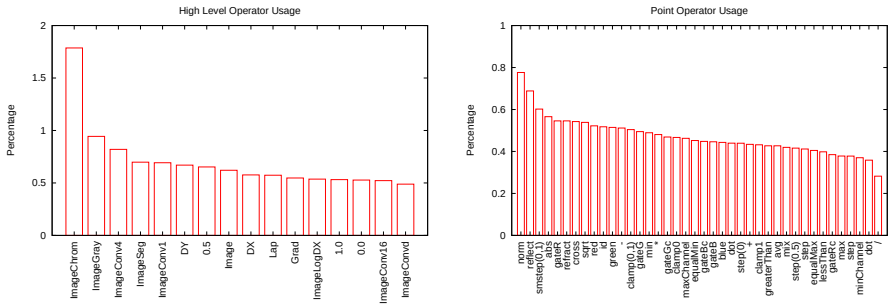
Fig. 5. (a) time required to evaluate a single individual depending on the representation used. (b) speedup.

may not be perfect but are able to locate the object approximately. Over time, this detector will be refined by evolution. As previous experiments have shown, the evolved detectors are very robust, i.e. the object is still located even though it was distorted or had changed its scale [4].

Figure 5(a) shows the time required to evaluate a single individual depending on the representation used. Results are shown for GPU accelerated processing as well as standard CPU processing. It is clear that the time is linear in the number of high level operators when CPU processing is used. For this experiment, the sequence of the radio controlled car was used. All of the individuals used a 2×2 processing matrix and only the number of high level operators was varied. The evaluated individuals were generated completely at random to ensure an unbiased sampling of the search space. Each input image had a size of 512×288 pixels. Figure 5(b) shows the speedup obtained.

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. With the 3–2⊗2 representation (see Figure 2), 78.5% of the total computation time are used to perform image processing on the GPU, 2.7% are used for rendering and the remaining 18.8% is used for all other computations on the CPU including the operations of the evolutionary algorithm.

Figure 6 shows how useful the different operators are for detecting the radio controlled car as well as the toy train. The plot was generated by tabulating how often each operator occurred during all experiments which were carried out to produce Figure 3. The most useful operator was `ImageChrom`. This operator is especially useful to create object detectors based on color as it can be used to compute chromaticities. Since some operators are more useful, than others, we conducted another set of experiments. The set of high level operators was limited to the 8 most useful high level operators and the set of low level point



(a) (b)

Fig. 6. Operator usage sorted from most often used to least often used. (a) high level operators (b) point operators

High level image processing operators	`Image`, `DY`, `ImageGray`, `ImageChrom`, `ImageConv1`, `ImageConv4`, `ImageSeg`, `0.5`
Low level point operations	`id`, `abs`, `sqrt`, `norm`, `clamp(0,1)`, `smstep(0,1)`, `red`, `green`, `gateR`, `gateG`, `-`, `*`, `min`, `cross`, `reflect`, `refract`

Fig. 7. Reduced set of image processing operators.

operators was limited to the 16 most useful point operators. This reduced set of operators is shown in Figure 7.

Removing unnecessary operators has the advantage that the pixel shaders which are used to map the genotype to an image processing algorithm can be considerably simplified. It is not possible to directly map an operator to an executable function as it is possible in C or C++. With a pixel shader, this needs to be done using a sequence of `if`-instructions. A case statement is not available. For our initial experiments, we had a total of 56 operators, i.e. 16 high level operators and 40 point operations. Hence, the pixel shader has to run through 56 `if`-statements for every output pixel that is computed. The `if`-instructions require considerable time to execute. In comparison, a CPU only implementation is of course able omit these operations per pixel. The CPU implementation decides once for each node which image processing operator is applied and then computes all of the output pixels using this image processing operator.

Figure 8 shows the results using the reduced operator set. Compared to Figure 3 we see that the evolved operators are more robust when the reduced operator set is used, i.e. evolution has to be turned on for fewer generations. The difference is statistically significant for half of the representations at 95%. For the remaining experiments, the difference is not statistically significant. In other words, when the reduced operator set was used, performance remained the same or improved. We also see that less re-starts are required (except for representations 2–1×2 and 6–2×6 for the radio controlled car and 2–2×2 for the toy train). On average good solutions are found within 8 images compared to

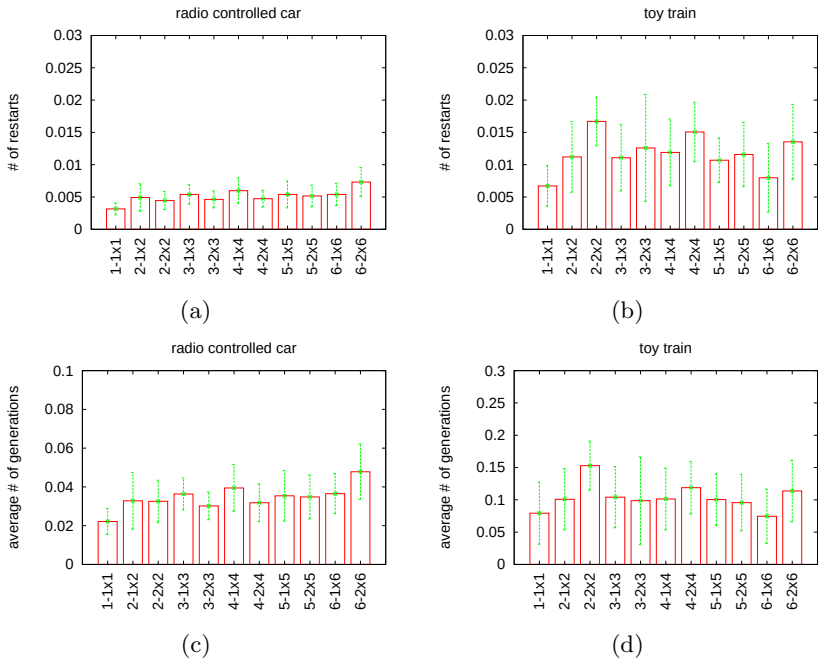
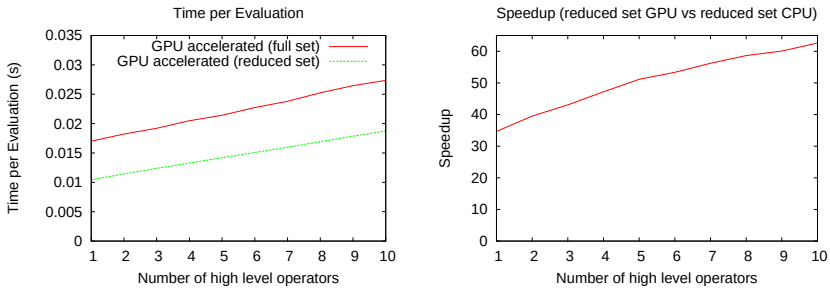


Fig. 8. (a/b) Average number of times that evolution had to be restarted using the reduced operator set. (c/d) Percentage of image frames that evolution was required using the reduced operator set. (a/c) Radio controlled car (b/d) Toy train.

10 image using the full instruction set. Figure 9(a) shows the time required to evaluate a single individual when the reduced operator set was used compared to the full set using GPU acceleration. On average, the time required to evaluate an individual was reduced by a factor of 1.5. Since the complexity of the pixel shader is considerably reduced with the reduced operator set, a lot less time was required to evaluate a single individual and hence the speedup improved. Figure 9(b) shows the speedup comparing the reduced operator set with and without GPU acceleration.

6 Conclusion

We have built an evolutionary object recognition system working at interactive rates using GPU processing. Initially, objects had to be manually identified by the user using the mouse pointer. The current system uses motion as a cue to detect moving objects in video sequences. The largest moving object provides the teaching input. The system evaluates a population of algorithms for each new image. The representation is a variant of the Cartesian Genetic Programming representation. First, a number of high level operators are applied. The output of these high level operators is then recombined using a processing matrix of point



(a) (b)

Fig. 9. (a) time required to evaluate a single individual using GPU acceleration. (b) speedup comparing the reduced instruction set with and without GPU acceleration.

operations. The processing matrix provides one or more transformed images as output. If more than one output image is computed, then these are averaged to obtain a single output image. The largest RGB response is the detected object position. Fitness is computed as the Euclidean distance between the detected and the actual object position. Offspring are generated using mutation and crossover operators. Half of the offspring are generated at random for each new image. This allows evolution to restart at any point in time.

The evolved object detectors receive only a single image as input. The size of the moving object is unknown to the detectors. However, since the objects move across the background, the surrounding of the object will constantly change. The evolved object detectors need to focus on features which remain constant over the course of several image frames. Object detectors focusing on varying features will be eliminated from the population. Only the detectors using robust detection strategies will remain.

With this contribution, we have thoroughly evaluated which operators are most useful for object detection. The speedup obtained through GPU processing was also analyzed. By reducing the number of operators, the speedup could be improved. The current system is a step towards a fully self-adapting/self-learning vision system. Object detectors are evolved using one cue (motion) but detect the object when this cue is not available. Future research will focus on the evolution of concepts.

References

1. W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming - An Introduction: On The Automatic Evolution of Computer Programs and Its Applications*. Morgan Kaufmann Publishers, San Francisco, CA, 1998.
2. S. Cagnoni. Evolutionary computer vision: a taxonomic tutorial. In *8th Int. Conf. on Hybrid Intelligent Systems*, pp. 1–6, Los Alamitos, CA, 2008. IEEE Comp. Soc.
3. M. Ebner. Engineering of computer vision algorithms using evolutionary algorithms. In J. Blanc-Talon, W. Philips, D. Popescu, and P. Scheunders, eds.,

Advanced Concepts for Intelligent Vision Systems, Mercure Chateau Chartrons, Bordeaux, France, pp. 367–378, Berlin, 2009. Springer.
4. M. Ebner. A real-time evolutionary object recognition system. In L. Vanneschi, S. Gustafson, A. Moraglio, I. D. Falco, and M. Ebner, eds., *Genetic Programming: Proc. of the 12th Europ. Conf., Tübingen, Germany*, pp. 268–279, Berlin, 2009. Springer.
5. M. Ebner. Towards automated learning of object detectors. In *Applications of Ev. Computation, Proc., Istanbul, Turkey*, pp. 231–240, Berlin, 2010. Springer.
6. M. Ebner and A. Zell. Evolving a task specific image operator. In R. Poli, H.-M. Voigt, S. Cagnoni, D. Corne, G. D. Smith, and T. C. Fogarty, eds., *Joint Proc. of the 1st Europ. Workshops on Evolutionary Image Analysis, Signal Processing and Telecommunications, Göteborg, Sweden*, pp. 74–89, Berlin, 1999. Springer-Verlag.
7. J. Fung, S. Mann, and C. Aimone. OpenVIDIA: Parallel GPU computer vision. In *Int. Multimedia Conf. Proc. of the 13th annual ACM int. conf. on Multimedia, Singapore*, volume 5, pp. 849–852. ACM, 2005.
8. J. Fung, F. Tang, and S. Mann. Mediated reality using computer graphics hardware for computer vision. In *Proc. of the 6h Int. Symp. on Wearable Computers*, pp. 83–89. ACM, 2002.
9. C. Harris and B. Buxton. Evolving edge detectors with genetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, eds., *Genetic Programming, Proc. of the 1st Annual Conf., Stanford University*, pp. 309–314, Cambridge, MA, 1996. The MIT Press.
10. R. Jain, R. Kasturi, and B. G. Schunck. *Machine Vision*. McGraw-Hill, New York, 1995.
11. M. P. Johnson, P. Maes, and T. Darrell. Evolving visual routines. In R. A. Brooks and P. Maes, eds., *Artificial Life IV, Proc. of the 4th Int. Workshop on the Synthesis and Simulation of Living Systems*, pp. 198–209, Cambridge, MA, 1994. The MIT Press.
12. A. J. Katz and P. R. Thrift. Generating image filters for target recognition by genetic learning. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 16(9):906–910, 1994.
13. J. R. Koza. *Genetic Programming. On the Programming of Computers by Means of Natural Selection*. The MIT Press, Cambridge, MA, 1992.
14. K. Krawiec and B. Bhanu. Visual learning by evolutionary and coevolutionary feature synthesis. *IEEE Trans. on Evolutionary Computation*, 11(5):635–650, 2007.
15. R. Lohmann. *Bionische Verfahren zur Entwicklung visueller Systeme*. PhD thesis, Technische Universität Berlin, Verfahrenstechnik und Energietechnik, 1991.
16. J. F. Miller. An empirical study of the efficiency of learning boolean functions using a Cartesian Genetic Programming approach. In W. Banzhaf et al., eds., *Proc. of the Genetic and Evolutionary Computation Conf.*, pp. 1135–1142, San Francisco, CA, 1999. Morgan Kaufmann.
17. NVIDIA. *CUDA. Compute Unified Device Architecture. Version 1.1*, 2007.
18. R. Poli. Genetic programming for image analysis. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, eds., *Genetic Programming, Proc. of the 1st Annual Conf., Stanford University*, pp. 363–368, Cambridge, MA, 1996. The MIT Press.
19. M. M. Rizki, L. A. Tamburino, and M. A. Zmuda. Evolving multi-resolution feature-detectors. In D. B. Fogel and W. Atmar, eds., *Proc. of the 2nd Am. Conf. on Evolutionary Programming*, pp. 108–118. Evolutionary Prog. Society, 1993.
20. R. J. Rost. *OpenGL Shading Language*. Addison-Wesley, Upper Saddle River, NJ, 2nd ed., 2006.

21. L. G. Shapiro and G. C. Stockman. *Computer Vision*. Prentice Hall, Upper Saddle River, New Jersey, 2001.
22. L. Trujillo and G. Olague. Synthesis of interest point detectors through genetic programming. In *Proc. of the Genetic and Evolutionary Computation Conf., Seattle, Washington, July 8-12*, pp. 887–894. ACM, 2006.
23. A. P. Witkin. Scale-space filtering. In *Proc. of the 8th Int. Joint Conf. on Artificial Intelligence, Karlsruhe, Germany*, pp. 1019–1022, 1983.