Real-Time Automatic Object Classification and Tracking using Genetic Programming and NVIDIA® CUDA™

Mehran Maghoumi

Department of Computer Science

Submitted in partial fulfillment
of the requirements for the degree of

Master of Science

Faculty of Mathematics and Science, Brock University
St. Catharines, Ontario

Abstract

Genetic Programming (GP) is a widely used methodology for solving various computational problems. GP's problem solving ability is usually hindered by its long execution time for large and complex problems. In this thesis, GP is applied toward real-time computer vision. In particular, object classification and tracking using a parallel GP system is discussed. First, a study of suitable GP languages for object classification is presented. To this end two main GP approaches for visual pattern classification were studied. These approaches include the block-classifiers and the pixel-classifiers. According to the experiments, the pixel-classifiers were generally more accurate and performed better. Next, using the results of these experiments, a suitable language was selected for the implementation of the real-time tracking system. The real-time system was implemented using NVIDIA CUDA. Synthetic video data was used in the experiments. The goal of the experiments was to evolve a unique classifier for each texture pattern that was present in the video. The experiments revealed that the system was capable of correctly classifying and tracking the textures in the video. Furthermore, the performance of the system was on-par with real-time requirements.

Acknowledgements

I would like to thank the following people for their support and encouragement during my research. Without their support, this thesis would have been impossible:

- Brian Ross for all his support, inspiration and understanding during the difficult days and his superb supervision
- My parents, Esmaeil and Fatemeh and my sisters, Maryam, Mojgan and Mahshad for being there for me at all times and providing me with the resources to develop my talents
- Marco Hutter for developing JCuda and providing quick technical support
- NVIDIA for their magnificent CUDA framework

Declaration

Chapter 4 of this thesis is based on an earlier research paper, *"Feature Extraction Languages and Visual Pattern Recognition"* by Mehran Maghoumi and Brian J. Ross (technical report # CS-14-03 [50]). The research in that paper was done by Mehran Maghoumi as a part of the M.Sc. thesis research in the Department of Computer Science at Brock University.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Evolutionary computation is an artificial intelligence problem solving paradigm inspired by the Darwinian evolution [6]. Genetic programming (GP) [41, 66] is a subfield of evolutionary computation. Here, computer programs are evolved and refined in order to solve problems of interest. Computer vision is a field which involves methods for acquiring, processing and analyzing visual information in computers [82].

Computational intelligence methods have been widely used for various computer vision problems such as target and object detection [32, 31, 83], face detection [88], mineral identification [72] and medical image analysis [65]. GP has also been used in various computer vision classification [19] problems such as texture classification [81, 77] and image and texture segmentation [78, 64, 79].

The advent of modern hardware and processors has enabled programmers to develop programs that benefit from the parallel processing capabilities that exist in such hardware. Using parallel programming frameworks a program can be executed hundreds of times faster than normal, a task which was only possible using powerful computing clusters. One such framework is NVIDIA CUDA [59]. NVIDIA CUDA allows programs to run on the highly-parallel graphics processing units (GPUs).

Evolutionary computation algorithms suffer from long execution times. Therefore, many attempts have been made to improve the runtime performance of these algorithms using parallel programming frameworks. GP has been successfully implemented using NVIDIA CUDA [45, 66, 70] and speed-ups of up to $\times 8$ were achieved.

The focus of this research is real-time object tracking using GP. Object tracking is a task within computer vision which involves detection of an object of interest and following its movements from frame to frame in a video [90]. Object tracking has recently gained much interest from researchers in computer vision field and it has numerous real-world applications. Automated surveillance, video indexing, traffic monitoring and human-computer interaction are among its applications to name a few. Object tracking can be performed either separately or jointly [90]. In the former method, the object is detected per-frame and the tracker is responsible for object correspondence in the subsequent frames. In the latter case, object location and region is iteratively updated using the information of the previous frame. Usually, object tracking is applicable to real-time scenarios. In the case of automated surveillance for instance, it is desirable to develop an object tracker that can work with a stream of video data coming from a surveillance camera. Hence, one of the considerations of any object tracking method must be its adaptability to real-time scenarios.

Computational intelligence methods have been widely used for object tracking and image or video analysis [88, 62, 34]. Most of the trackers that are developed using these methods are offline and supervised [73]; i.e. they are first trained using a predefined set of training data and they are explicitly told the expected output for each data instance in the training set. Afterwards, their resulting output is used to analyze the real-time data feed.

The aim of this thesis is to investigate a method for object tracking and classification using real-time GP. Unlike most work in the literature, our goal is to carry out object tracking in an online and automatic manner.

1.1 Goals and Motivations

Our goal is to investigate the object classification and tracking problem using GP. We have developed a system which is able to classify and track objects in real-time in an online manner. The system continuously monitors a stream of video data. The video data in question is a synthetic OpenGL generated animation. For each frame of the animation, the segments that exist in that particular frame as well as the background pattern of that frame are known. Using the texture of each segment, the GP system evolves a classifier that is tailored specifically to that segment. This texture classifier can discern the object from other objects currently in the scene. Using the data stream, the system continuously and dynamically evolves and improves these classifiers.

Since the proposed method is to be used in a real-time environment, the NVIDIA CUDA platform will be used in order to accelerate the system and achieve real-time performance. Undoubtedly, fitness evaluation and image processing will be the main bottlenecks of the system. Hence any effort for accelerating the system must primarily focus on accelerating these parts. Our main effort involves accelerating the evolution of texture classifiers.

One of the motivations for this research is to examine the suitability of a GP approach for online object classification and tracking. We are interested in evolving object trackers in real-time rather than pre-training systems and applying them in a real-time environment. Another motivation is to develop a system that works without user intervention. It is interesting to see a system that can automatically learn without supervision and can correct its behavior and adapt to ever-changing real-time situations. To the best of our knowledge, this is the first attempt in the literature that CUDA is used for creating an online GP system.

1.2 Thesis Structure

The thesis is structured as follows. Chapter 2 reviews the background information that are pertinent to GP, the computer vision problem. It also discusses the implementation considerations for using NVIDIA CUDA. Chapter 3 provides an overview of related research in the literature. Chapter 4 investigates various classification languages that have been used in the literature and provides a comparative study of these languages. Chapter 5 builds upon the result of Chapter 4 and uses the results of the comparative study to discuss the implementation of the real-time system. The experiments that were conducted as well as the results that were obtained are also presented in Chapter 5. Finally, Chapter 6 summarizes the results of our work and provides insight about possible research directions for the future.

Chapter 2

Background

2.1 Genetic Programming

Genetic Programming (GP) is an evolutionary algorithm-based methodology that develops computer programs by means of a simulation of the Darwinian evolution [41, 66]. It is a specialization of genetic algorithms (GA) that uses computer programs as chromosomes. The chromosomes in GP are sometimes referred to as *individuals*. A set of individuals called a population is maintained. These individuals are then selected for recombination or alternation to create a new population. The evolution of a population over time is called a generation. After a termination criterion is satisfied, the evolution process is terminated and the best individual (or a group of the best individuals) is selected as the final result of the system. Some terminologies that are frequently used in the literature is discussed in the sections that follow.

2.1.1 Representation

Instead of using actual lines of code, GP uses syntax trees to represent chromosomes. The nodes of these trees are either *functions* or *terminals*. An example of such tree would look like `MUL(ADD(a, b), c)`. This tree contains 5 nodes. First, the *a* and *b* terminals are added together and their result is multiplied by the terminal *c*. `MUL` and `ADD` are called functions because they are doing and operations on terminals and the value of their sub-tree depends on the value of the terminals in the sub-tree. *a*, *b* and *c* are dubbed terminals because they do not have any child nodes. A size limit can also be imposed on the trees to prevent working with unnecessary large trees. The terminal set and the function set together comprise the language of the GP system. The GP language needs to satisfy two requirements, namely closure and sufficiency [66]. Closure means that the function set elements must be able to work on the results that other functions or terminals create. Sufficiency implies that suitable solutions could be expressed for the problem at hand using the language elements.

2.1.2 Initialization

In GP, an initial population of random candidate solutions is first created [41]. The way this initial population is created is called *initialization*. Two main initialization methods exist and other methods use a variation of these two methods: the Full method and the Grow method. Both of these methods respect the tree size limit.

In the Full method, the nodes are randomly picked form the function set until the tree size limit is reached. At that depth, only the terminals are picked randomly from the terminal set. A tree that is generated using the Full method has the maximum number of nodes and all of the leaves will be at the same depth.

In the Grow method, nodes are selected randomly from both the terminal set and the function set and this selection continues until the size limit is reached. After that, the nodes are only selected from the terminal set. This method results in trees with more variety and different shapes and number of nodes.

A popular initialization method that is based on the Full and the Grow method is called *ramped half-and-half*: half of the population is created using the Full method and the other half of the population is created using the Grow method. Furthermore, some depth limits are also applied to the trees to ensure variety in the initial population.

2.1.3 Selection

GP is an evolutionary computation method. Therefore, a method for choosing individuals based on their fitness is used. Individuals are chosen using a fitness-based selection method. Among different methods, *tournament selection* is a popular method. Here, a set of k individuals is selected randomly from the population. The individual with the best fitness score in this set is then selected for evolution.

2.1.4 Reproduction Operations

In GP the individuals are evolved by a simulation of Darwinian evolution. The operations that are performed on individuals in an attempt to evolve them are called *Genetic Operations*. Individuals are selected and are either recombined together or altered in a specific way. The new individuals that are created are then moved to a new population forming a new *generation* of individuals. In GP two genetic operations are used.

Crossover

Crossover is used to recombine existing individuals and create offspring from them. In GP the recombination operator is called *crossover*. In crossover, two individuals are selected using the selection operator. Random nodes or an entire sub-tree of those individuals are swapped thus new individuals are created. Crossover is the main reproduction operation in GP and permits inheritance of traits from parents to offspring.

Mutation

Another way to create an individual based on an existing one is through mutation. In mutation, an individual is selected and then altered by selecting a random node in the tree and replacing it by a random node or a randomly created sub-tree.

2.1.5 Fitness Evaluation

Since GP is an evolutionary algorithm, a means for measuring the fitness of an individual is required in order to compare individuals and select the best among them. Fitness determines how suitable a candidate solution to the problem at hand is. The fitness of an individual in GP is determined by evaluating the expression tree of the individual and assigning a value to it based on some criteria. Over 85% of the evolution of a GP system is spent on fitness evaluation [9], therefore enhancing this process can effectively reduce the execution time of a GP system. Individuals that have better fitness rating can longer ”survive” and have a higher chance of being selected and moved to the next generation.

2.1.6 ECJ

ECJ [3] is an evolutionary computation research system developed in Java. It has the implementation of many evolutionary algorithms such as genetic algorithms, particle swarm optimization, and many of GP variants. It is open source, easily extensible, portable and also has very detailed documentation. For these reasons, we will use ECJ for our implementations.

2.2 Computer Vision

Computer vision is a field which involves methods for acquiring, processing and analyzing visual information in computers [82]. Unlike computers, human visual system has no problem identifying subtle differences and color variations in an image. This ability results in a correct segmentation of an object from its background. Computer vision is an example of an *inverse* problem, meaning that we intend to recover some unknown information from an image with insufficient information to fully represent the solution. This fact makes computer vision a difficult problem.

Image processing is any form of processing an image using a computer [22]. There is no general consensus about the boundaries of computer vision and image processing [22]. Some believe that image processing is a discipline in which the input and output of the operation is an image. Gonzalez et al. [22] believe that this boundary is too restricting because even the simple task of calculating the average intensity of an image – which results in a number – is not considered image processing. Furthermore, some believe that the goal of computer vision is to use computers to emulate human vision in a way that the computer is able to act based on visual perception. This view can also be restrictive as this whole process can be an incorporation of several sub-fields of AI. Based on these explanations, there is no fine line to separate image processing from computer vision.

Today, computer vision is being used in some areas of real-world problems. These areas include, but not limited to [82]:

- **Optical character recognition (OCR):** analyzing and reading handwritten text
- **Machine inspection:** inspection of parts using machines for quality insurance purposes
- **Medical imaging:** performing studies of people’s organs

- **Automotive safety:** detection of obstacles on the street, or fully automated driving
- **Fingerprint recognition:** for personnel authentication or forensic analysis

2.2.1 Object Tracking and Classification

Object tracking is the task of detecting an object of interest in a video feed and following its movements in the subsequent frames of the video [90]. Object tracking is usually performed in the context of more high-level problems known as video analysis. The three key steps in the video analysis are object detection, tracking the detected objects in the consequent frames, and analysis of the objects to study their behavior [90]. Object tracker can optionally provide additional information about the object of interest including its orientation, size or shape information. Due to the environment and the object of interest, object tracking is a challenging problem. Therefore, assumptions are made to simplify object tracking. One common assumption in the field of object tracking is that the objects of interest move smoothly without sudden changes in their trajectory [90]. Prior knowledge about the objects of interest can further simplify the task of object tracking. Object tracking has many applications [90] including surveillance, video indexing, human-computer interaction and etc.

Object tracking can be performed either separately or jointly [90]. In the former method, the object is detected per-frame and the tracker is responsible for object correspondence in the subsequent frames. In the latter case, object location and region is iteratively updated using the information of the previous frame.

For object tracking, various features of the objects of interest may be used. These features include color, textures or edges [90]. These features may be used for the detection of the objects of interest. Object detection can be done using various methods including background subtraction, image segmentation, supervised learning methods or object classification [90]. Object classification involves defining classes of objects of interest (e.g. wall, human, car etc.) and assigning a label to an object in an image based on the class that it belongs to [92]. It also involves determining whether an instance of such objects exists in an image (or a video frame) [92].

2.3 Machine Learning

Machine learning is a branch of artificial intelligence that studies the ways with which the computers can be made to modify or adapt their behavior in certain conditions so that their chosen actions more accurately resemble the pertinent action for that particular situation [51].

One model of machine learning is offline [74] (or batch) learning. In this model, the system accepts as input a set of observations or data. The internal model of the solution generated by the machine learning system is created to present the existing relationships between the inputs to generate the correct output. After the training phase is finished, this model does not perform any further learning. Conversely, in online learning the model has to make predictions about a sequence of inputs and is either rewarded or punished. The learner is only provided a description of instances. The goal of this learning model is to maximize the acquired reward. It is clear that in these systems, the learning process continues as new instances are encountered.

2.4 Parallel Computing

Parallel computing is a form of computation in which multiple operations and calculations are done at the same time [43]. Usually, large problems can be divided into smaller tasks that have little or no interaction and can be carried out at the same time [4]. Parallel software development is usually more difficult than sequential software development mainly because new problems such as synchronization arise when multiple parts of software are being executed at the same time, and so specialized parallel architecture is required. Parallel computing has received much attention in recent years and can be utilized for many computational-hungry problems. There are three types of parallelism [20]:

- **Fine-grained:** If the tasks of an application have to communicate with each other many times, that application exhibits fine-grained parallelism. These problems often require careful examination and planning so as to eliminate the synchronization problems that may occur.
- **Coarse-grained:** If the tasks of an application do not have to communicate with each other many times, that application exhibits coarse-grained parallelism.
- **Embarrassing parallelism:** If the tasks of an application rarely or never communi-

cate with each other, that application exhibits embarrassing parallelism. It is generally easier to convert these applications to parallel ones.

GP is usually considered “embarrassingly parallel” [66] since there are many places in GP that parallel computing can be incorporated (e.g. genetic operations, fitness evaluation, etc.)

Although parallel computation has lots of benefits, not many programmers try to develop parallel programs because of inherent difficulties in coding that arises from different method of computational thinking that is required. Furthermore, state-of-the-art parallel hardware such as grids or clusters are not always readily available. However, the advent of modern Graphics Processing Units (GPU) has made parallelism affordable and attainable. These specialized processors are able to manipulate large amounts of data at the same time, thus exhibiting parallel behavior [39].

Initially, programmers tried to harness the power of these special processors by using OpenGL or DirectX API functions and converting the problem at hand to a 3D rendering problem [39]. This method - which is known as GPGPU[1] - had its flaws: the graphical APIs were tedious to use, the performance of the final program was less than expected and the developed solution was not scalable and flexible for other similar problems. At this point a programming framework was needed to aid the programmers in developing parallel programs that can be executed on the GPU.

2.4.1 CUDA

CUDA[2] was introduced by NVIDIA in 2007. This framework gives programmers access to the virtual instruction sets and memory of parallel processing units in an NVIDIA GPU. Instead of using graphical API instructions, a program written in C/C++ code is directed to a specialized hardware in the GPU and that hardware manages the execution of that program on the GPU [39].

The CUDA framework is actually an extension to the C programming language. It introduces a few additional keywords that signifies that certain portions of code are to be executed on the graphics card. This way, programmers are not required to learn a new language or deal with complex instructions and data structures that are commonly used in

[1]General-purpose Programming using a Graphics Processing Unit
[2]Compute Unified Device Architecture

graphical APIs. The compiler that is responsible for compiling CUDA code is NVCC[3]. When C/C++ code is given as the input code for this compiler, it first analyses the code and separates the conventional C/C++ code and CUDA C code. The regular C/C++ code is compiled using the system's primary C compiler (GCC etc.) but CUDA C portions are compiled using NVCC. Thus, the compiling process is transparent for programmers and creating parallel programs is no different that the regular ones.

CUDA Program Structure

Every CUDA program has one or more parts that are either executed on the *host* (CPU) or the *device* (GPU). Usually, the parts of the program that do not require parallel computation (such as preparation phases) or exhibit little data parallelism are executed on the host. Those parts of the program that require huge computation are executed on the device. The NVCC compiler is responsible for generating the host and device code.

The host code usually prepares necessary data and transfers those data to device memory. This code, can only call certain functions of the device code. These functions are called *kernels*. The kernel function is responsible for launching many number of threads to exploit data parallelism.

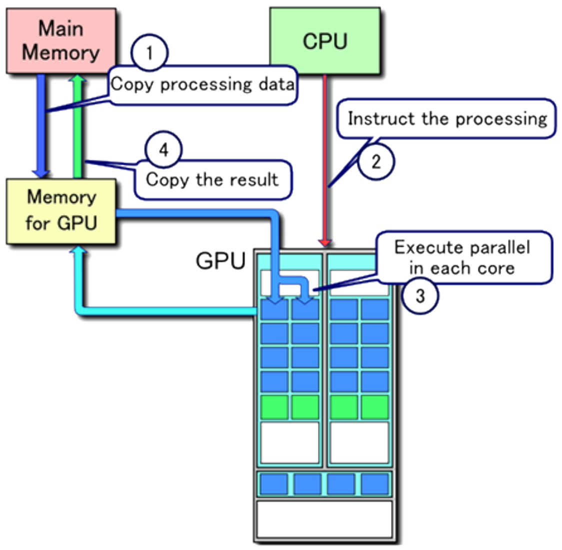
The kernel execution can be synchronous or asynchronous. If the execution is synchronous, when the kernel function has finished execution, the control is returned to the host code. Frequently, the device needs to return these results to the host. Thus, the host code is responsible for transferring this data from VRAM[4] to system's RAM. The figure below, represents a typical workflow of a CUDA program.

CUDA Threading Model

The kernel function can create millions of threads to solve a problem. In this regard, CUDA's programming model is SPMD (single-program multiple-data), i.e. the processing unit executes a single program on multiple parts of data. These processing units do not necessarily execute the same instruction at the same time. Note that this model is inherently different from the SIMD (single-instruction multiple-data) model. In SIMD, all processing units must execute the same instruction at the same time. This difference causes a problem formally known as *control flow divergence*. Specifically in CUDA, when a thread reaches

[3]NVIDIA C Compiler
[4]Video Ram

Figure 2.1: General workflow of a CUDA program (Illustration by Tosaka at en.wikipedia.org, redistributed under Creative Commons license)



a fork in its path (e.g. by executing an *if* statement), all of the threads that execute the *then* part of the conditional statement are processed first while the threads that need to execute the *else* part are suspended. After the first group of threads finish execution they are, in turn, suspended until the second group of threads finish their execution. If not thoroughly considered, thread divergence can cause a significant performance hit for a CUDA program [39].

In CUDA, the threads are organized in groups called *blocks*. A block can be thought of as an array that can have a maximum of 3 dimensions. Each element in this array is a thread. The dimensions of this array are defined by a kernel parameter called *block size*. These dimensions are accessible via *blockDim.x*, *blockDim.y* and *blockDim.z*[5].

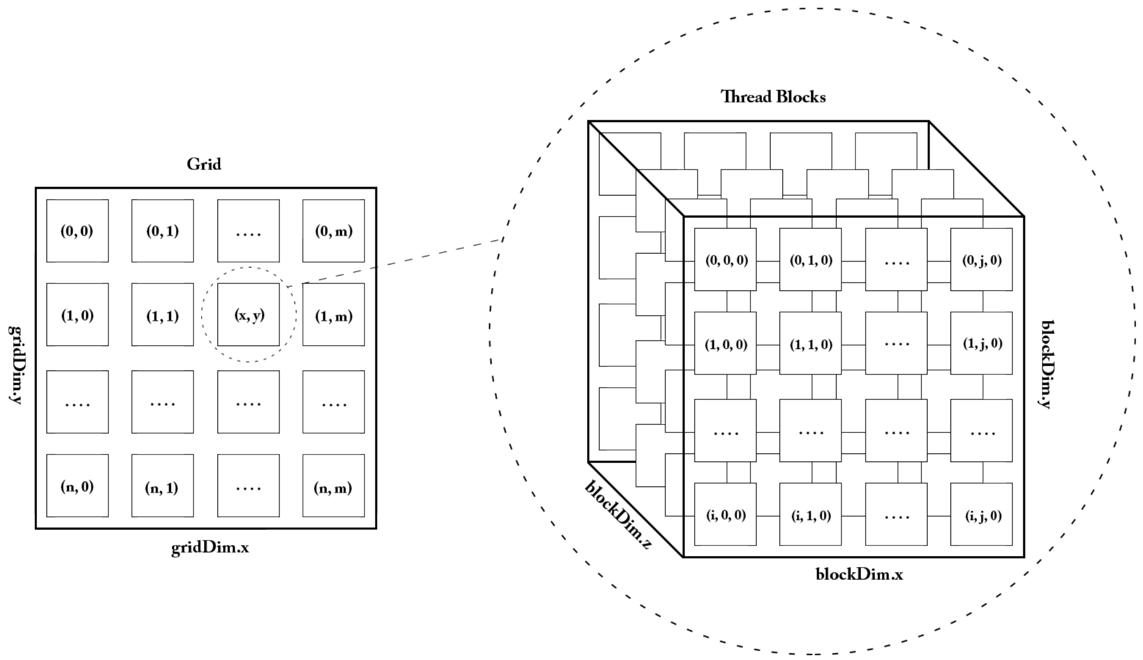
Every thread in a block is distinguished by its index (position). Thus, in a 3D block, each thread has *threadIdx.x*, *threadIdx.y* and *threadIdx.z* with which, the coordinates of the thread in a particular block is determined. The threads in a block share execution resources.

All of the thread blocks are organized in a group called *grid*. Like blocks, the grid can be thought of as an array but this array can have at most 2 dimensions. The dimensions of a grid is also determined by a kernel parameter called *grid size*. These dimensions are accessible via *gridDim.x* and *gridDim.y* values during runtime. Each element of this array is a block. Thus, the position of each block in the grid is determined by at most two

[5]These values are C constructs and are accessible in device code

indices: *blockIdx.x* and *blockIdx.y*. A graphical representation of this thread organization is as depicted in Figure 2.4.1.

Figure 2.2: CUDA thread organization



Before the kernel function is called, the dimensions of thread blocks and the grid needs to be specified. In current generation of NVIDIA graphics cards, each block can have a maximum of 1024 threads and the grid can have 65,536 blocks in total. These dimensions cannot be changed unless the execution of the kernel is finished and the kernel is relaunched. It is also worth mentioning that the number of threads in a block is the same across all blocks in the grid.

During execution time, threads are assigned to execution resources on a block-by-block basis [39]. In CUDA cards, execution resources are organized into SMs[6]. Depending on the specific hardware, multiple blocks are assigned to a single SM for execution. Once a block has been assigned to a SM, it is then divided into groups of 32[7] threads called *warps*. The threads in a warp are then executed simultaneously.

An important thing to note here is that a warp will always have exactly 32 threads. If the number of threads in a block is less than 32 the warp is still created using 32 threads but

[6]streaming multiprocessors

[7]the number of threads in a warp is implementation specific, however at the time of this writing, the warp size in all NVIDIA cards is 32

the extraneous threads are kept inactive. Note that inactive threads still occupy execution resources, hence as a performance consideration creating blocks with less than 32 threads should be avoided if possible.

CUDA Memories

Several different kinds of memory exist in CUDA. These memories have different scopes, lifetimes and latency. The table below summarizes different CUDA memories.

Table 2.1: CUDA memories

Memory	**Scope**	**Speed**	**Access Level**
Global	Grid	Slow	Host/Device
Shared	Block	Fast	Device
Constant	Grid	Fast	Host/Device
Register	Thread	Fast	Device
Local	Thread	Slow	Device

NVIDIA graphics cards do not usually have on-board cache, thus the programmer is responsible for caching the data that is accessed the most throughout the run in order to increase the overall performance of the system. The global memory, is slow but very large (typically 1GB or more in modern NVIDIA GPUs). Kernel's input and output data is usually stored in this memory. However, one should note that the latency of this memory is very high and overusing this memory in kernel code can severely degrade the performance of the system.

The shared memory on the other hand, is extremely fast but is limited to 48KB in most recent graphics cards. It is usually used as a cache for the global memory and programmers manually transfer most accessed data from the global memory to this one. The constant memory, is another fast memory that can be accessed from both the host and the device, however since it is a cached memory, if it is accessed in irregular patterns then its performance will drop significantly. Note that the constant memory in here is different that a constant defined in C using `const` keyword. In C, constants defined using `const` should have a definite value before the program is compiled. In CUDA, variables defined in constant memory need not have a value during compilation. Their value could be assigned in runtime but before a kernel call.

The variables in each thread are usually kept in registers. This memory is extremely fast but very limited and cannot be indexed. Therefore, structures that require large amount of memory (such as arrays) are stored in the local memory. The local memory is actually the global memory (VRAM) but access to it is limited to the thread that created it. Correct utilization of CUDA memories is an essential performance consideration.

Programming in CUDA

In this section we intend to provide simple guidelines and examples of writing programs in CUDA. We first begin by explaining what compute capability is and what the extensions that CUDA has added to the conventional C programming language are. Then we talk about some of the important functions that CUDA has provided for synchronization. Finally, we provide examples about the method of thinking that CUDA requires.

Compute Capability

Each NVIDIA graphics card has a specific compatibility version named compute capability. Compute capability defines what types of operations are supported on the card. Older cards only support compute capability v1. In these cards, many of the operations (such as atomics which will be explained later) are not supported and some of the thread capabilities are minimal. Generally higher versions of these cards are easier to program and different utility functions exist for them.

Based on compute capability version, the NVCC compiler generates different codes. Thus, it is usually a good idea to set the compute capability version of NVCC to the desired version before compilation. For compatibility, one should keep the version as low as possible whereas for ease of programming, higher versions are preferred.

Extensions to C

CUDA programs are written in C programming language with a few extensions to the original language. Table 2.2 summarizes some of the most commonly used extensions that CUDA has added to conventional C.

Built-in Functions and Atomics

No host function can be called in CUDA kernels. Therefore, CUDA SDK provides access to simple mathematical functions in kernel codes. Some of most useful functions are

Table 2.2: CUDA's extensions to C

Extension	Usage	Example
`__global__`	Signify kernel function	`__global__ void add (int *a, int *b, int *c)`
`__shared__`	Signify a shared resource	`__shared__ int input[10];`
`__constant__`	Allocate a resource in constant memory	`__constant__ double indCounts = 5;`
`gridDim`	A struct to access the dimensions of the grid in runtime	`int dimGrid = gridDim.y;`
`blockDim`	A struct to access the dimensions of the current block in runtime	`int dimBlock = blockDim.z;`
`blockIdx`	A struct to access the index of the current block	`int bid = blockIdx.x`
`threadIdx`	A struct to access the index of the current thread	`int tid = threadIdx.z`

presented in Table 2.3.

Table 2.3: Useful CUDA built-in math functions

Function	Definition
sin()	Sine function
cos()	Cosine function
abs()	Absolute value function
expf()	Exponential function

CUDA provides several functions for thread synchronization. The most useful function is the barrier synchronization function: `__synchthreads()`. Every thread in a block that reaches this function during its execution is suspended until all of the threads in that block reach this function. After all of the threads have reached this function, the execution of all of those threads in continued.

It is clear that barrier synchronization might not be adequate for some synchronization scenarios. Suppose that before the execution of the kernel is finished, every thread created by the kernel needs to add a value to a single variable. Obviously there should exist a method to synchronize the accesses to this single variable otherwise the final value of the thread might not be reliable. CUDA provides the *atomic* functions for these cases.

An atomic function is a function that is executed sequentially by all threads in the kernel. If multiple threads want to call an atomic function, only one of them is able to execute it. Other threads will have to wait until the current call to that function is ceased. It should be noted that atomic functions will degrade the performance of the system as they make the execution model sequential. Hence, use of atomic functions must be minimized.

Based on compute capability, CUDA atomic functions can only function on integers or floats. In older cards, only integer atomics are available while in newer cards, floating point versions of these functions also exist. Table 2.4 summarizes some of the most useful atomic functions in CUDA.

Table 2.4: Useful CUDA atomic functions

Function	**Usage**
`atomicAdd`	Add a value to a memory location
`atomicSub`	Subtract a value from a memory location
`atomicMin`	Stores the minimum of the two provided values in a memory location
`atomicMax`	Stores the maximum of the two provided values in a memory location

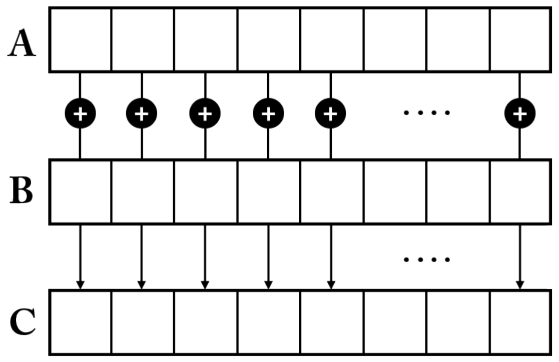
Method of Thinking

Although CUDA provides instruction sets similar to C programming language, it requires a different method of thinking for solving problems. CUDA must be thought of as a processor with many cores, each of which can solve a part of the problem. We demonstrate this method of thinking with an example[8].

Suppose that we want to solve a vector addition problem. Let us assume two vectors with 10 elements named A and B and an array with the final results named C. The goal of this problem is to add the values of each of the elements in the two input vectors and save the results in the third. In a conventional C program, one would write a *for* or a *while* loop to iterate over the elements in A and B, adding the elements in those two vectors together and saving it in the corresponding element in C. Figure 2.3 represents this process.

[8]This example was inspired by a similar example in [75]

Figure 2.3: Graphical representation of adding two vectors



If we were to solve this vector addition problem using CUDA, we would not need to use a loop to iterate over the elements of the vectors. Rather, we would assign each addition operation in Figure 2.3 to a single CUDA thread. We would use the thread's specific index to map each thread to a single vector element and launch a kernel with appropriate number of threads to do the operation. This is exactly what we mean by the method of thinking: loops should preferably be converted to a model which uses many threads for carrying out a task. Listing 2.1 shows a sample CUDA kernel to solve this problem. This kernel should be invoked using 10 or more threads to function properly.

Listing 2.1: Sample vector addition kernel in CUDA

```
__global__ void add(int *a, int *b, int *c) {
        int tid = threadIdx.x;
        if (tid > 10)
                return;
        c[tid] = a[tid] + b[tid];
}
```

2.4.2 JCuda

JCuda [33] is an open-source, cross-platform and freely available wrapper for CUDA in Java. A one-to-one mapping of all the functions in CUDA SDK is present in this library. Furthermore, other useful CUDA libraries (such as those for mathematical operations) are also available in JCuda. Because of its detailed documentation and active support, we have adopted this library for our purposes in this thesis. Furthermore, since both ECJ and JCuda are written in Java, interfacing the two libraries can be easily done.

A drawback of this library is that the kernel functions for CUDA programs still have to be written in CUDA C and these codes are compiled separately by NVCC. This makes debugging of the developed code somewhat difficult.

2.4.3 OpenCL

Similar to CUDA, OpenCL is another parallel programming framework. OpenCL was first introduced in 2008 by Apple Inc. [39]. The aim of OpenCL is to address significant limitations of heterogeneous parallel-computing systems. As mentioned before, CUDA in only limited to compatible NVIDIA cards. OpenCL, on the other hand, can be executed on any parallel hardware that supports OpenCL specifications. This hardware can be a CPU, a GPU or even a cluster of connected nodes. This makes programs developed using OpenCL much more portable than those developed using CUDA. However, several research have claimed that this portability comes with the cost of a minor performance penalty [14, 36].

Because OpenCL and CUDA share many similarities, a kernel function that is developed in either of the frameworks can be easily converted to the other [39]. Since OpenCL is relatively nascent, its resources are somewhat limited. Because of these reasons, we decided to use CUDA for the purposes of this thesis.

Chapter 3

Literature Review

This chapter reviews some of the previous works that are pertinent to our research. The application of the CUDA programming framework in the related research is also investigated. The CUDA programming model is proficient for problems that demonstrate huge amounts of data-parallelism. Today, many applications are comprised of two main parts: a sequential control logic component and an inherently parallel computation component. These computation components are usually data-parallel in nature [21] indicating that these programs can be accelerated using CUDA. Many applications have been accelerated using CUDA and the goal of this section is to provide a brief overview of such applications.

3.1 CUDA in Computational Intelligence

This section briefly presents some applications of CUDA in computational intelligence methods such as particle swarm optimization and neural networks.

CUDA in Particle Swarm Optimization

Particle swarm optimization (PSO) [38] is an optimization and problem solving paradigm inspired by the behavior of the birds flocking or fish schooling. PSO maintains a population of particles which are scattered in the problem search space. It then iteratively updates the position of particles in order to find suitable positions for particles and by extension, suitable solutions to the problem. Since PSO suffers from slow runtimes, researchers have implemented PSO systems suitable for parallel execution. This is due to PSO having many tasks that can be executed in a parallel manner. In [57] Mussi et al. proposed several models of parallelism for PSO.

One of the older methods is the island model parallelism in which several swarms of particles are maintained and processes separately in parallel. These swarms are connected to each other based on a predefined topology. After a certain criterion is met, best particles are moved (migrated) to other swarms with respect to the topology of the whole network. These particles are processed in their new swarm and then moved again until the termination criteria of the PSO algorithm is met. For CUDA implementation, each swarm is assigned to a single thread and different swarms are evaluated and processed in different threads. The worst pitfall of this method is that it does not map well to the data-parallel model that CUDA requires. If this method is implemented in CUDA, lots of computational resources of CUDA are wasted. Also, control flow divergence in this method is very high as the code that each thread executes, differs significantly from those of the other threads.

Another proposed parallel model for PSO is to assign each particle in the swarm to a thread. This way, a single thread is responsible for updating and evaluating an individual in the swarm. An immediate drawback of this method is that it limits the maximum number of particles that can exist in a swarm. This method requires synchronization and communication between all active CUDA threads which is currently impossible [2]. Only the threads that belong to the same thread block can communicate with each other via the shared memory.

The best model that was proposed for parallel implementation of PSO was to implement PSO using multiple kernels. Each portion of the PSO is performed using a separate kernel. For example, fitness evaluation is done using one kernel while velocity and position update is performed by another. This method requires a lot of access to global memory but the access latency can be mitigated by utilizing shared memory or flattening and caching parts of global memory.

CUDA in Artificial Neural Networks

Artificial neural networks (NN) [12] is a machine learning problem solving model inspired by the animals' central nervous system. The feed-forward neural networks, for example, consist of an input layer, a hidden layer and an output layer. Each layer is comprised of a defined number of nodes. The nodes in each layer are connected to the nodes in the next layer. Each connection between the nodes has a specified weight. The connection weights determines the model of the network as well as its capabilities. Training an NN, which involves determining the weight of the connections, is a time consuming process. Many attempts have been made to accelerate the training process of NN's using CUDA. There are many types of NN systems and each NN differs in topology and configuration from the other. As a result, there are various CUDA implementation of NN systems.

Jang et al. [35] used a Multi-layer perceptron (MLP) with 2 hidden layers. The calculations that were involved in their model contained lots of matrix multiplication which was accelerated on CUDA. Furthermore, sigmoid function calculations were also performed in parallel for all nodes in the each layers of the hidden layer.

Guzhva et al. [24] used CUDA to implement a back-propagation algorithm for training MPLs on the GPU. They trained the system for a geophysical problem. To test their system, 6720 MPLs were trained each of which had 1 hidden layer with 8 neurons, 1648 inputs and a single output. They were able to achieve speed-ups of up to 50$\times$ compared to a highly optimized CPU implementation.

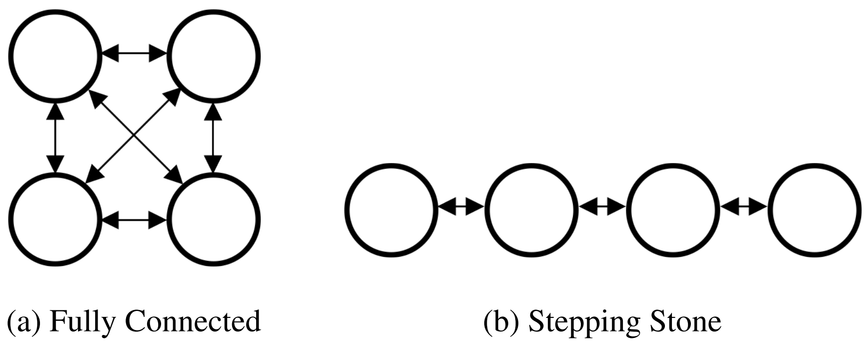
CUDA in Genetic Algorithms

GA demonstrates an embarrassingly parallel model. In GA, parallelism can be exploited in different ways [11]. The easiest model is the master-slave model which distributes fitness evaluation of chromosomes in GA among different processors (different threads in the case of CUDA). This kind of GA was implemented by Cantu-Paz [10]. Although this research was much older than the advent of CUDA, the authors were able to achieve reasonable amounts of speed-up using a cluster of connected nodes.

Another model of parallelism in GA is the island parallelism with multiple populations which is the focus of work in [68] by Pospichal et al. To simplify their implementations, the authors allocated each island to a single thread block. Each thread in that block is responsible for processing a single chromosome and shared memory was utilized to minimize the number of access to the global memory. The migration model that they occupied for their research was the asynchronous model; meaning that an island would not wait for other islands to finish their task before migrating its chromosomes. Furthermore, the topology of the islands was unidirectional; i.e. each island can only accept chromosomes from one neighbor. One notable point in their research is that they used CUDA for genetic operations as well. This way, not only they accelerated the evaluation of GA chromosomes, but also they were able to reduce the time for genetic operators.

Some research has also been done regarding the combination of several different parallel methods in GAs. These methods are called hierarchical GAs [11]. Zhang and He [91] used this type of parallelism. They successfully reduced the execution times more than any of their components alone. Moreover, they combined the master-slave parallelism with the island model. A notable fact about their research is the employment of two island models: the fully-connected model and the stepping-stone model. Figure 3.1 depicts these two models.

Figure 3.1: GA island models (circles denote islands)



In the fully-connected model, all islands can communicate with one another and can migrate their chromosomes to every other island. In the stepping-stone model, only adjacent islands can receive/send chromosomes to each other. In their implementation, every thread block in a grid is responsible for processing a single island and each thread in that block is responsible for a single chromosome. The blocks in a grid demonstrate a fully-connected island model behavior. Grids, however, demonstrate a stepping-stone model behavior. These grids are only connected to their adjacent grids and they only transfer their best chromosomes. Note that since all of the chromosomes are transferred to video memory at the beginning of the run, during evolution no data is transferred from video memory to RAM, resulting in reduction of congestion on system bus. As a result, full computational power of the graphics card can be utilized and maximum performance can be achieved.

CUDA in Genetic Programming

GP is “embarrassingly parallel” [46], and so it is very suitable for parallel programming. There have been many attempts in developing a parallel GP system and the resulting systems experienced significant amounts of speed-ups. Similar to GA, GP has several components that can be implemented in a parallel model. A computational study [9] measured the amount of time required by various GP phases. Table 3.1 presents the results of this study.

Table 3.1: Average Execution Time of GP Components

Phase	**Percentage**
Initialization	0.39
Initial population evaluation	8.57
Selection	0.01
Crossover	0.01
Mutation	0.03
Evaluation	85.32
Replacement	0.03
Control	5.64

As it can be seen in Table 3.1, 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 inevitably focus on improving the performance of the evaluation phase, i.e. the evaluation portion must be part of the device code and the rest of the parts are performed using the host code. Although these results are implementation specific, they provide a good insight of potential areas of interest in parallel implementation of GP.

The focus of this section is to review the literature regarding parallel GP. By parallel GP we refer to GP implementation on specialized parallel hardware and not distributed GP or island model parallelism. These kinds of parallelization are relatively straightforward and usually do not require specialized parallel hardware.

There are mainly two approaches to running GP on parallel hardware [45, 66, 70]:

1. Evaluating a single program using multiple fitness cases *(fitness-case parallel)*
2. Evaluating multiple programs in parallel *(population parallel)*

It is clear that the first approach reduces thread divergence because it maps closely to CUDA's SPMD model. Nonetheless, it is mostly suitable for systems that have large amounts of fitness cases. Harding and Banzhaf [26] have acknowledged this as a weakness of this approach. The authors state that not many GP problems deal with thousands of fitness cases and this creates a "gap" between what can be evaluated and what can be evolved. Moreover, it has been suggested that in order to solve real-world problems using GP, having large populations are more important than large data sets [42]. Thus, the fitness-case parallel scheme is not widely used in the literature. The population parallel approach, on the other hand, promotes thread divergence but is more applicable for systems with large population size and aids the development of systems with large population sizes.

Many attempts have been made to incorporate the use of GPUs for GP systems. According to [7], Meyer-Spradow *et al.* [52] were the first ones that attempted this. They evolved pixel shaders using short linear assembly language and used them for real-time interactive rendering. Although this effort did not directly accelerate the evolution of GP, their work is notable because of the real-time and interactive behavior of their GP system.

Harding and Banzhaf [26] were the first to run evolved programs on GPUs and accelerate the evaluation phase of GP. Since their attempt was before the advent of CUDA, they used Microsoft Accelerator [85] tool to compile and run GP individuals on the GPU. Furthermore, their approach involved evaluating a single individual on all fitness cases at the same time (fitness-case parallel).

Langdon and Banzhaf [46] implemented a SIMD interpreter for GP on GPUs. Contrary to Harding and Banzhaf's research, they employed the population-parallel scheme and treated each GP individual as a single program. They also converted the GP trees to postfix expressions. This eliminates the need for recursive calls and permits the evaluation of trees using an explicit stack. They implemented their system using the RapidMind parallel programming framework.

Another attempt to accelerate GP without the use of CUDA was done by Wilson and Banzhaf [87]. They developed a linear genetic programming system on Microsoft Xbox 360 video game console to solve a regression and a simple classification problem using the population parallel approach. Other than speed-up, their effort revealed that the highly integrated CPU and GPU of Xbox 360 has the potential of alleviating the decision of allocating specific functionalities to CPU or GPU.

Research conducted by Comte [13], is among other types of parallel GP systems that are not implemented in CUDA. There, Comte used a Sony PlayStation 3 video game console to evolve a population of 648 individuals on the console's CPU. The adopted GP model was the linear GP model and all evolutionary operations were implemented on the Cell CPU. It is worth mentioning that the parallel model the Cell CPU employs is SIMD, and thus careful considerations were required during the implementation to harness the full power of the aforementioned processor.

Among other parallel programming models, OpenCL has also received some attention from GP developers. Agusto *et al.* [5] used a population-parallel approach to evaluate all individuals in the population on an OpenCL compatible device. They employed prefix expression as a representation of individuals and they solved a simple regression problem and a classification problem.

Another recent and notable work regarding development of GP using OpenCL was done by Banzhaf and Harding [29]. They used GPU.NET as their development tool and they employed the Cartesian GP model. Banzhaf and Harding concluded that CGP is amenable to use with a GPU interpreter. Furthermore, they deduced that CGP is suitable for solving classification problems with GP.

One of the first efforts to incorporate CUDA in GP was made by Robilliard *et al.* [71]. Their research involved interfacing ECJ with CUDA to solve ECJ's example of symbolic regression problem. They used CUDA to evaluate a population of 1000 individuals in parallel. Their work is notable for combining the two GP parallelism approaches: the fitness-case parallel approach and the population parallel approach. They termed this approach *BlockGP*. Similar to previous works, they employed postfix expressions to represent GP individuals. In their implementation, they assigned each GP individual to a single CUDA block with 32 threads, hence invoking a kernel with 1000 blocks and 32 threads in each block. The threads in a block parse the postfix expression assigned to that block in parallel and evaluate it for one of the 32 fitness cases. This way they fill the pipelines of the GPU more effectively [71].

One important point revealed by Robilliard's et al. research was that when the population size is very large, CPU becomes very inefficient in breeding (i.e. combining and mutating) individuals. This inefficiency becomes so high that the breeding time would no longer be negligible compared to the evaluation time. To mitigate this problem, they suggested that the breeding phase be also implemented on GPU. Furthermore, they suggested the change of ECJ's tree representation. By default, ECJ uses a pointer-based representation which imposes high allocation and deallocation overhead in Java. The default tree representation also requires a translation phase. This phase translates a pointer-based tree to a postfix expression.

In a later work, Robilliard et al. [70] successfully changed ECJ's breeding module in way that it directly evolves linear postfix expressions. This implementation was based on guides and codes implemented by Langdon [44]. They showed that for large populations, the new representation increased the performance up to 7 times. Furthermore, they compared their BlockGP scheme with the conventional population-parallel scheme. They referred to this scheme as *ThreadGP*. In ThreadGP, each thread is responsible for evaluation of a single individuals. This way, the total number of threads in CUDA kernel is the population size. Although this implementation is more computationally extensive and will fill the GPU pipelines better, they predicted that its performance will be lower than BlockGP since it promotes divergence. Their experiments corroborated this prediction as ThreadGP was more suitable for experiments with small number of fitness cases. They concluded their research by optimizing memory accesses in their system. They observed speed-ups of up to 3 times when shared memory was utilized in their BlockGP scheme.

In 2010, Langdon [45] used a similar scheme to solve Boolean 20-multiplexor and 37-multiplexor with 137 billion fitness cases. He used SMCGP model to further utilize the bit-level parallelism of CUDA and achieved a tremendous throughput of 665 billion GP operations per second. Although it was estimated in [89] that the 20-multiplexor problem would take 4 years to solve using GP, Langdon with the aid of CUDA, was able to solve this problem in less than an hour. He was also able to solve 37-multiplexor problem in less than a day. Solving this specific problem using GP was never attempted before [45].

In an attempt to develop a distributed parallel GP system, Harding et al. [28] used a cluster of GPU equipped computers to compile and evaluate a population of GP individuals. This permitted efficient processing of large data sets (in the case of this research, 10 million rows with several hundred megabytes). They solved a classification problem for network intrusion and the model they employed for their GP system was the CGP model. Their host code was implemented in C# and CUDA.NET was used for C# and CUDA communications. The performance of their system was unexpectedly low at the rate of 86 individuals per second. They attributed this low throughput to the overhead of data transfers between cluster nodes and the fact that the GPU in each node was also shared with the operating system. Furthermore, they observed that the slowest node in the cluster greatly affected the performance of all other nodes.

3.2 Computer Vision

Much research has been conducted in the field of computer vision. The approach and methodology for various computer vision problems differ. We review some of the previous work that involve computational intelligence methods.

3.2.1 Computer Vision, PSO and NN

Owechko et al. [62] conducted research on classifier swarms. Their approach involved feature-based object classification with search mechanisms based on a variant of PSO called SNPSO[1]. Using their approach, they were able to detect multiple objects in their input image. Furthermore, their solution was very fast and robust.

Puranik et al. [69] used a combination of fuzzy logic with PSO to model the human perception of colors. The fuzzy set included three sets each of which corresponded to one of the colors used in the HSL[2] color space. The PSO was used to measure the appropriateness of each of the members of the fuzzy sets. Using this approach, they were able to segment an image and extract similar objects from it.

Engelbrecht et al. [61] used a PSO to perform pattern classification in images. In their research, they provided a PSO method for clustering. Then they addressed two other problems in pattern recognition: the color image quantization and spectral unmixing. These two problems were also tackled with two variants of PSO. Furthermore, they compared their PSO method with other popular methods for pattern classification and their results were generally better.

Face recognition is widely used and is a very practical problem. Intrator et al. [34] used a hybrid supervised/unsupervised neural network to tackle this problem. In their method, first the eyes and mouth were detected and then a spatial normalization of the image was performed. Finally the classification was done using their neural network.

Another widely used problem is fingerprint classification and recognition. Karu et al. [37] developed a classifier that was able to classify fingerprints into five categories [3]. Their neural network was able to classify 4000 images with an accuracy of 85.4 %. Furthermore, their classifier was independent of rotation, translation and small amounts of scale changes.

One of the applications of computer vision is in automotive safety. Escalera et al. [18] attempted to develop a sign recognition and analysis system using neural networks and genetic algorithm. They claimed that automotive systems with this capability can aid road maintenance and promote the development of intelligent autonomous vehicles. The detection phase in their research was performed using a genetic algorithm. A neural network was used for the classification of signs. They claimed that their method was extensible to use for other types of vision problems.

[1]Sequential Niching Particle Swarm Optimization

[2]Hue - Saturation - Lightness

[3]These categories include: arch, tented arch, left loop, right loop and whorl

3.2.2 Computer Vision and GP

Because of its flexibility, GP has been adapted for various computer vision problems. According to Poli [65], GP is particularly useful for image analysis. He suggested that GP is a useful approach for image analysis. In his research he regarded GP as a method to find optimal filters for various image analysis-related tasks. He outlined some important criteria that terminal sets, function sets and fitness evaluation methods should have for image analysis using GP.

According to Poli, the terminal set for a GP system used in this area should be small and must include a limited number of variables. This criterion makes the size of the search space amenable. These terminals must also be able to present the information included in the image at different scales. Furthermore, he states that the terminals must be easy to calculate and should not require huge amounts of computational resources. In order to be able to represent the whole image, these terminals could work on regions of the image; i.e. they should act like a function. They take a region of the image and return a single value representing some statistical property of that image. These functions are called *spatial functions*. Poli also mentions that in addition to closure and sufficiency, functions in the GP systems should also be efficient. Criteria mentioned for terminal sets must also apply for function sets. Poli suggests that the functions must not involve computationally intensive tasks.

One notable point about Poli's research is that he mentions the ”sensitivity/specificity dilemma” [67]. This dilemma states that no detection or segmentation algorithm can detect all points in the areas of interest (true positives) without incorrectly detecting points which belong to other areas (false positives) and missing some points in areas of interest (false negatives). In his research, Poli showed that even with a relatively small and simple function and terminal set, GP can demonstrate better performance than a complex neural network.

Song *et al.* [81] used GP for texture classification. In their research, they used raw pixel features in order to evolve texture classifiers. Moreover, they employed a block processing approach, in which rectangular blocks of the image are processed rather than individual pixels and a classification decision is made for all pixels in that block. Positive and negative examples are obtained using iterative sampling of the input images a predefined number of times. Each sample is a block of N$\times$N pixels which contains grey-scale pixel values and has a label. Using these grey-scale values, the GP system evolves a classifier. This classifier is then applied to the image in a sweeping window manner and classification decisions are made for the whole blocks.

Using the same approach, in [78] Song *et al.* used these texture classifiers for the texture segmentation problem. To this end, they evolved a classifier for each texture. Using a sweeping window, they executed all of these classifiers on an image that contained these textures. They then recorded the number of times each pixel in a block was classified positively by each classifier. They then labeled each pixel based on voting. If the majority of the output for a pixel was texture K, then that pixel is labeled as texture K. Finally, all pixels are colored based on their classification label.

In addition to using raw pixel values for texture classification, they later examined texture classifiers that used standard image processing features [77]. These classifiers performed better than the classifiers that only used raw pixel values. However, calculating some of the image processing features require computational power and slows down the execution times.

GP has also been used for mineral identification or terrain classification. Ross *et al.* [72] used genetic programming for evolving mineral identification functions for hyperspectral images. Each identifier was responsible for identifying one of the three mineral specimens. Leitner *et al.* [47] used Cartesian GP (CGP) [53] to automatically classify rocks, sands and etc. on Mars. CGP uses a linear chromosome representation. In CGP, each chromosome contains integers which are divided into groups of three of four each of which defines a position in a 2D array. A single integer in each group defines the location of the operator in the 2D array while other integers in that group define the location of the operands in the 2D array .CGP representation closely resembles graph-like representation in the sense that the output of each operator may be used more than once. Using a rich function set consisting of mathematical and prevalent image processing operators, they performed classification and segmentation in a pixel-based manner. These features allowed the incorporation of domain knowledge into the GP language.

Poli [64] used GP for feature detection and image segmentation. His approach was based on the idea that low-level image analysis tasks - such as enhancement or segmentation - can be thought of as image filtering problems which GP can solve by discovering the optimal filters. His approach was mainly the same as his work in [65]. He used an identical function and terminal set to his previous work and tested the system on X-ray images for detecting blood vessels. His system outperformed a neural network for this problem.

Many image processing tasks have been successfully performed using GP. Kowaliw *et al.* [40] used CGP to evolve image transforms. These transforms convert the images to a more classifiable form and can be used as features for other image processing tasks. They applied their evolved transforms to medical images for the task of classification. Harding *et al.* [30] demonstrated how CGP can be used for image processing. Using the OpenCV image processing library and a set of prevalent image processing operators (such as Gaussian blur, dilate, erode and Canny edge detection) they showed that robust and human-readable GP programs can be evolved. Moreover, these image operations allow incorporating domain knowledge into the GP language. They applied their method on problems such as noise reduction, medical imaging and robotics with satisfactory performance.

Among researchers, Tackett [84] was the first one to use GP for discriminating target and non-target objects in a series of images. Contrary to the researchers before him, he used actual images and not bitmaps. He used two GP systems for his work that shared a common function set. The first system used 7 primitive features of an image that were based on the intensities of two *"image windows"*.

In his work, he divided each image to mosaics of 62×62 pixel overlapping regions. He termed these regions the *large windows*. Each of these regions are also divided into 5×5 overlapping *small windows*. The first set of terminals that he used for these windows were based on intensity values of these regions. For his second system, he performed a 3:1 decimation filtering on the large windows. Then he extracted 20 moment and intensity based features from the resulting image. These features were used as terminal sets of his second system. His fitness function was based on the combination of the probability of incorrect classification and the probability of correct classification. This way, he was able to counteract a result that would classify everything as target. He compared his results against a back-propagation neural network. Both of the systems showed satisfying performances.

Howard et al. [32] used GP for detection of ships in ocean in synthetic aperture radar (SAR) imagery. For their terminals, they used spatial functions similar to those described by Poli and their GP system was steady-state rather than generational. Since the size of the images were large, the GP system was trained using only a portion of the pixels in the image. In order to choose the training pixels they used a strategy with three stages. In the first stage, a subset containing at most 1000 ocean pixels where selected at random. In this stage, GP evolves a detector that can detect target pixels in this subset. In the second stage, the best detector of stage 1 is applied to the entire image. This causes some false positive detection. These pixels are store for use in the final stage. In the final stage, a new GP run is performed. This run is responsible for discriminating between false positives and actual targets in the pixels generated by the second stage. The result of this run is another GP individual. The two individuals (one from stage 1 and the other one from stage 3) are then combined together using a *min* function. They executed their system using different parameters and compared them against one another. They concluded that for such problem, large population sizes result in better solutions. Furthermore, they claimed that their results are highly comparable to those obtained by other methods such as Kohonen neural network. In a similar work, Harvey et al. [31] used GP to detect golf courses in the aerial images. The reason that they chose golf courses was that they have similar color range and vegetation. Their final GP results were graphs rather than trees: it allowed the reuse of values computed in some of the sub-trees. Contrary to Poli's guidelines, their function and terminal set were comprised of image processing operators and included spectral, spatial, logical and thresholding operators.

Winkeler and Manjunath [88] used GP for object detection; more specifically, face detection. They deemed GP as a method that can evolve complex non-linear image analysis functions from a set of small and simple feature-based or statistical functions. The authors conducted two experiments. The first experiment used statistical features extracted from 20×20 pixel regions of either face or non-face images. The average value and standard deviations of these regions were treated as features. The second experiment, on the other hand, scanned gray-scale images once for detecting faces at all scales. The scheme they used for their first experiment was the island-model GP. Their motivation behind this approach was to reduce code complexity and preventing specialization of solutions. This system was able to detect faces with the accuracy of more than 93%. They tested their second system on an image that contained human faces as well as several hand-drawn pictures of faces. They included the parsimony pressure for their second system. Individuals were allowed to have 50 nodes and then penalized. This system was able to detect human faces but not the hand-drawn faces.

Smart and Zhang [76] used GP for real-time object tracking in streaming videos. Their approached consisted of evolving an object tracker using the first few frames of the video stream and using the evolved tracker to track the objects in the subsequent frames. For each frame in the video, the greyscale image data as well as the position of the tracked object in the previous frame was provided to the evolved program. Instead of the conventional single-valued GP nodes, each node of their expression trees contained a vector with an x and a y component. Similarly, the output of the whole GP tree was a vector. The goal of the evolution was to evolve a tree which when applied to the current frame, takes the position of the object in the previous frame and predicts the position of the object in the current frame. They used lightweight features for the system. These features were either intensity values of the pixels or edge detection values produced by a simple edge detector function. At the end of the evolution, the most suitable object tracker for the desired object in the video was selected and used to track the object in the subsequent video frames. Their results suggested satisfactory performance for human head tracking.

Song and Fang [80] used GP to evolve moving object detectors. They examined two different approaches: the single-frame and the multi-frame approach. In the former, each frame was analysed independently while in the latter, a frame is examined in the context of a sequence of frames. For the single-frame approach, they examined three GP languages which only differed in types of features that were provided to the system. Three sets of features were examined for each language, details of which follow. The pixel intensity features contained only the pixel intensity values. The pixel hue features contained the results of a sine transformation on the hue value of the pixels. The last set of features contained spatial features extracted from specific regions of the input image. For the multi-frame approach the motion plane feature was used which for every frame contains motion information about all of the preceding frames. Their system required two phases namely the evolution phase and the application phase. This indicates that their system was offline and that the GP individuals were pre-trained prior to the application. In the application phase, each frame of the video was swept by a moving window in a manner similar to the one that [78] used. They tested all of the language variations on two different videos. The results indicated that the multi-frame approach which utilized the motion plane features was the most robust method followed by the single-frame approach which used the pixel intensity values. Based on these observations, Pinto and Song [63] further experimented the motion plane features for more complicated scenarios such as detecting moving vehicles on a freeway. The experiments suggested the robustness of the motion plane features. Furthermore, all of the approaches met the real-time requirements.

3.3 Computer Vision using Parallel Computation

Many of the computer vision algorithms have training phases. Usually, this phase is the slowest part of the algorithm and many applications demonstrate large amounts of data-parallelism. This section focuses on the applications of parallel computing methods in computer vision. These methods are generally used to decrease the time required for the training phase. This speed-up, can provide the opportunity of developing real-time systems.

Since parallel computing frameworks (such as CUDA) have only been recently released and extensively used, the publications that incorporate these methods in computer vision are rather limited. Nevertheless, in this section we review some of the literature that use parallel computing frameworks along with computational intelligence methods.

CUDA has been extensively used in medical imaging [75]. It can be used to analyze hybrid medical scans to diagnose diseases such as cancer in their early stages. CUDA can also be used in various computer vision-related task in environmental sciences [75]. It can be used to accelerate the analysis of seismic images. These images are vastly used to locate underground supplies of gas and old and require weeks of computations.

Some researchers have attempted to implement computational intelligence techniques using CUDA. Mussi et al. [56] developed a road sign detection system using PSO. Their PSO system took into account both the shape and the color of signs to detect them. To speed-up their processing, they used three CUDA kernels in their code. These kernels were designed to minimize the transferring of data between CPU and GPU. Their first kernel was only assigned the task of position update. Their second kernel was responsible for fitness evaluation while the third kernel had the task of updating personal best and global best values. Furthermore, they used a PSO with three swarms. This way they were able to assign each of the swarms to a CUDA block. Moreover, each swarm in their system was responsible for detecting a special class of road signs.

Because they were able to achieve real-time performance, they field tested their system using a camera mounted vehicle. Their system only produced 2 false negatives and was able to process the environment at 40fps[4]. Moreover, they observed that their implementation did not saturate the GPU, leaving room for more sophisticated implementations and processing.

Mussi's et al. [58] research was primarily concerned with marker-less human body tracking using multi-view video sequences. They formulated this problem as a multi-dimensional nonlinear optimization problem and were able to solve it using PSO. In marker-less tracking, no special equipment is required and the subject can participate in the test using everyday clothing. This is in contrast with marker-based tracking in which the test subject has to wear body suit and special clothing so that the computer is able to track it. Their PSO implementation details were mostly the same as those used in their previous work [56].

The aim of Uetz's et al. [86] research was to develop a large-scale object recognition system; i.e. a system which given an image, can label existing objects in that image. For their research they used a hierarchical neural network. One particular problem with large-scale object recognition system is that they require large number of training cases to demonstrate

[4]frames per second

acceptable performance. This large number of training cases results in slow training phase. Uetz et al. decided to accelerate their NN by means of CUDA. Their final system had error rates of less than 3% and compared to a sequential implementation on a CPU, they experienced speed-ups of up to 82 times.

According to Mnih et al. [54], despite many years of effort on automatic road detection, as of yet there are no automatic road detection systems in existence. In their research, they used a NN with millions of trainable weights and CUDA to train the NN using two large urban data sets that had never been used before. Despite many previous works in the same area, they did not use fixed width edge detection techniques as they deemed this approach unsuitable.

The data set they used was carefully aligned and labeled by a human expert. To further improve their system, they used some post-processing procedures. These procedures which included unsupervised pre-training phases and supervised post-processing, are beyond the scope of this thesis. Their final results showed that their system was very proficient in detecting roads and only had minor errors.

Not much research exists about online and real-time GP systems. Nording and Banzhaf [60] used a real-time and online GP system to evolve programs that would control the movements of a miniature robot. The GP system directly evolved control programs. This was necessary due to the constraints imposed by the robot's architecture. The GP would take sensor readings as an input and generate control programs as an output. These control programs would be then sent to the motors and the fitness of each program was determined after the action sequence had concluded. Although due to the ever changing position of the robot this was an unfair way to evaluate each individual, they discovered that the surviving individuals were indifferent towards this as time progressed. They used a steady-state GP system for their implementation. Based on the fitness values, the GP system would discard less desirable individuals in favor of the more suitable ones. They tested the real-time system for two different problems, namely obstacle avoidance and object following. Both applications resulted in satisfactory performance.

Although researchers have successfully applied GP to many problems including computer vision, only a few publications exist regarding the incorporation of parallel methods in GP for solving computer vision problems. Harding [25] used CGP to evolve image filters. His goal was to evolve noise removal filters that performed better that the traditional median filter. Furthermore, he used a graphics card and the Microsoft Accelerator API in order to evaluate the fitness of the evolved filters. For each pixel, the filter used the values form the

9 adjacent pixels in the vicinity of the original pixel in order to remove the noise that was introduced to the image. In a later work, Harding and Banzhaf [27] attempted to use CGP in order to reverse engineer the filters that were performed on a series of images using GIMP[5]. The term "reverse-engineering" here refers to imitating the results of the filters that were performed by GIMP. As in [25], GPU and Microsoft Accelerator was used to accelerate fitness evaluation. They successfully evolved filters such as dilate, erode, emboss, motion and Sobel that were equivalent to their GIMP counterparts.

In [15] Ebner proposed an adaptive online evolutionary visual system that would be able to adapt itself to the environment. According to him, evolutionary systems suffer from a common drawback: if they are moved to a new environment, they often break. He suggested the creation of an evolutionary system that would adapt itself based on a feedback from the user. In [15] he proposed running multiple algorithms at once: a main algorithm and a few slightly modified variations of that algorithm in the background. Each algorithm would perform the necessary calculations on the input image and would present the results to the user. The user would then select the best result and thereafter, the algorithm with the best result would become the main algorithm for the next input image. Clearly, this intelligent approach requires enormous computational powers.

In [16], Ebner implemented his proposed approach. To fulfill the need for such high computational power, he used OpenGLSL[6] and the CGP method to evaluate individuals of the different algorithms. To implement the adaptive behavior of the system, the system provided the user with a GUI[7]. In this GUI, the user was able to signify the areas in which the object should be extracted by mouse. As long as the mouse button was pressed those areas were used for evolution and fitness evaluation. The output of the best individuals were always presented on the screen. The user could select the best output and the system would use its equivalent algorithm as the primary algorithm for the rest of the evolution. By utilizing the methods mentioned above, Ebner's final system was able to achieve real-time performance and was easily able to detect specific targets in a sequence of images.

Building upon his previous work, Ebner extended his adaptive system to an automated system. The aim of his latest work [17] was to omit user intervention and make the system fully automated. To accomplish this goal, the input to his system was changed from a single image to a continuous sequence of images. He stated that since for humans motion is an important hint for identifying existing objects, computer vision systems could also detect

[5]A freely-available and open source image editing software

[6]OpenGL Shading Language

[7]graphical user interface

interesting objects based on their movement in a sequence of images. Hence, his system could detect any moving object and track it. For his implementation he proposed a method for converting sequence differences into a 2D motion model, details of which are beyond the scope of this thesis. Much like his previous work, he used OpenGLSL for executing the code on the GPU and accelerating the system. He also followed his initial proposal of having a main algorithm and a few other variations of it in the background. Using his system, he was able to evolve an object detector that was able to track and detect any object that was moving in a sequence of images.

Chapter 4

Feature Extraction Languages

In its simplest form, and object tracking system should be able to track simple objects, each defined by a unique texture. To this end, an important task is to evolve classifiers that are able to distinguish and classify textures. In order to apply GP for the visual pattern classification problem, one needs to provide the GP system with a suitable GP language. The language in question must be able to extract useful features from the provided input so that meaningful solution could be evolved for the problem. This chapter discusses the methods by which GP could be used for the problem of visual pattern classification. We will also investigate various GP languages that could help with this problem.

4.1 Introduction

Pattern recognition and classification is a challenging computer vision problem. The goal of visual pattern recognition is the automatic identification and extraction of features of interest. As discussed in section 3.2.2, GP has been used for a variety of computer vision and classification problems. In the GP literature, pixel-based classifiers (sometimes called *kernel-based* or convolution-based[1] classifiers) are dominant. These classifiers examine a set of features defined for a specific pixel of an image and make a classification decision for that single pixel. These features may also include values from the pixels in the vicinity of the pixel in question. These classifiers are repeatedly applied to all pixels of an image and each pixel is classified according to the output of the classifier. Conversely, block-classifiers are executed on a region (usually a set of pixels) of an image and a single decision is made

[1]These terms frequently appear in the image processing literature

for the whole region. Since block-classifiers make a decision for a group of pixels, they are generally faster. This efficiency comes with a cost: their accuracy is generally lower than that of the pixel-classifiers.

Although block-classifiers process a smaller set of data in comparison with pixel-classifiers and by extension have lower accuracy, previous research [81, 78, 77, 79] has identified them as an effective approach for the texture classification problem. Our objectives in this chapter are twofold. First, we are interested in comparing the effects of the GP languages on the performance of the evolved classifiers. Second, we will do a comparative study of the accuracy of the pixel-classifiers versus that of the block classifiers. To this end, we re-implemented the GP classification system presented by Song *et al.* [81, 78, 77, 79]. We also implemented a pixel-based GP classification system based on the previous attempts in the literature (see Section 3.2.2). We test the languages on different data sets to evaluate and compare their performance.

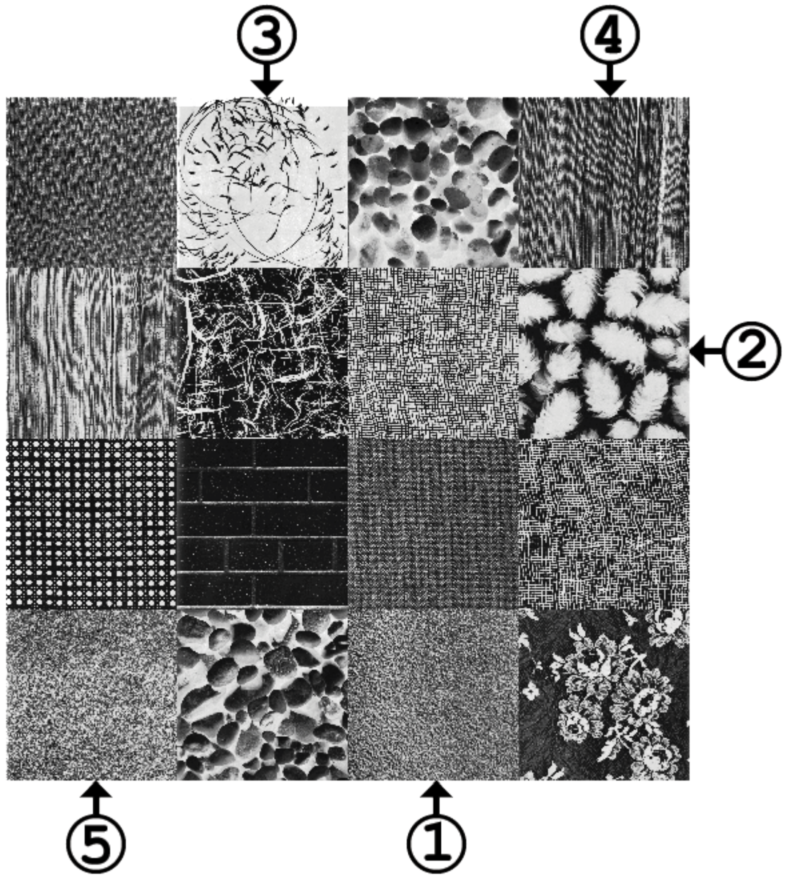
4.2 Data

The main computer vision problem here is the classification of the grey-scale textures in the Brodatz [8] texture database. This database was previously used in [81, 78, 77, 79]. Figure 4.2 shows the textures we have used. The 5 denoted textures are the ones that were used as positive examples in either of the GP systems; i.e. for one set of experiment, one of the labeled textures would be provided as the target for classification (hence, the positive example) while all other textures are provided as negative examples.

The original Brodatz textures were of size 640 $\times$ 640 pixels. In order to make the GP runs faster, these textures were scaled to 128 $\times$ 128 pixels. Although this process may reduce and distort the texture data, preliminary experiments showed that the GP system was mostly indifferent towards this reduction. Furthermore, these textures were selected to include various difficulty levels. For example, texture 1 and texture 5 are fairly similar to the human eye while both of them differ significantly from texture 3.

In addition to textures, we selected two other images to investigate the applicability of texture classification languages in other computer vision problems. One image is a color aerial image of boats near a port obtained from the satellite view of Google Maps [23]. The goal here is to classify boats in the image. Another image is a color photo of a group of people obtained using a camera. We aim to detect human faces in the image. Figures 4.2 and 4.2 show these images.

Figure 4.1: The Brodatz textures. The numbered textures are used as positive examples (target for classification) and the rest of the images are used as negative examples. Each texture patch is 128×128 pixels. Full image is 512×512 pixels

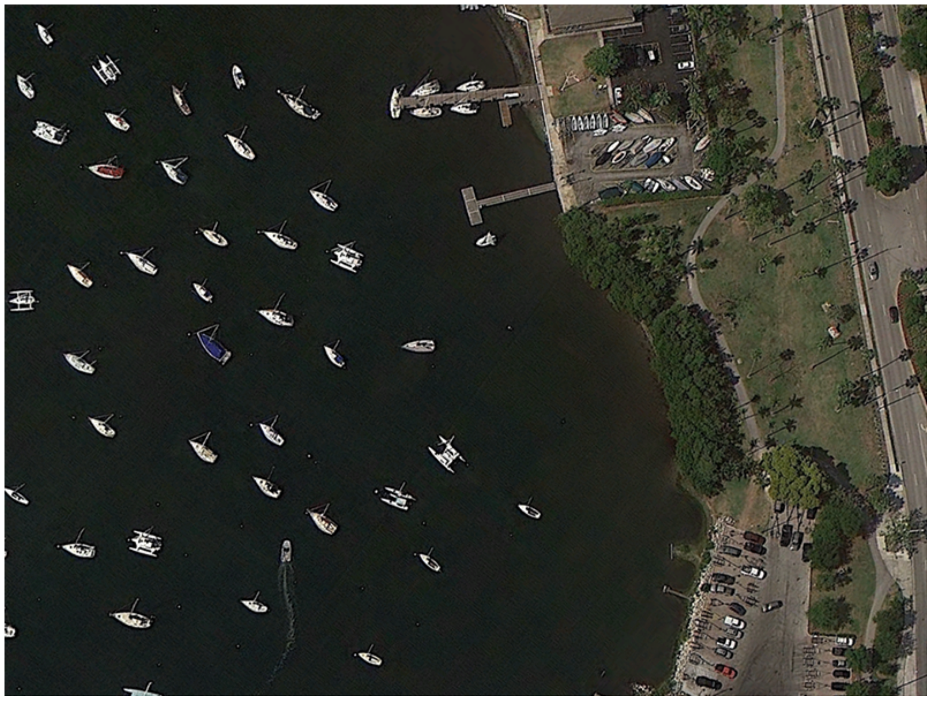


4.3 Block Classification Language

The block classification language was presented by Song *et al.* [77, 81] and was used for other computer vision applications [78, 79]. We implemented Song *et al.*'s original language. The system works as follows. Three images are provided to the system: a target image for positive classification (the positive image), the target for negative classification (the negative image), and a test image. For system training, a predefined number of sub-images are randomly sampled from the positive and the negative images. We refer to these sub-images as *blocks*. The blocks are of arbitrary sizes but for our purposes we have used blocks of 32×32 pixels[2]. Each training instance contains grey-scale (Brodatz) or color (boats, faces) information about the pixels of that block as well as the label of the instance,

[2]As demonstrated in [77], blocks of size 32×32 provide a good balance between classification performance and runtime performance

Figure 4.2: Aerial image of boats. Full image is 1692 $\times$843 pixels with 18891 positive pixels.

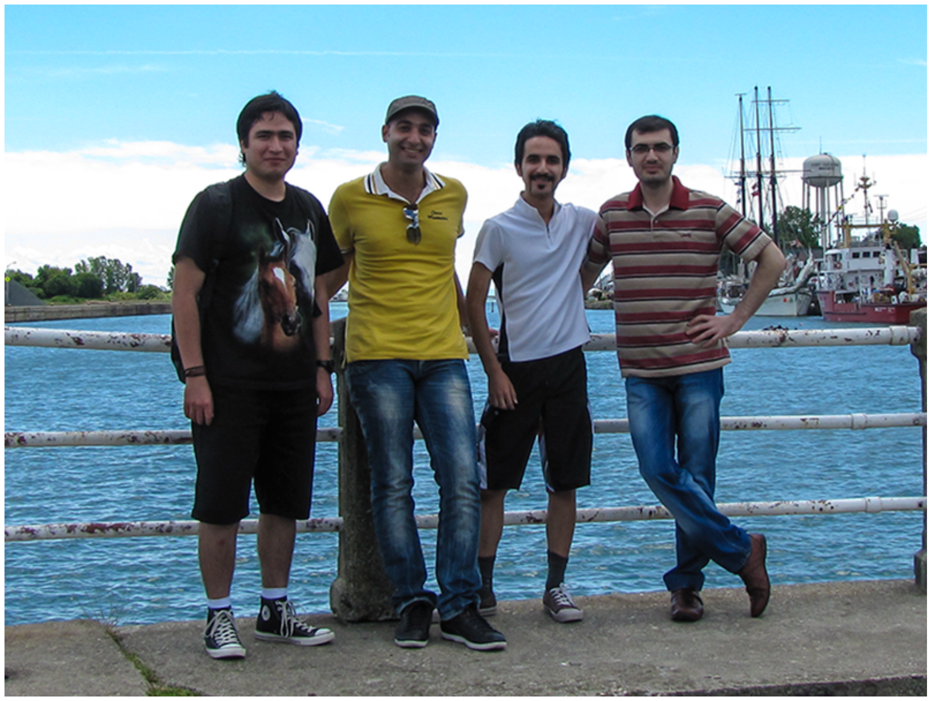


hence forming a feature vector. Using this feature vector, an evolved GP tree composed of various mathematical and decision making functions can extract useful features that are needed for classification. This results in a binary classifier which, when applied to an image region, makes a classification decision for that entire region. The goal for evolution is to evolve a classifier that correctly classifies positive and negative image instances, on a block-by-block basis.

The block classification language as used in [81, 77] is given in Table 4.3. This is a simple language that only uses pixel values of the feature vector to extract features that help the evolution of classifiers. Most functions and terminals are standard in the literature. Att[x] denotes the RGB value of the x^{th} pixel in the feature vector. For the implementation, strongly-typed GP was used [55]. The root of each GP tree is double, therefore the returned value of the tree must be converted to a binary value. We use static range selection: computed values greater than or equal to zero are interpreted as *true*, while negative values are *false*.

The distinguishing characteristic of block-classifiers is that they make a single decision for a block of 32$\times$32 pixels. When a decision is made, all 1024 pixels in that block are assigned that classification. This poses some difficulty in comparing block-classifiers with

Figure 4.3: Group photo. Full image is 1280×720 pixels with 7874 positive pixels.



pixel-classifiers. In order to make a more meaningful comparison, we use the following approach. During the training phase, the GP tree is executed on every possible 32×32 region of an image. We store the number of times that each specific pixel was classified as *true* within a tested block. This number is then divided by the total number of times that the pixel in question was processed. The result is the percentage value that each pixel was classified as *true*. If a pixel is classified as *true* the majority of time (more than 50%) then that pixel is classified as *true* as the final result of the classification operation for that pixel.

4.4 Pixel Classification Language

Our pixel-classifier works as follows. For a given data set, two images are provided to the GP system: an image to process and a ground truth image. In the ground truth image, the targets for positive classification are marked. Using the ground truth, the system randomly samples positive and negative examples from the input image a predefined number of times. These *center pixels* are then used for creating training instances. Using the coordinates of the center pixels, a block of n×n pixels is formed around these pixels in a way that the center pixels coincides with the coordinates ($\lfloor n/2 \rfloor$, $\lfloor n/2 \rfloor$). We refer to these blocks of pixels

Table 4.1: Block classification language. (D=Double, B=Boolean)

Name	Return type	Arg. type	Description
Add	D	D	addition
Sub	D	D	subtraction
Mul	D	D	multiplication
Div	D	D	protected division
If	D	B, D, D	if a true then b else c
$>=$	B	D, D	true if $a \geq b$
$=<$	B	D, D	true if $a \leq b$
Between	B	D, D, D	true if $b < a < c$
Random	D	-	random constant, -1 $\leq c \leq 1$
Att[x]	D	I	Value of attribute

as *grids*. For each pixel in the grid, spatial filter values (average and standard deviation) are calculated and stored for later access by GP expressions. To speed up processing, we compute these values using NVIDIA CUDA. Note that in computer vision literature, such filters are known as convolutions. Convolution filters are usually performed via moving a window of size $k{\times}k$ over every pixel in the image and mapping the value of the pixel in question to a new value by performing a series of calculations that use the value of the pixels that are inside the moving window. The moving window is commonly known as the *kernel* and k defines the kernel size. However, since kernel may be confused with CUDA kernel functions we used the term spatial filters instead in this research.

During training, a binary classifier is evolved using positive and negative training instances. The system's feature matrix is composed of the raw pixel values as well as the spatial filter values. GP uses this matrix in conjunction with mathematical and decision making functions to evolve a classifier. Integer offsets can be used to extract features from pixels in the area around the center pixel. The decision made by the classifier is applied to the center pixel. During testing, the classifier processes every pixel of the image. This contrasts to the block classifier, which assigns the classification to all the pixels in a block.

Tables 4.2 and 4.3 present the terminal set and the function set of our pixel-classifiers. We defined 3 pixel classification languages. These languages will use a subset of functions and terminals of Tables 4.2 and 4.3. As before, strongly-typed GP was used for the implementation of the pixel-classifier. Three data types are defined: double (D), integer (I), and channel (C). Terminals include ephemeral random constants for integers and doubles, channel index, and random terminals. Each time a random terminal is accessed, a new ran-

Table 4.2: Pixel language part 1. (I=integer, D=double, C=channel)

Name	**Return type**	**Arg. type**	**Description**
c	c	-	channel index (0,1,2,3)
ERC	D	-	ephemeral random constant in the range [0, 1]
Random	I	-	random integer in the range [0, 31]
GridERC	I	-	ephemeral random constant in the range [0, 31]
Input colour	D	C	channel value of the selected pixel
$Avg_{k=15,17,19}$	D	C	average of $k \times k$ area
$Stdev_{k=15,17,19}$	D	C	standard deviation of $k \times k$ area
Input colour	D	C, I, I	channel value C at (i,j) offset
$OAvg_{k=15,17,19}$	D	C, I, I	average of $k \times k$ area of channel C, offset (i,j)
$OStdev_{k=15,17,19}$	D	C, I, I	standard deviation of $k \times k$ area of channel C, offset (i,j)

dom value is generated and returned. Pixel values (RGB and/or grey-scale) are accessible, either directly for a center pixel, or for a specified integer offset within the 32×32 block. The channel argument c specifies the particular channel (R, G, B, grey-scale) to retrieve[3]. Spatial data (average, standard deviation) can also be accessed for center pixels and offsets near them for the specified channel. The area values (15, 17, 19) were determined by experimentation, as earlier attempts using smaller areas were not beneficial.

The four languages used for the experiments are summarized in Table 4.4. *Spatial operations* refer to the average and standard deviation functions, while *offsets* include the color and spatial operators that use (i,j) offsets. Therefore, the first two languages (Complete, No Offset) use spatial operators, while the others do not. Raw Features is essentially the Block Processing language with extended mathematical operators, but to be used in a pixel-classification manner.

4.5 Experiment Setup

The run parameters are summarized in Table 4.5. The classification accuracy [81] was used as the fitness function of all GP experiments. The classification accuracy is defined in Equation 4.1.

[3]Grey-scale images have R, G, and B removed

Table 4.3: Pixel language part 2. (I=integer, D=double)

Name	Return type	Arg. type	Description
Add	I/D	I/D	addition
Sub	I/D	I/D	subtraction
Mul	I/D	I/D	multiplication
Div	I/D	I/D	protected division
Neg	D	D	negation
Exp	D	D	*e* raised to the operand
IfGT	D	D,D,D,D	if a>b then c else d
Max	D	D,D	maximum
Min	D	D,D	minimum
Sin	D	D	sine
Cos	D	D	cosine

Table 4.4: Summary of the language variations

Name	Spatial operations	Offsets	Block processing
Complete	×	×	
No Offset	×		
Raw Features		×	
Block Processing		×	×

$$fitness = \frac{TP + TN}{TOTAL} \times 100 \tag{4.1}$$

In Equation 4.1, *TP* is the number of true positives, *TN* is the number of true negatives and *TOTAL* is the total number of cases.

4.6 Results

Table 4.6 summarizes the results of the experiments. The best evolved solutions for some of the experiments are presented in Appendix A. In order to make a meaningful statistical analysis, the experiments were run 20 times, each time with a different random number seed. Therefore, most scores are averaged over 20 runs, except for *best*, which is the overall image score of the single top performing classifier which was found during the evolution. All values are obtained from the testing phase of the system. Testing scores,

Table 4.5: Run Parameters

Parameter	**Value**
Population size	1024
Generation size	50
Crossover rate	90%
Mutation rate	10%
Selection method	Tournament selection
Tournament size	4
Elites	2
Number of runs	20
Pixel block size	32×32
Positive training examples (pixel-classifiers)	512
Negative training examples (pixel-classifiers)	1024
Positive training examples (block-classifiers)	400
Negative training examples (block-classifiers)	400

which exclude the training pixels, are the true positive and true negative scores for each image. *Performance* is the average of the true positives and the true negatives for testing. This was necessary to avoid biasing the scores towards a class. According to the "sensitivity/specificity dilemma" [67], the detection algorithms cannot produce produce true positives without generating false positives and false negatives. This means that any detection algorithm could always be biased towards one class (either positive or negative). For instance, in the experiments for Texture 1 there is only 1 positive instance and 15 negatives instances. These many negative examples may skew the results towards the negative instances. Best overall performing languages within a statistical significance measure [4] of 95%, are presented in boldface.

Although training scores are not reported, they were relatively similar to the testing scores for each set of experiments. Therefore, it can be concluded that over-training is unlikely to be occurring. The performance of different languages somewhat varies. For instance, in Texture 1 the Raw Features language has lower performance in positive identification compared to other languages. On the other hand, the Block Processing language is weak in negative identification. Based on these numbers it can be concluded that the spatial languages have performed better than other languages for Texture 1.

Texture 4 proved to be the most challenging data set in these experiments. All languages have performed significantly worse on this data set than all other data sets. The No Offset

[4]Statistical significance was measured with an unpaired t-test with unequal variance

Table 4.6: Experiment results. All scores are %. “Test +/-” scores are true positives and true negatives of the testing phase respectively. “Performance” is the average of “test +” and “test -”. All scores are averaged over 20 runs, except for “best”, which is the image score from single best classifier found. Top performing overall languages (within statistical significance of 95%) are highlighted in bold.

		Language			
Image	**Scores**	**Complete**	**No Offset**	**Raw Feat.**	**Block Proc.**
Texture 1	test +	79.40	89.91	39.69	95.73
	test -	93.67	93.73	83.75	35.33
	performance	86.54	**91.82**	61.72	65.53
	best	94.14	93.48	65.55	80.43
Texture 2	test +	75.55	58.20	25.76	97.40
	test -	85.74	90.20	90.64	60.68
	performance	**80.65**	74.20	58.20	**79.04**
	best	89.77	82.64	62.18	84.12
Texture 3	test +	96.76	96.39	73.68	94.66
	test -	97.68	97.64	87.78	98.05
	performance	**97.22**	**97.01**	80.73	96.35
	best	98.19	98.24	81.42	96.95
Texture 4	test +	27.33	67.51	2.41	94.23
	test -	91.94	88.70	98.23	41.95
	performance	59.64	**78.11**	50.32	68.09
	best	83.94	86.05	51.62	69.17
Texture 5	test +	81.85	87.37	20.09	89.79
	test -	86.32	89.41	90.28	57.14
	performance	84.09	**88.39**	55.19	73.46
	best	91.05	90.70	62.69	79.27
Boats	test +	94.53	94.74	77.25	95.89
	test -	96.21	96.45	93.35	67.11
	performance	**95.37**	**95.59**	85.30	81.50
	best	96.29	96.81	88.04	92.68
Face	test +	90.94	95.28	87.90	94.97
	test -	92.65	94.26	89.86	64.69
	performance	91.79	**94.77**	88.88	79.83
	best	95.66	96.93	90.25	92.49
	Total wins	3	6	0	1

language had the best performance for this set. It is surprising that although the Complete language was a super-set of the No Offset language, it had poor positive classification performance.

The No Offsets spatial language was the overall top performing language, having the best (statistically significant) average solution performance for 6 images. This is followed by the Complete language (3 images), and the Block Processing language (1 images). The two spatial languages, namely the Complete language and the No Offset language are the overall top performers.

The fact that the super-set language (Complete) performs worse than the No Offset language contradicts conventional wisdom that GP evolution will select the best language operators for a problem at hand. This can be attributed to the fact that a larger language can make the search space more complicated.

The Raw Features and Block languages, which do not use spatial operators mostly performed poorly in the experiments. It is clear that the block processing strategy we used with the Block Processor and the threshold value is advantageous for that language, as it is the main technical difference between it and the Raw Features language, which was the poorest performing language of the four studied.

Figure 4.4 show some texture image results of the experiments. Images (a-d) are the best solution image results for Texture 1 recognition, and (e-h) for Texture 2. It is evident that high positive scores were obtained for the Texture 1 area in images (a-d). Green overlay in other texture areas denotes false positive. It can be seen that most languages were confused by the pattern in the bottom-left corner. Admittedly, this pattern is very similar to the pattern of Texture 1. The Raw language was the worst-performing language studied for this data set and it has overlayed many parts of the data set. This has resulted in the low overall score of 65.60%.

Although Texture 2 seems to be very distinct compared to other textures, it was one of the most challenging textures for the GP system to recognize. The results in (e-h) show that it is usually recognized, but more mistakes arise in (e,f,g). In most experiments, the Block classifier had a high degree of false positives. A good result of the Block language on Texture 3 is shown in (i).

It is worth mentioning that boundaries can be difficult for some languages. The spatial functions, when used on boundary pixels, will contain information from both of the adjacent textures. We consider boundary artifacts to be acceptable noise.

Figure 4.5 show sample results for the aerial boat image. The No Offset result in (b) is very close to the ground truth (a), and has an 88.05% image score. False positive are mostly found in the ground clutter on the right-side. For comparison, a Block Processing result is shown in (c). As it can be seen in the details image in (d), the pixel classifiers have the ability to do precise classification and are more accurate compared to block-classifiers.

Figure 4.6 shows some results with the group photo. Although all existing faces are identified, there are many false positives on the arms and the background. The results are even worse for the block-classifier. Compared to the textures and the aerial image, the group photo is a much more difficult data set. This can be attributed to the fact that more features (especially, facial features) are required in order to do accurate face detection. These features are usually high-level image processing features which, given the simple mathematical GP-language, are probably too difficult for evolution to mimic.

Figure 4.8 shows the fitness performance for the Texture 1 runs. The top 4 curves are for the spatial languages, which show superior fitness to the non-spatial languages.

4.7 Discussion and Comparison

The results show that pixel-classifiers perform better than block-classifiers. Among pixel-classifiers, the ones that use the spatial operators are preferable to those that do not. Our block processing strategy used to analyze the results in Table 4.6 is not the manner in which other papers used block languages [77, 78, 79, 81]. By using exhaustive block overlays and thresholding, we improved the block language performance.

Examining research in [77, 81, 78, 79], we note that the GP expressions used for block processing were apparently not applied in a random-sampled manner to images. Rather, fixed coordinate positions for block overlays were used on training images. Computational performance in wall-clock speed is advantaged by the ability to classify large regions of images at once. Moreover, in block classifiers mistakes are costly since a single wrong classification decision affects a large number of pixels.

Our experience is that using a block classifier with random sampling of images during training and testing is more challenging for block classification. Since the simple block language samples only a relatively sparse number of pixel points in an image region, it is difficult to learn pattern concepts for complex images when the training points are limited. Conversely, spatial languages have the ability to extract additional features over an image region and they overcome this limitation. In our case, they extract features using average and standard deviation features which are similar to blur and edge filters. Furthermore, because a pixel-based spatial language can error for one pixel rather than an entire region, errors are far less costly.

Research in [77] provides the accuracy results of the original block processing language. Table 4.7 summarizes these accuracy values on the textures that we examined in our experiments. Comparison of the values in Table 4.6 with the ones in 4.7 suggests that the pixel-classifiers generally had better performance than the original block processing language. Note that the differences between our block processing language and the ones in Table 4.7 stem from the different sampling strategy that we employed. We used random sampling while the original block processing language used samples with fixed offsets.

Table 4.7: Accuracy values of the original block processing language

Texture	**Classification Accuracy (%)**
Texture 1	86.10
Texture 2	90.46
Texture 3	91.80
Texture 4	89.71
Texture 5	84.40

Comparing our results to other works in the literature, we can safely conclude that the classification accuracy of different GP languages studied here is satisfactory given the simplicity of the languages. Our languages share many similar elements to those available in the literature. Simple mathematical operators (addition, subtraction, etc.) as well as trigonometry functions, maximum and minimum function and the power and rooting functions are ubiquitous in almost all of the applications of GP in computer vision. Other functions such as the spatial operators are also frequently used in the literature. Harding [27] used spatial operations of size 3×3 for evolving image filters. Harding *et al.* [30] used more than 50 image processing operators including Gaussian blur, dilate, erode, Sobel and Canny edge detectors and etc. and achieved a classification accuracy of 98% in medical image classification. In addition to simple mathematical operators, Tackett [84] used decision making operators along with 20 different statistical features and achieved a classification accuracy of 96%. The closest implementation to ours is done by Howard *et al.* [32] for which they used average and variance filters of size 3, 5, 7 and 9. Unfortunately, they did not provide classification accuracy values of their experiments.

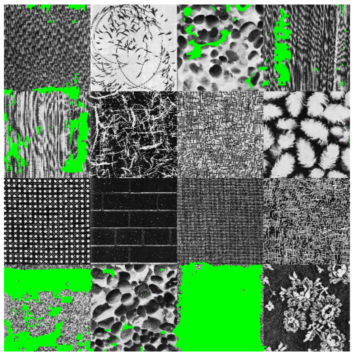
Comparison of our results in target detection (boats and faces) with other efforts in the literature reveals that considering the simplicity of our implementation, our accuracy of 95% is competitive with the other efforts. Moreover, our implementation is very lightweight as it eliminates the need of calculating computationally expensive spatial filters. This makes our implementation more suitable for real-time applications.

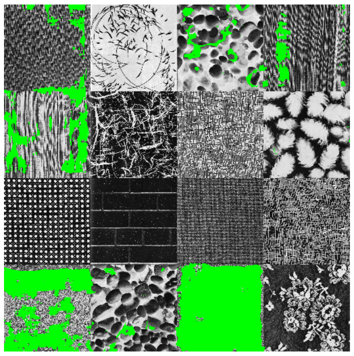
Examining previous research suggests that a means of improving the accuracy of a GP-based classifier is the incorporation of more high-level and specialized image processing features. These high-level features allow the incorporation of domain knowledge into the GP language. However, there is an inherent trade-off between accuracy and speed as using more features results in higher execution times which may prevent the use of system in real-time environments.

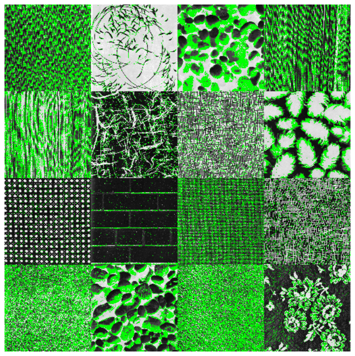
4.8 Conclusion

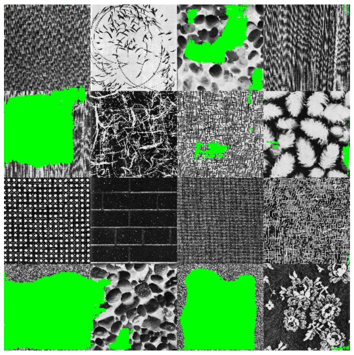
Our results show that pixel classifiers with offsets and spatial pixels are the best performers for the studied problem. The spatial operators are helpful in extracting sophisticated features from raw image data. Although block-classifiers have lower accuracy, they were competitive with the pixel-classifiers that we tested especially considering the very simple language that we studied.

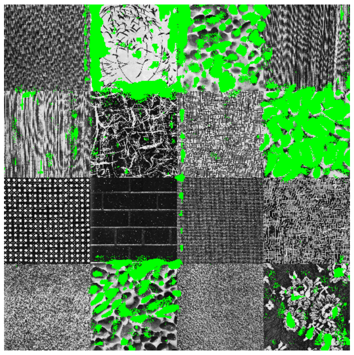
Since the purpose of this comparative study was to select the most suitable language for the real-time environment, we decided to use the No Offset language for the basis of our implementations in the next chapter. Evidently, this language defines a reasonably sized search space and this makes it easier for the GP system to evolve good solutions. Furthermore, the language is considerably lightweight since the offset language elements are removed compared to the Complete language. Another trait that makes this language suitable for our purposes is its simplicity when mapped to the CUDA framework, making the parallel implementation relatively straightforward. In the next chapter we discuss more implementation details of this language.

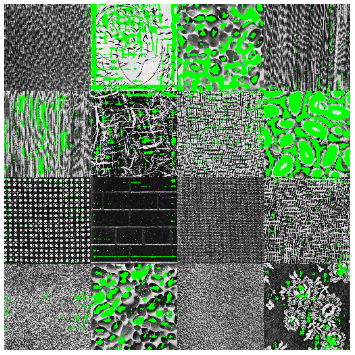

(a) Texture 1 - Complete
+ 94.74%, - 93.60
Total: 94.17%

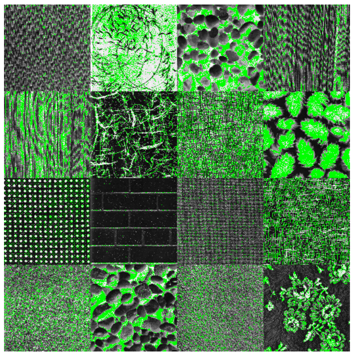

(b) Texture 1 - No Offset
+ 94.36%, - 92.67%
Total: 93.50%

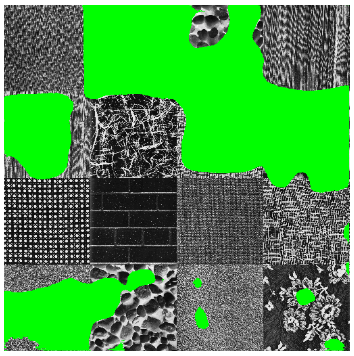

(c) Texture 1 - Raw Features
+ 51.58%, - 78.94%
Total: 65.26%


(d) Texture 1 - Block Proc.
+ 73.96% - 88.51%
Total: 81.23%


(e) Texture 2 - Complete
+ 94.10%, - 85.45%
Total: 89.77%


(f) Texture 2 - No Offset
+ 75.11% - 90.25%
Total: 82.68%


(g) Texture 2 - Raw Features
+ 41.24% - 83.18%
Total: 62.22%


(h) Texture 2 -Block Proc.
+ 99.01% - 70.81%
Total: 84.91%

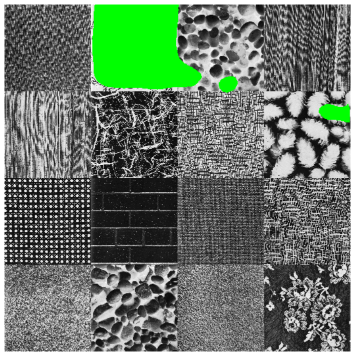
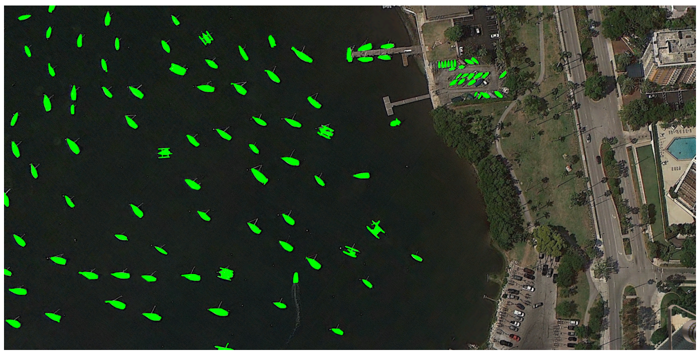
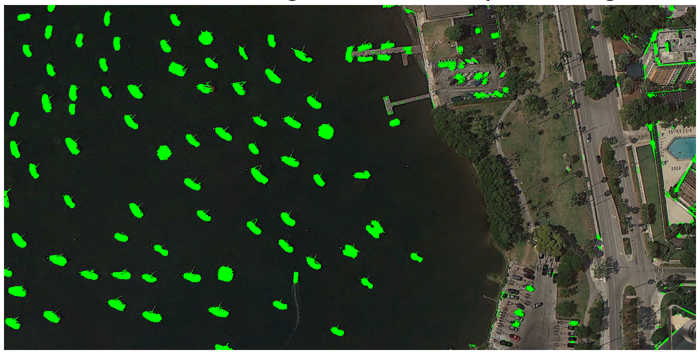
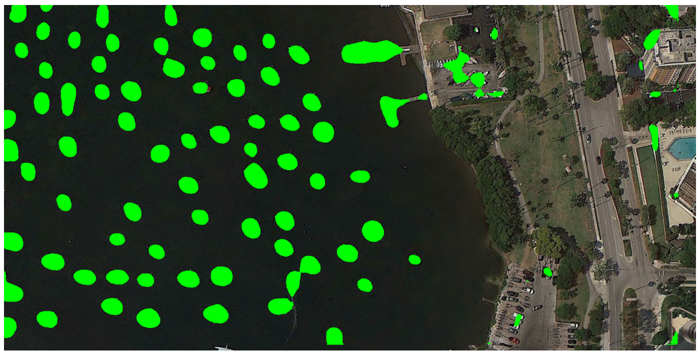

(i) Texture 3 -Block Proc.
+ 98.38%, - 97.64%
Total: 98.01%

Figure 4.4: Texture output images. The overall score is reported for each image. This score includes both testing and training. Green overlay indicates positive detection.


(a) Ground truth. Target areas overlayed with green.


(b) No Offset (+ 94.46%, - 96.45%, Total: 95.46%).


(c) Block Processing (+ 97.37%, - 90.57%, Total: 93.97%).

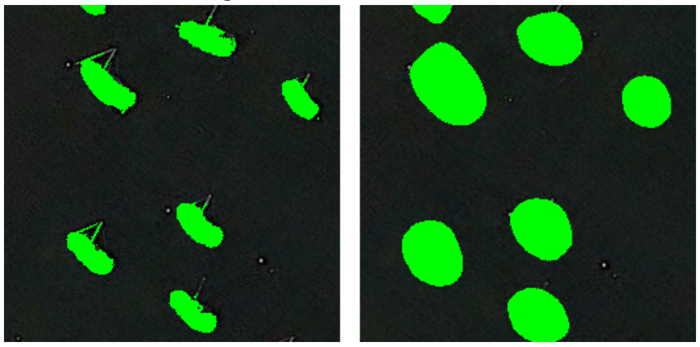
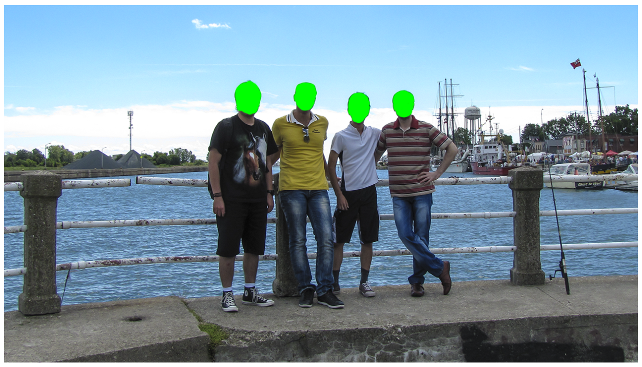
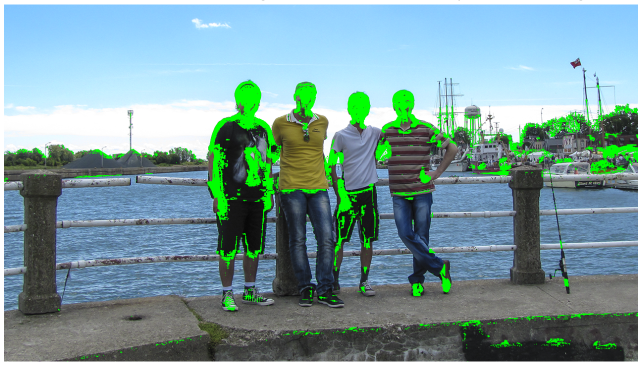
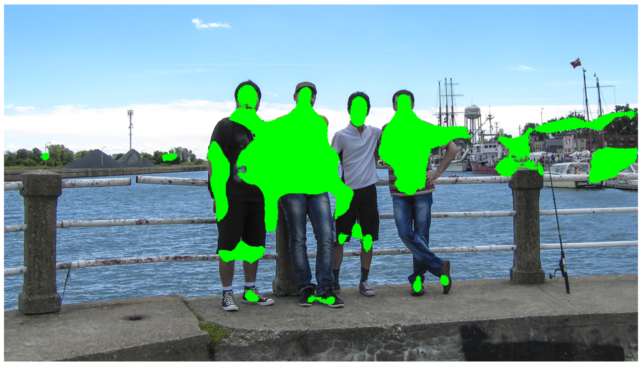

(d) Details: (left) No Offset, (right) Block Proc.

Figure 4.5: Boat output images. Green overlay indicates positive detection in (b), (c) and (d). The overall score is reported for each image. This score includes both testing and training.


(a) Ground truth. Target areas overlayed with green.


(b) No Offset. (+ 97.48%, - 96.44%, Total: 96.96%)


(c) Block Proc. (+ 98.29%, - 87.69%, Total: 92.99%)

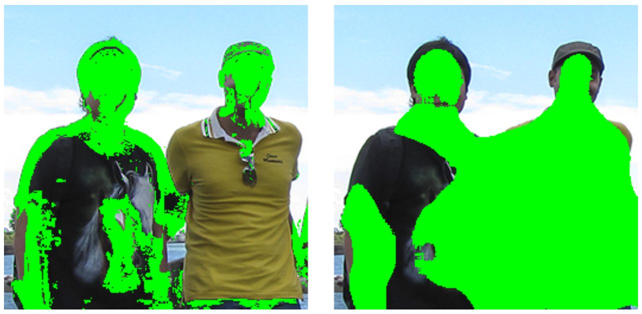

(d) Details: (left) No Offset, (right) Block Proc.

Figure 4.6: Face output images. Green overlay indicates positive detection in (b) and (c). The overall score is reported for each image. This score includes both testing and training.

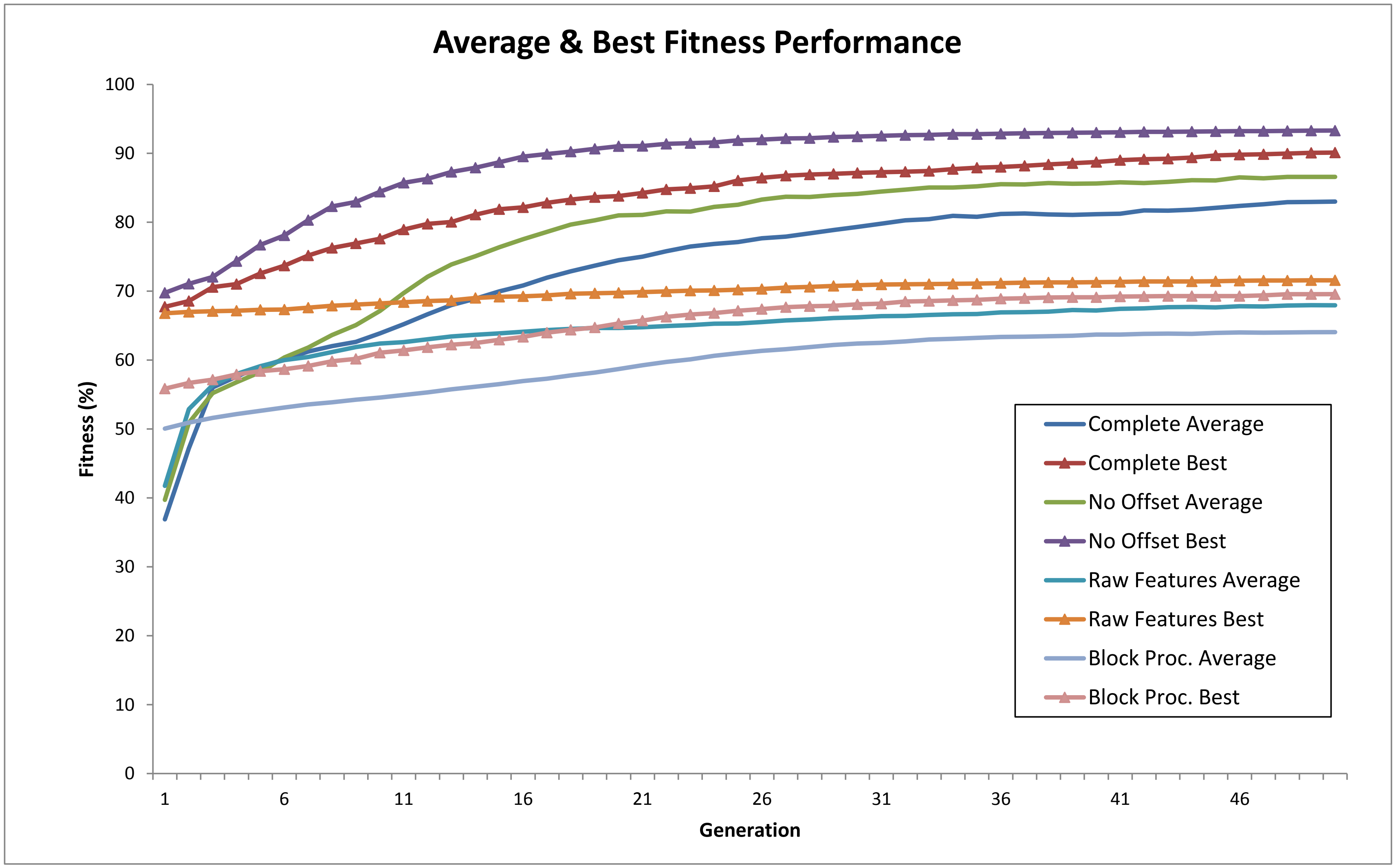


Figure 4.7: Fitness performance graph for Texture 1 runs (average 20 runs).

Chapter 5

The Real-Time Engine

5.1 Introduction

The focus of this chapter is the real-time computer vision engine. We discuss the architecture of the system along with some of its implementation details. We will also discuss the automatic learning environment that we have employed and also provide some insight about our object tracker.

5.2 Object Tracking and Automated Learning

Automated learning is a challenging task and in order to avoid some of the complexities that are associated with it, we have made several simplification which will be discussed shortly. According to [90], object tracking can be either performed jointly or separately. In the former, the object tracker processes every frame of the video and detects the object per frame while in the latter the object tracker detects the object of interest in the first frame and approximates its location in the subsequent frames. For the purposes of this research, the object tracking task is performed separately in the sense that the object tracker runs on every frame of the video and analyzes the segments in the video. For simplicity we have assumed that all segments in the video frame are unique and segments are not repeated. Furthermore, once a segment enters the animation canvas, it will remain on the canvas for the duration of the run. We are also supplied the location and the boundaries of each segment. In other words, the segments are predetermined and no object detection is performed. We needed to impose these limitations in order to simplify the vision problem at hand. Such implementation prepares the system for a more sophisticated implementation that involves segmentation algorithms in the future. As a result, in the implementation of the system we favored simplicity over optimization. This is especially true for CUDA kernels. Micro-optimization of the kernel code is required if heavy utilization of the GPU resources is the goal. This level of optimization usually breaks the simplicity of the implementation and it contradicts with our goals.

In each video frame, all segments as well as the background are extracted and for each segment, a classifier is evolved. When the classifiers are evolved, all of them are executed on all segments in the video frame. A segment is “claimed” by a classifier if that classifier positively classifies the majority (i.e. more than 50%) of the pixels of the segment in question. When a segment is claimed, a color overlay unique to the claiming classifier is painted on that segment, effectively distinguishing it from all other segments on the screen. This overlay tracks the segment in the video frame. If a segment is “orphan” (i.e. no classifier has claimed it) or a segment is claimed by more than one classifier or a classifier has claimed more than one segment, the retraining rules need to be applied.

After running all available classifiers on all segments of the video frame, several scenarios may occur. If all segments are uniquely classified by all classifiers and all classifiers have only claimed one segment, then are no problems and no retraining is necessary. If after running all available classifiers on the frame there is a segment that is orphan, then a brand new classifier should be evolved. The texture of the orphan segment is given to the classifier as a positive example while all other segments in the frame as well as the background pattern are given to it as negative examples. If there are no other segments, only the background pattern is assigned to the classifier as a negative example.

If after running all available classifiers on all segments a classifier has claimed more than one segment or a segment is claimed by more than one classifier, the system may have a problem. In this case, the classifiers that have problems are completely removed from the system. This way, the segments that were probably correctly claimed by them will become orphan segments and according to the rule that we just discussed, classifiers for these segment will be evolved when the next video frame becomes available. In other words, if there are any classifiers that are problematic, they are removed and their relevant segments are made orphan so that new classifiers could be evolved for them when the next video frame is available. To clarify the procedure that we just discussed, Algorithm 1 shows the pseudo-code for the automatic learning rules.

One thing to note is that the system resolves the conflicts in a step-by-step manner. In other

```
forall the segments in the current frame do
    Run all available classifiers on all segments (except for permanent orphans);
    Determine orphans;
end
Determine (new) permanent orphans and remove them from the list of orphans;
forall the evolved classifiers do
    if Classifier has not claimed anything or has claimed more than 1 segment then
        Mark this classifier to be destroyed;
    end
end
Destroy all marked classifiers;
if GP queue is empty then
    Evolve a classifier for one of the orphans
end
```

Algorithm 1: Pseudo-code for the automated learning environment

words, the request for the evolution of a new classifier is issued only if no other requests are currently being processed. This way we make sure that a problem is resolved before trying to solve another problem. Furthermore, after a few requests for evolution we may wind up with some classifiers that can be considered "failed" in the sense that they demonstrate conservative behavior and will classify everything as negative. As seen in Chapter 4, GP may sometimes evolve inadequate classifiers. These failed classifiers will be marked and removed from the system at the end of each cycle as they only degrade the performance of the system.

Initial experiments revealed that even after leaving the system running for a long time, it may fail to evolve a unique classifier for each segment. This is particularly true about the harder data sets. In such cases, the system will be stuck in a loop and the GP engine would be invoked many times. Therefore we imposed a limit on the maximum number of evolution requests per segment. For any segment that this limit is reached, that segment will be considered a permanent orphan and GP will not be invoked for that segment. We should mention that the language implementations in this chapter is based on the discussions that were presented in Chapter 4. This chapter is not meant to address the shortcomings of the languages that were discussed in Chapter 4; rather, we build upon the research in the previous chapter in order to show the application of a suitable GP-language in a real-time environment.

5.3 Modules

The developed system is comprised of various modules. These modules are discussed in this section. A block diagram of the system is as depicted in Figure 5.1.

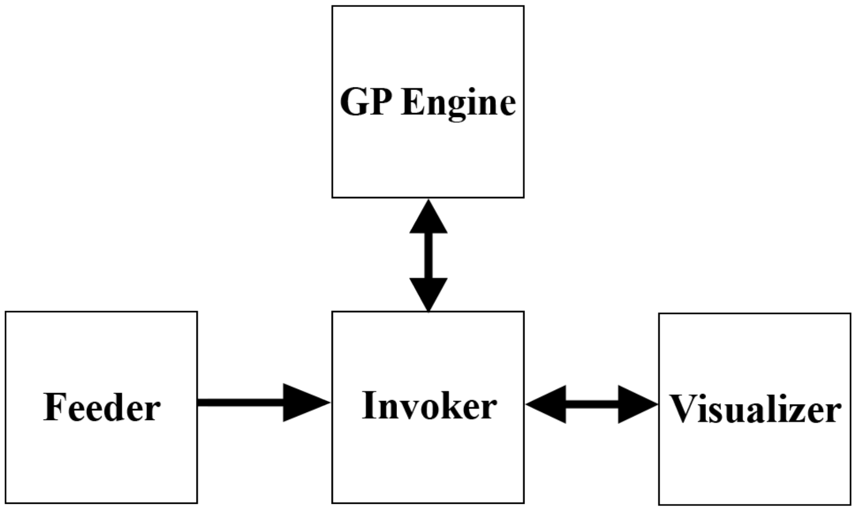


Figure 5.1: Block Diagram

5.3.1 The Parallel GP Engine

In the previous chapter we studied various GP-languages for texture classification. The language we chose was the No Offset language (see Table 4.4). This language showed satisfactory performance in our experiments and is relatively easy to implement in the CUDA framework.

Incorporation of CUDA

The original GP system that was discussed in the previous chapter was created using ECJ [49]. JCuda [33] was used for Java/CUDA interoperability. Following JCuda's convention, the CUDA kernel code for this system was written in C. Furthermore, all Java objects that are used in the kernel code were converted to equivalent C structures. All source codes for our implementation are available at ***https://www.github.com/Maghoumi/GP-Tracker***.

Following the guidelines in [9], the fitness evaluation part of this GP system was made parallel. The population-parallel approach that was discussed in 3.1 was used as the parallel implementation model.

Note that for the implementation, the ECJ system was actually extended; i.e. the population parallel approach has been implemented by extending existing ECJ components using guides in [48], hence by changing the kernel file and some minor modifications, one could easily use this GP system for other kinds of problems.

Before an individual is evaluated, it needs to be transferred to the GPU memory. Based on previous works in the literature [71, 46], we used the postfix notation for tree representation in our GP system. For simplicity, we added capabilities to ECJ to convert a GP tree to a postfix expression. This is not by any means an efficient implementation and as mentioned in [70], a more efficient method would be to change the breeding module in ECJ to evolve postfix expression rather than pointer-based trees. Also, we developed a stack-based postfix expression evaluator in CUDA. Whenever a tree needs evaluation, this evaluator parses the postfix expression and does the necessary calculations.

Since CUDA memory transfers are very slow, it is usually 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 data are flattened and are stored in contiguous memory addresses. This implementation was based on the guidelines available in [2, 1]. Furthermore, all allocated memories meet CUDA's alignment requirements as detailed in [1].

Whenever the population needs to be evaluated, the whole population will be evaluated in a single CUDA kernel call. To this end, for each generation, the number of individuals that need evaluation is determined. Then, each individual is converted to a postfix expression and stored in the GPU memory. The CUDA kernel is called using a reference to this memory space. The spatial filters (such as the average and standard deviation filters) are all calculated using CUDA in a separate kernel before the evaluation kernel is invoked.

For evaluation, we have adopted the method described in [71], i.e. each block is responsible to evaluate a single individual on all fitness cases. Depending on the total number of fitness cases, a thread is responsible for performing the required calculations for one or more of the fitness cases (again for the same individual). Therefore, the block size and the number of tasks assigned to each thread are determined dynamically. The grid size of the kernel is also determined dynamically which is equal to the number of individuals that require evaluation.

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 and ECJ assigns the fitness of each individual to it.

The Black Box Approach

We converted the GP system we discussed in the previous chapter to what we would refer to as the "black box". At creation, the evolution parameters are passed to the system. These parameters will remain constant throughout the rest of the execution unless changed manually. The GP system runs on its own thread and monitors a job queue. To initialize evolution, a job must be added to this queue. Each job contains a set of positive images and a set of negative images[1]. Whichever module that schedules a job could also define a set of call-back methods in order to be notified about various events in the GP system. One such event is the conclusion of a single generation or evolution of a new best individual.

As soon as a job is added to the GP system's queue, the evolution process begins asynchronously. Thus, additional jobs could be queued on the system while a job is being processed. When the evolution has concluded, the best trained classifier is returned via the aforementioned call-back methods. These evolved classifiers can be executed on arbitrary data using CUDA. Furthermore, the system has the capability of retraining a classifier by means of processing another job. A retrain job is similar to a regular evolutionary job with the exception of the existence of a classifier. The classifier in question could either be used to seed the initial population of the evolutionary run or it could be merely a placeholder for a newly evolved classifier.

Training Instances

Each evolutionary job has sets of positive and negative images. These positive and negative images are randomly sampled and the samples are stored as training instances for the system. These instances form a feature vector as described in Section 4.4. After selection, the instances are shuffled and transferred to the GPU memory. When the CUDA evaluation kernel is invoked, these samples are used as fitness cases for the evaluation function.

[1]The system can also get image/ground-truth as the input, but for the purposes of this thesis, the positive set and the negative set suffice

5.3.2 The Invoker Module

This 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 segmented video 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.

5.3.3 The Feeder Module

The sole purpose of the Feeder module is obtaining video frames from a video source (be it a video file or OpenGL generated video) and segmenting the frame. At the Invoker's request, the Feeder will segment the next video frame and pass it back to the Invoker. Each frame that the Feeder provides contains some data. One such data is the image data of the video frame. Another data is a set of detected segments in the frame in question. The background information of the frame is also available per frame. These data are used for creating a job object and queuing it on the GP Engine.

If the video source of the Feeder module is an OpenGL generated video, the data structure containing the pattern of the segments as well as their current location in the video frame is passed to the Invoker. Each frame will also contain a background pattern that was used in the frame. This background pattern will be always used as a negative example for the segments in the frame. This was done because in case a frame only contains a single segment, we are interested in distinguishing that segment from the background pixels.

5.3.4 The Visualizer

The Visualizer module is mainly used for running the evolved classifiers on the video frame. It serves other purposes as well. In addition to storing all evolved classifiers, it requests for the evolution of new classifiers as well as detecting conflicts between the outputs of the classifiers. In other words, the automated learning rules that were discussed in Section 5.2 are implemented by the Visualizer. The Visualizer calculates all the spatial filters for all segments in the current video frame, runs all classifiers on them, and does the thresholding to determine which classifier has claimed which texture. The Visualizer deletes the classifiers that are problematic and also handles the orphan segments.

One thing to note here is that the visualizer prioritizes the orphan segments over conflicts. In other words, classifiers are evolved for all orphan segments first and then the conflicts are resolved. The order in which the orphan segments selected to be queued for training is random in each cycle.

For the visualization of the results of the evolution, the GP trees are only executed on the pixels within the boundaries of all segments. This way, the overall performance of the system will improve as the trees are only executed on the regions of interest of the video.

5.4 Experiments

This section describes the details of the experiments that were performed on the developed system in order to benchmark its performance. All of the 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 (non-developer) drivers.

5.4.1 Textures

The textures that have been used for the experiments are the same as Section 4.2. However, we have prepared these textures in a different manner. The images that we used for our previous experiments were 128×128 pixels whereas for the experiments in this chapter, textures of size 96×96 are used. This improved the runtime performance of the experiments. However, at the size of 96×96 pixels the scaling process that we used in Chapter 4, significantly distorted the pattern of some textures. As a result, the textures that had a uniform pattern across the whole region of the image were simply cropped to 96×96 pixels while other patterns were scaled.

Based on the experiment results of the previous chapter, the textures are categorized into two sets: the easy set and the hard set. The easy set consists of the textures that were relatively easy for the GP system to classify while the hard set contains all of the textures from the easy set plus the more challenging textures. The easy set is depicted in Figure 5.2 and the additional textures in the hard set are depicted in Figure 5.3.

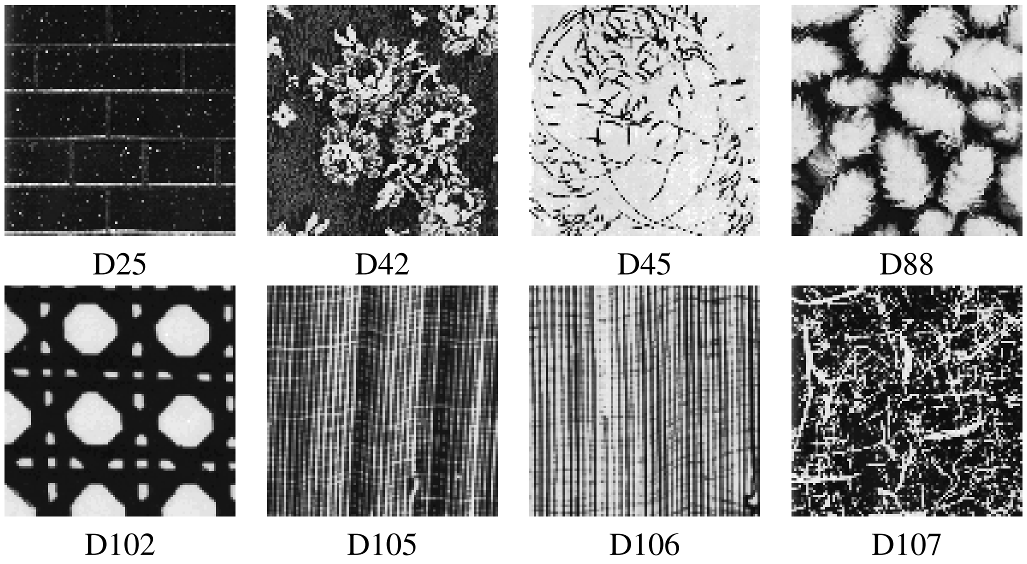


Figure 5.2: The easy set. Each image is 96×96 pixels

5.4.2 Experiment Procedure

Synthetic video is used as the primary source of data for the experiments. The video is generated in real-time using OpenGL and contains a selected number of textures from the easy or the hard set. For each cycle of the run, the location of each segment as well as the pattern of each segment are passed to the Invoker module. The Invoker module then passes the frame and the segments to the Visualizer and managed GP call requests that the Visualizer may issue.

Two main sets of experiments are performed. In the first set, which we will refer to as the “All” set hereinafter, the OpenGL generated video contains a predefined number of textures. Using the Feeder module, this video is passed to the Invoker module and the training session is started. This marks the beginning of a *session*. The number of textures in this session remains constant throughout the run. Using the automated learning rules that were discussed previously, the Visualizer attempts to evolve a unique classifier for each texture.

As previously mentioned, even after a significant amount of GP calls the GP engine may fail to evolve a unique classifier for each texture. We imposed a limit on the maximum number of GP calls that can be made for a specific texture. Should this limit exceed, the texture in question will no longer be queued on the GP engine and by extension, the system will not attempt to evolve a classifier for it. We refer to these textures as *permanent orphans*. The

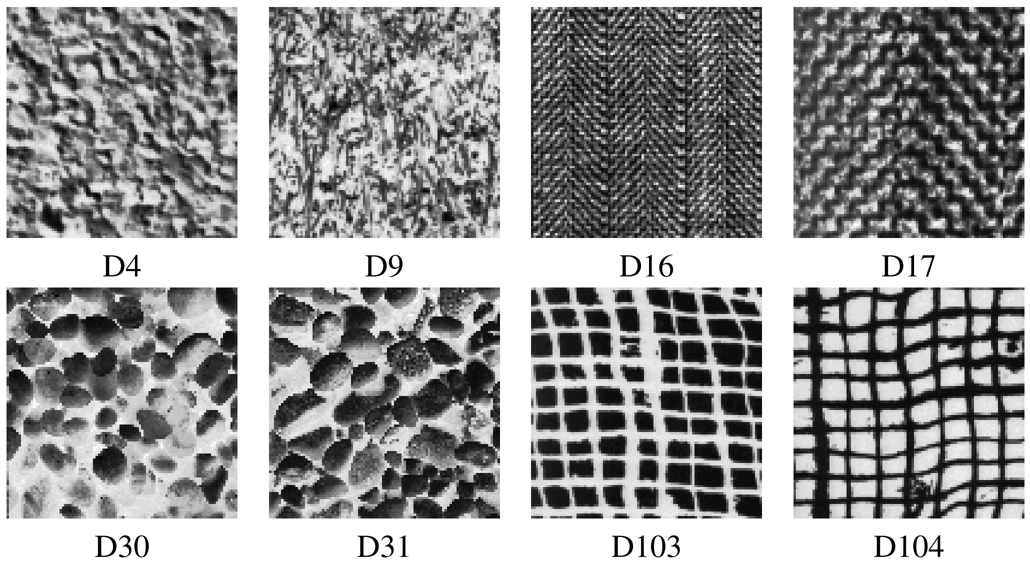


Figure 5.3: The hard set. Each image is 96×96 pixels

segment texture of the permanent orphans, however, are used as negative examples for the training of classifiers for other textures.

A session is considered successful if all textures – with the exception of permanent orphans – have a unique classifier that can distinguish them from the rest of the textures in the session. After a successful session, all statistical data that are required for analysis are gathered and the frame rate of the Visualizer is measured. In order to signify the results of the experiments that belong to this set, we prefix them by the label *"all"*.

The second set of experiment, which we will refer to as the "One" set hereinafter, consists of a video feed in which new segments are added to incrementally. For this set, a predefined number of textures are selected. At the beginning of the session in this set, the generated video contains only two of the selected textures. This *mini-session* is processed in a manner similar to the first set of the experiments; i.e. a classifier is evolved for each texture using the automated learning rules. A mini-session is considered successful, if all textures – again, with the exception of permanent orphans – have a unique classifier or if the maximum limit for the number of GP calls is reached.

As soon as the mini-session is successful, another texture is added to the video feed. When added, this texture will be an orphan texture. Thus, a classifier is evolved for it. Furthermore, the inclusion of the new texture will probably cause confusions for the existing classifiers in which case they are queued for retraining. The actual session will conclude if all of the originally selected textures have been added to the video feed and all mini-sessions are successful. Same as before, the statistical data are collected at the end of the session. In order to signify the results of the experiments that belong to this set, we prefix them by the label *"one"*.

In order to distinguish the different experiments we will 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. Table 5.1 summarizes all the experiments that were performed.

Table 5.1: Summary of the Experiment Sets

Set	Difficulty	Number of 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

5.4.3 Fitness Evaluation

The classification accuracy was used as the fitness function of all GP experiments. The classification accuracy is defined as follows:

$$fitness = \frac{TP + TN}{TOTAL} \times 100 \tag{5.1}$$

where *TP* is the number of true positives, *TN* is the number of true negatives and *TOTAL* is the total number of cases.

5.4.4 GP Language and Parameters

The GP language that was used for all experiments is the No Offset language discussed in Section 4.4 (see Table 4.4 for more details). Most of the core GP parameters that were used are exactly the same as the ones presented in Table 4.5. Table 5.2 summarizes the run parameters.

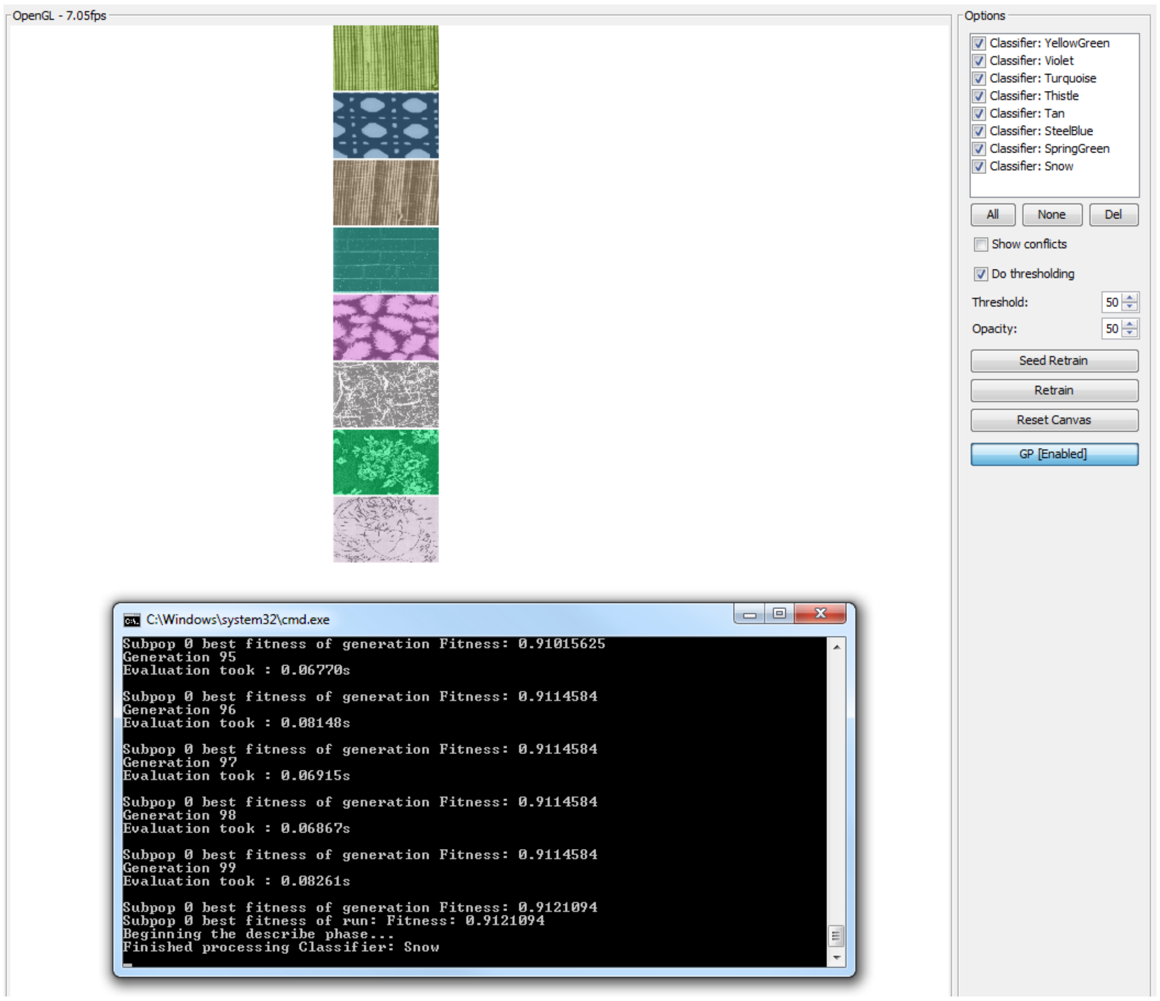
Table 5.2: Run Parameters

Parameter	**Value**
Population size	1024
Generation size	100
Crossover rate	90%
Mutation rate	10%
Selection method	Tournament selection
Tournament size	4
Elites	2
Number of Runs	20
Positive examples	512
Negative examples	1024
GP call limit	25

The only parameter that was changed in these experiments compared to Chapter 4 is the maximum number of generations (100 here as opposed to 50 which was used previously). The motivation behind this decision was the performance of the system. The real-time GP engine could easily process 50 generations in a fraction of a second and some preliminary experiments revealed that with a maximum of 100 generation, classifiers could achieve a higher level of accuracy on particular textures. These experiments further suggested that generally few GP calls were required for evolving suitable solutions and textures rarely required more than a handful of GP calls in order to evolve a unique classifier for them. Therefore, we imposed a limit of maximum 25 GP calls per texture. In case after 25 GP calls no unique classifier was evolved for a particular texture, that texture will be considered a permanent orphan.

5.5 Results

This section presents the results of the experiments. Where applicable, the results are averaged over 20 runs. Figure 5.4 shows a screen capture of the system during a run.



- [✓] Classifier: YellowGreen
- [✓] Classifier: Violet
- [✓] Classifier: Turquoise
- [✓] Classifier: Thistle
- [✓] Classifier: Tan
- [✓] Classifier: SteelBlue
- [✓] Classifier: SpringGreen
- [✓] Classifier: Snow

[ ] Show conflicts

[✓] Do thresholding

Threshold: 50

Opacity: 50

```
Subpop 0 best fitness of generation Fitness: 0.91015625
Generation 95
Evaluation took : 0.06770s

Subpop 0 best fitness of generation Fitness: 0.9114584
Generation 96
Evaluation took : 0.08148s

Subpop 0 best fitness of generation Fitness: 0.9114584
Generation 97
Evaluation took : 0.06915s

Subpop 0 best fitness of generation Fitness: 0.9114584
Generation 98
Evaluation took : 0.06867s

Subpop 0 best fitness of generation Fitness: 0.9114584
Generation 99
Evaluation took : 0.08261s

Subpop 0 best fitness of generation Fitness: 0.9121094
Subpop 0 best fitness of run: Fitness: 0.9121094
Beginning the describe phase...
Finished processing Classifier: Snow
```

Figure 5.4: Screen capture of the real-time system processing a video. The checkbox list shows the list of evolved classifiers along with their respective colors. Each segment in the video has been overlayed with the color of its classifier. The console window shows the outputs of the GP engine per generation.

5.5.1 GP Invocations

We studied the number of GP invocations that are required for each set of experiments. These results are summarized in Table 5.3. The confidence level value specifies the maximum and minimum number of GP invocations that are required for a particular data set with a confidence of 95%. The values in the normalized column have been calculated by dividing the average value by the number of textures in each experiment.

Table 5.3: Average number of GP invocations (95% confidence)

Experiment	Average		Normalized
all-easy-4	4.00	± 0.00	1.00
all-easy-8	17.00	± 4.02	2.12
all-hard-4	9.20	± 4.52	2.30
all-hard-8	39.80	± 9.57	4.97
all-hard-12	96.80	± 14.76	8.06
all-hard-16	185.52	± 10.76	11.59
one-easy-8	23.45	± 2.67	2.93
one-hard-16	192.25	± 9.27	12.01

As it can be seen in Table 5.3, more GP invocations are necessary for experiments with larger data sets. In other words, greater number of textures require more GP invocations so that a suitable classifier is evolved. These results coincide with what one would expect. One thing to note here is that on average, the experiments in which the textures are gradually added to the video feed (e.g. one-easy-8) require more GP invocations than the experiments where all textures exist in the video feed from the start of the experiment. This indicates that if all textures are available at the same time, it would be easier for the GP engine to evolve solutions. On the other hand, if textures are given one by one, the new textures may cause some confusions for the previously evolved classifiers. Therefore, these classifiers need to be retrained and this results in more GP invocations. Based on this observation it can be safely concluded that the *"All"* sets are generally easier than the *"One"* sets.

A close look at Table 5.3 reveals that the number of invocations is very sensitive to the number of textures in the experiment. Merely adding 4 additional textures to a data set results in a significant increase in the number of GP invocations (e.g. all-hard-12 vs. all-hard-16).

5.5.2 GP Invocations per Texture

Figure 5.5 depicts the average, the minimum and the maximum number of GP invocations per texture for each run. Figure 5.5 confirms the difficulty level of the selected textures set. The textures that we categorized as easy have been frequently recognized without much problem. The textures in the hard set, however, required more GP invocations. Figure 5.5 also reveals that the number of GP invocations mostly depends on the difficulty of the textures and not the number of textures in the experiment. Note that as (a) suggests, the textures in the all-easy-4 experiment only required a single GP invocation. This fact marks this set as the easiest set of all experiments.

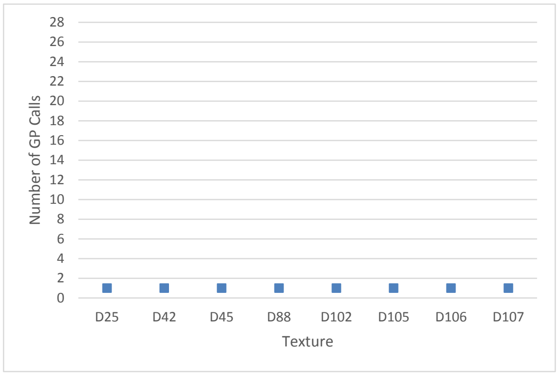
There seems to be a great variance in the number of GP invocations that some textures required. For instance, in (e) the minimum number of invocations for D107 was 1 while the maximum number of invocations was 26. This indicates that for some experiments, D107 was probably confused with other textures that were selected for that experiment.

Figure 5.5 also confirms that the "One" experiments were generally more challenging than the "All" experiments primarily because the textures required more invocations.

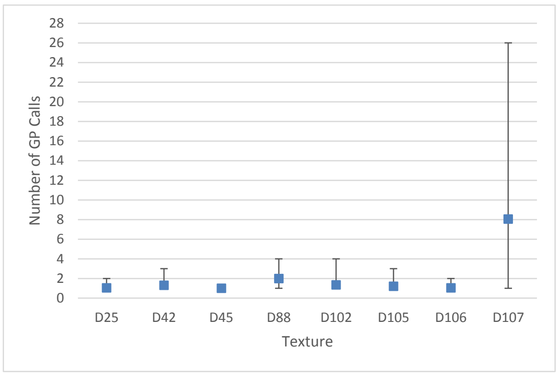
5.5.3 Permanent Orphans

One aspect of the experiments is the likelihood of getting permanent orphans in a run. Table 5.4 summarizes the number of permanent orphans at the end of the run for each experiment. We also calculated the confidence interval for each experiment at 95%. The normalized values have been calculated by dividing the value of average by the number of textures in each experiment. The normalized value is similar to the chance that each texture will be a permanent orphan in each experiment set.

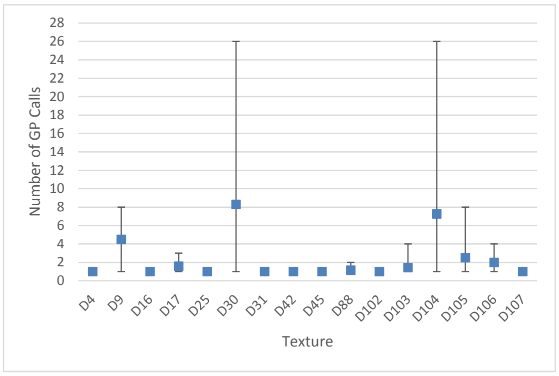
According to Table 5.4, higher number of textures will result in higher number of permanent orphans at the end of the run. An interesting thing to note here is that the likelihood of having a permanent orphan is more dependent on the number of textures in the video feed and less on the difficulty of the textures. In other words, although the all-hard-8 set contains more challenging textures that the all-easy-8 set, they roughly have the same number of permanent orphans at the end of the run. On the other hand, experiment sets which contain higher number of textures will conclude with higher number of permanent orphans. Also note that for the experiments all-hard-12, all-hard-16 and one-hard-16 the number of textures that were distinguished (i.e. non-permanent orphans) is roughly 10. This could



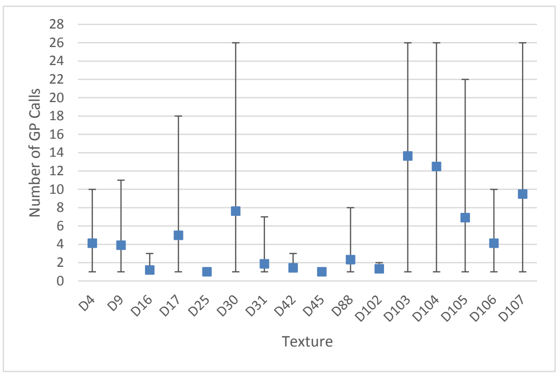
(a) all-easy-4



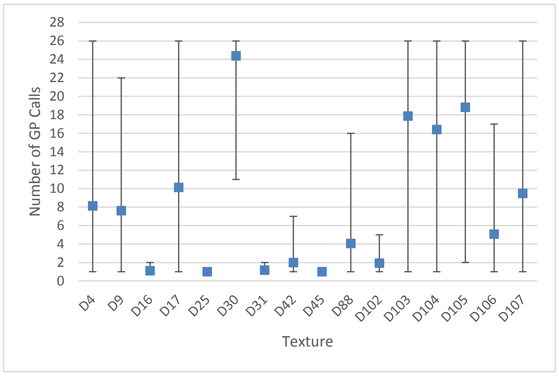
(b) all-easy-8



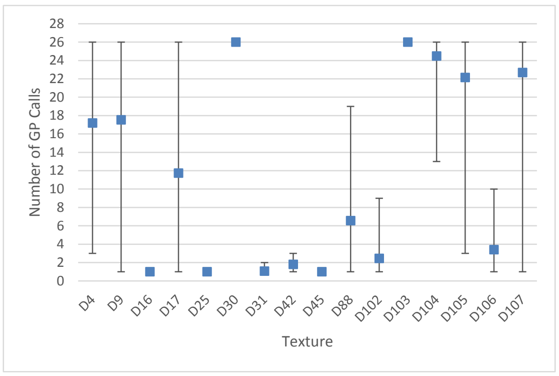
(c) all-hard-4



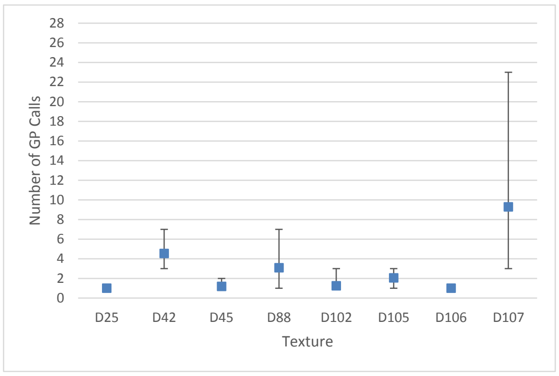
(d) all-hard-8



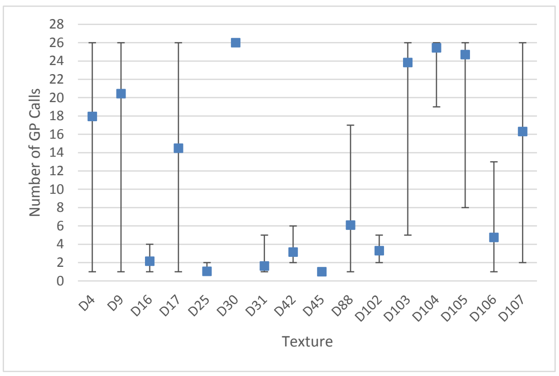
(e) all-hard-12



(f) all-hard-16



(g) one-easy-8



(h) one-hard-16

Figure 5.5: GP Invocations per Texture

Table 5.4: Average number of permanent orphans (95% confidence)

Experiment	Average	Normalized
all-easy-4	0.00 ± 0.00	0.000
all-easy-8	0.10 ± 0.14	0.012
all-hard-4	0.10 ± 0.14	0.025
all-hard-8	0.65 ± 0.38	0.081
all-hard-12	2.20 ± 0.61	0.183
all-hard-16	5.40 ± 0.53	0.337
one-easy-8	0.00 ± 0.00	0.000
one-hard-16	5.15 ± 0.43	0.321

indicate that for the texture data that was used for the experiments, the system can classify at most 10 different textures at the same time.

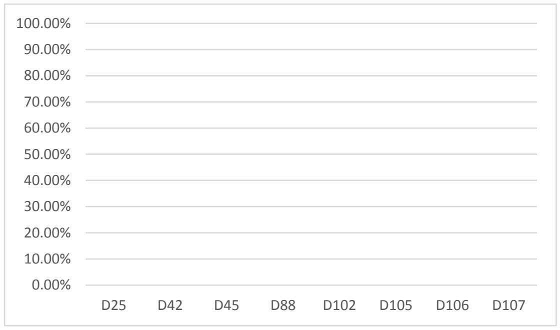
Figure 5.6 depicts the average percentage of time each texture was a permanent orphan during the run. Again, this figure confirms the difficulty of the textures in each set. Also note that some experiments are very unlikely to be concluded with permanent orphans. As (a) and (g) suggest, these experiments never had any permanent orphans in them while other sets (such as (c) or (b)) rarely ended with permanent orphans.

Combining the results of Table 5.3 and Table5.4, we can conclude that although the "one" and the "all" sets had similar number of permanent orphans at the end of the run, the "one" set required more GP invocations than the "all" set. Based on this, we can conclude that the experiments in the "one" set were generally more challenging than the experiments in the "all" set.

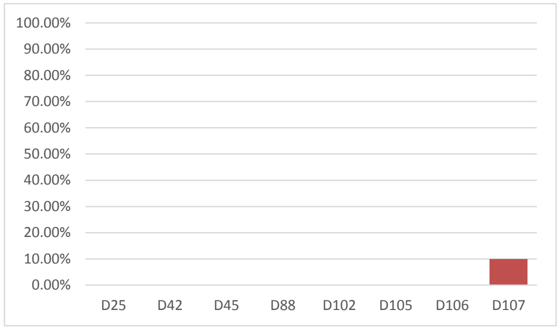
5.5.4 Gradually Added Textures

One relation of interest in the "one" set is that of the number of newly added texture and the number of orphans. Figure 5.7 projects this relation. This figure provides a timeline for the mini-sessions. Each step on the horizontal axis shows the current number of textures in the video. 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.

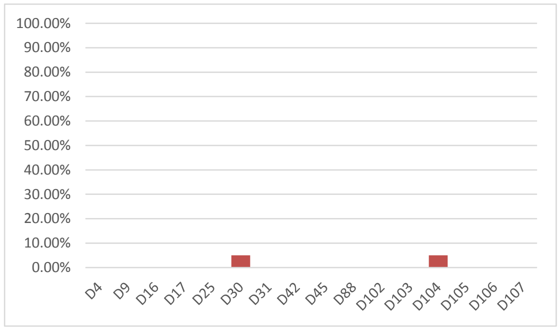
As can be seen in Figure 5.7, the number of orphans that are likely to be encountered increases as the total number of textures in the video increases. This indicates the increase



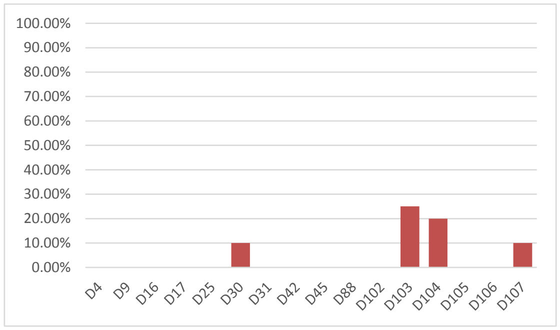
(a) all-easy-4



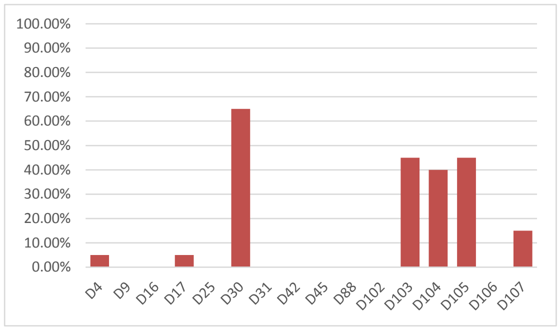
(b) all-easy-8



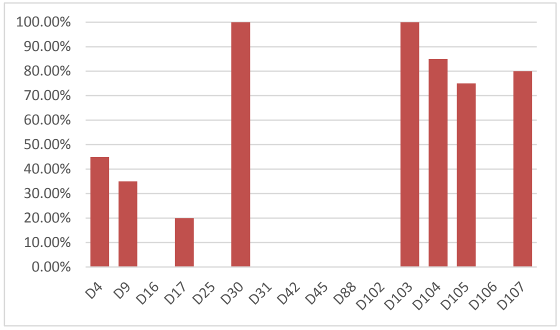
(c) all-hard-4



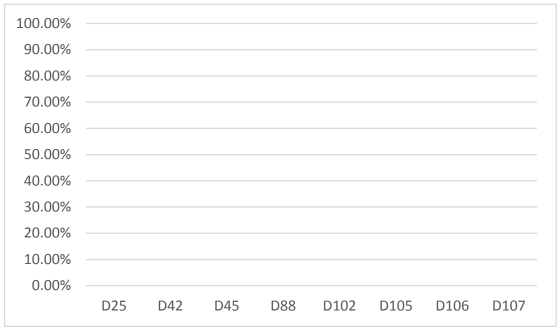
(d) all-hard-8



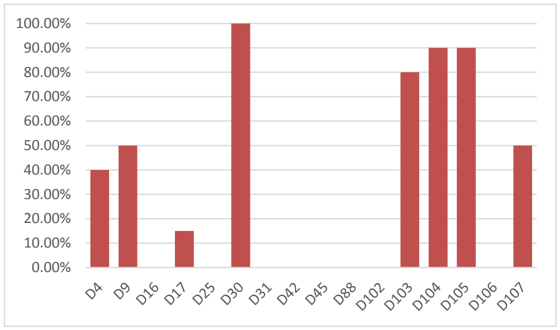
(e) all-hard-12



(f) all-hard-16



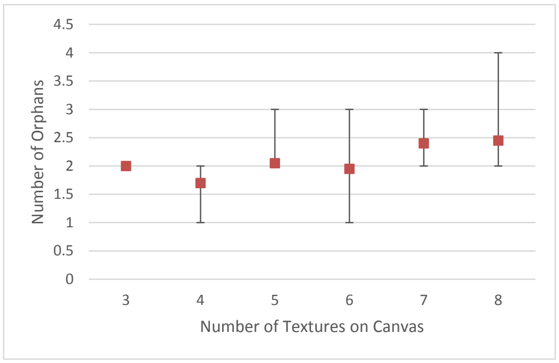
(g) one-easy-8



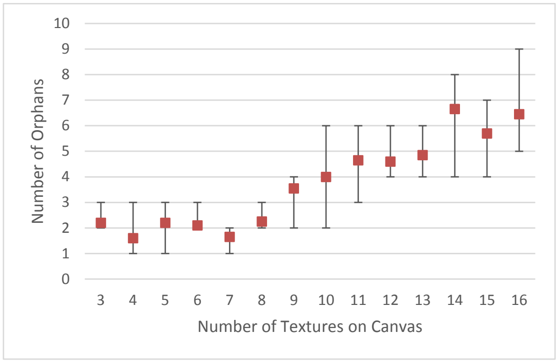
(h) one-hard-16

Figure 5.6: Percentage of Time a Texture was a Permanent Orphan

in the complexity of the search space and the need to evolve more sophisticated classifiers.



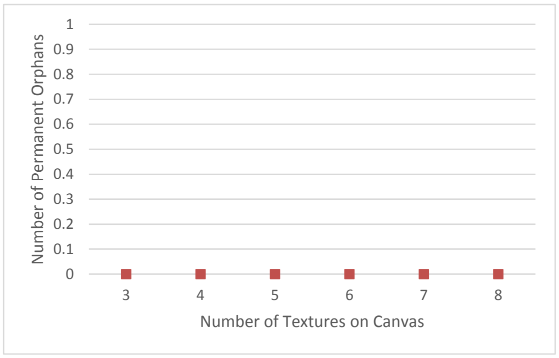
(a) one-easy-8



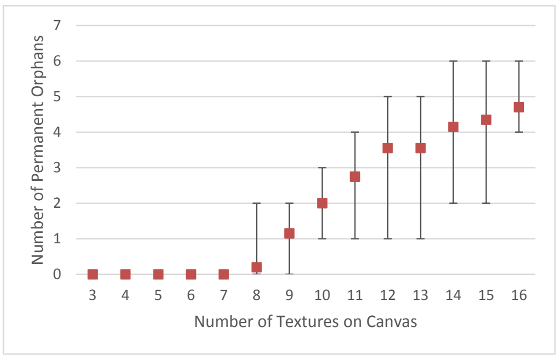
(b) one-hard-16

Figure 5.7: Number of Textures vs. Number of Orphans

Another aspect of the "one" experiment set is the relationship between the number of textures and the number of permanent orphans. Figure 5.8 presents this relationship. The number of permanent orphans in the system increases with the introduction of additional textures. This figure also confirms our previous observations: the system is capable of classifying around 10 different textures at the same time. Again, we stress that this observation is only true for the texture data that we used for these experiments. If other textures are used, the system may behave differently.



(a) one-easy-8



(b) one-hard-16

Figure 5.8: Number of Textures vs Number of Permanent Orphans

5.5.5 Frame Rate and Tree Size

An important aspect of a real-time system is its execution speed. For the developed system this aspect can be measured by 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 for higher number of textures. Each classifier should be executed on all textures so that its claimed texture is recognized. For M textures we require M classifiers and by extension, $M \times M$ classifier executions. Furthermore, with higher number of classifiers the thresholding operation needs to be performed more frequently. Hence, it is expected that the higher number of textures will result in lower frame rate at the end of the run.

Frame rate also depends on the tree size of the evolved classifiers. Larger trees are generally slower to evaluate than smaller trees. Moreover, larger trees will likely require more memory access. Since CUDA is used to evaluate these trees, higher numbers of memory access will affect the performance of the evaluation kernel.

The number of GP invocations also impacts the frame rate during the run. Currently, the same GPU is responsible for evaluating GP individuals as well as running the classifiers on the segments. Therefore, if a particular experiment requires many GP invocations, the frame rate that is measured during the evolutionary run is expected to suffer.

Table 5.5 presents the average frame rate as well as the average 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 call was issued and the GP engine started running. At this point, the GPU will not only be evaluating the fitness of the population but will also visualize the results. The final frame rate was measured when the session was successful and all no further GP calls were necessary.

A glance at this table reveals that GP invocation affects the frame rate since the values in the second column of Table 5.5 are lower than those in the third column. Furthermore, the more difficult a set is, the lower the frame rate is for that set.

For the values in Table 5.5 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, the sign of the correlation coefficient is negative, indicating higher frame rates are generally attained with lower tree sizes. However, the strength of the relationship between the frame rate and the tree size varies for each experiment. For sets with lower number of textures (e.g. for (a)) this relationship is strong whereas the relationship is weak for experiments that include higher number of textures (e.g. for (f)).

Table 5.5 also shows that the number of textures in each experiment affects both of the measured frame rates. In experiments with higher number of textures (e.g. all-hard-16) the average frame rate is much lower than the experiments with fewer number of textures (e.g. all-easy-4). This coincides with what one would expect because for more segments, there are more trees that require evaluation and this slows down the system.

To factor out the effect of the number of segments on the measured frame rate at the end of the run, we have normalized the values of the final frame rate of the system based on the values measured for experiments with 16 textures. In other words, if an experiment has 8 segments, its frame rate has been divided by 2 since the experiment with 16 segments has 2 times as many segments as the experiment with 8 segments. Using the normalized values we can see that easier experiments have higher normalized values. Moreover, the effects of the tree size can be better fathomed using the normalized values. Generally, experiments with lower average tree size have higher normalized values.

Table 5.5: 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

Using the normalized values in Table 5.5, we performed a two-sample t-test for all the possible pairs of experiments with 95% significance level based on 20 observations. Table 5.6 summarizes the results of these t-tests. For each pair, an arrow is used indicate if there was a significant difference between the pairs of data. The arrow points to the experiment set that performed better.

According to Table 5.6, all-easy-4 and all-hard-4 were the easiest while the one-easy-8 and one-hard-16 were the hardest experiments.

Figure 5.9 contains the plots of frame rate versus average tree size for each experiment. The trend line has also been added to the graphs to depict the relationship between the two. According to Figure 5.9, the performance is on-par with real-time requirements. Note

Table 5.6: Significance analysis of the normalized frame rates (95% significance). The arrow points to the faster experiment in each pair, a hyphen indicates no significant difference)

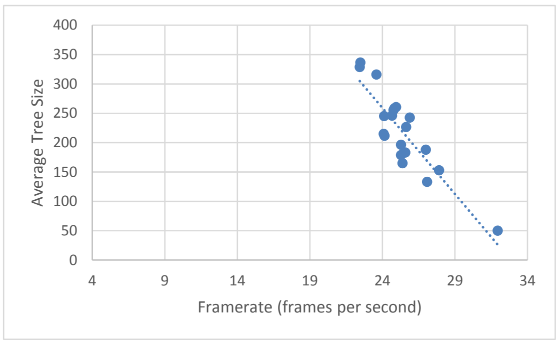
	all-easy-4	all-easy-8	all-hard-4	all-hard-8	all-hard-12	all-hard-16	one-easy-8
all-easy-8	↑						
all-hard-4	–	←					
all-hard-8	↑	–	↑				
all-hard-12	↑	↑	↑	↑			
all-hard-16	↑	–	↑	–	←		
one-easy-8	↑	–	↑	–	←	↑	
one-hard-16	↑	–	↑	–	←	–	–

that the GPU processes every single frame and for each texture in the frame, 6 relatively large[2] spatial filters are calculated on the fly. The filter calculations are coupled with tree evaluations as well. For M textures, M$\times$M classifiers are executed on 96$\times$96 pixels. In case there are permanent orphans in the system, their total numbers must be deducted from M. With a few simple calculations, the throughput of the system could be calculated. For the all-hard-16 experiment set, the number of evaluations (per frame) would be 16$\times$(16 - 5.40)$\times$96$\times$96 = 1,563,033.60. Considering that the average frame rate for that experiment is 5.46fps, the system is carrying out 8,534,163.456 tree evaluations per second. This throughput is ignoring the filter calculation overhead as well as any other overhead that is incurred by OpenGL rendering. Considering the average tree size of 240 nodes for each tree, this throughput could be calculated in terms of GP operations[3]. Multiplying the number of tree evaluations per second by the number GP operations in the tree yields the throughput of 2,048,881,962.51 or more than 2 billion GP operations per second. Using similar calculation, the throughput for each experiment set has been calculated and the results are shown in Table 5.7.

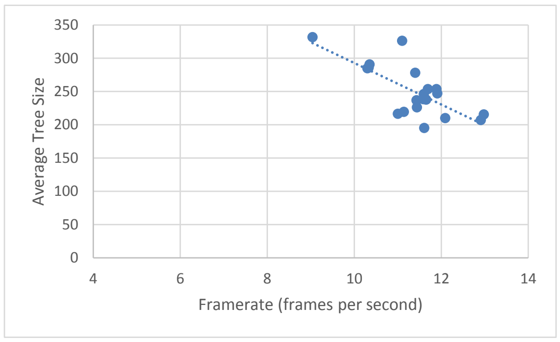
The throughput of the system could be increased by more implementation consideration. As mentioned before, for the sake of simplicity we favoured a more straightforward implementation over a heavily optimized one. Also, the conversion of GP trees to postfix expressions imposes an overhead on the system. By directly evolving trees on the GPU, this overhead would be minimized.

[2]As the size of the kernel of the spatial filters increases the calculations takes much longer

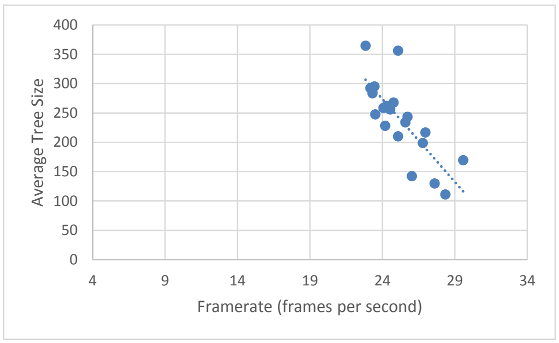
[3]Without knowing the number of leaf nodes in the tree, calculating the throughput in terms of FLOPS is not possible



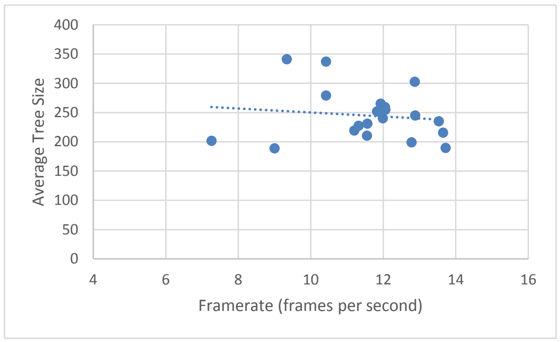
(a) all-easy-4



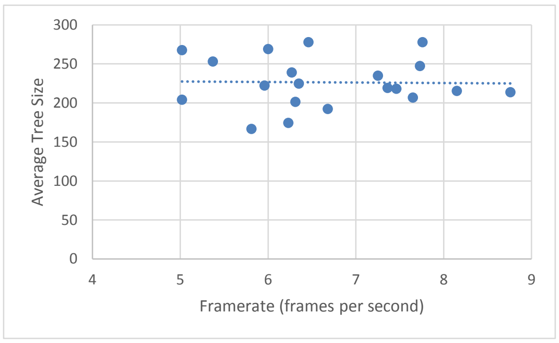
(b) all-easy-8



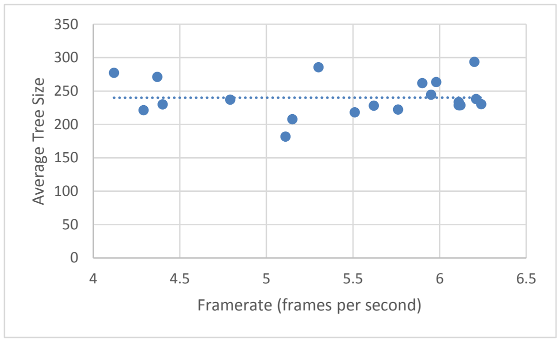
(c) all-hard-4



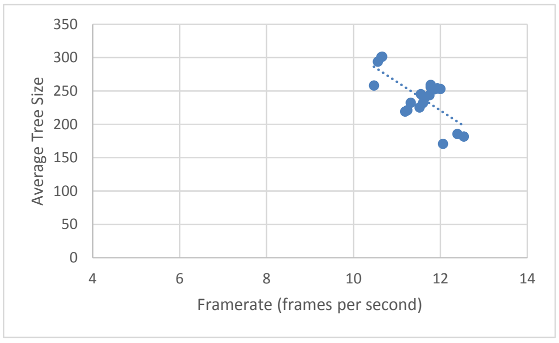
(d) all-hard-8



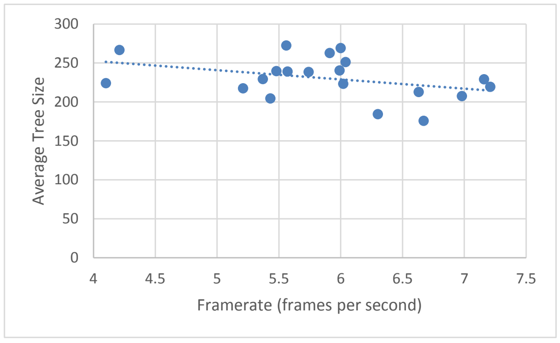
(e) all-hard-12



(f) all-hard-16



(g) one-easy-8



(h) one-hard-16

Figure 5.9: Frame rate vs. average tree size

Table 5.7: The throughput of each experiment

Experiment	Avg. Num. of Perm. Orphans	Throughput (Billion GP Ops/Sec)
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

5.5.6 Speed-up Comparison

In order to better grasp the advantage of using the GPU for tree evaluations, we performed additional runs. The all-easy-8 experiments were redone 40 times and for each generation of the evolution in each run, the tree evaluation time was recorded. The former 20 runs were exactly the same as the ones discussed in Section 5.4.2. The latter 20 runs were done without the aid of CUDA and used the CPU for tree evaluation. Experiment results showed that the runs that were done using CUDA were on average 80 times faster than the runs which used the CPU for evaluation. Figure 5.10 depicts the results of these experiments.

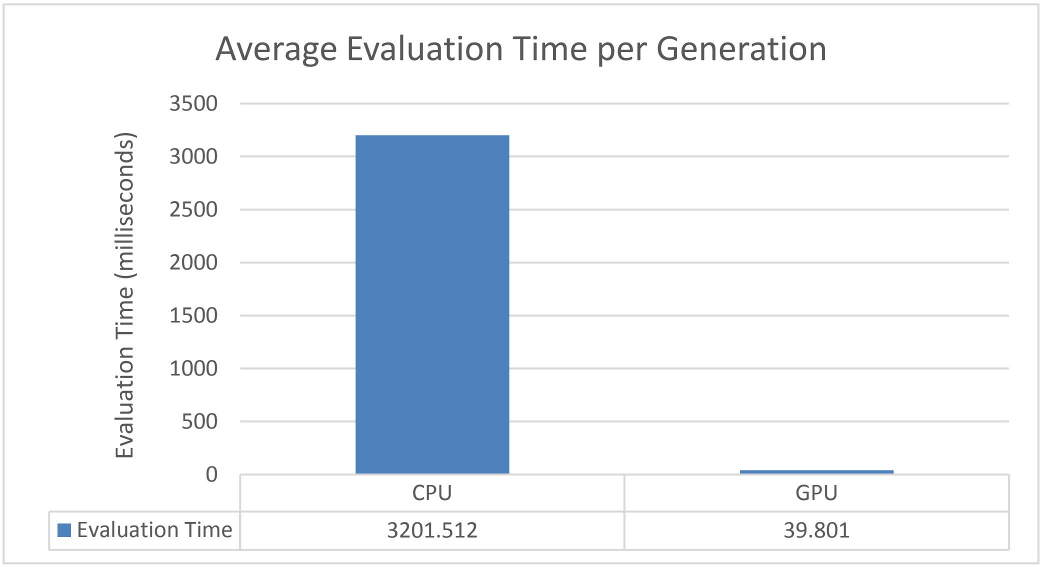


Figure 5.10: Average Evaluation Time per Generation

5.5.7 Eager Termination

In order to study the effects of the number of generations on the performance of the developed system we devised a new mode in the system. We call this mode the *eager termination* mode. In this mode, the maximum number of generations for each GP call is still 100. However, at any time during the evolution if the evolved individual of that generation with the highest fitness rating meets the majority classification condition (i.e. 50% positive classification), the evolution may be stopped. This way, if a good individual is evolved in (say) the 10th generation, that individual is used for the classification purposes.

For a meaningful comparison, we performed the all-hard-8 experiments 20 times with the eager termination option enabled. In addition to the permanent orphan information as well as the frame rate information, we recorded the total execution time of the 20 runs. 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 option enabled (see Table 5.4 for the number of permanent orphans). Conversely, the average frame rate and the execution times were greatly affected. Figure 5.11 summarizes the results of these experiments.

It can be seen that the system with the eager termination option enabled was about 8 times faster than the system without the option enabled. This coincides with what one would expect since for some textures, fewer generations are required for the evolution of a suitable classifier. The average frame rate of the system has also improved. This could be attributed to the lower sizes of the evolved trees. Fewer generations result in smaller evolved trees with less bloat. In other words, the trees that are evolved in the early generations are usually simpler than the ones evolved in the later generations which means that they are easier for being parsed and executed using CUDA.

5.6 Comparison

Research regarding real-time evolution of GP systems for computer vision applications is very scarce. Most of the previous efforts in the literature involve the application of an evolved solution in real-time. This kind of application is relatively straightforward as the solutions are evolved only once and executed many times. Furthermore, no additional training is performed in such systems which weakens their applicability to real-time scenarios.

Among the researches that we examined, [76, 80, 63] were focused on moving object analysis. Smart and Zhang [76] 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 which limits the applicability of the evolved solutions. Their system was trained using a few training images. After this phase, no additional training was done. This inherently differs from our research as our main goal was to create an online evolutionary system which would be able to correct its behavior even after the initial evolution has concluded. Moreover, their evolved solutions were only capable of detecting the moving objects and were unable to discriminate different moving objects from one another. This is another issue that our research aims to address. Similarly, the motion plane features that were used by Song and Fang [80] were unable to discriminate different objects. However, the pixel intensity features in that research was capable of detecting the motions of only a specific object. Pinto and Song [63] were able to use 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), it was incapable of discriminating the objects that belonged to the same class. We anticipate that the implementations in [76, 80, 63] could be used as a pre-processor for our implementation. By detecting object motion, the pre-processor would be able to provide the 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.

As mentioned in Chapter 3, Nordin and Banzhaf [60] 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 is similar to our work, such as multiple invocations of the GP system or real-time evaluation of the fitness using the problem's environment. However, the scale of these two systems are rather different. Their system was built for embedded applications which restricted the available computational power while ours requires an actual CUDA compatible hardware.

Similar to our work, Ebner [17] 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$\times$240 video [16]. One notable difference between his work and ours is the data that is being worked on. His implementation worked by classifying objects while ours depends on textures. A quick glance at the experiment results of Chapter 4 reveals that texture classification problem was generally more challenging that object classification problem. We were able to achieve classification accuracies of above 90% for boats and faces. This accuracy was rarely achieved in the texture classification problem that we studied. Furthermore, 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

	Purpose	**Online**	**Remarks**
Smart & Zhang [76]	Moving object detection	–	No discrimination among various objects
Song & Fang [80]	Moving object detection	–	No discrimination among various objects with some of the used features
Pinto & Song [63]	Moving object detection	–	Unable to discriminate objects of the same class
Nordin & Banzhaf [60]	Robot control	✓	Different scale/purpose Low computational requirement
Ebner [17]	Object detection	✓	Low performance: 320×240@4.5fps

Table 5.8: Comparison of various GP-based object analysis systems

engine. Table 5.6 summarizes the comparison of various GP-based object analysis systems.

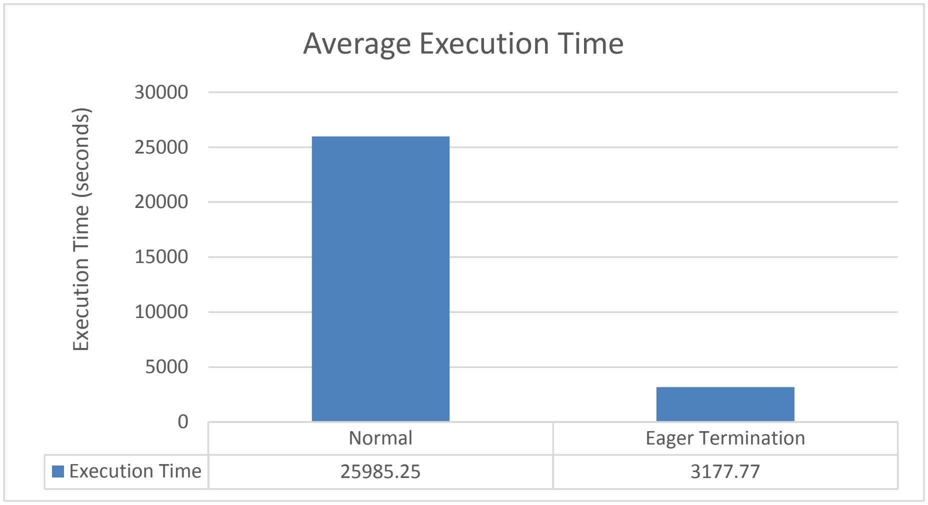
5.7 Conclusion

The system was capable of meeting real-time requirements and in the most difficult experiments achieved a throughput of around 2 billion GP operations per second. The implementation that we used favored simplicity over performance. Accessing the global memory of GPU causes the most significant bottleneck in the current system. By micro-optimizing the kernel code and manually caching the global memory, we could push the performance of the system beyond the current 2 billion GP operations per second. Another way to ameliorate the performance is to process blocks of pixels in the textures rather than process every single pixel. This manner of processing would be similar to the method employed by Song *et al.* [81] and has been proven to speed up the execution.

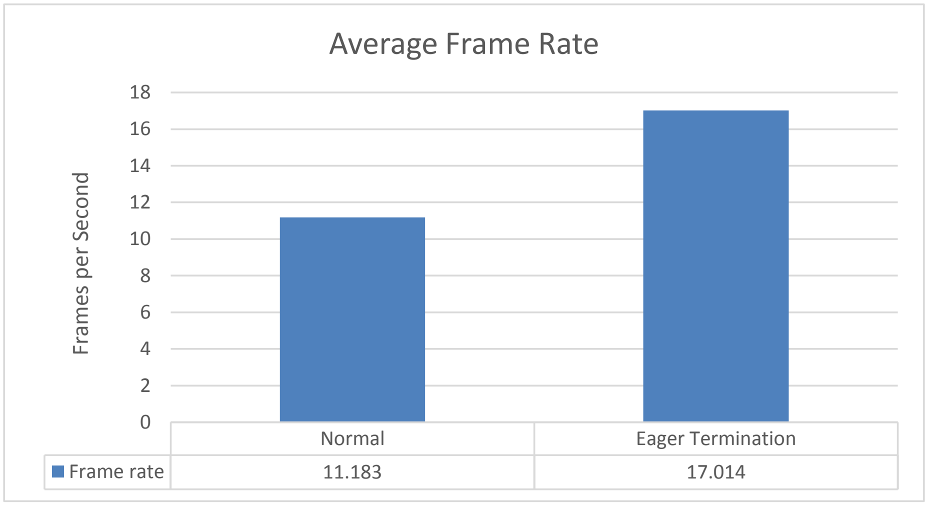
As the results showed, the frame rate of the system mostly depends on the number of textures. The system currently processes every single frame and every single pixel of the segments in that frame. In many applications of such systems, the video is processed at specific key frames. This would decrease the computational load of the system without adversely affecting the final results of the system.

With respect to the automated learning environment that we used, we also showed that the number of GP invocations mostly depends on the number of textures that are used in each experiment and not the difficulty of the textures. Similarly, the likelihood of having permanent orphans in the system depends on the number of textures in the video rather than the overall difficulty of the textures. Having permanent orphans or the need to call GP numerous times are GP classification issues (see Chapter 4). These issues are not directly related to the performance of the real-time system. Any improvements to the GP language will result in improvements to the real-time learning system that we discussed in this chapter.

By comparing the different experiment sets we can conclude that evolving classifiers for textures which are all available at once in the beginning of the run is generally easier than evolving classifiers for textures that are gradually added to the system. The results also indicated that the for the chosen texture data, the system was capable of uniquely classifying around 10 different textures at the same time.



(a) Average execution time



(b) Average frame rate

Figure 5.11: The effects of the eager termination option on the all-hard-8 experiments

Chapter 6

Conclusion and Future Work

This research proposed a real-time object tracking and classification system. The system is GP-based, works in an automatic manner, and is online which means that based on the input that is provided to the system, it can correct its behavior during the run.

Due to the complex nature of the object tracking problem, we implemented a system that works on an OpenGL-generated video. The video contained various number of segments each of which had its own unique texture. The textures were obtained from the Brodatz database. The goal was to evolve a classifier for each segment so that the object could be discerned from all other objects in the video. For implementation, NVIDIA CUDA was used to achieve adequate performance.

Using the results of the available research in the literature, different GP classification languages were studied. These languages were either block-classifiers or pixel-classifiers. Experiments showed that pixel-classifiers were generally more accurate while block-classifiers were generally faster. Inspired by these results, a GP language was selected and the real-time GP system was developed using that language.

To test the capabilities of the developed system, multiple sets of experiments were designed. Experiments introduced various difficulty levels and contained different number of textures. Moreover, for some experiments all of the selected textures were provided to the system at the beginning of the run while for others, only a couple of textures were provided and upon successful evolution of suitable classifiers, additional textures were gradually added to the video.

Experiments showed that the classification performance of the system was adequate. We concluded that we were able to track 10 different segments at the same time. The results also showed that the experiments in which the segments were gradually added to the system were more challenging than the experiments that contained all the segments from the start. Furthermore, we showed that the system had adequate runtime performance and reached the throughput of more than 2 billion GP operations per second.

There are many directions for future works. We assumed objects have constant sizes and they do not occlude one another. A more advanced task would be to classify and track the objects that can scale. For this task, a classifier that is specifically evolved for a texture should be robust enough that would be indifferent to texture scaling. This requires a more sophisticated GP language. In fact, improvement of the classification language is a possible direction for future research. We showed that by improving the classification language, more accurate results could be obtained. A possible means of such improvement is the addition of other image processing features that are standard in the literature. These features could aid the GP system in building more robust classifiers.

Another direction for future research is the incorporation of a segmentation algorithm. By incorporating a segmentation algorithm, the experiments could be performed on real video data. We made a lot of simplifications for the problem. In addition to assuming that all objects have the same shape or are of the same size, we assumed that the location of each segment is known in each frame and that segments have perfect boundaries. In our experiments, objects did not occlude each other and their size remained constant. These are not necessarily the case for real video and a system working on such data would need to work without these assumptions. In a real video, objects can have arbitrary sizes or shapes and can occlude each other. These make segmentation a challenging task. There is no perfect segmentation algorithm. Also, more computational power is required for better segmentation results which may affect the real-time performance. If working on real video is desired, the system should be able to work around the mistakes that the segmentation algorithm has made.

Using motion detectors as a pre-processor for our implementation is another possibility. As described in Section 5.6 the research in [76, 80, 63] was focused on evolving programs that were able to detect object motion in a video. These programs could act as a pre-processor for our classification engine. Using motion information, the processor would supply the GP engine with a list of objects that are currently moving. Using such information, GP is able to evolve a unique classifier for each moving object.

The implementation of the system could also be improved. We preferred a more straightforward implementation over an optimized one. By optimizing the kernel functions, the performance of the system could be pushed beyond the 2 billion GP operation per second. Obviously, this improves the runtime performance and could result in a more robust system. Another possibility is the implementation of the system for use on embedded devices. With the release of CUDA v6.0 and the development of high performance mobile processors such as the NVIDIA Tegra K1 processor, CUDA applications could be ported to mobile processors for embedded applications. In fact, we are currently looking into the ways to port our system for the NVIDIA Jetson TK1 development board. Moreover, one could implement the system using other parallel programming frameworks such as OpenCL. This increases the portability of the system and eliminates the need for specialized or proprietary hardware.

Bibliography

[1] *CUDA C Best Practices Guide: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide*. Accessed: 2014-05-30.

[2] *NVIDA CUDA C Programming Guide: http://docs.nvidia.com/cuda/index.html*. Accessed: 2014-05-30.

[3] Ecj: A java-based evolutionary computation research system. http://cs.gmu.edu/ eclab/projects/ecj/, may 2013. ECJ v21, Accessed: 2014-05-30.

[4] G. S. Almasi and A. Gottlieb. *Highly parallel computing*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1989.

[5] Douglas A Augusto and Helio JC Barbosa. Accelerated parallel genetic programming tree evaluation with opencl. *Journal of Parallel and Distributed Computing*, 73(1):86–100, 2013.

[6] Thomas Back, David B. Fogel, and Zbigniew Michalewicz, editors. *Handbook of Evolutionary Computation*. IOP Publishing Ltd., Bristol, UK, UK, 1st edition, 1997.

[7] Wolfgang Banzhaf, Simon Harding, William B Langdon, and Garnett Wilson. Accelerating genetic programming through graphics processing units. In *Genetic Programming Theory and Practice VI*, pages 1–19. Springer, 2009.

[8] Phil Brodatz. *Textures: a photographic album for artists and designers*, volume 66. Dover New York, 1966.

[9] Alberto Cano, Amelia Zafra, and Sebastián Ventura. Speeding up the evaluation phase of gp classification algorithms on gpus. *Soft Computing*, 16(2):187–202, 2012.

[10] Erick Cantu-Paz. *Designing efficient and accurate parallel genetic algorithms (parallel algorithms)*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1999. AAI9952979.

[11] Erick Cantu-Paz. *Efficient and accurate parallel genetic algorithms*, volume 1. Kluwer Academic Pub, 2000.

[12] A. Cochocki and Rolf Unbehauen. *Neural Networks for Optimization and Signal Processing*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1993.

[13] Pascal Comte. Design & implementation of parallel linear gp for the ibm cell processor. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, GECCO '09, pages 1:1–1:8, New York, NY, USA, 2009. ACM.

[14] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming. *Parallel Computing*, 38(8):391–407, 2012.

[15] Marc Ebner. An adaptive on-line evolutionary visual system. In *Proceedings of the 2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, SASOW '08, pages 84–89, Washington, DC, USA, 2008. IEEE Computer Society.

[16] Marc Ebner. A real-time evolutionary object recognition system. In *Proceedings of the 12th European Conference on Genetic Programming*, EuroGP '09, pages 268–279, Berlin, Heidelberg, 2009. Springer-Verlag.

[17] Marc Ebner. Towards automated learning of object detectors. In *Proceedings of the 2010 international conference on Applications of Evolutionary Computation - Volume Part I*, EvoApplicatons'10, pages 231–240, Berlin, Heidelberg, 2010. Springer-Verlag.

[18] A. De La Escalera, J. M A Armingol, and M. Mata. Traffic sign recognition and analysis for intelligent vehicles. *Image and Vision Computing*, 21:247–258, 2003.

[19] P.G. Espejo, S. Ventura, and F. Herrera. A survey on the application of genetic programming to classification. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 40(2):121–144, 2010.

[20] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[21] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel computing experiences with cuda. *Micro, IEEE*, 28(4):13–27, 2008.

[22] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2001.

[23] Google. Google earth (v6.2) program. http://www.google.com/earth/index.html. Accessed: 2013-01-04.

[24] Alexander Guzhva, Sergey Dolenko, and Igor Persiantsev. Multifold acceleration of neural network computations using gpu. In Cesare Alippi, Marios Polycarpou, Christos Panayiotou, and Georgios Ellinas, editors, *Artificial Neural Networks ICANN 2009*, volume 5768 of *Lecture Notes in Computer Science*, pages 373–380. Springer Berlin Heidelberg, 2009.

[25] Simon Harding. Evolution of image filters on graphics processor units using cartesian genetic programming. In Jun Wang, editor, *2008 IEEE World Congress on Computational Intelligence*, Hong Kong, 1-6 June 2008. IEEE Computational Intelligence Society, IEEE Press.

[26] Simon Harding and Wolfgang Banzhaf. Fast genetic programming on gpus. In *Proceedings of the 10th European conference on Genetic programming*, EuroGP'07, pages 90–101, Berlin, Heidelberg, 2007. Springer-Verlag.

[27] Simon Harding and Wolfgang Banzhaf. Genetic programming on GPUs for image processing. *International Journal of High Performance Systems Architecture*, 1(4):231 – 240, 2008.

[28] Simon Harding and Wolfgang Banzhaf. Distributed genetic programming on gpus using cuda. In Jos L. Risco-Martn and Oscar Garnica, editors, *WPABA'09: Proceedings of the Second International Workshop on Parallel Architectures and Bioinspired Algorithms (WPABA 2009)*, pages 1–10, Raleigh, NC, USA, September 12-16 2009. Universidad Complutense de Madrid.

[29] Simon Harding and Wolfgang Banzhaf. Implementing cartesian genetic programming classifiers on graphics processing units using gpu.net. In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, GECCO '11, pages 463–470, New York, NY, USA, 2011. ACM.

[30] Simon Harding, Jrgen Leitner, and Jrgen Schmidhuber. Cartesian genetic programming for image processing. In Rick Riolo, Ekaterina Vladislavleva, Marylyn D Ritchie, and Jason H. Moore, editors, *Genetic Programming Theory and Practice X*, Genetic and Evolutionary Computation, pages 31–44. Springer New York, 2013.

[31] NealR. Harvey, Simon Perkins, StevenP. Brumby, James Theiler, ReidB. Porter, A. Cody Young, AnilK. Varghese, JohnJ. Szymanski, and JeffreyJ. Bloch. Finding golf courses: The ultra high tech approach. In Stefano Cagnoni, editor, *Real-World Applications of Evolutionary Computing*, volume 1803 of *Lecture Notes in Computer Science*, pages 54–64. Springer Berlin Heidelberg, 2000.

[32] Daniel Howard, Simon C. Roberts, and Richard Brankin. Target detection in sar imagery by genetic programming. *Adv. Eng. Softw.*, 30(5):303–311, May 1999.

[33] Marco Hutter. Jcuda: Java bindings for cuda. http://www.jcuda.org/. Accessed: 2014-05-30.

[34] Nathan Intrator, Daniel Reisfeld, and Yehezkel Yeshurun. Face recognition using a hybrid supervised/unsupervised neural network. *Pattern Recogn. Lett.*, 17(1):67–76, January 1996.

[35] Honghoon Jang, Anjin Park, and Keechul Jung. Neural network implementation using cuda and openmp. In *Computing: Techniques and Applications, 2008. DICTA'08. Digital Image*, pages 155–161. IEEE, 2008.

[36] Kamran Karimi, Neil G. Dickson, and Firas Hamze. A performance comparison of cuda and opencl. *CoRR http://www.arxiv.org/pdf/1005.2581*, abs/1005.2581, 2010. Accessed 2013-05-12.

[37] Kalle Karu and Anil K. Jain. Fingerprint classification. *Pattern Recognition*, 29(3):389–404, 1996.

[38] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Neural Networks, 1995. Proceedings., IEEE International Conference on*, volume 4, pages 1942–1948 vol.4, Nov 1995.

[39] David B Kirk and W Hwu Wen-mei. *Programming massively parallel processors: a hands-on approach, 2nd Edition*. Morgan Kaufmann, 2012.

[40] T. Kowaliw, W. Banzhaf, N. Kharma, and S. Harding. Evolving novel image features using genetic programming-based image transforms. In *Evolutionary Computation, 2009. CEC '09. IEEE Congress on*, pages 2502–2507, May 2009.

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

[42] John R. Koza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[43] Vipin Kumar. *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[44] W. B. Langdon. Source code for gp using gpu. http://www-dept.cs.ucl.ac.uk/staff/w.langdon/ftp/gp-code/gpu_gp_2.tar.gz, 2008.

[45] W. B. Langdon. A many threaded cuda interpreter for genetic programming. In *Proceedings of the 13th European conference on Genetic Programming*, EuroGP'10, pages 146–158, Berlin, Heidelberg, 2010. Springer-Verlag.

[46] W. B. Langdon and Wolfgang Banzhaf. A simd interpreter for genetic programming on gpu graphics cards. In *Proceedings of the 11th European Conference on Genetic Programming*, EuroGP'08, pages 73–85, Berlin, Heidelberg, 2008. Springer-Verlag.

[47] J. Leitner, S. Harding, M. Frank, A. Forster, and J. Schmidhuber. icvision: A modular vision system for cognitive robotics research. In *5th International Conference on Cognitive Systems (CogSys)*, Feb 2012.

[48] Sean Luke. *The ECJ Owner's Manual*. Department of Computer Science, George Mason University, zeroth edition, October 2010.

[49] Sean Luke, Liviu Panait, Gabriel Balan, Sean Paus, Zbigniew Skolicki, Jeff Bassett, Robert Hubley, and A Chircop. Ecj: A java-based evolutionary computation research system. *George Mason University's Evolutionary Computation Laboratory*, 2007.

[50] Mehran Maghoumi and Brian J. Ross. Feature extraction languages and visual pattern recognition. Technical Report CS-14-03, Brock University, jan 2014. http://www.cosc.brocku.ca/sites/all/files/downloads/research/cs1403.pdf.

[51] Stephen Marsland. *Machine Learning: An Algorithmic Perspective*. Chapman & Hall/CRC, 1st edition, 2009.

[52] Jennis Meyer-Spradow and Jörn Loviscach. Evolutionary design of BRDFs. In M. Chover, H. Hagen, and D. Tost, editors, *Eurographics 2003 Short Paper Proceedings*, pages 301–306, 2003.

[53] JulianF. Miller and Peter Thomson. Cartesian genetic programming. In Riccardo Poli, Wolfgang Banzhaf, WilliamB. Langdon, Julian Miller, Peter Nordin, and TerenceC. Fogarty, editors, *Genetic Programming*, volume 1802 of *Lecture Notes in Computer Science*, pages 121–132. Springer Berlin Heidelberg, 2000.

[54] Volodymyr Mnih and Geoffrey E. Hinton. Learning to detect roads in high-resolution aerial images. In *Proceedings of the 11th European conference on Computer vision: Part VI*, ECCV'10, pages 210–223, Berlin, Heidelberg, 2010. Springer-Verlag.

[55] D.J. Montana. Strongly Typed Genetic Programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[56] Luca Mussi, Stefano Cagnoni, Elena Cardarelli, Fabio Daolio, Paolo Medici, and PierPaolo Porta. Gpu implementation of a road sign detector based on particle swarm optimization. *Evolutionary Intelligence*, 3(3-4):155–169, 2010.

[57] Luca Mussi, Fabio Daolio, and Stefano Cagnoni. Evaluation of parallel particle swarm optimization algorithms within the cuda architecture. *Information Sciences*, 181(20):4642–4657, 2011.

[58] Luca Mussi, Spela Ivekovic, and Stefano Cagnoni. Markerless articulated human body tracking from multi-view video with gpu-pso. In *Proceedings of the 9th international conference on Evolvable systems: from biology to hardware*, ICES'10, pages 97–108, Berlin, Heidelberg, 2010. Springer-Verlag.

[59] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008.

[60] Peter Nordin and Wolfgang Banzhaf. Real time control of a khepera robot using genetic programming. *Control and Cybernetics*, 26:533–562, 1997.

[61] MahamedG.H. Omran, AndriesP. Engelbrecht, and Ayed Salman. Particle swarm optimization for pattern recognition and image processing. In Ajith Abraham, Crina Grosan, and Vitorino Ramos, editors, *Swarm Intelligence in Data Mining*, volume 34 of *Studies in Computational Intelligence*, pages 125–151. Springer Berlin Heidelberg, 2006.

[62] Y. Owechko, S. Medasani, and N. Srinivasa. Classifier swarms for human detection in infrared imagery. In *Computer Vision and Pattern Recognition Workshop, 2004. CVPRW '04. Conference on*, pages 121–127, 2004.

[63] Brian Pinto and Andy Song. Motion detection in complex environments by genetic programming. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, GECCO '09, pages 2125–2130, New York, NY, USA, 2009. ACM.

[64] Riccardo Poli. Genetic programming for feature detection and image segmentation. In TerenceC. Fogarty, editor, *Evolutionary Computing*, volume 1143 of *Lecture Notes in Computer Science*, pages 110–125. Springer Berlin Heidelberg, 1996.

[65] Riccardo Poli. Genetic programming for image analysis. In *Proceedings of the First Annual Conference on Genetic Programming*, pages 363–368. MIT Press, 1996.

[66] Riccardo Poli, W William B Langdon, and Nicholas F McPhee. *Field Guide to Genetic Programming*. Lulu Enterprises Uk Limited, 2008.

[67] Riccardo Poli and G. Valli. Hopfield neural nets for the optimum segmentation of medical images. In E. Feisler and R. Beale, editors, *Handbook of Neural Computation*. Oxford University Press, 1996.

[68] Petr Pospichal, Jiri Jaros, and Josef Schwarz. Parallel genetic algorithm on the cuda architecture. In *Proceedings of the 2010 international conference on Applications of Evolutionary Computation - Volume Part I*, EvoApplicatons'10, pages 442–451, Berlin, Heidelberg, 2010. Springer-Verlag.

[69] Parag Puranik, Preeti Bajaj, Ajith Abraham, Prasanna Palsodkar, and Amol Deshmukh. Human perception-based color image segmentation using comprehensive learning particle swarm optimization. In *Emerging Trends in Engineering and Technology (ICETET), 2009 2nd International Conference on*, pages 630–635. IEEE, 2009.

[70] Denis Robilliard, Virginie Marion, and Cyril Fonlupt. High performance genetic programming on gpu. In *Proceedings of the 2009 workshop on Bio-inspired algorithms for distributed systems*, BADS '09, pages 85–94, New York, NY, USA, 2009. ACM.

[71] Denis Robilliard, Virginie Marion-Poty, and Cyril Fonlupt. Population parallel gp on the g80 gpu. In *Proceedings of the 11th European conference on Genetic programming*, EuroGP'08, pages 98–109, Berlin, Heidelberg, 2008. Springer-Verlag.

[72] B.J. Ross, A.G. Gualtieri, F. Fueten, and P. Budkewitsch. Hyperspectral Image Analysis Using Genetic Programming. *Applied Soft Computing*, 5(2):147–156, 2005.

[73] Claude Sammut and Geoffrey I. Webb. *Encyclopedia of Machine Learning*. Springer Publishing Company, Incorporated, 1st edition, 2011.

[74] Claude Sammut and Geoffrey I. Webb. *Encyclopedia of Machine Learning*. Springer Publishing Company, Incorporated, 1st edition, 2011.

[75] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[76] Will Smart and Mengjie Zhang. Tracking object positions in real-time video using genetic programming. In *Proceeding of Image and Vision Computing International Conference*, pages 113–118, 2004.

[77] A. Song. *Texture Classification: a Genetic Programming Approach*. PhD thesis, RMIT University, April 2003.

[78] A. Song and V. Ciesielski. Fast texture segmentation using genetic programming. In *Evolutionary Computation, 2003. CEC '03. The 2003 Congress on*, volume 3, pages 2126–2133 Vol.3, 2003.

[79] A. Song and V. Ciesielski. Texture analysis by genetic programming. In *Evolutionary Computation, 2004. CEC2004. Congress on*, volume 2, pages 2092–2099 Vol.2, 2004.

[80] Andy Song and Danny Fang. Robust method of detecting moving objects in videos evolved by genetic programming. In *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*, GECCO '08, pages 1649–1656, New York, NY, USA, 2008. ACM.

[81] Andy Song, Thomas Loveard, and Vic Ciesielski. Towards genetic programming for texture classification. In Markus Stumptner, Dan Corbett, and Mike Brooks, editors, *AI 2001: Advances in Artificial Intelligence*, volume 2256 of *Lecture Notes in Computer Science*, pages 461–472. Springer Berlin Heidelberg, 2001.

[82] Richard Szeliski. *Computer vision: Algorithms and applications*. Springer, 2010.

[83] Walter Alden Tackett. Genetic programming for feature discovery and image discrimination. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 303–311, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[84] Walter Alden Tackett. Genetic programming for feature discovery and image discrimination. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 303–311, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[85] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: simplified programming of graphics processing units for general-purpose uses via data-parallelism. Technical report, Microsoft Research, MSR-TR-2005-184, 2005.

[86] R. Uetz and S. Behnke. Large-scale object recognition with cuda-accelerated hierarchical neural networks. In *Intelligent Computing and Intelligent Systems, 2009. ICIS 2009. IEEE International Conference on*, volume 1, pages 536–541, 2009.

[87] G. Wilson and W. Banzhaf. Linear genetic programming GPGPU on microsoft's xbox 360. In *CEC 2008*, pages 378–385, 2008.

[88] Jay F. Winkeler and B.S. Manjunath. Genetic programming for object detection. In *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 330–335. Morgan Kaufmann, 1997.

[89] Masayuki Yanagiya. Efficient genetic programming based on binary decision diagrams. In *CEC, 1995.*, volume 1, pages 234–246. IEEE, 1995.

[90] Alper Yilmaz, Omar Javed, and Mubarak Shah. Object tracking: A survey. *ACM Comput. Surv.*, 38(4), December 2006.

[91] Sifa Zhang and Zhenming He. Implementation of parallel genetic algorithm based on cuda. In *Proceedings of the 4th International Symposium on Advances in Computation and Intelligence*, ISICA '09, pages 24–30, Berlin, Heidelberg, 2009. Springer-Verlag.

[92] Xin Zhang, Yee-Hong Yang, Zhiguang Han, Hui Wang, and Chao Gao. Object class detection: A survey. *ACM Comput. Surv.*, 46(1):10:1–10:53, July 2013.

Appendix A

Best Evolved Solutions

The best evolved solutions for some of the experiments in Chapter 4 are presented below.

Listing A.1: Texture 1 (Complete)

```
(− (+ (min (min (/ (min (max 0.25587136 0.13157782) (+ (min (/ (min (+ (sin (∗ (max
    0.25587136 SmallAvg BW)) (/ 7 (GridSmallAvg BW 25 (∗ 16 0.5167))))) (min (/ (min (
    max 0.25587136 0.13157782) (LargeAvg BW)) (+ 0.037915953 (GridSmallAvg BW
    25 (∗ 16 0.5167)))) (SmallAvg BW))) (LargeAvg BW)) (SmallAvg BW)) (sin (/ (min (
    max 0.25587136 0.13157782) (/ (min (LargeAvg BW) (MediumSd BW)) (SmallAvg
    BW))) (min (GridLargeAvg BW (∗ (SmallSd BW) (max 0.25587136 0.13157782)) 19)
    (− (+ (sin (LargeAvg BW)) (LargeAvg BW)) (SmallSd BW)))))) (− (GridLargeSd BW (/
     (∗ (max 0.5409668 0.72863007) (GridSmallAvg BW 16 12)) (− (exp 0.508604) (
    LargeSd BW))) 22)))) (SmallAvg BW)) (+ 0.037915953 (min (min (/ (min (+ (min (/ (
    min (max 0.25587136 0.13157782) (LargeAvg BW)) (+ 0.037915953 (GridSmallAvg
    BW 25 (∗ 16 0.5167)))) (SmallAvg BW)) (− (LargeSd BW))) (SmallSd BW)) (
    SmallAvg BW)) (SmallAvg BW)) (GridSmallAvg BW 29 (∗ 7 (/ 7 (GridSmallAvg BW 25
     (∗ 16 0.5167)))))))) (+ 0.037915953 (min (min (/ (min (+ (min (/ (min (max
    0.25587136 0.13157782) (min (SmallAvg BW) (MediumSd BW))) (GridSmallAvg BW
    29 30)) (min (/ (min (max 0.25587136 0.13157782) (− (LargeAvg BW) (SmallSd BW)
    )) (SmallAvg BW)) (+ 0.037915953 (SmallAvg BW)))) (− (SmallSd BW))) (SmallSd
    BW)) (SmallAvg BW)) (SmallAvg BW)) (GridSmallAvg BW 29 (/ 7 (SmallAvg BW))))))
    (− (LargeSd BW))) 0.20875005)
```

Listing A.2: Texture 1 (No Offset)

```
(min (− (− (+ (∗ (− (− (+ (+ (max (∗ (LargeAvg BW) (MediumAvg BW)) (− (LargeSd BW)
    (MediumAvg BW))) (LargeSd BW)) (∗ (MediumAvg BW) (LargeAvg BW))) (
    MediumAvg BW))) (∗ (exp (sin (/ (max (− (max 0.8812905 0.23369434)) (− (+ (
    LargeSd BW) 0.6977402) (MediumAvg BW))) (LargeSd BW)))) (− 0.38508502))) (
    LargeSd BW)) (MediumAvg BW))) (min (− (max (sin (/ (− 0.12895341 (LargeAvg BW
    )) 0.12895341)) (max (− (− (max (sin (/ (− 0.12895341 (SmallAvg BW)) 0.12895341)
    ) (− (+ (max (∗ (SmallAvg BW) (MediumAvg BW)) (− (LargeSd BW) (MediumAvg BW
    ))) (LargeSd BW)) (MediumAvg BW))))) (− (+ (max (∗ (LargeAvg BW) (MediumAvg
    BW)) (− (LargeSd BW) (MediumAvg BW))) (LargeSd BW)) (MediumAvg BW))))) (− (
    sin (sin (∗ (exp (LargeSd BW)) (min (min (exp 0.6881742) (sin (/ (SmallAvg BW)
    0.12895341))) 0.12895341)))))))
```

Listing A.3: Texture 1 (Raw Features)

```
(IfGT (/ 0.3172324 (sin (sin (+ (max 0.98145795 0.7297438) (− (− (/ (exp 0.5542153) (−
    0.19313586))) (+ (exp (InputColor BW)) (/ (max 0.27591744 0.2910664) (−
    0.4115312)))))))) (∗ (cos (IfGT (− 3 0.40140963) (− (IfGT (InputColor BW)
    0.019957572 (sin (sin (sin (+ (max 0.98145795 0.7297438) (− (− (/ (exp 0.5542153)
    (− 0.19313586))) (+ (exp (InputColor BW)) (/ (max 0.27591744 0.2910664) (−
    0.4115312)))))))) (min (− 0.30420583) 0.6619685)) (InputColor BW)) (IfGT
    0.35978308 0.31461608 0.250934 0.7107353) (min 0.3172324 0.73327917))) (+ (sin
    0.5912478) (− (− (/ (exp 0.5542153) (− 0.19313586))) (+ (exp (InputColor BW)) (/ (
    max 0.27591744 0.2910664) (− 0.4115312)))))) (− (− (/ (exp 0.5542153) (−
    0.19313586))) (∗ (IfGT (InputColor BW) 0.019957572 (sin (min (max (− 0.28632703)
    (+ 24 0.27350658)) (exp (cos 0.594859)))) (min (− 0.30420583) 0.6619685)) (exp (
    sin 0.7004277)))) (+ (InputColor BW) (− (− (∗ (/ (cos 0.41280842) (InputColor BW)) (
    min (− (IfGT 0.024303162 0.4653577 0.883157 0.49006754)) 0.73327917)) (IfGT (
    sin (InputColor BW)) (IfGT (max 0.41771677 0.40548402) (+ (InputColor BW) (− (
    IfGT (InputColor BW) (IfGT 0.30142406 0.187506 (∗ (cos (sin 0.5912478))
    0.8892455) 0.018796612) (− 0.43561268) (sin 0.7004277)) (cos (+ (InputColor BW)
    (− (/ 0.3172324 (sin (sin (max 0.40760446 0.9557756)))) (cos (− 0.43561268))))))) (
    max 0.16322891 0.56093717) (InputColor BW)) 0.3172324 (IfGT (exp (cos (max (
    InputColor BW) (− (cos 0.5574061) (IfGT 0.024303162 0.4653577 0.883157
    0.49006754))))) (+ (− (+ (min 0.89282537 0.4114711) (min 0.017604753 (− (− (/ (
    exp 0.5542153) (− 0.19313586))) (/ 1 (∗ (− (IfGT 0.22313471 0.5432788 0.7739634
    0.683356)) (min (InputColor BW) (− 16 9)))))))) 0.48381236) (− (min (− (IfGT
    0.35978308 0.31461608 0.250934 0.7107353) (exp 0.12654763)) (sin 0.5912478)) (
    cos (sin 0.014672659))) (cos 0.36419708)))) (cos (− 0.43561268)))))
```

Listing A.4: Texture 1 (Block Proc.)

```
(+ (∗ (∗ (∗ (− (+ Att(216) Att(75)) (− (− (∗ (− (+ (∗ (− Att(580) (− (∗ (∗ Att(635) Att(35)) (−
     Att(458) Att(735))) (∗ (+ Att(531) Att(980)) (+ Att(542) Att(955))))) (∗ (+ Att(168) Att
    (935)) (+ Att(727) Att(47)))) (− (∗ (∗ Att(858) Att(855)) Att(489)) (If (>= Att(44) Att
    (165)) Att(769) Att(374)))) (∗ Att(1014) Att(791))) (∗ (+ Att(738) Att(168)) (∗ (+ Att(26)
    Att(553)) (+ Att(840) Att(202))))) Att(262)) (+ Att(286) Att(822)))) (∗ (+ Att(220) Att
    (556)) (+ Att(490) Att(449)))) (∗ (− Att(488) (− (∗ (+ Att(547) Att(369)) (− (∗ (If (>= Att
    (502) Att(122)) Att(101) Att(361)) Att(793)) (∗ (+ Att(883) Att(947)) (+ Att(551) Att
    (263))))) (∗ (+ Att(903) Att(268)) (+ Att(693) Att(684))))) (+ Att(951) Att(135)))) (∗ (∗ (∗
    (∗ (+ Att(127) Att(390)) (∗ (+ Att(427) Att(956)) (∗ (+ Att(866) Att(23)) (∗ (+ Att(282) Att
    (33)) (∗ (+ Att(624) (+ (+ (∗ Att(911) (+ Att(237) Att(943))) (− (∗ (+ Att(617) Att(98)) Att
    (832)) (If (>= Att(920) Att(395)) Att(760) Att(24)))) (∗ (+ Att(490) Att(127)) (∗ (+ Att
    (160) Att(766)) (+ Att(82) Att(265)))))) (∗ (− (+ Att(409) Att(76)) (− (∗ (If (>= Att(918)
    Att(375)) Att(56) Att(874)) Att(852)) (∗ (+ Att(526) Att(951)) (+ Att(83) Att(401))))) (∗ (+
     Att(75) Att(570)) (+ Att(580) Att(119))))))))) (+ (+ (∗ (− Att(470) (− (∗ (+ Att(585) Att
    (564)) (− Att(509) Att(756))) (∗ (+ Att(75) Att(207)) (+ Att(445) Att(726))))) (+ Att(632)
    Att(626))) (− (∗ (+ Att(776) Att(995)) (+ Att(331) (+ (∗ (+ Att(629) Att(455)) (+ Att(245)
    (+ Att(379) Att(540)))) (If (>= Att(283) (0.3056736309156052)) Att(754) Att(872))))) (If
     (>= Att(761) Att(277)) Att(806) Att(236)))) (∗ (+ Att(180) Att(409)) (∗ Att(10) (+ Att
    (409) Att(5)))))) (+ Att(163) Att(402))) (∗ (− (+ Att(684) Att(522)) (− (∗ (If (>= Att(29)
    Att(838)) Att(119) Att(72)) Att(335)) (∗ (+ Att(585) Att(842)) (+ Att(6) Att(101))))) (∗ (∗
    (∗ (∗ (+ Att(387) Att(306)) (∗ (+ Att(279) Att(940)) (∗ (+ Att(927) Att(889)) (∗ (+ Att
    (643) (∗ Att(128) (+ Att(429) (+ Att(110) Att(466))))) (+ Att(893) Att(1014)))))) (+ Att
    (57) Att(396))) (+ (∗ Att(52) (0.2626698437092614)) (∗ (+ (+ (+ Att(341) (+ Att(510)
    Att(402))) (+ (+ Att(254) (+ Att(301) Att(230))) Att(292))) Att(429)) (+ Att(111) Att(389)
    )))) (∗ (∗ (∗ (∗ (+ Att(138) Att(664)) (∗ (+ Att(423) Att(594)) (∗ (+ Att(574) Att(653)) (+
    Att(75) Att(97))))) (+ Att(279) (+ Att(608) Att(46)))) (+ (∗ Att(8)
    (−0.04808655152155117)) (∗ (+ (+ Att(73) (+ Att(460) Att(32))) Att(281)) (+ Att(40)
    Att(94))))) (+ (+ Att(207) (+ Att(85) Att(226))) Att(218))))))) (− (/ Att(996) (If (= Att(449)
     Att(585)) Att(367) Att(825))) Att(51)))
```

Listing A.5: Boats (No Offset)

```
(/ (min 0.32039502 0.59971356) (max (IfGT (∗ (exp (− (MediumSd G) (IfGT (InputColor B
    ) (SmallSd G) 0.25834635 (MediumAvg B)))) (SmallSd R)) (LargeAvg G) (IfGT
    0.18081726 (LargeSd R) (LargeAvg G) (IfGT 0.18081726 (LargeSd G) (− (− (− (− (
    SmallSd B) (min 0.32039502 (− (min 0.32039502 0.59971356) (min 0.19697833
    0.54763806))))) (min 0.19697833 0.54763806)) 7) (min 0.32039502 (− (max (max (
    IfGT (∗ (+ (sin 0.76269704) (SmallAvg G)) (MediumAvg B)) (LargeAvg G) (exp (
    LargeSd B)) (− (− (SmallSd B) (min 0.19697833 0.54763806)))) (/ (InputColor B) (
    cos (− (SmallSd R))))) (− 0.239065 (max (SmallAvg G) (IfGT (SmallSd B) (− (
    SmallSd BW)) (− (MediumSd R) (LargeSd R)) (sin 0.76269704))))) (min 0.19697833
    0.54763806))))) (− (/ (LargeSd B) (max (IfGT (∗ (+ (∗ (+ (sin 0.76269704) (InputColor
     B)) (MediumAvg B)) (LargeAvg G)) (max (IfGT (∗ (exp (− (MediumSd R)
    0.18114069)) (MediumAvg B)) (LargeAvg G) (cos (SmallSd BW)) (− (− (MediumSd
    G) (min 0.19697833 0.54763806)))) (− (MediumAvg B)))) (LargeAvg G) (IfGT
    0.18081726 (exp (− (SmallSd G) (min 0.19697833 0.54763806))) (− 0.18114069) (
    SmallAvg G)) (− (max (max (IfGT (∗ (+ (sin 0.76269704) (InputColor B)) (MediumAvg
     B)) (LargeAvg G) (exp (sin (MediumAvg B))) (− (− (SmallSd B) (min 0.32039502 (−
    (min 0.32039502 0.59971356) (min 0.19697833 0.54763806)))))) (/ (InputColor B) (
    cos (− (SmallSd R))))) (− 0.239065 29)) (min (sin 0.5713174) (cos 0.8419428)))) (−
    (∗ (max (InputColor B) 0.89565814) (MediumAvg B))))) (IfGT (LargeSd G) (SmallSd
    G) 0.25834635 0.4666736))) (− (∗ (min 0.19697833 0.54763806) (SmallSd B)))))
```

Listing A.6: Boats (Block Proc.)

```
(+ (If (Between (/ AttBlue(914) AttRed(22)) (/ AttGreen(797) AttGreen(478)) (+ (If (
    Between (If (>= AttBlue(446) AttBlue(752)) AttRed(898) AttGreen(147)) (/ AttGreen
    (819) AttGreen(722)) (− AttBlue(555) AttGreen(38))) (+ (+ AttBlue(390) AttBlue(439))
     (If (>= (0.8642865530237154) AttGreen(627)) AttRed(671) AttRed(778))) (− (/
    AttBlue(1022) AttRed(362)) (/ AttRed(546) (0.4375763735052711)))) (+ (If (Between
    (If (>= AttBlue(512) AttBlue(1)) AttRed(673) AttGreen(328)) (/ AttGreen(351)
    AttGreen(450)) (− AttBlue(220) AttGreen(475))) (+ (+ AttBlue(782) AttBlue(1018)) (+
    AttRed(471) AttRed(776))) (+ (If (Between (If (>= AttBlue(626) AttBlue(504)) AttRed
    (826) AttGreen(538)) (/ AttGreen(617) AttGreen(72)) (− AttBlue(219) AttGreen(339)))
     (+ (/ AttGreen(477) (/ AttRed(561) AttGreen(671))) (+ AttRed(453) AttRed(21))) (− (+
     (− AttBlue(985) AttRed(35)) (If (Between (0.4608019303248223) AttGreen(677)
    AttGreen(120)) (∗ (− (∗ AttRed(584) AttBlue(655)) AttBlue(192)) AttBlue(448)) (−
    AttBlue(143) AttGreen(823)))) (If (Between (0.40944523214730255) AttRed(217)
    AttGreen(776)) (−0.48116696774714174) AttRed(399)))) (+ (If (Between (If (>=
    AttBlue(125) AttBlue(486)) AttRed(423) AttGreen(719)) (/ AttGreen(463) AttGreen
    (595)) (− AttBlue(231) AttGreen(109))) (+ (+ AttBlue(472) AttBlue(26)) (If (>=
(−0.19685687651893802) AttGreen(702)) AttRed(716) AttRed(589))) (− (/ AttBlue(359)
    AttRed(220)) (/ AttRed(629) (0.5101751992467203)))) (+ (If (Between (If (>= AttBlue
    (13) AttBlue(7)) AttRed(558) AttGreen(871)) (/ AttGreen(403) AttGreen(650)) (−
    AttBlue(513) AttGreen(215))) (+ AttBlue(709) (+ AttRed(54) AttRed(563))) (− (/
    AttBlue(864) AttRed(19)) (If (Between (−0.7883957014264851) AttRed(494)
    AttGreen(556)) (0.8325241089401507) AttRed(504)))) (+ (− AttBlue(967) AttRed(16)
    ) (− (+ (− AttBlue(252) AttRed(483)) (If (Between (−0.7743022172034295) AttGreen
```

(992) AttGreen(610)) (∗ (− (∗ AttRed(869) AttBlue(662)) AttBlue(788)) AttBlue(910))
(− AttBlue(501) AttGreen(756)))) (If (Between (0.2231492323396118) AttRed(358)
AttGreen(836)) (−0.8812526230449083) AttRed(730)))))))) (If (Between (If (>=
AttBlue(707) AttBlue(225)) AttRed(111) AttGreen(102)) (/ AttGreen(16) AttGreen
(226)) (− AttBlue(507) AttGreen(969))) (+ (If (>= AttBlue(64) AttBlue(382)) AttRed
(929) AttGreen(310)) (+ AttRed(165) AttRed(195))) (− (/ AttBlue(139) AttRed(28)) (If
(Between (0.7167730627345168) AttRed(375) AttGreen(416))
(−0.6710763806359519) AttRed(744))))))) (+ (+ AttBlue(395) AttBlue(393)) (+ AttRed
(96) AttRed(789))) (− (+ (If (Between (If (>= AttBlue(386)
AttBlue(117)) AttRed(121) AttGreen(534)) (/ AttGreen(278) AttGreen(125)) (− AttBlue
(982) AttGreen(195))) (+ (+ AttBlue(753) AttBlue(930)) (+ AttRed(251) AttRed(264)))
(− (+ (+ (If (Between (If (>= AttRed(692) AttBlue(917)) (− AttBlue(267) AttRed(881)
) (If (>= AttBlue(52) AttRed(833)) (If (Between AttRed(947) AttGreen(181) AttBlue
(585)) AttBlue(13) AttBlue(631)) (− AttBlue(841) AttRed(164)))) (If (Between
(−0.39919222323477177) AttRed(720) AttGreen(552)) (−0.5059131871833056)
AttRed(500)) (/ AttBlue(346) AttRed(313))) (+ (+ AttBlue(214) AttBlue(75)) (If (>=
(0.5717531097947044) AttGreen(111)) AttRed(988) AttRed(313))) (+ (If (Between (
If (>= AttBlue(962) AttBlue(442)) AttRed(614) AttGreen(892)) (/ AttGreen(628)
AttGreen(374)) (− AttBlue(48) AttGreen(564))) (+ (+ AttBlue(962) AttBlue(177)) (+
AttRed(676) AttRed(920))) (− (/ AttBlue(155) AttRed(425)) (If (Between
(0.30822415075949205) AttRed(898) AttGreen(712)) (0.5979795677596398)
AttRed(953)))) AttGreen(49))) (+ (If (Between (If (>= AttBlue(167) AttBlue(204))
AttRed(522) AttGreen(833)) (/ AttGreen(392) AttGreen(709)) (− AttBlue(404)
AttGreen(72))) (+ (+ AttBlue(521) AttBlue(238)) (+ AttRed(525) AttRed(247))) (− (/
AttBlue(343) AttRed(17)) (If (Between (0.5074934129497561) AttRed(466) AttGreen
(587))
(−0.6621277029714565) AttRed(884)))) (If (Between (−0.46922396685697243)
AttGreen(689) AttGreen(166)) (∗ (0.14618744006119044) AttBlue(373)) (− AttBlue
(584) AttGreen(354))))) (If (>= AttBlue(657) AttRed(606)) (+ (If (>= AttRed(661) (−
AttBlue(613) AttRed(20))) (− AttBlue(248) AttRed(913)) (/ AttGreen(483) (/ AttRed
(157) AttGreen(619)))) (+ (− AttBlue(979) AttRed(623)) (If (= AttRed(505) AttBlue
(177)) (∗ (−0.9858450882054335) AttBlue(537)) (− AttBlue(944) AttGreen(176)))))
(− AttBlue(63) AttGreen(84)))) (If (Between (0.9243155021285949) AttRed(355)
AttGreen(665)) (−0.9257837489315537) AttRed(876)))) (+ (− AttBlue(603) AttRed
(190)) (If (Between (0.39736836943903886) AttGreen(950) AttGreen(699)) (∗ (− (∗
AttRed(939) AttBlue(822)) AttBlue(871)) AttBlue(245)) (− AttBlue(742) AttGreen
(713))))) (If (Between (0.5844946512604088) AttRed(668) AttGreen(862))
(0.38066035277206756) AttRed(1023)))) (+ AttBlue(176) (+ (If (Between (If (>=

```
        AttBlue(844) AttBlue(728)) AttRed(17) AttGreen(305)) (/ AttGreen(127) AttGreen
        (500)) (− AttBlue(475) AttGreen(953))) (+ (+ AttBlue(426) AttBlue(438)) (+ AttRed
        (575) AttRed(696))) (+ (If (Between (If (>= AttBlue(800) AttBlue(740)) AttRed(765)
        AttGreen(134)) (/ AttGreen(1021) AttGreen(813)) (− AttBlue(445) AttGreen(260)))
        (+ (/ AttGreen(388) (/ AttRed(835) AttGreen(797))) (+ AttRed(360) AttRed(567))) (−
         (+ (− AttBlue(784) AttRed(856)) (If (Between (0.8504126889212269) AttGreen
        (955) AttGreen(841)) (∗ (− (∗ AttRed(457) AttBlue(106))
    AttBlue(836)) AttBlue(419)) (− AttBlue(238) AttGreen(373)))) (If (Between
         (−0.5062279328155481) AttRed(443) AttGreen(964)) (−0.3585972224028242)
         AttRed(208)))) (+ (If (Between (If (>= AttBlue(909) AttBlue(812)) AttRed(122)
         AttGreen(446)) (/ AttGreen(875) AttGreen(444)) (− AttBlue(435) AttGreen(61))) (+
          (+ AttBlue(636) AttBlue(709)) (If (>= (0.506033942003973) AttGreen(573))
         AttRed(803) AttRed(308))) (− (/ AttBlue(379) AttRed(43)) (/ AttRed(691)
         (0.25160538108247743)))) (+ (If (Between (If (>= AttBlue(418) AttBlue(626))
         AttRed(478) AttGreen(309)) (/ AttGreen(606) AttGreen(516)) (− AttBlue(1018)
         AttGreen(457))) (+ AttBlue(316) (+ AttRed(164) AttRed(0))) (− (/ AttBlue(870)
         AttRed(798)) (If (Between (0.2559386405628019) AttRed(111) AttGreen(302))
         (0.2878164932456684) AttRed(471)))) (If (Between
  (0.11393750520390689) AttGreen(960) AttGreen(568)) (∗ (−0.22343014945372985)
        AttBlue(366)) (− AttBlue(484) AttGreen(526))))))) (If (Between (If (>= AttBlue(506)
        AttBlue(163)) AttRed(998) AttGreen(151)) (/ AttGreen(96) AttGreen(824)) (−
        AttBlue(361) AttGreen(353))) (+ (If (>= AttBlue(866) AttBlue(365)) AttRed(767)
        AttGreen(797)) (+ AttRed(29) AttRed(703))) (− (+ (+ (If (Between (If (>= AttBlue
        (418) AttBlue(981)) AttRed(807) AttGreen(472)) (/ AttGreen(641) AttGreen(1016))
        (− AttBlue(223) AttGreen(820))) (+ (+ AttBlue(171) AttBlue(108)) (If (>=
        (−0.8623329900056751) AttGreen(268)) AttRed(551) AttRed(415))) (− (/ AttBlue
        (267) AttRed(480)) (/ AttRed(284) (−0.4663958323190218)))) (+ (If (Between (If
        (>= AttBlue(243) AttBlue(401))
    AttRed(169) AttGreen(793)) (/ AttGreen(428) AttGreen(798)) (− AttBlue(474)
         AttGreen(604))) (+ (+ AttBlue(42) AttBlue(814)) (+ (+ AttBlue(687) (/
         (−0.7142708998033824) (+ AttRed(344) (−0.8666838513652819)))) AttRed(939)
         )) (− (/ AttBlue(69) AttRed(784)) (If (Between (−0.46286589155371916) AttRed
         (215) AttGreen(495)) (−0.7789330897495346) AttRed(582)))) (If (Between
         (−0.1696545544052137) AttGreen(990) AttGreen(677)) (∗
         (0.6316166105268348) AttBlue(71)) (− AttBlue(485) AttGreen(707))))) AttBlue
         (501)) (If (Between (0.23947949894684717) AttRed(534) AttGreen(545))
         (−0.20519087271114) AttRed(772)))))))
```

Listing A.7: Face (No Offset)

```
(IfGT (sin (SmallSd BW)) (IfGT (LargeAvg R) (MediumAvg BW) (exp (max 0.39245507
    0.7760926)) (LargeSd BW)) (– 0.5118121) (IfGT (sin 0.22295298) (IfGT (IfGT (sin
    0.22295298) (IfGT (LargeAvg R) (MediumAvg BW) (exp (– 0.5118121)) (LargeSd
    BW)) (– 0.5118121) (IfGT (sin 0.22295298) (IfGT (IfGT (sin 0.22295298) (IfGT (
    LargeAvg R) (MediumAvg BW) (exp (+ (exp (exp 0.853431)) (+ (LargeSd BW) (
    LargeAvg G)))) (LargeSd BW)) (– 0.5118121) (IfGT (sin 0.22295298) (IfGT (
    MediumAvg BW) (MediumAvg G) (+ (IfGT (sin 0.22295298) (IfGT (LargeAvg R) (
    MediumAvg BW) (exp (+ (max 0.99123424 0.7760926) (SmallSd BW))) (LargeSd BW
    )) (– 0.5118121) (IfGT (sin (MediumSd G)) (IfGT (– 0.5118121) (MediumAvg G) (+
    (+ (LargeAvg R) 0.19481498) (SmallSd BW)) (LargeSd BW)) (– 0.5118121) (∗ (max
    0.39245507 0.7760926) (LargeAvg BW)))) (SmallSd BW)) (LargeSd BW)) (–
    0.5118121) (+ (max 0.99123424 0.7760926) (SmallSd BW)))) (MediumAvg G) (+ (
    LargeAvg BW) (SmallSd BW)) (LargeSd BW)) (– 0.5118121) (∗ (max 0.39245507
    0.7760926) 0.7760926))) (MediumAvg BW) (sin 0.22295298) (– (SmallSd BW))) (–
    0.5118121) (max (SmallSd BW) 0.7760926)))
```

Listing A.8: Face (Block Proc.)

```
(If (= (If (= (/ AttRed(192) AttGreen(540)) (/ AttGreen(660) AttGreen(951))) (/ AttGreen
    (517) AttGreen(676)) (+ (– AttBlue(463) AttRed(340)) (+ AttRed(380) AttBlue(831))))
    (If (= (/ AttRed(764) AttGreen(256)) (– (/ AttBlue(971) AttBlue(43)) (/ AttBlue(346)
    AttBlue(937)))) (If (= (/ AttRed(293) AttGreen(82)) (– (/ AttBlue(254) AttBlue(932)) (/
    AttBlue(500) AttBlue(473)))) (/ AttBlue(787) AttGreen(488)) (+ (– AttBlue(364)
    AttRed(1012)) (+ AttRed(406) AttBlue(625)))) (+ (– AttBlue(545) AttRed(364)) (+
    AttRed(395) AttBlue(268))))) (If (= (–0.9093787106426408) AttBlue(663)) (– AttRed
    (405) AttBlue(282)) (If (= (/ AttRed(225) AttGreen(848)) (/ AttGreen(156) AttGreen
    (698))) (If (>= (– AttRed(403) AttGreen(848)) AttGreen(141)) (If (= (/ AttRed(550)
    AttGreen(642)) (+ (– AttBlue(680) AttRed(198)) (+ AttRed(501) AttBlue(885))))
    AttGreen(706) AttBlue(623)) (– (If (= (/ AttRed(666) AttGreen(519)) (/ AttGreen(898)
    AttGreen(864))) (If (= (/ AttRed(131) AttGreen(115)) (/ AttGreen(932) AttGreen(532)))
     (– (If (= (If (= (/ AttRed(938) AttGreen(1014)) (/ AttGreen(104) AttGreen(68))) (/
    AttBlue(243) AttGreen(317)) (+ (+ (/ (+ (– AttBlue(996) AttRed(232)) (+ AttRed(346)
    AttRed(814))) (– AttGreen(125) AttRed(832))) (+ (∗ AttGreen(939) AttBlue(305)) (–
    AttBlue(131) AttRed(794)))) (+ AttRed(346) AttBlue(296)))) (/ AttGreen(341) AttGreen
    (996))) (If (= (0.22762937599716082) AttBlue(522)) (– AttRed(405) AttBlue(282)) (If
    (= (/ AttRed(72) AttGreen(918)) (/ AttGreen(303) AttGreen(503))) (– (If (= (/ AttRed
    (460) AttGreen(418)) (/ AttGreen(104) AttGreen(657))) (/ (– AttRed(43) AttBlue(89))
    (0.8060862594850176)) (+ (– AttBlue(994) AttRed(755)) (+ AttRed(90) AttRed(709))
    )) (+ (/ (− AttBlue(801) (−0.7681504245834911)) (/ AttRed(702) AttGreen(111))) (+
    (∗ AttGreen(317) AttBlue(194)) (− AttBlue(708) AttRed(394))))) (+ (− AttBlue(434)
    AttRed(380)) (+ AttRed(846) AttBlue(706))))) (+ (If (= AttRed(1010)
    (0.9522145893154064)) (0.0) AttBlue(520)) (+ AttRed(957) AttBlue(546)))) (+ (/ (If (
    Between AttBlue(845) AttGreen(849) AttBlue(926)) AttBlue(439) AttRed(767)) (−
    AttGreen(30) (− AttRed(574) AttGreen(34)))) (+ (∗ AttGreen(341) AttBlue(280)) (−
    AttBlue(855) AttRed(755))))) (+ (− AttBlue(202) AttRed(830)) (+ AttRed(707) AttBlue
    (549)))) (+ (− AttBlue(78) AttRed(607)) (+ (If (= (/ AttRed(187) AttGreen(880)) (/
    AttGreen(70) AttGreen(411))) (/ AttBlue(631) AttGreen(0)) (+ (− AttBlue(25) AttRed
    (361)) (+ AttRed(709) AttBlue(185)))) AttBlue(680)))) (+ (/ (− (/ (/ AttRed(145)
    AttGreen(695)) (− AttGreen(370) AttBlue(756))) (/ AttGreen(755) AttGreen(438))) (−
    (/ (/ AttBlue(880) AttGreen(1004)) (If (= AttBlue(754) AttGreen(467)) AttGreen(178)
    AttRed(28))) (/ (0.5862547844867398) AttRed(494)))) (+ (∗ AttGreen(912) AttBlue
    (783)) (− AttBlue(313) AttRed(125)))))) (+ (− AttBlue(638) AttRed(746)) (+ AttRed
    (833) AttBlue(308))))) (+ (If (= AttRed(518) (−0.2665281166389988)) (0.0) AttBlue
    (587)) (+ AttRed(431) AttBlue(728))))
```