Brock University

Department of Computer Science

Online Image Classification Using Graphics Processing Unit-Based Genetic Programming

Mehran Maghoumi, Brian J. Ross

Technical Report # CS-16-02
August 2016

Brock University
Department of Computer Science
St. Catharines
Ontario Canada L2S 3A1
www.cosc.brocku.ca

Online Image Classification Using Graphics Processing Unit-based Genetic Programming

Mehran Maghoumi, Brian J. Ross

***Abstract*—A texture classification vision system implemented with Graphics Processing Unit-based genetic programming is described. An online learning environment is implemented, in which genetic programming is automatically invoked when unclassified texture instances are present in an image stream. Once a segment is positively classified, the genetic programming classifier expression is reapplied to frames of the image stream. System performance is enhanced using population-parallel evaluation on a graphics processing unit. Various experiments with textures of varying difficulty were performed. Real-time performance was often seen in cases using 4-segments of texture data, as correct classifiers were evolved within seconds. In all cases, once evolved, classifiers were easily applied to the image stream in real-time. This research shows that high-performance real-time learning environments for image classification are attainable with genetic programming.**

***Index Terms*—image classification, genetic programming, GPU programming, online learning.**

I. Introduction

Computer vision research is developing new methods to automatically acquire, process, and analyze visual information [1]. Object classification is the automatic recognition and identification of an object. Genetic programming (GP) [2][3] has been used in computer vision problems. Object classification has been studied with GP, in which an evolved expression is applied to image data to be classified. Examples of research in GP and computer vision include target discrimination [4], image segmentation [5][6], face detection [7], texture classification [8][9], and real-time object tracking in videos [10][11].

Although GP is distinguished by its ability to evolve executable programs, one of its known shortcomings is its lengthy computational time required to evolve programs. As with all evolutionary computation algorithms, GP spends the majority of its processing time on fitness evaluation [12], since the evaluation of large tree expressions can be expensive. Consequently, researchers have been accelerating GP execution by using parallel hardware, and in particular, the graphics processing unit (GPU) found on graphics cards [13][14]. A popular GPU programming interface is NVIDIA CUDA [15], and GP has been successfully implemented in CUDA [16][3][17].

The usual motivation for using GPU-based GP is to solve more complex classes of problems. GPUs enable increases in population sizes, maximum generation limits, and training set sizes, all with minimal effect on overall execution time. A second, and perhaps less-often considered, benefit of GPU-based GP is to improve the computational responsiveness of GP processing itself. Rather than tackle more difficult problems, GPU-based GP can be used to solve conventional problems more quickly. In this sense, it is interesting to consider whether GPU-based GP could become widely accepted for real-time online learning environments.

Online learning is a machine learning approach in which incremental learning proceeds immediately when training data is available [18]. Subsequent data will enable the model to be refined and corrected as necessary. This contrasts to offline learning, in which a model is learned from all the available data during a training phase, and thereafter is not further refined. Online learning is conducive in dynamic environments, where new and changing data may arise at any time. It also lends itself well to real-time, interactive environments, since learning can occur on the fly, and refine itself as necessary. The GP literature in computer vision mentioned earlier uses offline learning, in which GP is pre-trained on example cases, after which the evolved solution is used to solve the problem. GP training may take minutes, hours, or longer, before a solution is obtained. There are no GP execution time requirements in such situations.

We use GPU-based GP for online texture classification. Classification is done on images from the Brodatz texture library [19], which is a standard image set used in computer vision research [20]. Brodatz textures are also used in Song *et al*.'s seminal GP-based texture classification research [8][6][21][22]. We implemented a GPU-based GP system in NVIDIA CUDA [23]. The system inspects synthesized image stream frames, and when unclassified or mis-classified textures are detected, GP is immediately invoked to obtain classifiers. The evolved GP classifiers are applied to the image stream as they become available, resulting in a real-time online learning environment.

The main motivation of this research is to explore the feasibility of using GP for real-time texture classification. We are interested in the speed at which GP can generate and apply classifiers. Our GP system design (language, fitness evaluation, training) is inspired by the work of Song [21], which was highly effective at Brodatz texture classification. Our results will show that GPU-based GP is a fast, real-time machine learning paradigm, at least for the smaller problem instances that we studied. This is a significant result, given that GP is renown for being one of the slowest machine learning

Department of Computer Science, Brock University, St. Catharines, ON, Canada, L2S 3A1

algorithms. Computer vision problems are also computationally involved, and showing that GP can be efficient in these problems is noteworthy.

This research is not intending to compare GP's performance with other computer vision techniques. There is much related research, for example, real-time Brodatz texture classification using mainstream computer vision algorithms [20], non-real-time Brodatz classification using neural networks [24], and Brodatz texture boundary detection for real-time tracking [25]. Other approaches may be faster and more accurate than ours. Although a comparative study of GP and other techniques is overdue and worth pursuing, it is outside the scope of this research. Further details about this research are in [26].

II. Literature Review

A. Genetic Programming and Image Classification

Because of its flexibility, GP has been adapted for various image classification tasks. The research in GP and image processing by Song *et al*. is the inspiration for our own work. Song *et al*. [8][21] used GP for Brodatz texture classification. A block processing approach is used, in which rectangular blocks of raw pixel data are processed by a simple GP classifier using basic mathematical operators, and a classification decision is made for the entire block. Song and Ciesielski [6] used similar texture classifiers to perform texture segmentation. They executed multiple classifiers on an image, and recorded the number of times each pixel in a block was classified correctly by each classifier. More recently, Song *et al*. [27][11] used GP to evolve object detectors and trackers.

Other examples of using GP in computer vision applications include the following. Tackett [4] was one of the first to use GP in image classification, where GP was used to discriminate target and non-target objects in a series of images. Poli [5][28] used GP for feature detection and image segmentation. Winkeler and Manjunath [7] used GP for face detection. Howard *et al*. [29] used GP for detection of ships in synthetic aperture radar (SAR) imagery. Harvey *et al*. [30] used GP to detect golf courses in the aerial images. Zhang and Smart [31] use GP for multiclass image classification using dynamic decision boundaries. Smart and Zhang [10] used GP for real-time object tracking in streaming videos. Krawiec and Bhabu [32] use linear GP to evolve real-world 3-dimensional object recognizers for raw image data. Zhang *et al*. [33] used GP for object detection. Ross *et al*. [34] used GP for mineral identification in hyperspectral images. Kowaliw *et al*. [35] used Cartesian GP (CGP) to evolve image transforms. Al-Sahaf *et al*. [36] presented a two-tier GP system in which the first tier extracted complex features from images and the second tier was responsible for image classification. Later, Al-Sahaf *et al*. [9] used GP to extract features inspired by local binary pattern [37] from textures. The extracted features were then used as the input to various classifiers. Leitner *et al*. [38] used CGP [39] to automatically classify geological features on Mars. Harding *et al*. [40] demonstrated the robustness of CGP for image processing.

B. GPU-based Genetic Progamming

Parallel computing is a form of computation in which multiple operations and calculations are done simultaneously [41]. *Embarrassing parallelism* is a model of parallelism in which the tasks of an application rarely or never communicate with each other. GP is also an example of an embarrassingly parallel algorithm [3].

There are many examples of parallelizing GP, and the resulting systems experienced significant speed gains. According to Cano *et al*. [12], more than 85% of the execution of a GP system is spent on fitness evaluation, while less than 1% of a GP run is spent on genetic operations. Consequently, any parallel model must focus on improving the performance of the evaluation phase. There are mainly two approaches to running GP on parallel hardware [16][3][17]. The first approach is evaluating a single program on parallel fitness cases *(fitness-case parallel)*. The alternative approach is to evaluating multiple programs in parallel *(population parallel)*. Population parallelism is more common in the literature.

In recent years, GPUs have been extensively used for accelerating applications that demonstrate parallelism and GP is no exception. Meyer-Spradow *et al*. [42] evolved pixel shaders using short linear assembly language and used them for real-time interactive rendering. Harding and Banzhaf [43] ran evolved programs on GPUs to accelerate the evaluation phase of GP in a fitness-parallel setup. Langdon and Banzhaf [44] implemented a population-parallel scheme on GPUs, and treated each GP individual as a single parallel program. Wilson and Banzhaf [45] developed a linear GP system on Microsoft Xbox 360 video game console to solve regression and classification problems using population parallelism. Comte [46] used a Sony PlayStation 3 video game console to evolve a parallel population. Agusto *et al*. [47] used a population-parallel approach to evaluate all individuals in the population using OpenCL [48]. Banzhaf and Harding [49] implemented the Cartesian GP model with GPUs.

The CUDA framework by NVIDIA has also been used for accelerating GP. Every CUDA program has one or more parts that are either executed on the *host* (CPU) or the *device* (GPU). The code that executes on the device is called the *kernel* function. The kernel function can create many threads to solve a problem. These threads are organized in groups called *blocks*, and blocks are organized into *grids*.

One of the first efforts to incorporate CUDA in GP was by Robilliard *et al*. [50]. They used CUDA to evaluate a parallel population of 1000 individuals. Their work is notable for combining the fitness and population parallelism together, which they termed *BlockGP*. Later, Robilliard *et al*. [17] successfully changed their breeding module in way that it directly evolves linear postfix expressions. They showed that this representation increased the performance up to 7 times. Langdon [16] used a similar scheme to solve Boolean 20-multiplexor and 37-multiplexor with 137 billion fitness cases. Although it was estimated in [51] that the 20-multiplexor problem would take 4 years to solve using GP, he solved it in less than an hour.

C. Real-time Genetic Programming

At the time of writing, publications regarding real-time and online GP systems are scarce. An example of a system which used parallel methods is the work by Harding [52]. He used CGP to evolve image filters and his goal was to evolve noise removal filters that performed better than the traditional median filter. In a later work, Harding and Banzhaf [53] used CGP to reverse engineer the image processing filters. Nording and Banzhaf [54] used a real-time and online GP system to evolve programs that would control the movements of a miniature robot. Ebner [55][56] proposed an adaptive online evolutionary visual system that could recognize various objects. He later expanded the system in [57] and was able to make the system fully automated and real-time.

Real-time evolutionary algorithms for computer vision have been implemented. Kaufmann *et al*. [58] used a real-time evolutionary algorithm for hand posture recognition. Boumaza and Louchet perform real-time parameter exploration for 3D robot vision using an evolutionary strategy called The Fly algorithm [59].

III. The Learning Environment

We are primarily concerned with evaluating GP's performance in quickly obtaining good-quality image classifiers within an online setting. We do not intend to study all the aspects of computer vision that are required in this problem domain. For example, topics such as image extraction from real-world data, noise removal, image segmentation, and advanced tracking algorithms, are outside our focus. Therefore, we simplified the visual environment so that we could concentrate exclusively on the issues of online GP classification.

One simplification we make is to use a synthesized image stream, in which image data (frames) are generated by an OpenGL application. This permits the quick generation of noise-free images for training and testing. It gives us complete control of the size and location of image objects in the frames. It is also convenient for post-processing frames for evaluation purposes, say, by colour rendering objects that have been classified. We introduce new segments into the image stream after previous frames have been fully processed. This results in a data stream that gradually increases in difficulty during the learning session.

The following constraints are placed on the visual environment:

1) Segments are squares (96-by-96 pixels), whose locations are known in the image frame. The size and locations of segments are used during GP training.
2) Each segment in a frame contains a unique texture pattern. Duplicate textures are not permitted between segments.
3) Once a segment is introduced to the image stream, it never disappears.
4) Segments never occlude each other.

Point 1 represents the use of a "perfect" segmentation algorithm. Points 2 and 3 help the machine learning environment determine when classifiers are erroneous (they recognize common segments, or do not recognize any segment). Point 4 simplifies pixel data extraction and analysis.

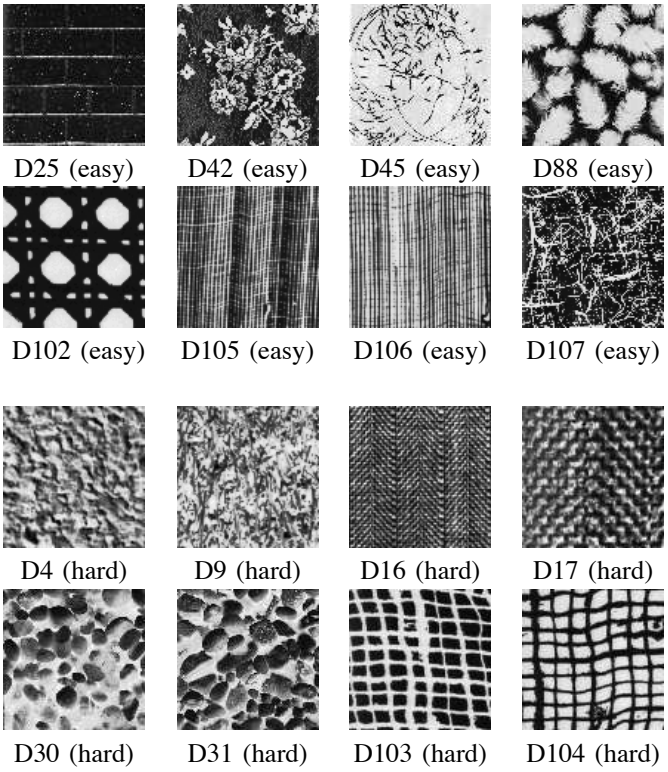


Fig. 1. The selected Brodatz textures. Each image is 96×96 pixels and 256-grey scale. Easy and hard categories based on work in [60]

Our image data is comprised of Brodatz textures [19], which are 96-by-96 pixel 256-level grey-scale images that were studied extensively by Song *et al*. [21][6][8]. The data we used is in Figure 1. The labels "easy" and "hard" approximately identify the degree of difficulty found with the images. We assign these difficulty categories based on our earlier research with GP to classify Brodatz textures, where some textures were more challenging to classify than others [60]. The varying difficulty of texture classification is consistent with other research with Brodatz textures [24].

The learning environment is shown in Algorithm 1. The GP-based vision system functions in an automated and unsupervised manner, with a minimal level of human intervention. The main goal is to obtain a collection of classifier expressions. To perform multiclass classification with GP, we employed the *one-vs.-all* scheme [61]. As such, there will be as many classifiers as there are segments, and each classifier evolved by GP will uniquely identify one of the segments in the current frame. So long as each segment in the image is uniquely paired with one classifier, the overall machine learning task has been satisfied.

The GP expression is applied to the pixels of an image, in order to determine its classification. Given a frame in the image stream, all segments within it are identified. The goal is that each segment should be associated with a single classifier expression that uniquely identifies or "claims" that segment. A classifier claims a segment if it positively classifies the majority (>50%) of the segment pixels as true. If each classifier uniquely claims one segment, then the image data is correctly classified.

Algorithm 1: Automated learning environment

```
classifiers ←{}
segments ←{}

while true do
    frame ←NextFrame()
    Append(segments, SegmentsOf(frame))
    orphans ← {}

    foreach s in segments do
        foreach c in classifiers do
            Classify(c, s)
        end

        if NotClassified(s) then
            Remove(segments, s)
            Append(orphans, s)
        end
    end

    foreach c in classifiers do
        count ← 0
        foreach s in segments do
            count ← count + Classify(c, s)
        end

        if count ≠ 1 then
            Remove(classifiers, c)
        end
    end

    if IsIdle(GP-Engine) then
        foreach s in orphans do
            Append(classifiers, EvolveClassifier(s))
        end
    end

end
```

In order to evolve classifiers in the environment, the following occurs. Should a segment be unclaimed by any classifier, then it becomes an "orphan", and requires a classifier. Similarly, if multiple classifiers claim the same segment, then that segment is being misclassified, and it is an orphan. An orphan segment needs a classifier to be evolved for it, and an invocation to GP is required. The system will use the existing segments to construct positive and negative training data for GP. After the GP run is through, an evolved classifier will be returned. Hopefully the classifier will correctly classify the segment, and it will no longer be identified as an orphan.

Sometimes errors will happen with classifiers. A classifier expression may claim more than one segment, or claim no segments at all. Such erroneous classifiers will be deleted, and new invocations of GP will attempt to generate correct classifiers to replace them. It is also possible that some challenging textures may confound GP. For example, the "hard" images in Figure 1 can be difficult to classify. We set a maximum limit of 25 failed attempts in invoking GP to evolve a classifier for a segment. Should a segment still be unclassified after 25 GP invocations, then it is labelled a "permanent orphan", and will be ignored for the rest of the session.

To summarize the processing done by Algorithm 1, the following situations may occur during a learning session:

1) If all segments are uniquely classified by separate classifiers, then are no problems and no GP training is necessary.
2) If after running all available classifiers on the frame there is a segment that is orphan, then a new classifier must be evolved for it. The texture of the orphan segment is given to GP as positive example data, while all other segments in the frame, and background pattern, are treated as negative image data.
3) If a classifier has claimed more than one segment or a segment is claimed by more than one classifier, the problematic classifier(s) are removed. Segments that were claimed by them will become orphans, and new classifiers will be evolved for them.
4) Conservative classifiers that do not claim any segments are removed.

Note that the system resolves conflicts in a sequential manner. Only one classifier is evolved at a time by GP. Orphan segments automatically invoke GP training, and so the evolution of a new classifier is issued on a "need only" basis. Likewise, should all segments be uniquely identified by the current ensemble of classifiers, GP will not be invoked.

Each classifier is applied to all the segments in the frame. The resulting information from the classifier (true or false) will detemine whether a segment claimed by the classifier, or is a negative segment. When a segment is claimed by a single classifier, it is rendered on the frame with a distinguished color. This indicates that it is correctly classified from the other segments.

Our environment can cope with dynamic changes in the image stream. For example, as soon as a new segment is introduced, it will be detected as a new orphan, and GP will be automatically invoked to find a classifier for it. Similarly, should a texture of a segment suddenly change, a new invocation of GP can occur.

IV. System Architecture

Our design philosophy was to make a straight-forward system architecture, and basic GPU implementation. In the future, a more sophisticated implementation involving micro-optimization of the kernel code would result in greater performance gains. Even with our simple but sub-optimal use of the GPU hardware, impressive performance gains will be seen (Section VI).

The GP system uses the Java-based ECJ environment [62]. The GPU coding uses NVIDIA CUDA v5.5 [15]. JCuda [63] is used for Java/CUDA interoperability. Following JCuda's convention, the CUDA kernel code for this system is written in C. All source code for our implementation is available, including a small library called *TransScale*, which allows agile CUDA development for CUDA using JCuda, and CUDA extensions for ECJ [1].

Population parallelism is used, in which the fitness of as many individuals as possible is evaluated in parallel. Before an individual is evaluated, it needs to be transferred to the GPU memory. As done in [44][50], we use postfix notation for tree representation in our GP system. A stack-based postfix expression evaluator was implemented in CUDA. To evaluate

[1] https://www.github.com/Maghoumi/

a GP tree, it is first converted to postfix in ECJ, transferred to the GPU, and the CUDA evaluator parses and executes the postfix expression.

Since CUDA memory transfers are very slow, it is a good practice to pre-allocate all the memory that is required. Therefore, for any evolutionary run, the coordinates of the training points along with all the data that are required for fitness calculation, are transferred to the GPU memory, and a reference to the allocated space is stored throughout the run. All the data is flattened and stored in contiguous memory addresses. Thus, the whole population will be evaluated in a single CUDA kernel call.

For evaluation, we adopted the method in [50], in which each block is responsible for evaluating a single individual on all fitness cases. Depending on the total number of fitness cases, a CUDA thread is responsible for performing the required calculations for one or more of the fitness cases. When the kernel has concluded, an array of fitness values is returned from the kernel. Each element of this array corresponds to the fitness value of a single individual in the population. This fitness array is copied back to the main memory, which ECJ uses to assign fitness to the population.

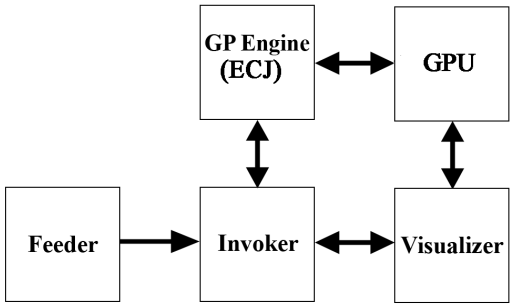


Fig. 2. System architecture. The Invoker takes image data from the Feeder, and calls the GP Engine whenever segments need to be classified. The GP Engine (ECJ) evolves a classifier for a segment. It calls the GPU using CUDA to quickly evaluate classifier expressions on the image stream. The Visualizer applies evolved classifiers to image frames, and uses the GPU to accelerate classifier execution.

The system is comprised of various modules (Figure 2). The GP Engine (ECJ) manages the GP evolution call requests. The GP system runs on its own host thread and monitors a job queue. To initialize evolution, a job must be added to this queue. Each job contains a positive and negative image segment(s). As soon as a job is added to the GP system's queue, the evolution process begins asynchronously. When evolution has concluded, the solution classifier is returned.

The GPU takes postfix classifier expressions from the GPU engine via CUDA, and runs them on image frames during fitness evaluation. The Visualizer will invoke the GPU in order to quickly evaluate evolved classifier expressions on an image.

The Invoker module is the link between all other modules in the system. This module manages the communications between the Visualizer and the GP Engine. The Invoker module obtains a single image frame from the Feeder, passes that frame to the Visualizer and handles the GP Engine invocation requests. When the evolution results are ready, the Invoker takes the results and passes them back to the Visualizer.

The purpose of the Feeder module is to obtain image frames from a host and to pre-process the frame. At the Invoker's request, the Feeder will pass the next processed frame to the Invoker. As discussed in Section III, the segment locations and boundaries are known in advance. In the future, an actual segmentation algorithm could be used.

The Visualizer module applies the evolved classifiers on the image frame. It enforces the automatic learning rules that were outlined in Algorithm 1. The Visualizer prioritizes the evolution of classifiers for orphan segments, and resolves conflicts between classifiers otherwise. The order in which the orphan segments are selected for training is arbitrary. For the visualization of the results of the evolution, currently the GP trees are only executed on the segment pixels, and not the image background .

V. Experiments

A. GP Language

The GP language used is shown in Table I. The language performs pixel classification, in that it makes a classification decision about a single pixel of interest. It is strongly-typed [64], and supports float and integer expressions. Operators consider both spectral (single pixel values) and spatial (image area, shape) features of the image. When a GP tree is evaluated, positive or zero values from the root are considered as positive classification, and negative values are negative classification. Most of the operators are standard in the GP vision literature. Protected division is used to prevent divide-by-zero errors; zero is returned in such cases. The Avg_k and $Stdev_k$ terminals are convolution filters that evaluate k-by-k square areas of the image surrounding the centre pixel, where k can be 15, 17 or 19. Average will blur an image, while standard deviation is similar to an edge detection filter. Both filters are computed on the fly during tree expression execution on the GPU.

We chose the language in Table I based on a comparative study of GP classification languages [60]. We found that this language was one of the best performing one in terms of solution quality and efficiency. It is also easy to implement in CUDA. The language is inspired by Song's GP language used in Brodatz texture classification [8][21]. He used a simple set of mathematical and conditional operators to analyze blocks of pixels, and makes a classification decision for the entire block at once. Our language differs from Song's in that we classify single pixels, rather than entire pixel blocks. We also use spatial image processing filters (Avg_k and $Stdev_k$), unlike Song. Using the same Brodatz image data used in this paper, we found that this language outperformed Song's and other variations [60].

B. Training and Fitness Evaluation

Each GP run has sets of positive and negative segment instances or examples. Examples are randomly sampled pixels used for training. The negative examples are uniformly sampled from all negative segments. Each instance is assigned a

TABLE I
GP LANGUAGE

Type	Name	Ret. Type	Arg. Type	Description
Function	Add	I/F	I/F	addition
	Sub	I/F	I/F	subtraction
	Mul	I/F	I/F	multiplication
	Div	I/F	I/F	protected division
	Neg	F	F	negation
	Exp	F	F	e raised to the operand
	IfGT	F	F,F,F,F	if $a > b$ then c else d
	Max	F	F,F	maximum
	Min	F	F,F	minimum
	Sin	F	F	sine
	Cos	F	F	cosine
Terminal	ERC	F	-	ephemeral random constant between [0, 1]
	Intensity	F	-	pixel luminosity
	Avg	F	-	average of $k \times k$ area
	Stdev	F	-	standard deviation of $k \times k$ area

Ret. Type and *Arg. Type* are the return and the argument types respectively (I=integer, F=float, $k \in \{13, 15, 17\}$)

classification label. After selection, the instances are shuffled and transferred to GPU memory. When the CUDA evaluation kernel is invoked, these samples are used as fitness cases for the evaluation function.

We use Song's classification accuracy [8] as the fitness function for the experiments. The classification accuracy is defined as:

$$\text{fitness} = \frac{\text{TP} + \text{TN}}{\text{Total}} \times 100$$

where *TP* is the number of true positives, *TN* is the number of true negatives, and *Total* is the total number of pixels tested. We used a 1:3 ratio of positive to negative classes, which we found gave better performance.

C. GP Parameters

TABLE II
RUN PARAMETERS

Parameter	Value
Population size	1024
Generation size	100
Crossover rate	90%
Mutation rate	10%
Selection method	Tournament (size=4)
Elites	2
Positive examples	512
Negative examples	1024
GP invocation limit	25
Number of runs/experiment	20

Table II summarizes the GP system parameters. Most of them are standard in the literature [2]. Positive and negative examples refer to the number of pixel samples from the segments used for training.

TABLE III
SUMMARY OF THE EXPERIMENT SETS

Mode	Difficulty	Num. Textures	Label
All	Easy	4	All-easy-4
		8	All-easy-8
	Hard	4	All-hard-4
		8	All-hard-8
		12	All-hard-12
		16	All-hard-16
One	Easy	up to 8	One-easy-8
	Hard	up to 16	One-hard-16
Eager	various	various	various

All experiments are performed on a Windows 7 x64 machine with an Intel Core-i5 3570 processor (four cores running at 3.40GHz), 8GB of RAM, GeForce GTX 660 graphics processor (960 CUDA cores clocked at 1033MHz with 2GB of GDDR5 RAM clocked at 6008MHz), CUDA v5.5 with GeForce v335.23 drivers.

D. Experiment Variations

Three types of environments for experimentation were used. In the first set, which we call "All-at-once", the image stream presents all the segments simultaneously. Using the automated learning rules from Section III, the Visualizer attempts to evolve unique classifiers for all the segments. A session is successful if all segment textures – with the exception of permanent orphans – are uniquely classified.

Another session variant, named "One-at-a-time", involves an image stream in which new random segments are added incrementally. A session begins with two textures. This *mini-session* is processed until both textures (other than permanent orphans) have unique classifiers. When the mini-session ends, another texture is added to the mix. Then a classifier is evolved for it, using the procedure in Algorithm 1. Since the appearance of a new texture will likely cause errors for existing classifiers, retraining will be necessary. This process continues until a maximum number of textures have been added. Since the ordering of textures can affect classifier performance, we randomize the ordering in each run.

A final environment was designed, which is called "Eager". The All-at-once and One-at-a-time sessions will always run GP until the maximum generation limit. However, Eager sessions quit GP evolution immediately when a correct classifier is found.

Table III summarizes the experiments undertaken. Experiments vary in the number of textures selected, and the difficulty of textures used. "Easy" experiments will select random textures exclusively from the textures labelled easy in Figure 1. "Hard" experiments randomly select from both the easy and hard textures. We use the notation *[S-D-C]* in which *"S"* signifies the experiment set, *"D"* signifies the difficulty of the textures used in the experiment and *"C"* signifies the number of textures used in the experiment. Therefore the notation *"All-hard-12"* means that the experiment was carried out for the first experiment set, with 12 textures from the difficult set.

VI. Results

This section examine different performance features of the experiments. Where applicable, the results are averaged over 20 runs.

There are two closely related aspects to consider when evaluating performance of the system. The first is the quality of solutions obtained by GP. This factor relates to the difficulty of the image classification problem as presented to GP. Should GP have difficulty in finding acceptable classifiers, then the online learning environment will require multiple invocations of GP runs, which of course results in delays in the session. The second aspect is the speed at which GP runs are executed on the GPU hardware, as well as the speed at which evolved classifiers are executed by the GPU on the image stream. Faster throughput will result in an overall speedup of sessions.

A. GP Invocations Required for Training

TABLE IV
Average Number of GP Invocations (95% confidence).

Experiment	Invocations		Evaluations	Normalized
	Avg.	±		
All-easy-4	4.00	0.00	409,600	1.00
All-easy-8	17.00	4.02	1,740,800	2.12
All-hard-4	9.20	4.52	942,080	2.30
All-hard-8	39.80	9.57	4,075,520	4.97
All-hard-12	96.80	14.76	9,912,320	8.06
All-hard-16	185.52	10.76	18,997,248	11.59
One-easy-8	23.45	2.67	2,401,280	2.93
One-hard-16	192.25	9.27	19,686,400	12.01

Evaluations is the number of GP trees evaluated: avg # invocations × 100 (generations) × 1024 (population size). *Normalized* values are the average values divided by the number of textures in the experiment.

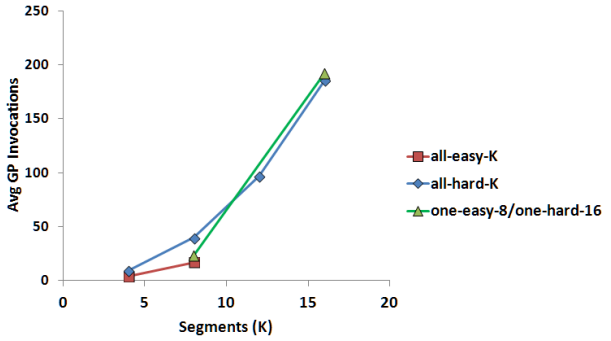
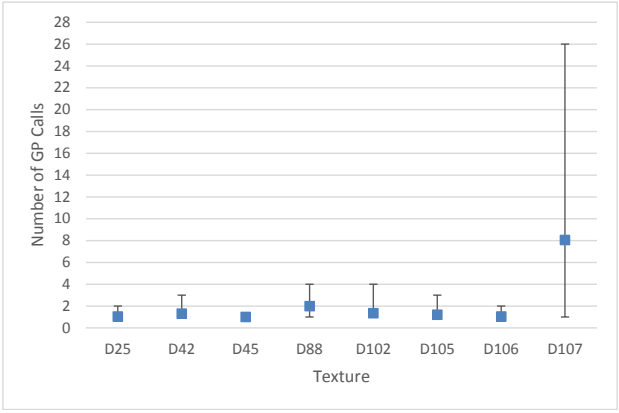
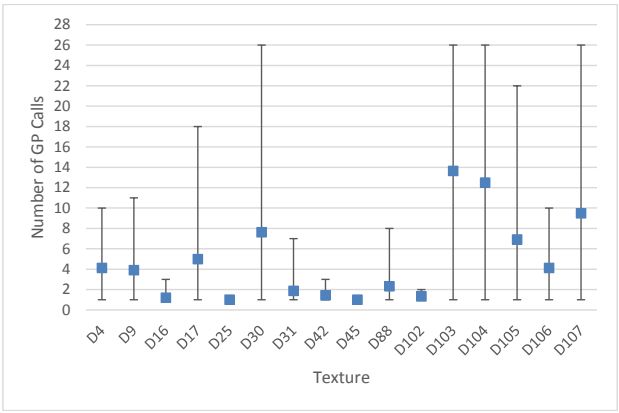


Fig. 3. Plot of average GP invocations per texture. In this and other plots, the bars represent the range of measured values.

Table IV shows the number of GP invocations required for each set of experiments. The confidence level value specifies the range of numbers of GP invocations required for a particular data set with a confidence of 95%. Figure 3 plots the data according to experiment type. The plot shows a trend with respect to the number of segments K. The 3 "easy" texture experiments show lower invocations than the "hard" ones for the same K values. Examining the table, the "easy" texture experiments require between half to a third of the GP invocations than their "hard" equivalents. The number of GP invocations is also proportional to the number of textures being considered. This makes sense, given that a greater number of texture objects represent a more difficult classification problem. The One-at-a-time runs require more invocations than the All-at-once. Gradually adding textures acts like a "moving target" for evolution. New texture objects introduce errors with existing classifiers, requiring their replacement. In All-at-once runs, the learning task is unchanged from the start.



(a) All-easy-8



(b) All-hard-8

Fig. 4. GP invocations per texture

Figure 4 shows the average and range of GP invocations per texture for the All-easy-8 and All-hard-8 runs. This confirms the difficulty level of the selected textures set. Note that the "easy" texture D107 is apparently more difficult than we had previously assumed.

B. Permanent Orphans

A permanent orphan is a segment that cannot be classified correctly after 25 separate invocations of GP. Table V summarizes the number of permanent orphans seen at the end of each experiment run. The supplied ranges show the confidence interval at 95%. Higher number of textures will result in a greater number of permanent orphans. The likelihood of having a permanent orphan is more dependent on the number of textures in the image frame and less on the difficulty of the textures. Although the All-hard-8 set contains more

TABLE V
AVERAGE NUMBER OF PERMANENT ORPHANS (95% CONFIDENCE)

Experiment	Permanent Orphans	
	Avg.	±
All-easy-4	0.00	0.00
All-easy-8	0.10	0.14
All-hard-4	0.10	0.14
All-hard-8	0.65	0.38
All-hard-12	2.20	0.61
All-hard-16	5.40	0.53
One-easy-8	0.00	0.00
One-hard-16	5.15	0.43

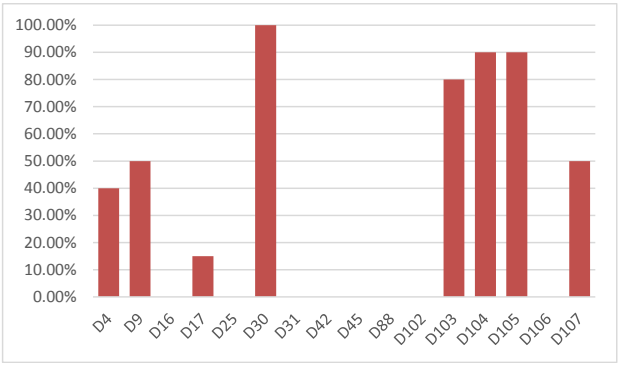


Fig. 5. Percentage of time a texture was a permanent orphan (One-hard-16).

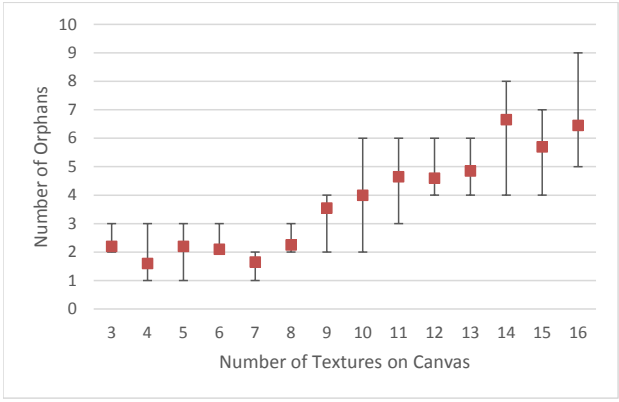


Fig. 6. Number of textures vs. number of orphans: One-hard-16

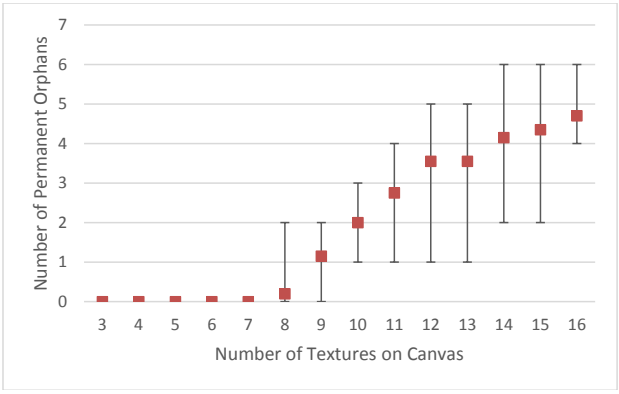


Fig. 7. Number of textures vs number of permanent orphans: One-hard-16

challenging textures than the All-easy-8 set, they result in the same number of permanent orphans. For some of the harder experiment sets, the number of textures that were distinguished (i.e. non-permanent orphans) is roughly 10. This suggests that 10 is a practical maximum number of textures that we can competently classify, at least using our system setup and image data. Improvements to the system design (GP language, training strategy, etc.) would have to be explored to increase this limit.

Figure 5 depicts the average percentage of time each texture was a permanent orphan during the One-hard-16 experiments. Note that most of the "easy" textures from Figure 1 were never permanent orphans, other than D105 and D107. Those textures became more challenging when other "hard" textures were included. Permanent orphans never arose in the One-easy-8 runs.

C. Incremental Introduction of Textures

The One-at-a-time runs also show interesting behaviour with regards to the number of orphans seen. Recall that a mini-session is the portion of a session during which a newly added segment is classified. Figure 6 provides a timeline for the mini-sessions. Each step on the horizontal axis shows the current number of textures in the stream. Since we start the experiments with 2 initial textures, the system always has 2 orphans at the beginning of the run. (This step has been omitted from the plots for simplicity.) After the mini-session containing the two textures has concluded successfully, one texture is added until the maximum number of textures is reached. As can be seen in Figure 6, classification difficulty increases as the number of textures increases. This indicates that the problem complexity is increasing, and that earlier evolved classifiers often become erroneous when new textures arrive.

Figure 7 shows the relationship between the number of textures and the number of permanent orphans for the One-hard-16 experiments. The number of permanent orphans in the system increases with the introduction of additional textures. This is again a product of increasing problem complexity, as there were no permanent orphans in the One-easy-8 runs.

D. Eager Termination

We did some experiments using Eager mode. Although the maximum number of GP generations is still 100, an Eager run immediately terminates when a correct classifier is found. We ran the All-hard-8 experiments with Eager termination enabled.

We found that Eager termination is a sensible means for improving performance. The average execution time of Eager version of All-hard-8 runs were an average of 8.2 times faster than the normal versions. Frame rates of these Eager runs were an average of 17.0 frames/sec, while normal runs had frame rates of 11.2 frames/sec. This is due to trees being smaller in the Eager runs, since for some textures, fewer generations are necessary to find suitable classifiers.

A t-test at the 95% significance level revealed that the number of permanent orphans did not change significantly in the experiments with the eager termination.

E. Frame Rate and Tree Size

TABLE VI
AVERAGE FRAME RATE AND TREE SIZE FOR DIFFERENT EXPERIMENTS.

Experiment	FPS (during)	FPS (final)	Tree Size	Correl. Coeff.	Normalized
All-easy-4	10.85	25.35	219.46	-0.88	6.33
All-easy-8	6.67	11.37	250.13	-0.72	5.68
All-hard-4	9.97	25.24	238.49	-0.77	6.31
All-hard-8	5.76	11.56	244.81	-0.12	5.78
All-hard-12	4.72	6.68	226.38	-0.02	5.01
All-hard-16	3.97	5.46	240.08	0.00	5.46
One-easy-8	8.63	11.52	241.21	-0.70	5.76
One-hard-16	4.93	5.87	230.40	-0.37	5.87

Normalized is the final frame rate multiplied by $K/16$, for any case where $K < 16$. Normalization is done separately for the all and one experiments.

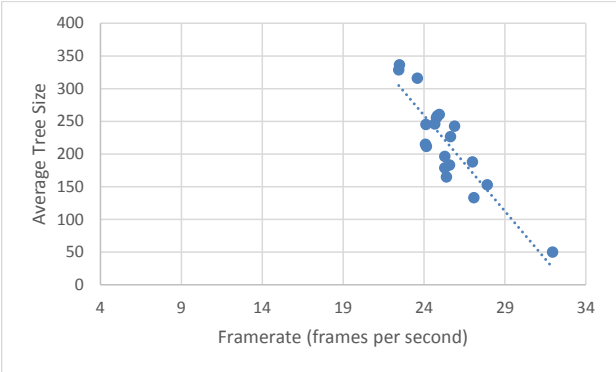


Fig. 8. Frame rate vs. tree size: All-easy-4

We also examined the execution speed of the system. One measure of responsiveness is the frame rate of the visualizer during and at the end of a session. The frame rate is affected by the number of textures that were used for the run. More classifiers need to be evolved and executed for higher number of textures. Frame rate is also affected by the tree size of evolved classifiers, as well as the number of GP invocations.

Table VI shows the average frame rate and tree size of each experiment both during the evolutionary run and at the end of a session. The "during" frame rate was measured when a GP invocation was issued and the GP engine started running. The "final" frame rate was measured when the session was successful and all no further GP invocations were necessary. The table reveals that GP invocation affects the frame rate, since the values in the second column of Table VI are lower than those in the third column. Furthermore, the more difficult a set is, the lower is the corresponding frame rate. We calculated the correlation coefficient of frame rate (at the end of the run) and the average tree size based on 20 observations. As expected, higher frame rates generally correlate with smaller trees.

Table VI also shows that the number of textures in each experiment affects the measured frame rates. To account for this, we normalized the values of the final frame rate for experiments having K textures ($K < 16$), by multiplying the scale factor $K/16$. This is done separately for the "all" and "one" experiment variations. For example, if an experiment has 8 textures, its frame rate is divided by 2 since the experiment with 16 textures has twice as many textures. The normalized frame rates are closer, and show that the raw final framerate is indeed inversely proportional to the number of textures being processed.

Figure 8 plots the frame rates versus average tree size for the All-easy-4 experiment. The trend lines show the relationship between the two, and confirm the correlation between frame rate and tree size. Other experiments showed similar linear correlations.

TABLE VII
EXPERIMENT THROUGHPUT

Experiment	Average Permanent Orphans	Throughput
All-easy-4	0.00	0.82
All-easy-8	0.10	1.65
All-hard-4	0.10	0.86
All-hard-8	0.65	1.53
All-hard-12	2.20	1.63
All-hard-16	5.40	2.04
One-easy-8	0.00	1.63
One-hard-16	5.15	2.16

Hardware is Intel Core-i5 3570 at 3.4 GHz, and GeForce GTX 660 at 1.033 GHz. *Throughput* is billion GP operations per second.

Table VII shows the total throughput of the system for different experiments. Note that the GPU applies every classifier to every segment in each image: for m textures, $m \times m$ classifiers are executed on 96×96 pixels. For example, for 8 textures, there are $8 \times 8 \times 96 \times 96 = 589824$ applications of classifiers to pixels *per frame in the image stream*. These rates are in line with similarly reported ones for GPU-accelerated GP in image processing [65]. During a classifier expression execution, six relatively large spatial filters are calculated on the fly, possibly multiple times. Therefore, there is clearly significant amount of computation occuring during sessions.

To better grasp the advantage of using the GPU for tree evaluations, we re-ran the All-easy-8 experiments on the main CPU without GPU support. The average evaluation time per generation was 39.8 milliseconds for GPU runs, and 3.20 seconds for non-GPU accelerated runs. GPU accelerated runs are 80 times faster on average.

F. Example Videos

Example videos showing the system in action are available[2]. Figure 9 is a screen capture of one such video. The image shows the end of a session in which 8 textures have been correctly classified. As they animate left-to-right, they are rendered in different colours, indicating their success. In the videos, the text window will scroll, showing the current status of the GP system. Usually the text flies by fast, showing the

[2]http://www.cosc.brocku.ca/~bross/GPVisionCuda/

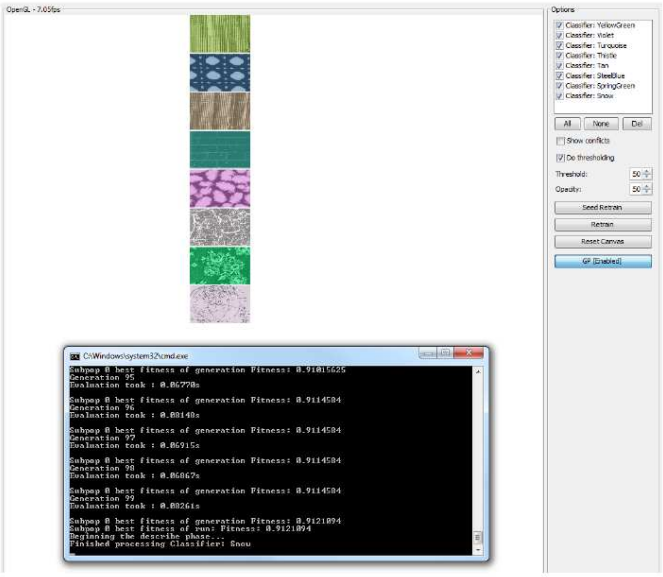


Fig. 9. Screen capture of a session.

speed at which GP is executed. However, typical runs will show the segment colours change according to classifier errors and difficulties.

Within all our videos, the 4-texture runs, and particularly the Eager variants, show very fast completion of classification. In many instances, solutions are obtained within seconds. In all instances, evolved classifier expressions are applied efficiently to the image data via the GPU, resulting in real-time frame rates.

The speed of runs degrades when 8 or more textures are used, due to the number of GP invocations required to find correct classifiers. The OpenGL video application also shows the current frame rate as the session proceeds, which we find to be a useful visual cue to the computation effort being undertaken by the GPU.

VII. Evaluation and Comparisons

The results in Section VI revealed two important insights. Firstly, texture classification with GP is a difficult problem. We knew that this would be the case, as earlier work in [60] showed that multi-texture classification of Brodatz textures is challenging. In this work, we found that the practical maximum number of textures that could be classified was 10. Furthermore, the style of learning session (All-at-once vs. One-at-a-time) impacted the effectiveness of evolution. The quality of our GP system is also affected by factors such as image data, GP parameters, training strategies, and GP language. Although our experimental design decisions were based on earlier trials, we do not claim that we have found an ideal configuration. In fact, it is likely that a more sophisticated GP language would produce better classifiers.

The second insight is that our quest for "real-time" GP was successful, but only for the smallest 4-segment experiments undertaken. In various 4-segment runs, correct classifiers were often obtained within seconds. The Eager experiments were particularly fast. We feel that this represents a significant milestone towards considering GP as a contender as a real-time machine learning paradigm for computer vision. On the other hand, it is debatable what it means for a system to be "real-time". Whether this means that the system should determine a solution in a fraction of a second, or a few seconds, or a few minutes, is unclear. Perhaps a system is real-time when the generation of a solution does not try the patience of the human, and does not result in a catastrophe within the application environment in which it is running.

In the course of our research, we discovered that system scalability was quickly overwhelmed by larger and more complex data sets. Thanks to the GPU support, frame rates seen in more complex experiments were often acceptable, and individual GP invocations might not take long to finish. However, overall session time suffered due to the dozens, if not 100's, of GP invocations. required to find correct classifiers for multiple texture objects. Scalability was hindered by the difficulty of correctly classifying greater numbers of textures.

Research regarding real-time evolution of GP systems for computer vision applications is scarce. Most of the literature involves the application of an offline pre-evolved solution in a real-time environment. We too found that our evolved solutions could be executed with real-time performance. Related to our work is research in GP and tracking in [10][27][11]. Smart and Zhang [10] used GP to evolve vectors that could best describe the location of the object in the new frame. While their evolved programs were suitable for real-time applications, their learning method was offline. In contrast, our online system which can correct its behavior dynamically. Moreover, their evolved solutions were only capable of detecting the moving objects, and were not intended to perform classification on them. Similarly, the motion plane features that were used by Song and Fang [27] were unable to discriminate different objects. Their pixel intensity features detected the motions of a specific object. Later, Pinto and Song [11] used the motion plane features for more complex videos. Although their evolved solutions were successful at detecting the motion of a specific class of objects (e.g. moving cars on a freeway), they did not discriminate between objects that belonged to the same class.

We anticipate that the implementations in [10][27][11] could be used as a pre-processor for our implementation. By detecting object motion, the pre-processor would be able to provide the image segment information to the GP engine. This segment information is then used for evolving classifiers that can uniquely identify each segment from all other segments. This would result in a full tracking environment.

Nordin and Banzhaf [54] used a real-time and online GP system to evolve control programs for a miniature robot. Although the problems they worked on are inherently different from ours, many aspects of their system are similar, such as multiple invocations of the GP system or real-time evaluation of the fitness using the problem's environment. Their system was built for embedded applications which restricted the available computational power, while ours requires an actual CUDA compatible hardware.

Ebner [57] developed an automated GP system for object detection. His initial research required user intervention, and was capable of achieving 4.5 fps on a 320×240 video [56].

His implementation worked by classifying objects, while ours depends on finer-detail texture images. His approach relied on frame differencing in order to detect moving objects while our implementation requires a segmentation algorithm that can feed segment information to the GP engine.

We are unaware of any studies comparing GP with other techniques in computer vision applications. The Brodatz texture library is well known in computer vision research. Picard *et al*. use principal components analysis and autoregressive algorithms on the Brodatz database [20]. They report 99% accuracy and real-time performance, although no time measurements are given. Diazi-Pernas *et al*. trained neural networks in a supervised, non-real-time environment on sets of Brodatz textures [24]. Classification accuracy ranged from 100% to less than 60%, depending on the texture images. They mentioned that texture D30 was one of the most difficult textures, which we also found to be the most challenging one (Figures 4 and 5). This gives some circumstantial evidence that neural networks and genetic programming may have similar classification capabilities in this problem domain. Other researchers have examined real-time learning with neural networks [66]. Many are working on GPU acceleration of neural networks, for example, for object tracking [67]. We are certain that GPU acceleration will benefit neural networks in computer vision applications such as this one.

VIII. Conclusions and Future Work

We have described an online GP-based texture classification system. The system showed real-time performance in the simpler experiments. Although the more difficult experiments were not real-time, they achieved a throughput of around 2 billion GP operations per second.

There are many directions for future work. GP's classification power will greatly benefit with the use of a more sophisticated GP language for image classification. Some of the image processing done elsewhere could be incorporated into the language [20][24]. The effect of noise on images should also be considered [68]. The OpenCV CUDA library could be worth consideration in accelerating a more advanced GP classifier language. Many spatial operators are implemented in OpenCV, and using them could be beneficial. More advanced training strategies (e.g. fitness sharing) are also worth considering. Other kinds of image data, including colour images, could be considered instead of the Brodatz textures.

In the future, the realization of real-time GP will benefit from the fact that graphics cards continue to get faster and cheaper. By micro-optimizing the CUDA kernel, and expanding the kinds of parallelism used, we would see even greater performance gains.

Another direction for future research is to improve the sophistication of the computer vision architecture of our system. Our synthetic image stream could be replaced with real video, in which a segmentation algorithm analyzes the frames before being processed by GP. This would be a natural "next stage" of this research, and especially if the segmentation implementation could be GPU supported. Motion detectors could be used in a pre-processor stage for our implementation. Using ideas from [10][27][11], the processor supplies the GP engine with a list of objects that are currently moving. Using this information, GP can evolve a unique classifier for each moving object. This could be used for object tracking.

Acknowledgements: Thanks to Cale Fairchild for his assistance with hardware issues. This research was partially supported by NSERC DG 138467.

References

[1] R. Szeliski, *Computer Vision: Algorithms and Applications*. Springer, 2010.

[2] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: MIT Press, 1992.

[3] R. Poli, W. W. B. Langdon, and N. F. McPhee, *Field Guide to Genetic Programming*. Lulu Enterprises Uk Limited, 2008.

[4] W. A. Tackett, "Genetic programming for feature discovery and image discrimination," in *Proc 5th Intl Conf on Genetic Algorithms*. San Francisco, CA: Morgan Kaufmann, 1993, pp. 303–311.

[5] R. Poli, "Genetic programming for feature detection and image segmentation," in *Evolutionary Computing*, ser. LNCS, T. Fogarty, Ed. Springer Berlin Heidelberg, 1996, vol. 1143, pp. 110–125.

[6] A. Song and V. Ciesielski, "Fast texture segmentation using genetic programming," in *Proc. CEC '03*, vol. 3. IEEE, 2003, pp. 2126–2133.

[7] J. F. Winkeler and B. Manjunath, "Genetic programming for object detection," in *Proc. Genetic Programming 1997*. Morgan Kaufmann, 1997, pp. 330–335.

[8] A. Song, T. Loveard, and V. Ciesielski, "Towards genetic programming for texture classification," in *AI 2001: Advances in Artificial Intelligence*, ser. LNCS, M. Stumptner, D. Corbett, and M. Brooks, Eds. Springer Berlin Heidelberg, 2001, vol. 2256, pp. 461–472.

[9] H. Al-Sahaf, M. Zhang, M. Johnston, and B. Verma, "Image descriptor: A genetic programming approach to multiclass texture classification," in *Evolutionary Computation (CEC), 2015 IEEE Congress on*. IEEE, 2015, pp. 2460–2467.

[10] W. Smart and M. Zhang, "Tracking object positions in real-time video using genetic programming," in *Proc. of Image and Vision Computing Intl Conf.*, 2004, pp. 113–118.

[11] B. Pinto and A. Song, "Motion detection in complex environments by genetic programming," in *Proc GECCO '09 Late Breaking Papers*. New York, NY: ACM, 2009, pp. 2125–2130.

[12] A. Cano, A. Zafra, and S. Ventura, "Speeding up the evaluation phase of GP classification algorithms on GPUs," *Soft Computing*, vol. 16, no. 2, pp. 187–202, 2012.

[13] W. Banzhaf, S. Harding, W. B. Langdon, and G. Wilson, "Accelerating genetic programming through graphics processing units." in *Genetic Programming Theory and Practice VI*. Springer, 2009, pp. 1–19.

[14] W. B. Langdon, "Graphics processing units and genetic programming: An overview," *Soft Comput.*, vol. 15, no. 8, pp. 1657–1669, Aug. 2011.

[15] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008.

[16] W. B. Langdon, "A many threaded CUDA interpreter for genetic programming," in *Proc. EuroGP'10*. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 146–158.

[17] D. Robilliard, V. Marion, and C. Fonlupt, "High performance genetic programming on gpu," in *Proc. BADS '09*. ACM, 2009, pp. 85–94.

[18] C. Sammut and G. I. Webb, *Encyclopedia of Machine Learning*, 1st ed. Springer, 2011.

[19] P. Brodatz, *Textures: a photographic album for artists and designers*. Dover New York, 1966, vol. 66.

[20] R. Picard, T. Kabir, and F. Liu, "Real-time recognition with the entire Brodatz texture database," in *Proc. IEEE Conf on Computer Vision and Pattern Recognition*, 1993, pp. 638–639.

[21] A. Song, "Texture classification: a genetic programming approach," Ph.D. dissertation, RMIT University, April 2003.

[22] A. Song and V. Ciesielski, "Texture analysis by genetic programming," in *Proc. CEC '04*, vol. 2. IEEE, 2004, pp. 2092–2099.

[23] D. B. Kirk and W. H. Wen-mei, *Programming massively parallel processors: a hands-on approach, 2e*. Morgan Kaufmann, 2012.

[24] F. Diaz-Pernas, M. Anton-Rodriquez, and J. Diez-Huguera, "Texture classification of the entire Brodatz database through an orientational-invariant neural architecture," in *Proc. IWINAC 2009*. Springer, 2009, pp. 294–303.

[25] A. Shahrokni, T. Drummond, and P. Fua, "Texture boundary detection for real-time tracking," in *Proc. ECCV 2004*. Springer, 2004, pp. 566–577.

[26] M. Maghoumi, "Real-time automatic object classification and tracking using genetic programming and NVIDIA CUDA," Master's thesis, Brock University, Dept of Computer Science, 2014.

[27] A. Song and D. Fang, "Robust method of detecting moving objects in videos evolved by genetic programming," in *Proc. GECCO '08*. New York, NY: ACM, 2008, pp. 1649–1656.

[28] R. Poli, "Genetic programming for image analysis," in *Proc. of the First Annual Conference on Genetic Programming*. MIT Press, 1996, pp. 363–368.

[29] D. Howard, S. C. Roberts, and R. Brankin, "Target detection in SAR imagery by genetic programming," *Adv. Eng. Softw.*, vol. 30, no. 5, pp. 303–311, May 1999.

[30] N. Harvey, S. Perkins, S. Brumby, J. Theiler, R. Porter, A. Cody Young, A. Varghese, J. Szymanski, and J. Bloch, "Finding golf courses: The ultra high tech approach," in *Real-World Applications of Evolutionary Computing*, ser. LNCS, S. Cagnoni, Ed. Springer Berlin Heidelberg, 2000, vol. 1803, pp. 54–64.

[31] M. Zhang and W. Smart, "Multiclass object classification using genetic programming," in *Proc. EvoWorkshops 2004*. Springer, 2004, pp. 369–378.

[32] K. Krawiec and B. Bhanu, "Visual learning by evolutionary and coevolutionary feature synthesis," *IEEE Trans on Evolutionary Computation*, vol. 11, no. 5, pp. 635–650, October 2007.

[33] M. Zhang, U. Bhowan, and B. Ny, "Genetic programming for object detection: A two-phase approach with an improved fitness function," *Electronic Letters on Computer Vision and Image Analysis*, vol. 6, no. 1, pp. 27–43, 2007.

[34] B. Ross, A. Gualtieri, F. Fueten, and P. Budkewitsch, "Hyperspectral image analysis using genetic programming," *Applied Soft Computing*, vol. 5, no. 2, pp. 147–156, 2005.

[35] T. Kowaliw, W. Banzhaf, N. Kharma, and S. Harding, "Evolving novel image features using genetic programming-based image transforms," in *Proc. CEC '09*. IEEE, May 2009, pp. 2502–2507.

[36] H. Al-Sahaf, A. Song, K. Neshatian, and M. Zhang, "Extracting image features for classification by two-tier genetic programming," in *Proc. CEC 2012*. IEEE, 2012, pp. 1–8.

[37] T. Ojala, M. Pietikäinen, and D. Harwood, "A comparative study of texture measures with classification based on featured distributions," *Pattern recognition*, vol. 29, no. 1, pp. 51–59, 1996.

[38] J. Leitner, S. Harding, M. Frank, A. Forster, and J. Schmidhuber, "icVision: A modular vision system for cognitive robotics research," in *Proc. 5th CogSys*, Feb 2012.

[39] J. Miller and P. Thomson, "Cartesian genetic programming," in *Proc. EuroGP 2000*, ser. LNCS, R. Poli, W. Banzhaf, W. Langdon, J. Miller, P. Nordin, and T. Fogarty, Eds., vol. 1802. Springer Berlin Heidelberg, 2000, pp. 121–132.

[40] S. Harding, J. Leitner, and J. Schmidhuber, "Cartesian genetic programming for image processing," in *Genetic Programming Theory and Practice X*, R. Riolo, E. Vladislavleva, M. D. Ritchie, and J. H. Moore, Eds. Springer, 2013, pp. 31–44.

[41] V. Kumar, *Introduction to Parallel Computing*, 2nd ed. Boston, MA: Addison-Wesley Longman, 2002.

[42] J. Meyer-Spradow and J. Loviscach, "Evolutionary design of BRDFs," in *Eurographics 2003 Short Paper Proceedings*, M. Chover, H. Hagen, and D. Tost, Eds., 2003, pp. 301–306.

[43] S. Harding and W. Banzhaf, "Fast genetic programming on GPUs," in *Proc. EuroGP'07*. Springer-Verlag, 2007, pp. 90–101.

[44] W. B. Langdon and W. Banzhaf, "A SIMD interpreter for genetic programming on GPU graphics cards," in *Proc. EuroGP'08*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 73–85.

[45] G. Wilson and W. Banzhaf, "Linear genetic programming GPGPU on Microsoft's Xbox 360," in *Proc. CEC 2008*. IEEE, 2008, pp. 378–385.

[46] P. Comte, "Design & implementation of parallel linear GP for the IBM Cell Processor," in *Proc. GECCO '09*. ACM, 2009, pp. 1–8.

[47] D. A. Augusto and H. J. Barbosa, "Accelerated parallel genetic programming tree evaluation with OpenCL," *Journal of Parallel and Distributed Computing*, vol. 73, no. 1, pp. 86–100, 2013.

[48] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *IEEE Des. Test*, vol. 12, no. 3, pp. 66–73, May 2010.

[49] S. Harding and W. Banzhaf, "Implementing Cartesian genetic programming classifiers on graphics processing units using gpu.net," in *Proc. GECCO '11*. New York, NY: ACM, 2011, pp. 463–470.

[50] D. Robilliard, V. Marion-Poty, and C. Fonlupt, "Population parallel gp on the G80 GPU," in *Proc. EuroGP'08*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 98–109.

[51] M. Yanagiya, "Efficient genetic programming based on binary decision diagrams," in *Proc. CEC '95*, vol. 1. IEEE, 1995, pp. 234–246.

[52] S. Harding, "Evolution of image filters on graphics processor units using Cartesian genetic programming," in *Proc. CEC '08*, J. Wang, Ed. Hong Kong: IEEE, 1-6 Jun. 2008, pp. 1921–1928.

[53] S. Harding and W. Banzhaf, "Genetic programming on GPUs for image processing," *International Journal of High Performance Systems Architecture*, vol. 1, no. 4, pp. 231 – 240, 2008.

[54] P. Nordin and W. Banzhaf, "Real time control of a Khepera robot using genetic programming," *Control and Cybernetics*, vol. 26, pp. 533–562, 1997.

[55] M. Ebner, "An adaptive on-line evolutionary visual system," in *Proc. SASOW '08*. Washington, DC: IEEE, 2008, pp. 84–89.

[56] ——, "A real-time evolutionary object recognition system," in *Proc. EuroGP '09*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 268–279.

[57] ——, "Towards automated learning of object detectors," in *Proc. EvoApplicatons'10*. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 231–240.

[58] B. Kaufmann, J. Louchet, and E. Lutton, "Hand posture recognition using real-time artificial evolution," in *Proc. EvoApplications 2010*. Springer, 2010, pp. 251–260.

[59] A. Boumaza and J. Louchet, "Dynamic flies: Using real-time Parisian evolution in robotics," in *Proc. EvoWorkshops 2001*. Springer, 2001, pp. 288–297.

[60] M. Maghoumi and B. J. Ross, "A comparison of genetic programming feature extraction languages for image classification," in *Proc. SSCI*. Orlando, FL: IEEE, December 2014, pp. 1–8.

[61] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[62] S. Luke, "ECJ: A Java-based evolutionary computation research system," 2015, version 23, Accessed: 2015-07-28. [Online]. Available: http://cs.gmu.edu/~eclab/projects/ecj/

[63] M. Hutter, "JCuda: Java bindings for CUDA," may 2014, accessed: 2014-05-30. [Online]. Available: http://www.jcuda.org/

[64] D. Montana, "Strongly typed genetic programming," *Evolutionary Computation*, vol. 3, no. 2, pp. 199–230, 1995.

[65] W. Langdon, "Large-scale bioinformatics data mining with parallel genetic programming on graphics processing units," in *Massively Parallel Evolutionary Computation on GPGPUs*, ser. Natural Computing Series, S. Tsutsui and P. Collet, Eds. Springer, 2013, pp. 311–347.

[66] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, "Real-time learning capability of neural networks," *IEEE Trans on Neural Networks*, vol. 17, no. 4, pp. 863–878, July 2006.

[67] M. Tarkov and S. Dubynin, "Real-time object tracking by CUDA-accelerated neural network," *Journal of Computer Sciences and Applications*, vol. 1, no. 1, pp. 1–4, 2013.

[68] S.-S. Liu and M. Jernigan, "Texture analysis and discrimination in additive noise," *Computer Vision, Graphics, and Image Processing*, vol. 49, pp. 52–67, 1990.