UNIVERSIDADE Đ
COIMBRA

Francisco José Rodrigues Baeta

GENETIC PROGRAMMING IN GRAPHIC PROCESSING UNITS WITH TENSORFLOW

Dissertation in the context of the Master in Informatics Engineering, Specialization in Intelligent Systems, advised by Professor João Nuno Gonçalves Costa Cavaleiro Correia and Professor Tiago Filipe dos Santos Martins and presented to the Faculty of Sciences and Technology / Department of Informatics Engineering.

October 2020

This page is intentionally left blank.

Faculty of Sciences and Technology

Department of Informatics Engineering

Genetic Programming in Graphic Processing Units with TensorFlow

Francisco José Rodrigues Baeta

Dissertation in the context of the Master in Informatics Engineering, Specialization in Intelligent Systems, advised by Professor João Nuno Gonçalves Costa Cavaleiro Correia and Professor Tiago Filipe dos Santos Martins and presented to the Faculty of Sciences and Technology / Department of Informatics Engineering.

October 2020

1 2 9 0

UNIVERSIDADE Ð COIMBRA

This page is intentionally left blank.

Abstract

Genetic Programming (GP) is an automatic problem-solving technique inspired by nature, that optimizes solutions through the evolution of a population of computer programs [39]. This process of optimization, although computationally expensive, is also classified as “embarrassingly parallel”. For many years researchers attempted to exploit this tendency towards concurrency by employing various types of parallel programming paradigms. Particularly, vectorization of fitness evaluation data points was shown to benefit from the use of commercially available parallel architectures such as the Graphics Processing Unit (GPU), to speed up many GP related systems.

In this work, we take advantage of the TensorFlow framework to investigate the benefits of applying this data vectorization method to different scenarios and determine respective GPU performance gains over a Central Processing Unit (CPU). For this purpose, an independent GP engine was developed, TensorGP, along with a testing suite to extract comparative results across different iterative and vectorized approaches.

Symbolic regression problems and expression-based image evolution were used to validate system functionality. Our performance benchmarks demonstrate that by exploiting the TensorFlow execution model, performance gains of up to two orders of magnitude can be achieved on the GPU when compared to a CPU, for the vectorization of large problem domains. Furthermore, exploratory experimentation in the context of evolutionary art illustrates the engine’s feasibility towards artistic production.

Keywords

GP-GPU, Genetic Programming, TensorFlow, Vectorization, Parallel, Evolutionary Art

This page is intentionally left blank.

Resumo

Inspirada pela natureza, a Programação Genética (GP) constitui um método automático de resolução de problemas que otimiza soluções através da evolução de um conjunto de programas de computador [39]. Este processo de otimização, apesar de computacionalmente dispendioso, é também extremamente paralelizável. Ao longo dos anos, vários modelos de programação foram aplicados a GP de modo a aproveitar este potencial de paralelização. Mais especificamente, foi demonstrado que a vetorização de dados relativos à avaliação de aptidão dos indivíduos em GP beneficia do uso de processadores com capacidades de paralelização como as Unidades de Processamento Gráfico (GPUs).

No presente trabalho recorremos à plataforma TensorFlow para investigar as vantagens da aplicação deste método de vetorização a vários problemas de GP, assim como para analisar ganhos de desempenho da GPU sobre o CPU. Para este propósito, foi desenvolvido um motor de GP independente, TensorGP, assim como um ambiente de testes com o objetivo de extrair resultados comparativos entre diversas abordagens tanto vetorizadas como iterativas.

Problemas de regressão simbólica e evolução de imagens foram usados para validar o correto funcionamento das funcionalidades implementadas. Os resultados experimentais obtidos demonstram que tirando partido do modelo de execução do TensorFlow para a vetorização de domínios de avaliação extensos, ganhos de desempanho até duas ordens de grandeza são alcançáveis em GPU comparativamente com o CPU. Para além do mais, testes exploratórios realizados no âmbito de arte evolucionária comprovam a viabilidade do motor para efeitos de produção artística.

Palavras-Chave

GP-GPU, Programação Genética, TensorFlow, Vetorização, Paralelização, Arte Evolucionária

This page is intentionally left blank.

Acknowledgements

To my advisor, Professor João Nuno Gonçalves Costa Cavaleiro Correia and second advisor, Professor Tiago Filipe dos Santos Martins I would like to thank the knowledge and insight bestowed upon me. Without their permanent guidance and availability this work would not have been completed.

To my parents, for their unconditional love and support, as well as remaining family members and friends that, even if indirectly, helped me throughout this project.

This page is intentionally left blank.

Contents

Acronyms

AI Artificial Intelligence. 1

API Application Programming Interface. 20

AVA Atomic Visual Action. 56, 58

AVX Advanced Vector Extensions. 33, 34

Cg C for Graphics. 15

CGP Cartesian Genetic Programming. 11, 14

CNN Convolutional Neural Network. 56

CPU Central Processing Unit. iii, 2, 13–16, 19, 32–35, 40–43, 48, 50, 51, 53, 56, 58, 61, 62

CSV Comma Separated Values. 36

CUDA Compute Unified Device Architecture. 21, 32–34, 62

DAG Directed Acyclic Graph. 12, 21, 22, 48

EA Evolutionary Algorithm. 5–7

EC Evolutionary Computation. 1, 3, 5–9, 13, 48

FPGA Field Programmable Gate Arrays. 14

GP Genetic Programming. iii, 1–3, 5, 7–16, 18–20, 23, 28, 35, 36, 40, 44, 47, 48, 50, 51, 55, 57, 61, 62

GPops Genetic Programming operations per second. 13, 14

GPU Graphics Processing Unit. iii, 1, 2, 13–16, 19, 21, 22, 32, 33, 35, 36, 40–43, 48–51, 53–58, 61, 62

HPC High Performance Computing. 11

HSV Hue Saturation Value. 30

JPEG Joint Photographic Experts Group. 41, 44, 61

Mops Million of operations per second. xvii, 34, 35

MSE Mean Squared Error. 41, 42, 45

MTD Maximum Tree Depth. 8

NaN Not a Number. 51

NFL No Free Lunch. 6

NIMA Neural Image Assessment. 56–58

PDGP Parallel Distributed Genetic Programming. 11

RAM Random Access Memory. 12

RGB Red-Green-Blue. 19, 30, 56

RHH Ramped Half-and-Half. 22, 23, 49

RMSE Root Mean Square error. 45–47

RPN Reverse Polish Notation. 14

SIMD Single Instruction Multiple Data. 14, 32–34

SPMD Single Program Multiple Data. 13

SSE Streaming SIMD Extensions. 14

SSIM Structure Similarity Index Measure. 45–47

VRAM Video Random Access Memory. 33, 49, 51, 58

XLA Accelerated Linear Algebra. 35

This page is intentionally left blank.

List of Figures

This page is intentionally left blank.

List of Tables

This page is intentionally left blank.

Chapter 1

Introduction

In recent years, the growth of Artificial Intelligence (AI) research and development enabled the implementation of increasingly sophisticated systems. However, the growing complexity that these problems incur also leads to a higher degree of criticality and an increased need for computational resources. For instance, autonomous vehicles require the correct identification of countless objects as fast as possible in order to avoid obstacles and prevent accidents. This need for evaluation speed has been a constant throughout the area of computational intelligence, of which Evolutionary Computation (EC) is a field. Inspired by Darwin's theory of natural evolution, EC aims to evolve candidate solutions in a continuous process of stochastic optimization.

In this work, we investigate some methodologies capable of speeding up the evolutionary process with a special focus on the step of assessing how fit a candidate solution is. This step, referred to as fitness evaluation, is a dominant concern to the performance of all genetic algorithms. Particularly, Genetic Programming (GP), which targets the evolution of actual computer programs, is known to require large amounts of computation resources, as all individuals in the population need to be executed and tested against the objective. As a result, fitness evaluation is widely regarded as the most computationally costly operation in GP for most practical applications [18].

Despite this high demand for computational resources, GP is beyond doubt a powerful evolutionary technique, as it is capable of tackling every problem solvable by a computer program without the need for domain-specific knowledge. This domain independence is a desirable property as no assumptions need to be made in advance about the form or structure of the optimal solution [39]. Additionally, because the same program needs to be evaluated for all data points in our problem domain, the process of fitness assessment itself holds great parallelization potential. Consequently, many methods have been proposed to take advantage of parallel hardware in an attempt to speedup this process.

The last decade saw the exponential growth of computing power proposed by Gordon Moore back in 1965 [36] start to break down. As we start meeting the limits of physics, further scaling in frequency and power becomes commercially unfeasible for the microprocessor industry, forcing a paradigm shift towards multi-core computing and parallelization. With the rise of parallel computing, devices such as the Graphics Processing Unit (GPU) have become ever more readily available. GPUs are designed with a throughput-oriented architecture in mind that maximizes data processed per unit of time, allowing for multiple tasks to be executed simultaneously. Whilst these devices are commonly targeted towards personal computer gaming, the high floating-point performance they offer can be applied to various high-performance scientific tasks, GP included.

One way to parallelize the fitness phase is by vectorizing the set of data points to be evaluated. In this data parallel approach, the evaluation works on the full array of fitness cases, performing a vector or tensor operation for each function in our GP program. These tensor operations are highly optimized on GPUs as they are necessary for the various stages of the graphical rendering pipeline. Therefore, it makes sense to couple this data vectorization approach with a GPU architecture. Tensor operations are highly optimized on GPUs as they are necessary for the various stages of the graphical rendering pipeline. Therefore, it makes sense to couple this data vectorization approach with a GPU architecture.

A number of interpreted languages like Matlab, Perl and Python already possess frameworks specifically designed to deal with vectorized operations. Namely, TensorFlow is a numerical library for the Python programming language that has proven handy for the study and representation of a wide spectrum of machine learning problems as it seemingly distributes computational efforts [1]. However, there are clear benefits of exploring the vectorization capacity of this framework to speed up GP. Staats et al. [46] have done precisely that by using TensorFlow to benchmark various GP related classification problems on both GPU and Central Processing Unit (CPU) architectures with the aid of the KarooGP engine. Their experimental results clearly demonstrate the feasibility of this framework for fitness data vectorization on parallel hardware.

1.1 Objectives

Our work aims to validate the versatility of this approach and extend it by studying a broader range of GP related topics such as classical symbolic regression problems, targeted image evolution, evolutionary art, etc. Mainly, we intend to determine how each one of the scenarios analyzed benefits from the use of a GPU and to what extent. Furthermore, TensorFlow has evolved as a platform since the implementation of KarooGP, introducing novelties in the underlying execution model. For this reason, another objective will be to compare different TensorFlow execution modes and understand how these can affect evolution in GP.

In order to achieve the aforementioned goals, the following tasks were executed:

1. Study of necessary concepts and compilation of related literature.

2. Using TensorFlow to prototype a GP engine capable of operator vectorization, TensorGP.

3. Feature validation through a series of Symbolic regression experiments.

4. Adjustments to the engine as well as implementation of extra features for evolution analysis.

5. Creation of a GitHub repository [5].

6. Benchmarking of different GP approaches.

7. Exploration of an Evolutionary Art scenario with parameter exploitation.

1.2 Outline

The present document is organized as follows. In Chapter 2, the theoretical overview regarding fundamental EC and GP topics is provided followed by a comprehensive list of related literature in Chapter 3. In Chapter 4 we define the approach alongside the representation needed to meet engine requirements. Next, in Chapter 5 we detail the main features and operators implemented in TensorGP. Then, we compile all the experimentation done involving the engine for Chapter 6. Finally, Chapter 7 highlights the main conclusions from this study and some orientation for future work.

This page is intentionally left blank.

Chapter 2

Background

This chapter aims to provide an overview of the necessary concepts that will be targeted throughout this work. Following a top-down layout, we start by introducing Evolutionary Computation (EC) to then funnel our focus towards the main theme of this project which is Genetic Programming (GP).

2.1 Evolutionary Computation

For many years, scientists have strove to apply some of the aspects regarding biological evolution to other contexts such as problem-solving, computer simulation and system modeling. Initially, EC emerged as an alternative to find near-optimal solutions to problems that could not be solved in polynomial time, the so called NP-hard problems.

These problems are often unfeasible to solve through classical analytical methods as they often comprise a vast search space and/or slow solution assessment, which in turn makes finding the absolute best solution a daunting task. Instead of scanning through the entire search space to assess this best solution, EC employs heuristic approaches to gradually find better solutions that converge to the global best as time goes on. In fact, an Evolutionary Algorithm (EA) is nothing more than a continuous optimization process in which a population of candidate solutions evolves according to specific problem constraints.

In the realm of Computer Science, evolutionary processes are comprised of four main phases: Initialization, Selection, Reproduction (through genetic operators) and Termination [15]. Figure 2.1 shows the flow of phases within an evolutionary cycle.

In the initialization phase, the starting population of candidate solutions is generated, oftentimes at random (although instantiation from specific starting points can be used if we possess some prior knowledge of the task at hand) [4].

When implementing evolutionary techniques it is important to understand the difference between phenotypes and genotypes in order to properly choose the representation that best suits the problem to solve. The genotype of an individual is the set of its genetic attributes (genes) that encodes behavioral traits. These encoded traits have observable manifestations and characteristics that make up the phenotype. Any organism can, therefore, be viewed as a mapping between their genotype and the respective phenotype. The selection phase is a two-step process where, in a first instance, the existing population is evaluated according to a fitness function (always calculated over the phenotype) that indicates how viable each individual is. After assigning a fitness value to every member of the population,

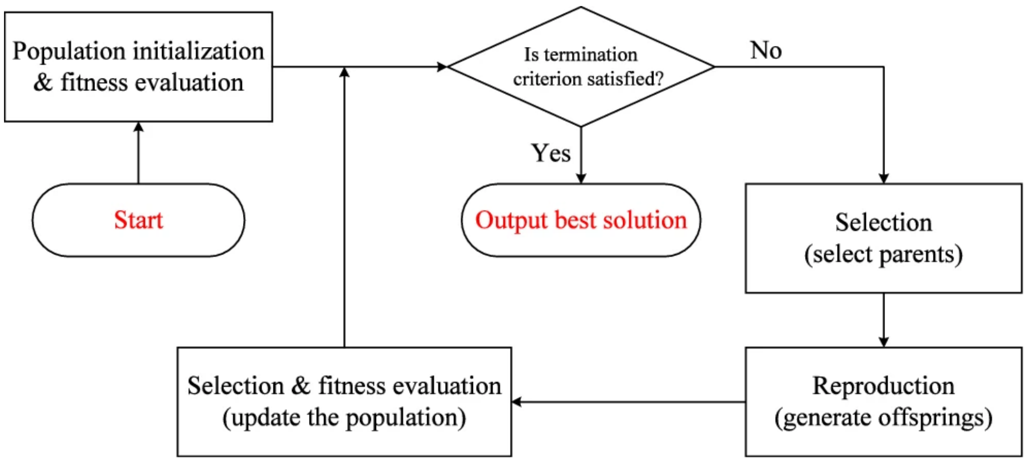

Population initialization & fitness evaluation
Start
Is termination criterion satisfied?
No
Yes
Output best solution
Selection (select parents)
Reproduction (generate offsprings)
Selection & fitness evaluation (update the population)

Figure 2.1: Representation of the Initialization, Selection, Reproduction and Termination phases of the evolutionary cycle [9].

the top-scoring individuals are chosen to build the next generation. This way, selection drives phenotypes as close to the optimum as possible with each passing generation [44]. Typically, this process is implemented through a tournament that is iteratively applied over subsets of the whole population and selects pairs of individuals to breed.

With the selected members from the previous phase, a new generation is created using the genetic material of the parents. The two most commonly used genetic operators that achieve both mixing and variation of genetic material are crossover and mutation, respectively. Whereas crossover is necessary to produce different combinations of genes, mutation ensures that small variations are introduced in the gene pool to prevent premature convergence of solutions to local extrema. Generally, we want a higher exploration rate of the search space in the early steps of the evolutionary process while promoting exploitation in later stages.

Finally, the evolution process must come to an end. There are numerous measures to determine the best moment to stop these algorithms, some of which are: performance stagnation (also called the threshold of performance, which happens when the evolution ceases to produce better fitted individuals), maximum runtime, maximum generations or when an acceptable error from the global optimum is reached.

Even though evolutionary approaches can be great problem solving tools, there are several aspects to consider. Probably the most obvious limitation of any evolutionary process stems from the already mentioned sub-optimality of solutions found. Being based on heuristics, EA mostly sacrifices accuracy and optimality for speed. It is not hard to conclude that there exists no algorithm for solving all (e.g. optimization) problems that is, on average, superior to any other contending algorithm for all possible cases [50]. This is called the No Free Lunch (NFL) theorem. In reality, EA is frequently said to be the second best approach to most problems. If for a given problem there is no traditional solving method that runs in a feasible amount of time, then an EA would be a good alternative to find a "good-enough" solution in a sane amount of time. If however, there already exists an analytical method that solves the problem in a reasonable time frame, then there is little space for EC in that task.

Perhaps another aspect worth mentioning is that EA provides the necessary flexibility to be incorporated either in black-box models or in very specific situations such as dynamical environments where the goals and problem constraints keep changing over time.

One last aspect to note is the apparent robustness of EA to parameter variation. While it is true that parameter optimization (such as mutation and crossover probabilities, tournament distribution, population size, etc) can play a significant role in algorithmic performance, wrong parameter settings usually do not prevent fairly good results [44]. This added robustness can help to ease the adoption of such algorithms by other fields of research.

2.2 Genetic Programming

GP is a subfield of EC that concerns itself with the evolution of actual computer programs. This means that, while other EC approaches require knowledge about the internal structure inherent to a problem, GP "automatically" solves the problem by optimizing a program to solve it, consequently being regarded as a domain- independent technique. As with all evolutionary processes, GP is stochastic as it involves the probabilistic selection of the best individuals and introduces variations in the set of candidate solutions (or programs in this case) through crossover and mutation. Again analogously to EC, although this incorporation of random variations allows escaping from local extrema, it also means that there is no guarantee as to solution optimality.

In GP, fitness assessment is done by running the actual program and evaluating their quality compared to a form of ideal quality criteria. For example, if we are trying to find a program that computes the average of two inputs, we may calculate the numerical difference between the result from our evolving GP program and the correct formula for computing averages. However, by only calculating the difference for a specific input, our solution will be biased as we aren't assessing fitness for other sets of inputs. In reality, for a program with n inputs, we want to assess performance within a n-th dimensional domain by calculating the summation of the differences across all points.

Commonly, programs in GP are not represented by a text string but rather through syntax trees. These trees are comprised of terminal nodes that contain the variables and constants, as well as non-terminal nodes that contain allowed operators/functions for that program. The set of allowed operators is called the function set while the set of possible variables leaf nodes is called the terminal set. The terminal and function sets together form the primitive set. It should be noted that 0-arity operators (functions with no parameters such as "rand" or "jump") are considered part of the terminal set and are therefore treated as leaf nodes.

In this section, we analyse some desired properties of GP systems in regards to function set creation and tree generation, as well as making some remarks about areas where GP is particularly useful.

2.2.1 Function set properties

In order for a GP system to work efficiently, we need to ensure some properties regarding the function set, namely closure and sufficiency. Closure is a property of the operators that ensures the validity of functions generated by the program whatever combination they appear in. This property is needed so that programs don't cause errors in the fitness assessment phase. To achieve Closure of the function set, we must provide type consistency and evaluation safety. Evaluation safety is mainly a way to eliminate undefined behavior that may occur for some input combinations. Let's assume we wish to include division in our function set. If the terminal set contains the constant 0 then there is the possibility of generating a program that attempts division by zero, which is undefined behavior that inevitably leads to a run-time error. In this scenario, we would have to implement evaluation safety programmatically by performing regular division with non zero dividers and returning a predefined value (such as 0 or even -1) to prevent run-time errors.

The other property of closure is Type Consistency. With Type Consistency, we make sure that all functions allowed in the program only return and accept data of the same type. This is often a requirement as the genetic operators applied to GP may swap nodes corresponding to different subtrees, generate new subtrees and even delete existing ones, making it possible for every operator to serve as input to any element of the function set (including the operator itself).

The extent to which GP type consistency is still an issue in untyped languages (such as Python and Perl) depends on the degree of internal type consistency already ensured upon the set of operators provided by the language itself. Nevertheless, languages that ensure this property over all existing data types are rare not to mention that ensuring type consistency across different data structures can lead to several ambiguities and is an even more complex task altogether. For that reason, the best practice is to stick to a single type of data and explicitly define all non-trivial cases.

Another sought after property of GP operators is sufficiency. Sufficiency is needed to guarantee that at least one solution to the problem can be reached (or feasibly approximated) given the function set. Suppose that we take the previously discussed example and try to evolve a program to compute the average of two values in a system where the function set only contains addition. We will never be able to reach a sensible solution as all we can do is sum some combination of the inputs. To emulate division, we need either the division operator itself or some derivation like multiplication. Unfortunately, there are several problems where proving sufficiency for a given function set might be impractical and, for such problems it is commonplace to add a few redundant operators in an attempt to ensure this property.

2.2.2 Tree generation and manipulation

Generally, in order to generate an individual in GP, the two most used methods are Grow and Full (Figure 2.2). Both methods employ random initialization of the nodes and ensure a maximum depth (maximum distance in edges from a leaf to the root node). The full method ensures that leaf nodes only appear in the defined Maximum Tree Depth (MTD) by continuously feeding operators into one another until that point. Although this generates a tree with a full topology, the size (number of nodes) of the resulting tree may change as different operators might have a different number of inputs (child nodes). Whereas the full method selects operators from the function or terminal set according to the current tree depth, the grow method simply chooses a random node from the whole primitive set. This allows for leaf nodes to exists above MTD, leading to shorter trees (fewer nodes) on average. Oftentimes, both of these methods are used in conjunction to generate the initial population in a process known as Ramped half-and-half. In this approach, half of the population is generated using the full method while the other half is generated using grow.

During the evolutionary process, the program population is subject to recombination and mutation operations just like in EC. In GP, crossover is typically done in a subtree fashion. In this technique, we replace a randomly chosen subtree in the first parent with a subtree

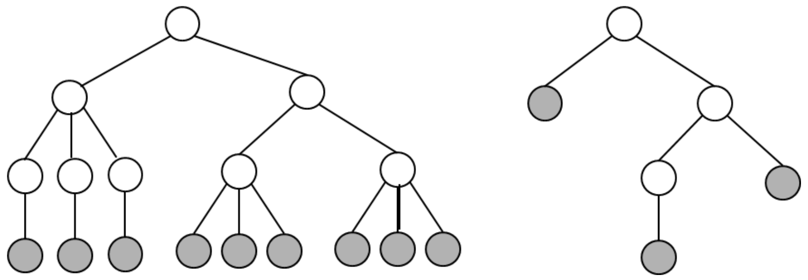


Figure 2.2: Example of trees generated with the full and grow methods (left and right, respectively) [31].

of the second parent. Randomly choosing a node in the whole tree makes leaves much more prone to crossover which causes a small amount of genetic material to be modified. To account for this, Koza [25] proposed the widely accepted approach of only selecting leaf nodes 10% of the time.

When it comes to mutation operators there are a few operators commonly used, probably the most popular being subtree mutation. This operator replaces the subtree corresponding to a randomly selected node with another randomly generated subtree. Subtree mutation tends to generate offspring with the same size (on average) as their parents, although some variations exist [39]. On the other hand, Hoist mutation tends to create smaller individuals by generating offspring that are copies of a randomly chosen subtree of the parent. Likewise, Shrink mutation also monotonously decreases program size by replacing a randomly chosen subtree with a terminal. Another highly acclaimed mutation operator is Point mutation (or node replacement mutation) where we select a random node (either function or terminal) and replace the matching primitive with a different random primitive of the same arity. Many EC problems benefit from Permutation mutation, which is a way to shuffle the genes of an individual by repeatedly performing swaps in the genotype. This type of mutation is also applied in GP, although to a lesser degree, by swapping the arguments of a randomly chosen node from the function set.

2.2.3 GP Applications

Akin to EC, GP methods are regarded as good alternatives to problems where optimal solutions are not attainable at all or in a reasonable amount of time. According to [39], GP seems to excel in problems with at least some of the following properties:

- The relationships among the relevant variables are unknown or poorly understood.
- Finding the size, topology or representation of the optimal solution is a major part of the problem.
- Significant amounts of test data are available in computer-readable form.
- There are good simulators to test the performance of tentative solutions to a problem, but poor methods to directly obtain good solutions.
- Conventional mathematical analysis does not, or cannot, provide analytic solutions.
- An approximate solution is acceptable (or is the only result that is ever likely to be obtained).
- Small improvements in performance are continuously measured and highly prized.

Therefore, GP has done well in a variety of tasks ranging from conventional symbolic regression and signal processing problems to other domains such as medicine and generative art.

Generative art is commonly regarded as any art practise that is set in motion with some degree of autonomy through a set of rules which are executed by a machine, typically a computer [17]. Evolutionary art can therefore be defined as a subset of all generative art that encompasses some type of evolutionary process.

As explained by J. Romero [41], GP in specific is a commonly used evolutionary art technique. This is due to the fact that any mathematical expression evolved with GP can be evaluated by plugging in pixel coordinates to produce an image phenotype.

Even though defining art is outside the scope of this work, a general consensus is that the perception of art is highly subjective. This can be problematic in the context of evolutionary art because every evolutionary process needs to assess the fitness of its candidate solutions. In fact, this happens to be the major difficulty for art generation with evolutionary techniques. [32, p. 24]. In Chapter 6 we will revisit this topic by employing an automatic evaluation method to evolutionary art in GP.

Chapter 3

State of The Art

In this chapter, we delve further into GP and contextualize the role of parallelization in this area while providing a compilation of related literature.

3.1 Graph-based GP

Tree based GP follows a very well defined structure where each node calls primitives from the depth level immediately below. While it is true that a tree is a special kind of graph, efforts have been made to model GP systems in a graph-like manner where each subtree has the possibility of being called from more than one location. This approach makes sense as it is frequent for a program to have common parts that are derivations of other highly fit individuals. These common parts represent copies of the same subtree that can be found within an individual.

To evolve programs with this high degree of parallelism, Poli [38] proposed Parallel Distributed Genetic Programming (PDGP), which introduced the reuse of subtrees. In PDGP, programs as seen as graphs with nodes representing primitives and links representing both the flow of program control and results. The results labelled in the links allow for PDGP to transcend the generalization of standard tree-based GP and explore the evolution of more complex programs like logic networks, neural networks, finite state automata, etc.

Another interesting graph-based form of GP systems is Cartesian Genetic Programming (CGP). In this representation initially proposed by Miller [34], a program's genotype is an integer list that represents the functions and connections between graph nodes. This list of integers is subdivided into groups, each corresponding to a position in a 2-D array. While one integer in a group defines the primitive to use, others point to the location of the input values in the genome. Effectively, CGP implements an unusual form of redundancy, in which genes may be switched on or off. This redundancy proved to have unexpected beneficial results in evolutionary search [35].

3.2 Faster GP

Being that we must run every program in the population for every fitness case, it comes as no surprise that GP is regarded as a High Performance Computing (HPC) task when applied to complex problems. In this section, we analyse some methods used to curb the use of computational resources and time, be it by speeding up or parallelizing GP search.

3.2.1 Fitness Acceleration through Caching

As discussed in the previous section, graph-based GP has the ability to reuse partial results avoiding the repeated evaluation of the same subtree and therefore speeding up the execution of a given program. However, GP programs usually share code amongst different individuals and not only within a single program [22]. This comes as a direct consequence of the evolutionary process that promotes the genetic material of fit individuals, leading to many copies of successful code throughout the population. Fitness caching refers to the process of storing results of calculated subtrees thereby avoiding the re-calculation of data. In particular, Handley [19] implemented fitness caching by saving the computed value by each subtree for each fitness case. This allowed for computational savings both by not recomputing subtrees that appear more than once in a generation and by not recomputing subtrees that are copied from one generation to the next. Furthermore, Handley represented the population of parsed trees as a Directed Acyclic Graph (DAG) rather than a collection of separate trees, consequently saving memory by not duplicating structurally identical subtrees.

Although many GP applications can greatly benefit from fitness caching, there are several exceptions worth mentioning. First and foremost, in order to effectively reuse a past result, we must remember both the inputs and the corresponding output for a given subtree. There are, however, GP systems where subtrees have side effects in other parts of the program. In such systems, the output of a subtree depends not only on the input parameters but also on the rest of the program. Langdon [27] exploited syntax rules about where in the code these side effects could lie so as to implement caching under these constraints. Nevertheless, because of the unpredictable nature of these GP systems, fitness caching is more widely used outside this domain.

Another implementation concern of this caching approach are the genetic operators involved. Suppose that during the recombination phase we add a new subtree to a terminal node of an already existing subtree. By doing this, the caches of every node between the new code and the root node may now be invalid. Although at first this might look like a complex problem, it should be noted that only direct ancestors of a node need to be updated. As a result, imagining that the crossed over code is inserted at depth d, we only need to update d nodes [39]. Considering that to evaluate a node we can use the cached values of its children, this task is not so daunting after all.

Even if we rule out the performance overhead incurred by subtree caching, memory is still an issue. Because the number of unique subtrees is a dynamical consequence of a run, the size of the memory allocated for caching should be dynamic as well. This is a potential problem in applications with a limited amount of available Random Access Memory (RAM). It is therefore desirable to devise a mechanism where the maximum size of the subtree cache can be set beforehand. Keijzer [23] proposed two cache update and flush methods to deal with fixed size subtree caching. The first method uses a postfix traversal of the tree and implements a bottom-up approach to decide whether a subtree should be added to or deleted from the cache. The second method follows the DAG approach, where inside the DAG a fixed set of fitness evaluations are cached. Realistically, as memory is a limited resource, the cache size is always limited and because of this hit-rates and search times become determining factors to performance. Wong and Zhang [51] developed a caching mechanism based on hash tables to estimate algebraic equivalence between subtrees. This proved efficient in reducing the time taken to search for common code by reducing the number of node evaluations.

Caching methods are particularly useful in scenarios with greater memory requirements and where code re-execution is more time consuming. As an example, Machado and Cardoso [33] applied caching to the evolution of large sized images (up to 512 by 512 pixels) in the NEvAr evolutionary art tool, confirming the benefits of caching for instances with large sets of fitness cases.

3.2.2 GP in Parallel Hardware

The field of GP, although computationally intensive in nature is also "embarrassingly parallel" [2]. As a matter of fact, this is not a property exclusive to GP but valid to the area of EC as a whole. Because of this, many efforts have been made to implement GP systems using parallel hardware.

There are two main ways to apply parallelization to EC. We can attempt to speed up both the evolutionary process itself or the fitness evaluation phase. For instance, Yu et al. [52]. implemented a parallel genetic algorithm where not only the fitness evaluation was parallelized, but also the genetic operators. Nevertheless, parallelizing genetic operators like crossover means providing exclusive access to the genetic material of the parents, which might perform suboptimally with traditional GP implementations. Perhaps the most straightforward way to parallelize GP is to do so in the fitness phase, which is (in most instances) the most computationally costly operation in GP [18]. Various methods to speed up fitness evaluation with different parallel architectures are herein analysed, with a special focus on Graphics Processing Unit (GPU)s.

3.2.3 GPU versus CPU

Whereas a Central Processing Unit (CPU) strives to minimize the latency of operations, a GPU mainly focuses on maximizing data throughput as shown in Figure 3.1. Data throughput refers to the amount of information that gets process in a given time unit, no matter how fast each operation is performed, which translates to Genetic Programming operations per second (GPops) in GP.

In recent years, we have seen steady growth in the computing capability of GPUs. Even CPUs have seen an increasing concern with overall throughput maximization resulting in higher core counts. While this is true, even top-of-the-line CPUs are still nowhere near the floating point performance offered by a standard GPU. Typically, GPUs operate according to a model called Single Program Multiple Data (SPMD), where many processors simultaneously run the same program/instruction on different inputs. This applies to fitness evaluation in GP in the sense that every generation we need to run every program in the population on the same set of data points.

As an example, Cano et al. [8] proposed a scalable model to massively parallelize the evaluation of individuals using the NVIDIA Compute Unified Device Architecture (CUDA) GPU programming model. Their test results revealed a significant reduction of computational time compared to a sequential approach, reaching speedups of up to 820 fold for certain classification problems. Similarly, Augusto and Barbosa [3] ported many GP parallelization strategies into the open source and multi platform OpenCL programming framework. By comparing the results of these parallel strategies on CPU and GPU processors from different vendors they concluded that overall, the GPU is considerably faster and more power efficient than the CPU as even a mid-range GPU can beat a very high-end CPU.

So far, we have looked at benchmarks comparing optimized GPU code against trivial

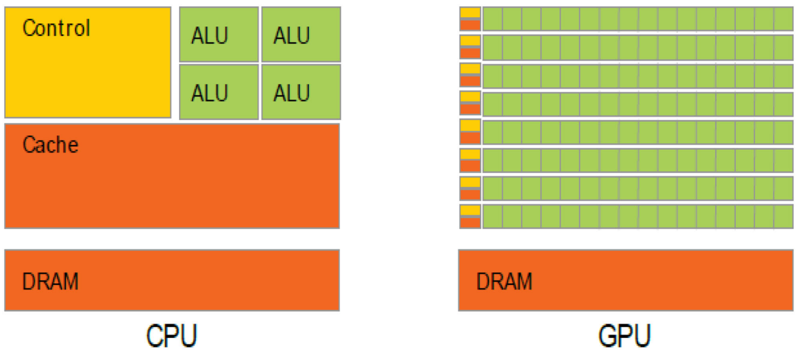


Figure 3.1: A CPU architecture is optimized for low latency accesses, while GPUs are optimized for higher throughput [29].

sequential CPU implementations. However, this is quite unfair for the CPU counterparts as these devices have seen growing support in Single Instruction Multiple Data (SIMD) optimizations. As Chitty [12] demonstrates, a multi-core CPU implementation of GP can yield performance levels that match and even exceed those made for the latest graphics cards. Indeed, despite the higher throughput of GPUs, identical performance values were obtained for a sequential stack implementation of GP using Streaming SIMD Extensions (SSE) and loop unrolling in a Multi-core CPU.

Additionally, GPUs are not the only parallelization capable devices that have been applied to GP. For instance, Field Programmable Gate Arrays (FPGA) are chips that contain large matrices of configurable logic blocks connected via programmable interconnects. These interconnects allow for the chip functionality and connectivity to be changed through software by simply writing a configuration into the chip's static memory [39].

In particular, FPGAs have been applied to fitness evaluation in GP to evolve sorting networks [24]. The problem of checking whether a candidate network (a sequence comparison-exchange operations executed in a fixed order) is a valid sorting network is an exceptionally burdensome problem when it comes to computational resources, hence the use of an FPGA. Still, Koza et al. managed to find optimal sorting networks to sort 7, 8 and even 9 items.

3.2.4 Interpretation vs Compilation

In order to run these programs in the GPU, we can either compile them for this platform or build an interpreter capable of translating the program. There is clearly a trade-off choice: the cost of iterating over code to translate it needs to be balanced against the compilation overhead and reduced cost of iterating over compiled instructions.

Langdon and Banzhaf [26] demonstrated a SIMD interpreter which runs a quarter of a million programs simultaneously on the NVIDIA GeForce 8800 GTX GPU using RapidMind's GNU C++ OpenGL framework. By employing a Reverse Polish Notation (RPN) rather than a prefix-based representation, Langdon replaced recursive calls by an explicit stack thus making the interpreter readily applicable to CGP and increasing performance. In one case, more than one billion GPops were interpreted. Analogously, Cano and Ventura [7] created a parallel subtree interpreter that extends on common parallelization approaches by concurrently evaluating different code segments of the same individual. As demonstrated in their work, the interpreter managed to outperform similar models, achieving up to 21 billion GPops using a NVIDIA GTX 480 GPU. In a similar fashion, Harding and Banzhaf [20, 21] used fragment shader programs to interpret GP individuals. This is done through a case statement at the evaluation of each node that determines what function to apply to the input values. The shader is executed for each individual and the resulting texture is converted back into an array for fitness assessment. This was proved beneficial to increase performance both in classification and regression problems.

On the other hand, many works resort to compilation approaches. Chitty [11] used a conversion technique to automatically convert each GP tree into a fragment program that would then be compiled for the GPU on the CPU. After compilation, the program is transferred to the GPU where it is executed for fitness evaluation. This is achieved by using both OpenGL and C for Graphics (Cg) to convert chromosomes to Cg source code and render the evaluation data to a texture. The texture is then transferred back to the CPU where it is compared to the desired output and assigned a fitness value. This implementation proved to be much faster than many data parallel approaches.

3.2.5 Parallelization approaches and Thread Divergence

However we choose to run GP programs in the GPU (be it interpreted or compiled), we can also run multiple programs at once. As a result, we can parallelize both the evaluation of programs (population parallel approach) or the set of fitness cases (data parallel approach).

Robilliard et al. [40] analysed this problematic by implementing two different interpretation schemes: BlockGP and TheradGP. Threads in a GPU architecture are typically subdivided into groups. These groups have different designations according to GPU manufacturers (Streaming Multiprocessor for NVIDIA, Compute Unit for AMD, etc). For the sake of simplicity, we shall call these groups of threads multiprocessors. In the BlockGP scheme, each multiprocessor would only interpret one GP tree at a time, which allowed for multiple GP programs to be evaluated in parallel. The second scheme would spread the individuals of the population by the available threads on the GPU. This is to say that, if the population number was equal to the number of threads of the GPU, each thread would be responsible to run one single GP program. Their test results showed that, for all the considered test cases, BlockGP continuously outperformed ThreadGP.

This was demonstrated to be partly due to a phenomenon called divergence. Although in parallel computing each thread workload should aim to be as independent as possible, many algorithms cannot escape the need for thread synchronization. Let's say that we required that all threads evaluate an if statement before continuing code execution. In this scenario, there may be threads that evaluate the conditional statement as false on their own data and are then put on idle as they wait for the remaining ones. This process leads to divergence and consequently to a loss of computing power. This was precisely the problem in the ThreadGP scheme as many threads interpreting different programs were run on each multiprocessor In their comparative benchmarks, Augusto and Barbosa [3] also investigated the impact of branch divergence in GPU performance. Accordingly, they concluded that the population-parallel per multiprocessor strategy is clearly the most efficient approach.

3.2.6 Data Vectorization

Up until this point, we verified that there is a clear motivation in parallelizing the fitness phase. More specifically, we exemplified approaches where various fitness cases were assessed simultaneously through parallelization. However, we can also run all fitness cases at the same time but by vectorization of the fitness domain instead. Whatever the dimensionality of the problem, we can always represent the fitness domain as an n-dimensional matrix of values.

In this paradigm, running a GP program consists of nothing more than a successive component-wise application of the operators the program is made of. In his work regarding subtree caching Keijzer [23] also studies the benefits of vectorized evaluation for symbolic regression over the standard case-by-case evaluation method. As the author points out, such a vectorized evaluation effectively reduces the running time of a program to the number of nodes it contains. This makes overhead independent from the size of the fitness domain, which is especially useful in problems with millions of data points (*i.e.* the evolution of large images via GP [10]).

Furthermore, some interpreted languages such as Matlab, Python and Perl already have extensive support for vectorized evaluation in an attempt to reduce computational impact. In fact, many numerical computation libraries like Python's TensorFlow have been used to aid domain vectorization [46]. Moreover, TensorFlow has the added benefit of being readily available for GPU with minimal to none code modifications. GPUs, as the name implies, are optimized for graphics processing, which mostly entails matrix and vector operations. Therefore, it makes sense to use this kind of hardware to accelerate vectorized evaluation even further. Staats et al. [46] demonstrated the benefits of using TensorFlow to vectorized GP fitness data in both CPU and GPU architectures. Experimental results showed that by vectorizing data with KarooGP (the GP engine used) in a problem consisting of 90,000 data points, performance increases up to 875 fold are possible under specific scenarios. Moreover, while runs of the KarooGP system on the GPU fail to outperform similar runs on the CPU for small problems, the results are inverted when running complex problems (5.5 million data points) with GPU speedups averaging 1.3 times. Once again, this last benchmark validates that larger fitness domains do benefit from the higher data throughput offered by massively parallel architectures such as the GPU.

This page is intentionally left blank.

Chapter 4

Approach

The fitness evaluation process in Genetic Programming (GP) consists of two phases: genotype to phenotype translation through the calculation of domain data points followed by fitness assessment based on the obtained phenotype. In this work, we intend on speeding up the fitness evaluation process by mainly focusing on the vectorization of data points to be assessed during the genotype to phenotype translation phase. The advantages and motivations for this approach are described in section 3.2.6 “Data Vectorization”. To achieve our objectives, we resort to the TensorFlow numerical library to create a GP engine called capable of implementing the mentioned techniques for accelerating tree evaluation, as proposed by [46].

While a few other independent GP engines already use this framework, this new implementation is justifiable as we wish to extend the applicability of this system to study other problems (i.e., in the domains of image evolution) and possibly provide a general-purpose tool that makes parallelization seemingless for experimentation within GP research. Besides, other engines using TensorFlow (including KarooGP) mostly use outdated versions of TensorFlow that operate over graphs which can oftentimes be non-optimal as far as GP is concerned (as we will analyse in the following sections). Moreover, we will also compare our vectorized approach against iterative baselines and other standard GP frameworks in the field (such as DEAP) in order to determine performance gains.

4.1 Parallelization of Fitness Evaluation

The domain of fitness data points to be evaluated is fixed for all individuals across all generations, making the vectorization of this data trivial using a tensor representation. In essence, a tensor is a generalization of scalars (that have no indices), vectors (that have exactly one index), and matrices (that have exactly two indices) to an arbitrary number of indices [42]. The output from the evaluation phase will be a tensor, given as a function of the coordinates for each data point. These coordinates will also be represented as tensors, each defining coordinates for one of the dimensions.

Before moving on, we must define the variables of the terminal set. For a problem concerning a domain with n dimensions, there will be n variables in the terminal set as this is the amount necessary to index any element within that domain.

In a serial approach x, y, z, ..., are actually variables because they change according to which element of the domain we are evaluating. However, in a vectorized approach we want to evaluate the domain in one go and as a result x, y, z, ..., will be tensors containing coordinate values. This means that, for example, every element in x will have the value corresponding to the position of that element in the tensor along the x axis. For this reason, the word "variable" is a rather misleading terminology in the context of parallel GP. Nevertheless, in an effort to stay consistent with existing literature, we will continue to refer to these as variables of the terminal set.

In the same manner that there are variables in the terminal set, there are also constants/s-calars that either can take the same value for the whole problem domain or have specific values according to a given dimension. Typically this dimension is the depth, in order to emulate color channels like Red-Green-Blue (RGB).

To calculate the output tensor of a program, we first need to represent the program flow as a data structure composed of operators from the function set as well as variables and ephemeral constants of the terminal set. This is done by first parsing the expression containing the code of the program into a tree structure that is recursively traversed to calculate the result tensor. TensorFlow then seemingless distributes computational efforts across available hardware, Central Processing Unit (CPU) and/or Graphics Processing Unit (GPU).

Finally, fitness is calculated by comparing this result with the optimal fitness measure. We should note that this last comparative step of the fitness evaluation phase can also be vectorized and run on the GPU. Additionally, as the target to compare is constant, we can save this data as a tensor to GPU memory and eliminate the overhead of memory transfer.

4.2 TensorFlow Intermediate Caching

When running the vectorized operations, TensorFlow performs some limited caching of intermediate nodes to avoid later code re-execution.

As previously mentioned, it is frequent for a population of GP programs to share common code. In fact, this is a result of the evolutionary process that promotes highly fit code which in turn gets mutated and transmitted across generations. By caching the fitness values of common segments of code, we avoid re-executing the same code time and time again, possibly leading to a significant increase in efficiency and algorithmic performance.

In TensorFlow 1 this can be done through an intermediate graph of operations in what TensorFlow calls graph execution mode. With further developments in TensorFlow 2 there was a shift in execution paradigm and the explicit intermediate graph was abolished in favor of just-in-time tensor execution, also known as eager execution mode.

In upcoming sections, we aim to explore and exploit the caching mechanisms employed by both TensorFlow execution modes to hopefully obtain more meaningful speedups, both in GPU and CPU.

Chapter 5

Implementation

TensorGP takes the classical approach of most other GP applications and expands on it by using TensorFlow to vectorize operations and consequently speed up the domain evaluation process through the use of parallel hardware. The engine herein described was implemented in Python 3.7 using the TensorFlow 2.1 framework and is publicly available on GitHub [5].

TensorFlow is still an evolving tool with version 2.0 coming out in late 2019. By this time, most of the TensorFlow dependent part of the engine was already developed and, for this reason, some features of the engine had to be rewritten to comply with both the new TensorFlow Application Programming Interface (API) and execution models.

In this section, we describe the implementation details of the incorporated GP features and the efforts of integrating some of said features with the TensorFlow platform.

5.1 Genotype and Phenotype

In its simplest form, each individual can be represented as a mathematical expression which corresponds to a program that one would come across in any traditional functional programming language. Our engine follows the traditional GP approach of representing this expression as a tree graph, which facilitates the modification of our symbolic expressions through genetic operators. This way, to build an individual, a tree structure is created and nodes are recursively added to the tree until a certain depth range and other criteria are met.

The next step is to go through the entire genotype data to produce phenotypes. This is where vectorization really shines. In our case, the phenotype is a rank 3 tensor which results from evaluating our expression over the entire problem domain. Because this result tensor has three dimensions, we can represent the phenotype as an image. The first two dimensions of the result tensor are the width and height of the image while the third dimension corresponds to the color channels. The classical approach of calculating an individual's expression as many times as there are evaluation points in the problem domain can be a time consuming task even for modern processors. This is especially true for larger domains.

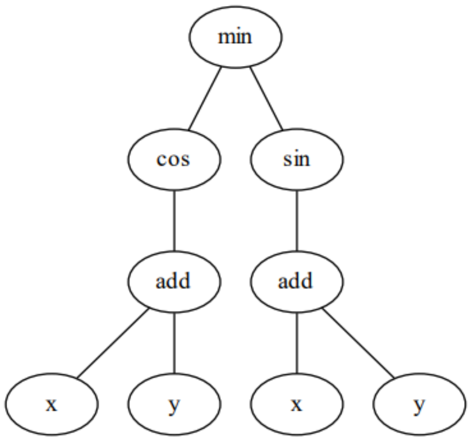
For instance, a 1,000 by 1,000 pixel image translates to 1 million evaluation points for which we would have to evaluate an expression. Moreover, it is not uncommon for GP experiments to have populations with hundreds or even thousands of individuals, which would mean that just to get to the phenotypes, we could potentially be looking at billions of expression evaluations per generation. Vectorization eases the burden of this task by taking advantage of the parallel nature of the evaluation process. With the aid of TensorFlow primitives, we can apply an operation to all domain points at the same time. It's noteworthy that the evaluation itself may not happen at the same time in hardware, however. Internally, the tensor information is passed to the processor, where it is segmented and distributed between all available parallel processors. This step is done with the help of Compute Unified Device Architecture (CUDA) if using the GPU. TensorFlow provides an abstraction layer for this segmentation process.

5.1.1 TensorFlow Integration

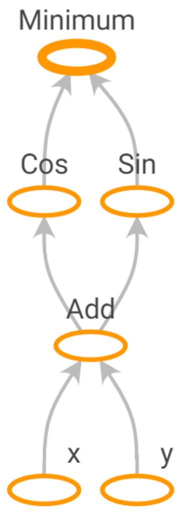
As mentioned, TensorFlow can either execute in an eager or graph-oriented mode. When it comes to graph execution, TensorFlow internally converts the tree structure into a graph before actually calculating any values. This is done in order to cache potential intermediate results from subtrees, effectively generalizing our tree graph structure to a Directed Acyclic Graph (DAG).

$$\begin{aligned}&min(\\&\qquad cos(add(x,y)),\\&\qquad sin(add(x,y))\\&)\end{aligned}$$

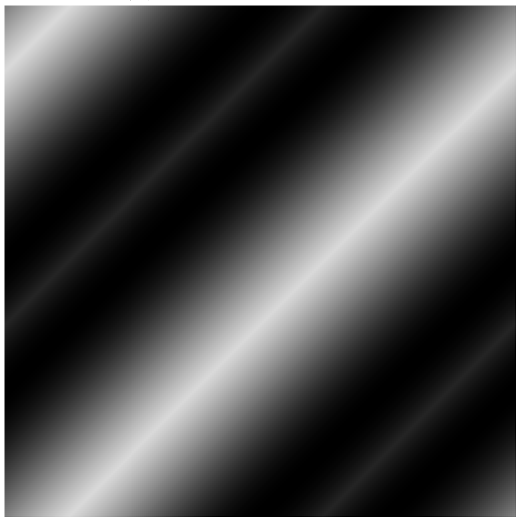
(a) Program as a string.



(b) Tree representation.



(c) Internal TensorFlow DAG.



(d) Image phenotype.

Figure 5.1: Genotype and Phenotype representation in TensorGP.

Figure 5.1 shows the different possible stages of genotype to phenotype translation in the engine. The string to tree translation phase will only happen if the population is given as input by the user. Moreover, we can verify that in the DAG representation, the addition of x and y variables is reused and fed into nodes that use that same code (the cosine and sine operators, in this case).

Throughout most of TensorFlow version 1, this process of compiling desired operations into a graph was the only mode of execution and had to be done explicitly in code. Before the announcement of version 2, TensorFlow introduced a new processing model called "eager execution", which allows for the immediate execution of a tensor operation without the need for graph building. Therefore, the tree to DAG translation phase does not exist when running TensorFlow in eager mode. It worth noting this eager execution model is enabled by default in TensorFlow 2.x and supports GPU acceleration [45].

Although graph oriented execution allows for many memory and speed optimizations, there are scenarios where the tensors to run change frequently. In such scenarios, the removal of this intermediate graph layer might be beneficial.

Because the individuals we are evolving are constantly changing from generation to generation, we would be inclined to think that eager mode would instead be a good fit for tensor execution. For this reason, in Chapter 6 we include some performance comparisons between both these TensorFlow execution modes.

5.2 Tree Generation

The three main methods for randomly generating tree expressions are supported by the engine: grow, full and Ramped Half-and-Half (RHH). These methods recursively grab elements from the function and terminal set in order to generate a batch of tree structures (the initial population). This batch of trees must obey a minimum and maximum tree depth as well as a certain set of rules.

For clarity and future reference, depth is defined as counting edges, not nodes.

5.2.1 Grow and Full

As mentioned in Section 3, with the full method all individuals are generated by filling the tree with elements from the function set until the maximum tree depth is reached. Concerning the grow method, a similar process is implemented using both the function and terminal set instead, leading to shorter trees on average.

As a sidenote, because the primitives of the function set can have different arities, two trees generated using the same maximum depth can be rather different in size (number of nodes). For instance, assuming the full method using a maximum depth n, a tree with a arity-3 root node will generate three n - 1 depth subtrees which will most likely be bigger than a tree with a single argument root node (only one n - 1 depth subtree). The grow method is also affected by such differences, although to a lesser extent due to the random distribution of terminal nodes.

5.2.2 Ramped Half-and-Half

In the first few iterations of the engine, RHH was implemented by simply having one half of each tree generated with the full method and the other half generated using the grow method. While this might be the most straightforward implementation, it has the disadvantage of always generating trees with a specific depth. To overcome this, the tree depth is defined *a priori* (typically using a random value from 2 to 6 [39]).

Nonetheless, there are other versions of RHH that uniformly distribute generated trees across a given depth range, achieving a good range of both randomly sized and randomly shaped individuals. One of those variations is described by [49] and it consists of evenly dividing the population in blocks, so that each block generates individuals of different depths ranging from the specified minimum to maximum tree depths. Half of each block is generated using the full method, while the other half uses the grow method. The main difference is that using this algorithm, all individuals are generated using one and only one method (either full or grow).

This RHH variation is no longer a way of generating trees individually, but rather a way of generating an entire population, achieving the same effect as the first implementation but with a more uniform depth and size distribution (as opposed to a random one). There are more complex ways to mix the full and grow methods to achieve true uniformity amongst trees such as the algorithm proposed by [6], which guarantees the creation of a tree chosen uniformly from the full set of all possible trees of a given size. However, we opted to use the block driven RHH algorithm because of the balance it offers between complexity/performance and distribution of results.

The tree generation function is used to either initialize a population or to inject individuals into an already existing one, in a process known as immigration (which the engine also supports). Typically, this process is executed to promote diversity in situations of prolonged fitness plateaus.

All generated programs must be evaluated for fitness before entering the current population. Custom initial individuals can also be passed to the engine through a text file containing the corresponding expressions as we will later describe.

5.3 Genetic Operators

In any application involving evolutionary computation, there must be a way to mix and change the genetic material of candidate solutions. This can be achieved with crossover (also known as recombination) and mutation operators. The idea behind crossover is that by combining the genetic data of two highly fit individuals, we produce a different individual which, from a statistical standpoint, should also be fairly fit. On the other hand, mutation methods introduce small changes to the genes in order to explore new solutions and prevent the stagnation of the genetic pool in a given population.

5.3.1 Crossover

As previously seen, the common approach of using a tree to represent individuals in GP eases the implementation of both crossover and mutation operators. TensorGP implements one crossover and four mutation operators. The used crossover operator implements a variation of a common technique called subtree crossover. This technique involves selecting a node from one parent and inserting that node into the second parent. The selected node from the first parent is chosen randomly either from the function set with 90% probability or from the terminal set with 10% probability (according to Koza's rule [25]). Then, a node is randomly chosen from the function set of the second parent and within that node, a random children is chosen to be substituted.

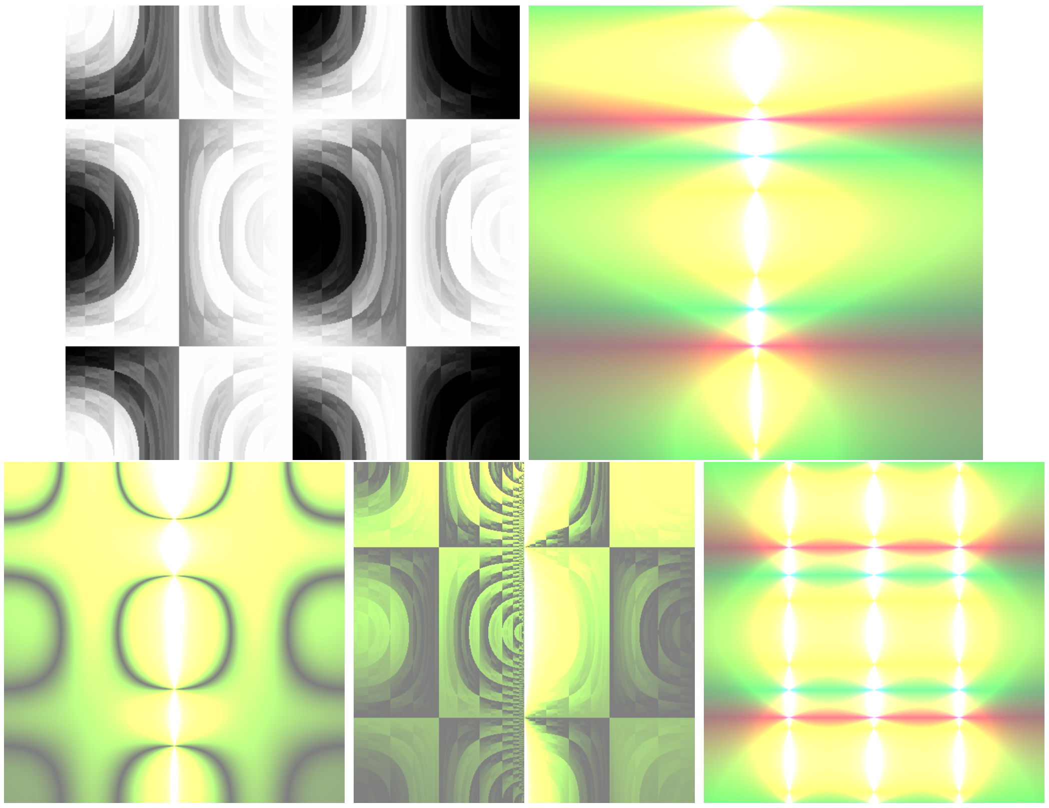


Figure 5.2: Example of the crossover result between the top row images, resulting in the images of the bottom row.

It is worth mentioning that this algorithm preserves both parents, and unlike other variations, only one new individual is created by copying the second parent and performing the aforementioned substitution process. Figure 5.2 exemplifies this crossover method. As shown, the image produced is a mix of both its parents because it combines features and shapes from both images into a single individual.

5.3.2 Mutation

When it comes to mutation, the four implemented algorithms are subtree, point, promotion and demotion mutation. All these operators create a single new individual, that results from copying the parent and applying the corresponding algorithms. In early versions of the engine, only subtree mutation was implemented. However, in order to cope with the increasing evolutionary needs of experimentation, three more operators were added. A description of all currently implemented mutation operators is subsequently provided.

Subtree mutation randomly chooses a non-terminal node and replaces its rooted subtree by a randomly generated one of the same depth. In early versions of the engine, only subtree mutation was implemented. However, the destructive nature of this operator coupled with its inability to explore different depths, lead to the implementation of the remaining operators.

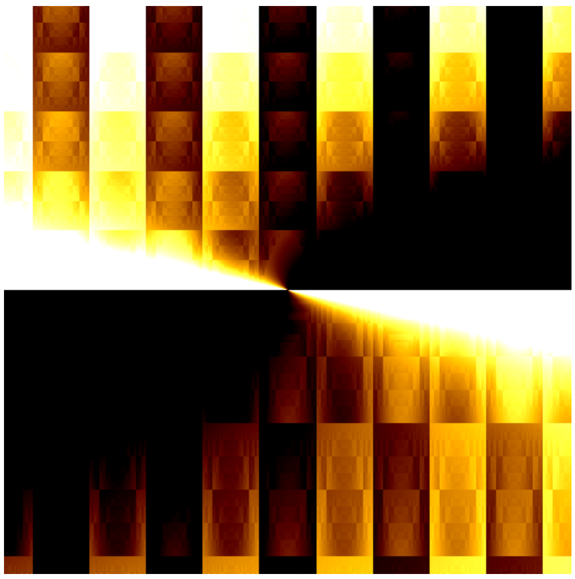


Figure 5.3: Parent program for the mutation processes exemplified in Figures 5.4 and 5.5.

Point mutation randomly chooses a node (terminal or not) and replaces said node with another one from the same set. In other words, if a terminal node is chosen we replace it by another terminal whereas if a function is chosen, we replace it by another function. We need to take additional caution for the nonterminal case because we must replace the chosen node with another primitive of the same arity. This means that we can't replace an *if* primitive (which takes three arguments) with a *multiply* (which only takes two). In the special case that there is only one primitive of the specified node's arity, the mutation process generates an individual equal to the parent.

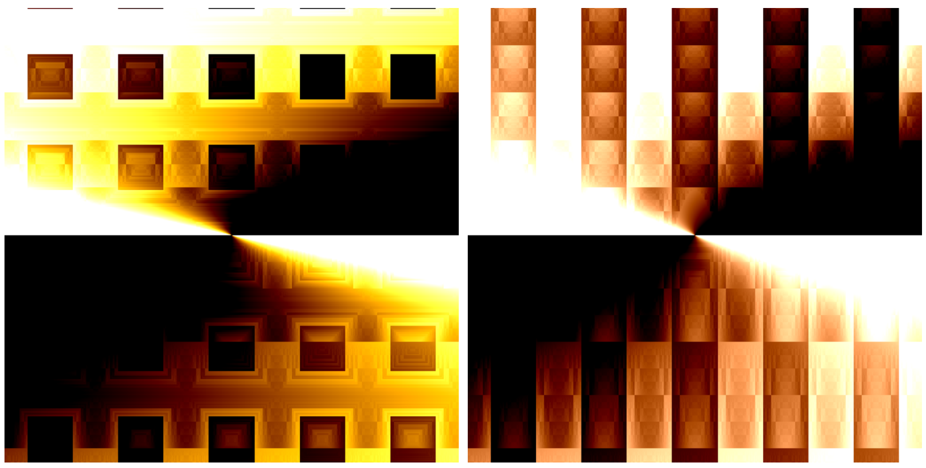


Figure 5.4: Images resulting from Subtree (right) and Point mutation (left).

One peculiarity of these two operators is that they both preserve the depth of the original parent. However, we might want want to dynamically vary this parameter in order to avoid biasing the evolutionary process towards the depth distribution of the initial population. For this reason, promotion and demotion mutation were implemented.

Promotion mutation aims to decrease the size of a tree by erasing one of its nodes. In this method, we take a randomly chosen non-terminal node and replace one of its children with the parent. This will also erase any other children of the older node. Moreover, this mutation never deletes terminals because we could not generate a valid tree without adding another primitive in its place.

On the contrary, demotion mutation serves the purpose of increasing the size of a tree. At first, a random node is chosen (terminal or non-terminal) and is replaced by a randomly selected primitive from the function set. If the node has children, one of them will be the old replaced node, while the remaining children (if any) are randomly generated terminals.

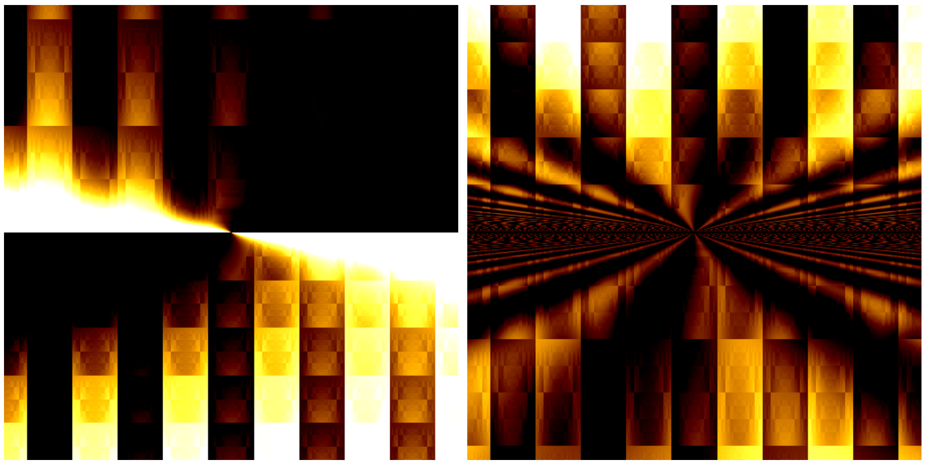


Figure 5.5: Images resulting from Promotion (right) and Demotion mutation (left).

5.3.3 Operation Application

The application of genetic operators is mutually exclusive, meaning that the engine applies at most one of the operators described to each individual in each generation. The population generated for the next generation is based on the current population. For each new individual, a tournament selection is performed over the current population to select individuals which will be the parents of any genetic operators applied. The operator to be applied to each new individual is affected by both the mutation and crossover rates. The algorithm is as follows.

Algorithm 5.1: Application of genetic operators.

```
input: max_retrie_cnt, child_depth
i ← 0

while (min_depth ≤ child_depth ≤ max_depth) and
      (i < max_retrie_cnt)
        parent ← tournament_selection()
        random_number ← random()
        child ← parent

        if random_number < crossover_rate
                parent_2 ← tournament_selection()
                child ← crossover(parent, parent_2)
        else if crossover_rate ≤ random_number < crossover_rate
                child ← mutation(parent, mutation_method)
        end if

        child_depth ← depth(child)
        i ← i + 1
end while
```

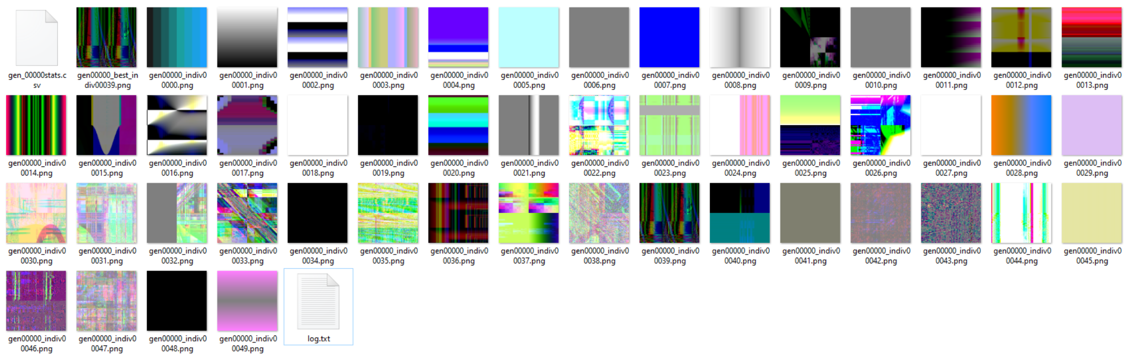
First, a random number is generated from 0 to 1. If the generated number is lesser than the crossover rate, then we select two parents and perform the crossover operator. Otherwise, if the random number is between the crossover rate and the sum of both the mutation and crossover rates, one of the four possible mutation operators is randomly chosen and applied. Else, if the random number is within the sum of both rates and 1 no operator is applied. In order for this selection to work as intended, the sum of the mutation and crossover rates must be less than 1, or else the selection will be biased towards choosing the crossover operator.

As discussed previously, the promotion and demotion mutation operators do not guarantee depth preservation and because of this, generated trees might not meet the depth range criteria. If this is the case, the engine will retry applying the operator selection process for a few more iterations. If after a number of tries no valid tree is found, the algorithm simply gives up and copies the individual resulting from the tournament selection. The pseudocode in Algorithm 5.1 illustrates the whole selection process for genetic operators.

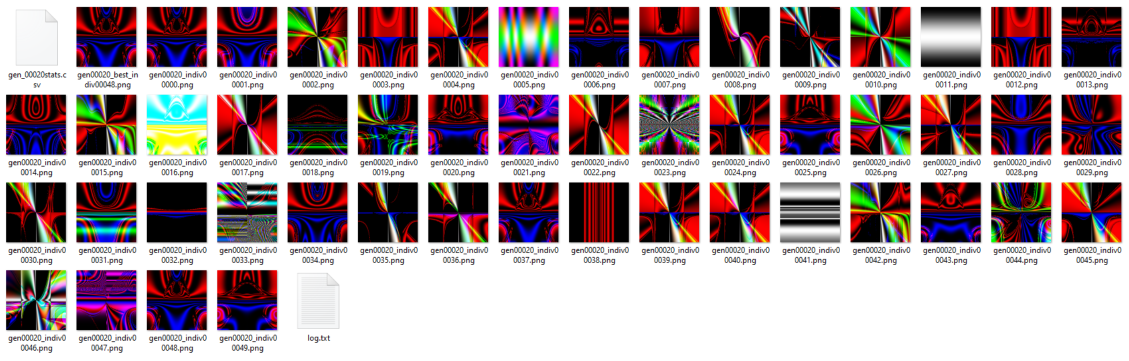
5.4 Fitness Assessment

Fitness is assessed every time new individuals enter the population. This happens when the population is initialized, between generations or when immigration is used. The fitness function itself is written by the user and passed to the engine as a reference.

Before assessment, the engine passes the population to a wrap function that enables the TensorFlow device to be used for evaluation, only then the actual fitness function is called. The vectorization process for fitness evaluation is described in detail in Chapter 4. The next population of individuals is generated by mutating and mating individuals of the previous generation according to Algorithm 5.1.



(a) Initial population of 50 random individuals.



(b) Population after 20 generations.

Figure 5.6: Population evolution within an evolutionary run (experiment with NIMA classifier - see Chapter 6).

One of the most important aspects of evolutionary algorithms in general is the method of selecting individuals. Our GP engine follows the most commonly used tournament selection method [14]. Tournament selection works by taking a sample for the population and choosing the fittest individual within that sample. The size of this sample is called tournament size and determines the evolutionary pressure to exert onto the population (a tournament size of 3 is the engine's default). A larger tournament size is more likely to choose the overall best, leading to more evolutionary pressure and greater convergence.

5.5 Primitive Set

In order to provide a general purpose GP tool and ease evolution towards more complex solutions, the primitive set implemented goes beyond the scope of simple mathematical and logic operators. This way, we attempt to provide sufficiency through redundancy of operators for as many problems as possible. Moreover, the fact that our engine predominately deals with images calls for the addition of image specific operators.

Operators must also be defined for all possible fitness domains, which in some cases means implementing protection mechanisms for certain values. Table 5.1 enumerates the different types of operators and respective special cases. All operators are applied to tensors and integrated into the engine through composition of existing TensorFlow functions.

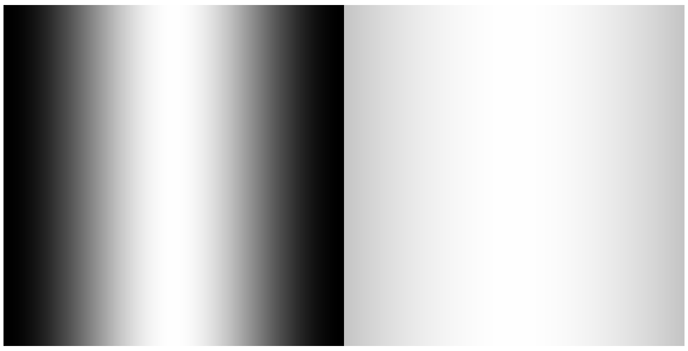
Besides the specified protective mechanism there are noteworthy implementation details for some of these operators. As an example, when calculating trigonometric operators, the input argument is first multiplied by π. The reasoning behind this is that most problem domains are not defined in the $[-\pi, \pi]$ range, but are otherwise normalized to either [0, 1] or [-1, 1].

As a matter of fact, it is common in GP not to represent image data in the range of its width, height and depth but with these values normalized to [-1, 1]. This makes it so that the argument to the trigonometric operators is in the $[-\pi, \pi]$ range, which covers the whole output domain for these operators, as can be observed by the examples in Figure 5.7. Generally speaking, keeping the problem domain normalized helps maximize operator coverage and achieve better results (or more aesthetic ones, as far as image evolution is concerned).

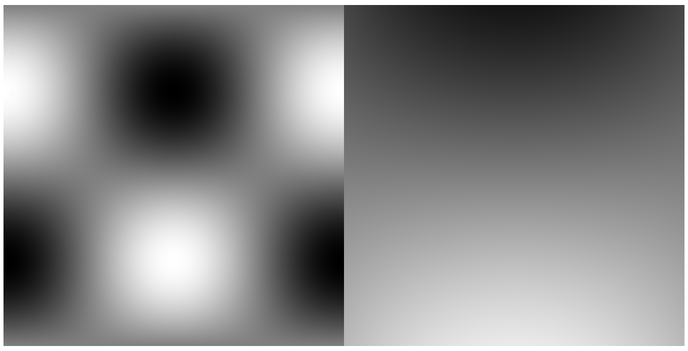
On another note, bitwise operators also need modifications due to incompatible data types. We are interested in bitwise operations between integers. However, the values passed between operators in the engine are floats. We can overcome this by interpreting the float input values as integers, removing the decimal separator and then dividing the final result. For instance, assuming the *xor* operator for floats with 6 effective digits of precision, we can approximate the integer result by calculating:

$$f(x, y) = \frac{int(x \cdot 10^6) \oplus int(y \cdot 10^6)}{10^6} \tag{5.1}$$

TensorGP implements a lot of operations that despite useful, are not meant to be applied to all kinds of problems. For this reason, before calling the engine, the user may specify a subset of the described operators to be used according to the task at hand. Moreover, operators like *clip* and $frac$ are mostly helper functions used when implementing other operators and may therefore be used even if not directly defined the user-specified subset.



(a) $f(x, y) = cos(x)$.



(b) $f(x, y) = cos(x)sin(y)$.

Figure 5.7: Images generated with π multiplied by the trigonometric functions input as implemented in the engine (left) and respective images generated without adjusting for π (right).

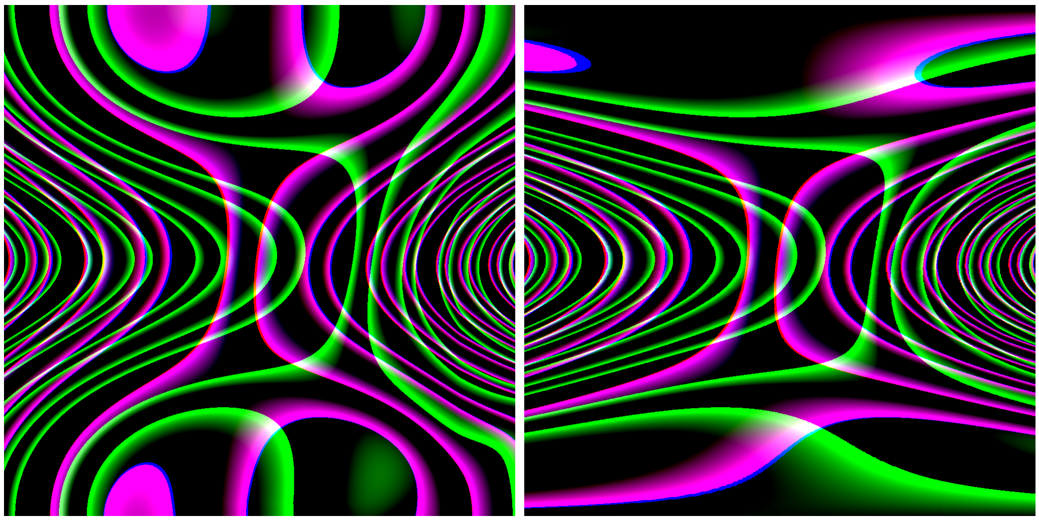
5.5.1 Warp

While still testing operator functionality in the beginning of this work, the *warp* operator stood out as a source of some bugs and implementation struggles. Perhaps the main difficulty is that, unlike other operators, *warp* data accesses are not performed element-wise. In other words, whereas other operators perform calculations over elements of the same index, *warp* needs to be able to access any element of an input tensor independent of the index being calculated. Because of this, the TensorFlow implementation of this operator needs to be expressed as a "not so obvious" composition of existing primitives. This makes the *warp* probably the slowest operator of the whole primitive set. Hence, in this subsection, we shall more rigorously define this operator as well as analyse it from a performance perspective.

The *warp* operator is defined as a transformation that maps every element of a tensor to a different coordinate. There are two main arguments to this operator for a tensor with rank n: the tensor with the elements to transform (hereafter referred to as image tensor) and an array of n tensors of rank n, each with the new coordinate position for that dimension (hereafter referred to as coordinate tensors). The bi-dimensional version of this operator is often used to distort shapes in an image.

Figure 5.8 exemplifies this distortion process. The original image (left) is warped using a trigonometric expression to transform values along the x axis. We can see that the top and bottom features of the original image seem to become stretched while features towards the middle stay similar.

Any colored image can be thought of as a rank 3 tensor. In this representation, the first



(a) Original Image I. (b) $T = warp(I, mult(cos(y), x), y)$

Figure 5.8: Image transformation using warp.

two dimensions would correspond to the width and height of the image, while the last dimension encodes the data for the color channels, which corresponding to the "depth". Color data can be encoded in RGB, Hue Saturation Value (HSV), or any other format (TensorGP uses RGB).

In order to deform shapes in an image while preserving color data, the coordinate tensors have to map points along the third dimension to themselves. In other words, we will have to perform a null transformation along the color channel (here corresponding to the depth) of the tensor. We can achieve this by plugging in the color channel input a tensor filled with the depth values for each of its elements.

It is worth noting that a tensor resulting from a *warp* operation can either lose or maintain the amount of information relative to the original tensor, but never gain additional information. This happens because *warp* transforms data by gathering elements from the main input tensor, therefore never generating more unique elements in the result.

If the transformation process (defined by the input tensors) gathers every element exactly once, the amount of information is maintained, otherwise at least some information is lost. Because of this, some properties that are commonplace in other simple math operators (such as associativity and inversion) either do not hold for all warp transformations or simply do not apply.

Identity is however a valid *warp* property (exists for every tensor). Let I be the image tensor of rank n and C the array of coordinate tensors corresponding to the variables of the terminal set. Then, the expression $Identity = warp(I, C)$ holds true for every I. For $n = 2$, this translates to $Identity = warp(I, x, y)$. Besides, other common matrix operations can easily be expressed as a *warp* transform. For instance, a transpose can be emulated by switching the coordinate tensors and defining $Transpose = warp(I, y, x)$.

When it comes to performance, because we must perform a gather from the input for every element of the result tensor, the *warp* operator has a very high parallelism potential. Thus, we might be enticed to write a straightforward implementation based on the transformation process for each individual element (as presented in Algorithm 5.2).

It should be noted that, even though tensors can be viewed as multidimensional matrices, for implementation purposes it is easier to represent them as a single array of contiguous elements. In this representation, we can iterate over all tensor elements with a simple for loop and locate a certain element within the tensor based on its coordinates.

For example, assuming the coordinates (c_1, c_2, c_3) and dimensions (n_1, n_2, n_3) we can get the index by calculating $index = ((c_1 \times n_1) + c_2) \times n_2) + c_3$. Index calculation can be generalized for n dimensions either recursively or using a for loop, as demonstrated by Algorithm 5.2.

Algorithm 5.2: Parallel *warp* algorithm for an arbitrary number of dimensions.

```
function get_coordinate(offset, index)
        domain_delta ← max_domain − min_domain
        el ← input_tensors[index][offset]
        domain ← constrain(min_domain, el, max_domain) − min_domain
        domain ← domain × (dimensions[index] − 1) × domain_delta;
        return round(domain)
end function

function warp(input_tensors, image, dimensions)
        result ← create_tensor(tensor_rank)

        foreach index in [0, number_tensor_elements]
                coordinate ← get_coordinate(index, 0)

                for i in [1, tensor_rank]
                        coordinate ← dimension[i] × coordinate
                        coordinate ← coordinate + get_coordinate(index, i)
                end for

                result[index] ← image[coordinate]
        end for

        return result
end function
```

Where “input_tensors” is an array of tensors ($T[i][n]$ represents the nth element of the ith input tensor), “image” is the tensor to gather values from, “dimensions” refers to the shape of tensors and “tensor_rank” to the number of dimension of tensors.

This implementation has the advantage of being easily adapted to machines with parallel processors by segmentation (or even elimination) of the outer for loop.

However, while TensorFlow possesses a plethora of operators to cater to our vectorization needs, it lacks a warp-like operator. Therefore, to implement it, we need to express the whole transformation process as a composition of existing TensorFlow functions. Luckily, TensorFlow has a “gather_nd” operator, which accepts a tensor of indices and grabs data from another tensor according to the given indices. This way, the only thing that we have to do in preparation is to build this tensor of indices and pass it to the “gather_nd” operator.

Algorithm 5.3: Parallel warp algorithm for an arbitrary number of dimensions.

```
def tensorflow_composite_warp(input_tensors, image, dimensions):

    n = tensor_rank
    domain_delta = max_domain − min_domain

    # Replace non−defined values with zero
    for t in input_tensors:
        t = tf.where(tf.math.is_nan(t), tf.zeros_like(t), t)

    # Create a list of tensor coordinates
    T = create_list(n)

    for i in range(n):

        # Constrain all tensor values to domain
        T[i] = tf.clip_by_value(input_tensors[i], 0.0, dimensions[k] - 1)
        T[i] = tf.math.subtract(T[i], 1.0)

        # Map domain to coordinates
        T[i] = tf.math.multiply(T[i], 1.0 / domain_delta)
        T[i] = tf.round(T[i])


    # Stack all coordinate tensors in a single tensor along a new axis
    coordinates = tf.stack(T, axis = n)
    coordinates = tf.cast(coordinates, tf.int32)

    # Gather elements from the image tensor given the calculated coordinates
    return tf.gather_nd(image_tensor, coordinates)
```

In turn, the process of building the index tensor involves the following phases: iterating over the coordinate tensors and constraining all elements to the problem domain, mapping their elements from the domain values of the problem to the dimensions of the output tensor (*e.g.* image resolution) and finally rearranging coordinate tensor data by stacking the tensor list along a new axis, effectively creating a new single tensor acceptable by "gather_nd". A pseudocode of this operator composition approach is described in Algorithm 5.3.

The previous code is written in Python and the TensorFlow functions used are marker by the prefix "tf.". It should also be noted that arithmetic operators in TensorFlow such as $tf.math.add()$ only accept tensor inputs. In order to include scalars in such operations, we need to create a constant tensor with the function $tf.constant()$. For the sake of readability, however, the above pseudocode assumes the automatic creation of that constant. As an example: $tf.add(T, 1)$ adds 1 to all tensor elements without the need to explicitly create the constant tensor.

Despite being easy (and arguably more comprehensive) to implement *warp* as a composition of operations, we cannot deny the inefficiency of this method versus writing a custom operator. In fact, TensorFlow allows for the creation of custom operators using C++ for the CPU code and CUDA for an optimal GPU version.

In order to better understand the performance implications of these algorithms and verify the potential for parallelism of the *warp* operator, Figure 5.9 compares 6 different approaches. The objective being to check how a TensorFlow 2 implementation fares against more low-level implementations as well as some other iterative baselines.

The entries corresponding to the TensorFlow platform implement *warp* with operator composition (Algorithm 5.3), while other entries are based upon explicit parallelization (Algorithm 5.2). The actual implementations used are slightly optimized versions of both algorithms. All approaches were tested using rank 3 tensors with dimensions $[x, x, 4]$, x varying from 64 to 4,096. Results are computed from an average of 100 runs (Table 5.4).

Two approaches are GPU based: one of them uses TensorFlow composition (TF GPU) while the other implements explicit parallelization with an independent CUDA kernel (GPU CUDA). For the CPU, there is the classical serial approach (CPU), a Single Instruction Multiple Data (SIMD) version (CPU AVX2), another SIMD version with multithread-

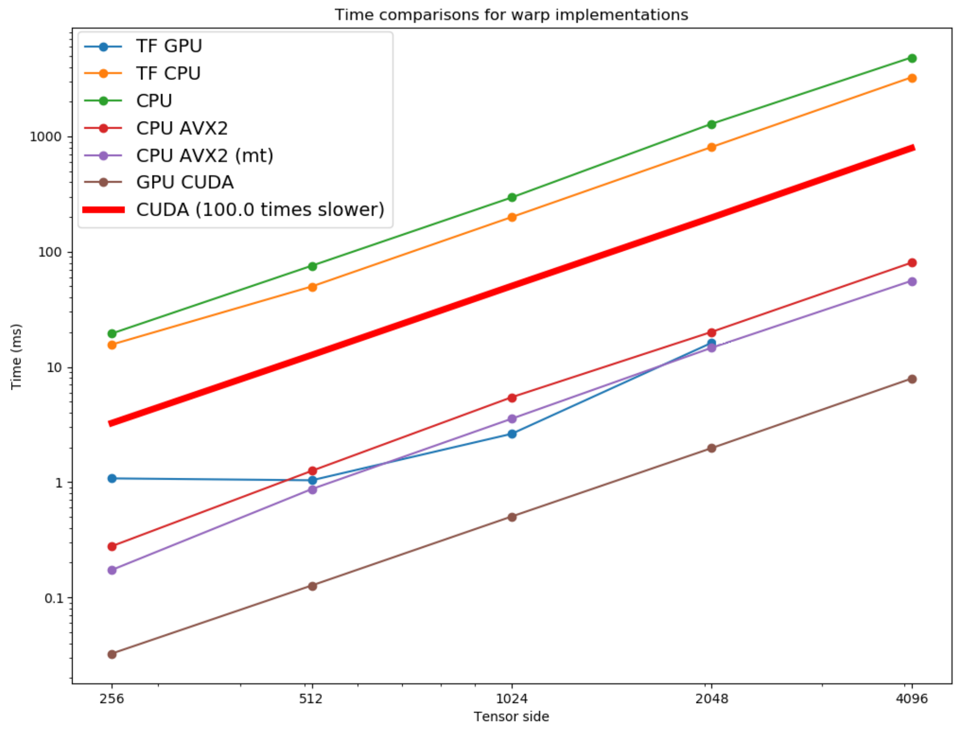


Figure 5.9: Time comparisons for various warp implementation approaches across different tensor sizes.

ing added (CPU AVX2 mt) and finally, the TensorFlow approach (TF CPU). Specifically, the SIMD instruction set used was Advanced Vector Extensions (AVX) 2 (also known as "Haswell New Instructions"). Moreover, all TensorFlow approaches were executed in eager mode.

Due to the many orders of magnitude that these test results encompass, our best bet for graphical representation is to use a logarithmic scale as it would be otherwise impossible to distinguish results (as exemplified by Figure 5.10).

Figure 5.9 shows average timings across different tensor sizes. Regarding the CPU, we observe a big discrepancy between serial (CPU) and vectorized implementations (like AVX2), which only confirms how parallelism prone this operator is. Furthermore, almost every approach follows a linear time increase across tensor side, apart from the TensorFlow approach on GPU (TF GPU) which exhibits some starting overhead along with performance penalties for larger domains (see Table 5.2).

In Figure 5.11 we can more easily analyse the performance curve for this approach. Smaller domains display a lower operation per second count caused by the initial overhead. In the meantime, larger domains values seem to indicate performance degradation, likely due to GPU memory constraints. In reality, the last test case corresponding to the largest problem domain, did not finish in TensorFlow GPU due to insufficient Video Random Access Memory (VRAM) (therefore not being present in the figures and tables). The fact that this test case completes in a basic CUDA application seems to indicate that TensorFlow stills performs some level of cashing for intermediate operator composition results which occupies more memory (even when executing in eager mode).

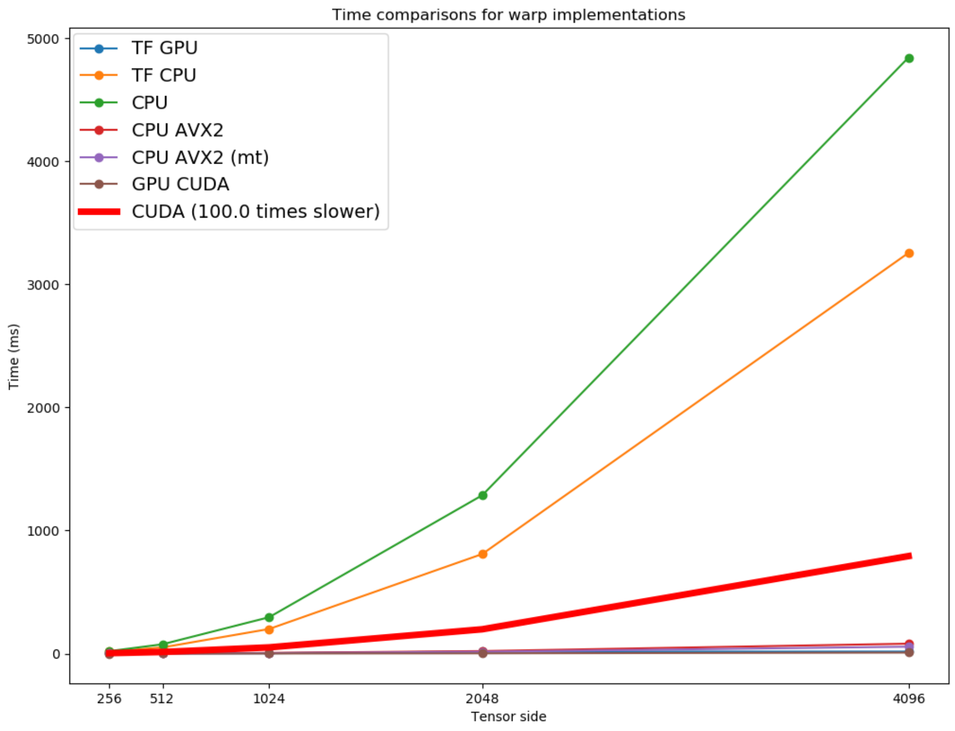


Figure 5.10: Time comparisons for various warp implementation approaches on a linear scale.

Additionally, still looking at Figure 5.11, we see that the two SIMD approaches using AVX2 scale slightly better for smaller domains, tendency that is inverted in the TensorFlow approach for CPU that achieves lower relative Million of operations per second (Mops) for tensors with size 256 when comparing with larger test cases (see Table 5.3).

Overall, the most performant approach (CUDA) was around 300 times faster than the slowest one (CPU, serialized), reaching almost 10,000 Mops. In both figures, the red line represents a threshold for approaches two orders of magnitude slower than CUDA. Apart from the traditional iterative CPU approach, only the TensorFlow CPU version did not meet this threshold. This is most likely because our specific TensorFlow build tested was not compiled to support SIMD instructions.

	CUDA	**TF (GPU)**	**TF (CPU)**	**CPU (AVX2 MT)**	**CPU (AVX2)**	**(CPU)**
256	0.033	1.077	15.631	0.173	0.278	19.486
512	0.127	1.037	49.856	0.870	1.252	75.467
1024	0.503	2.625	199.636	3.549	5.454	295.192
2048	1.973	16.171	808.798	14.652	20.115	1286.380
4096	7.919	DNF	3255.874	55.819	80.252	4843.459

DNF stands for “Did Not Finish”

Table 5.2: Average timing values(in milliseconds) for different warp implementations across tensor sizes.

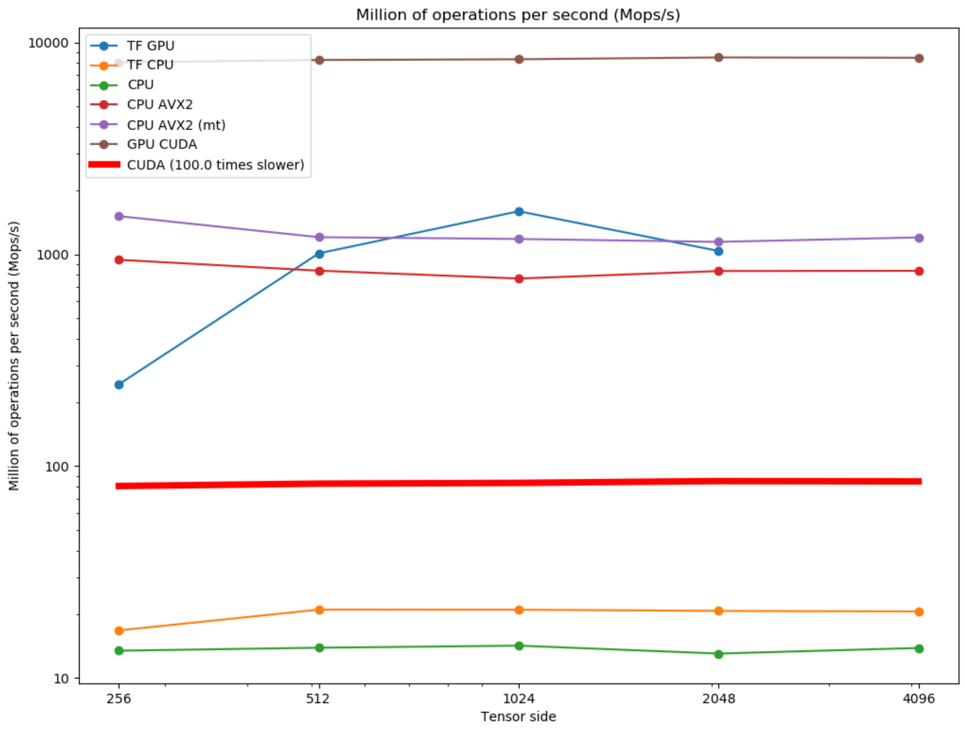


Figure 5.11: Operations per second comparison for various warp implementation approaches across different tensor sizes.

The implementation used in our GP engine was the second fastest for the 1,024 domain test case. This surprising performance even when resorting to operator composition might be explained by the TensorFlow internal Accelerated Linear Algebra (XLA) integration. XLA allows to internally compile a sequence of operators into a single GPU kernel call, substantially reducing related overheads. Analysing XLA performance in TensorFlow is, however, not the aim of this work.

	CUDA	**TF (GPU)**	**TF (CPU)**	**CPU (AVX2 MT)**	**CPU (AVX2)**	**(CPU)**
256	8058.283	243.372	16.771	1515.178	942.761	13.453
512	8270.179	1011.006	21.032	1205.135	837.527	13.895
1024	8336.091	1597.630	21.010	1181.827	768.962	14.209
2048	8503.404	1037.515	20.743	1145.046	834.073	13.042
4096	8474.518	DNF	20.612	1202.258	836.231	13.856

DNF stands for "Did Not Finish"

Table 5.3: Average operations per second values(in Mops) for different warp implementations across tensor sizes.

Even though these results seem impressive, the performance differences shown are not indicative of real-world performance. Tree evaluation often chain dozens or even hundreds of different operators, leading to other problems in memory management between GPU and CPU. Besides, program evaluation is only one step within GP and, because of this, performance improvements of many orders of magnitude are not to be expected for full evolutionary runs. In 6, we will analyse GPU performance improvements for both tree evaluation and complete evolutionary runs.

Parameter	***Value***
CPU	Intel® Core™ i7-6700 (3.40 GHz)
GPU	NVIDIA GeForce GTX 1060 3GB
Runs	Average of 100

Table 5.4: Experimental setup for the warp operator benchmark.

5.6 Features and Parameterization

In this section, we provide a comprehensive description of TensorGP's main features along with the functionality of the parameters that control them. This will hopefully provide a clear overview of engine functionality and capabilities from a user's perspective.

5.6.1 Features

When a GP run is initiated on the engine, a special ID based on the current time is generated for that run and a folder with this ID is created in the local file system (the path can be user specified). This folder holds files which register information about the evolution of the experiment, including a folder for each generation. Inside the generation folders are the images corresponding to every program within that generation. By default all generations are logged, however, the user can specify an interval between which the population data is not saved (this can be useful to avoid writing too much information to disk in longer runs).

In each generation, the engine keeps track of depth and fitness for all individuals. This information is written to a Comma Separated Values (CSV) file inside the corresponding generation folder along with the image of the fittest individual of that generation, a text file with the genotype information (in the form of a string) and images of the remaining population if the user specifies so. Furthermore, when the run is over, the engine saves the overall stats for the experiment in a CSV file and automatically generates the depth and fitness evolution graphics across generations.

Besides, the engine keeps an updated state with all the important parameters and evolution status. The idea is to be able to pause the evolutionary process at any time and close the program to resume the experiment later. To do this, when the engine is called it starts by creating a configuration file for that experiment with the appropriate settings (population size, generations, mutation rate, etc). Then, with each passing generation, the engine updates the file with information regarding evolution status (current population, generation, best fitness, etc). When it is time to resume the experiment, the engine simply loads the corresponding configuration file (using the ID that uniquely identifies the experiment), gathering the latest generational and population data.

Although the default engine behaviour is to generate the initial population according to a given or otherwise random seed, the user can choose to pass a text file containing string-based programs to the initial engine call in order to specify a custom initial population.

Even though the most traditional method of determining when a GP run ends is to set a generational limit, there are a number of other ways to do so, including number of evaluations, acceptable error, fitness stagnation or loss of diversity. Currently, there are only two stop criteria implemented: the generation limit (which the engine defaults to) and acceptable error. In the acceptable error method, the experiment comes to an end if the best fitted individual achieves a fitness value specified by the user. The conditional check for this value is made differently depending on whether we are dealing with a minimization or maximization problem, which leads to the next main feature.

The user can define whether the values calculated for fitness are to be minimized or maximized. The implementation of this feature is rather simple because the only change needed is to the conditional fitness checks: when maximizing we check if the fitness is greater than the current best while when minimizing we check if its lesser than the best. This translates into small changes to the tournament selection and fitness assessment methods. It is worth noting that the fitness evaluation function passed to the engine by the user must be in agreement with the objective specified (maximizing or minimizing).

Moreover, It is possible to define custom operators for the engine. The only requirement for the implementation of any operator is that it must return a tensor generated with TensorFlow and have $dims = []$ as one of the input arguments (in case the tensor dimensions are needed). After implementing the function, the user is required to add an entry to the function set with the corresponding operator name and arity.

5.6.2 Parameters

With all the default values defined in the engine, the only strictly necessary argument that must be passed to the initial call is the fitness function itself. Nevertheless, different problems more often than not benefit from different evolutionary processes, at which point parameter adjustment is not only recommended but may even be necessary.

Table 5.5: Parameterization of the current engine implementation.

Name	**Default values**	**Accepted values**	**Description**
`fitness_function`	-	Function pointer	Function to be used when assessing individual performance
`population_size`	100	Positive Int	Number of individuals (programs) in the population
`tournament_size`	3	Positive Int	Number of individuals that participate in a tournament during the selection process
`mutation_rate`	0.1	$[0, 1]$	Probability of applying any of the mutation operators to an individual
`crossover_rate`	0.9	$[0, 1]$	Probability of applying crossover to an individual
`min_tree_depth`	0	Positive Int	Minimum depth of individuals generated by the engine
`max_tree_depth`	8	Positive Int	Maximum depth of individuals generated by the engine
`max_init_depth`	None	Positive Int	Maximum depth of individuals generated by the engine in the initial population
`max_init_depth`	“ramped half-and-half”	String	Choose tree generation method: “ramped half-and-half”, “full” or “grow”
`terminal_prob`[a]	0.1	$[0, 1]$	Probability of generating terminal nodes when conditions to do so are met

`scalar_prob`[a]	0.33	$[0, 1]$	Probability of generating a constant terminal while generating a terminal
`uniform_ scalar_prob`[a]	0.5	$[0, 1]$	Probability of generating a scalar constant terminal while generating a constant terminal
`stop_criteria`	"generation"	String	Method to stop the evolution process; "generation" means to stop after *stop_value* number of generations
`stop_value`	10	Positive Int	Threshold to stop the evolution process
`ojective`	"minimizing"	String	Whether we are dealing with a maximization or minimization problem
`immigration`	∞	Positive Int	Generation interval to bring back best individuals from previous generations
`target_dims`	[128, 128]	Positive Int array	Problem domain dimensions
`max_nodes`	-1	Positive Int	Maximum number of nodes to be generated for each tree (-1 means no limit)
`seed`	None	Positive Int	Seed to be used by the random number generator in the engine
`debug`	1	$[0, \infty]$	Print debug information to console (higher numbers translate to more verbose outputs)
`save_graphics`	True	Boolean	Generate fitness and depth evolution graphics if True
`show_graphics`	True	Boolean	Show generated fitness and depth evolution graphics if True
`device`	"/cpu:0"	String	Which physical device to run TensorFlow operators on (along with some other features)
`save_to_file`	10	Positive Int	Generation interval to save engine state data to
`read_init _from_pop_file`	None	String	If not None read population from specified file instead of randomly generating it

[a] A terminal is a tensor taken from the function set; a constant terminal is a terminal which is always constant within at least one dimension; a scalar constant terminal is constant along all dimensions (all tensor values are equal).

Table 5.1: Description of implemented engine operators.

Type	**Subtype**	**Operator** (engine abbreviation)	**Arity**	**Functionality**
Mathematical	Arithmetic	Addition (add)	2	$x + y$
		Subtraction (sub)	2	$x - y$
		Multiplication (mult)	2	$x \times y$
		Division (div)	2	x/y 0 if denominator is 0
	Trigonometric [a]	Sine (sin)	1	$\cos(x\pi)$
		Cosine (cos)	1	$\sin(x\pi)$
		Tangent (tan)	1	$\tan(x\pi)$
	Others	Exponentional (exp)	1	e^x
		Logarithm (log)	1	$\log x$ -1 if $x < 0$
		Exponentiation (pow)	2	x y 0 if x and y equal 0
		Minimun (min)	2	$min(x, y)$
		Maximun (max)	2	$max(x, y)$
		Average (mdist)	2	$(x + y)/2$
		Negative (neg)	1	-x
		Square Root (sqrt)	2	$\sqrt{x}$ 0 if $x < 0$
		Sign (sign)	1	-1 if $x < 0$ 0 if x equals 0 1 if $x > 0$
		Absolute value (abs)	1	-x if $x < 0$ x if $x \geq 0$
		Constrain (clip)[b]	3	ensure $y \leq x \leq z$ or $max(min(z, x), y)$
		Modulo (mod)	2	$x \bmod y$ remainder of division
		Fractional part[b] (frac)	1	$x - \lfloor x \rfloor$
Logic	Conditional	Condition (if)	3	if x then y else z
	Bitwise [c]	OR (or)	2	logic value of $x \vee y$ for all bits
		Exclusive OR (xor)	2	logic value of $x \oplus y$ for all bits
		AND (and)	2	logic value of $x \wedge y$ for all bits
Image	Transform	Warp (warp)	n	Transform data given tensor input [d]
	Step	Normal (step)	1	-1 if $x < 0$ 1 if $x >= 0$
		Smooth (sstep)	1	$x^2(3 - 2x)$
		Perlin Smooth (sstepp)	1	$x^3(x(6x - 15) + 10)$
	Color	Distance (len)	2	$\sqrt{x^2 + y^2}$
		Linear Interpolation (lerp)	3	$x + (y - x) \times frac(z)$

[a] Input argument in radians, [b] These are mostly help operators, [c] Transformation to integer is needed
[d] More details in section 5.5.1

Chapter 6

Experimentation

This chapter compiles a series of experiments performed with our engine, ranging from the early phases of feature validation to CPU against GPU performance benchmarks, as well as some real-world image evolution applications.

In the first section, efforts towards operator and evolution validation are described. Next, a comprehensive benchmark of CPU against GPU timings is studied, first for a controlled environment with only tree/tensor evaluation, followed by a real-world GP problem contemplating evolution. Finally, we investigate applications in the area of evolutionary art using an image assessment classifier.

The software and processing hardware used for the execution of these experiments is as follows:

- **CPU** Intel® Core™ i7-6700 (3.40 GHz)
- **GPU** NVIDIA GeForce GTX 1060 3GB
- **RAM** 2 × 8GB @2,666 MHz
- **Operative System** Windows 10
- **Execution Environment** Command Prompt

More details regarding specific experimental setups are defined in tables located at the end of the corresponding section/subsection. GP parameters not specified in these setup tables follow the defaults defined in Table 5.5.

6.1 Validation Testing

This section describes some of the validation efforts performed on the engine to ensure both evolutionary behaviour and the correctness of implemented operators. As mentioned, the engine suffered a major overhaul with the upgrade to version 2 of TensorFlow. At this point, TensorGP's implementation was already in an advanced stage, which meant testing and validating most features twice.

An initial batch of tests was performed to verify the integrity of Tensorflow vectorized operators. This test consisted of a series of asserts performed over desired operator properties to test different use cases. All of the protection mechanisms described in Table 5.1 were validated at least once through this method.

In the context of validating the evolution process itself, three problems were studied. The first problem tested on the engine concerned the file size maximization of Joint Photographic Experts Group (JPEG) generated images. In this experiment, the main focus was solely to confirm the engine's ability for evolution. Later, TensorGP was applied to the symbolic regression task of approximating a polynomial function using a simple Mean Squared Error (MSE) metric for fitness assessment. Besides confirming evolution, this experiment also analysed performance differences between CPU and GPU.

Along with the TensorFlow 2 overhaul, some quality of life features regarding management of the primitive set and operator definition were added to the engine as well as a few genetic operator improvements, which meant feature re-validation. For this reason, the engine was then applied to three symbolic regression problems in the context of image evolution.

It is worth noting that tests conducted in the first two experiments (JPEG compression and polynomial symbolic regression) were performed with version 1 of TensorFlow using graph execution mode. For this reason, performance is not up to par with the latest engine implementation which uses version 2 in eager execution mode. Still, we must reiterate that experiments in this section are not meant to showcase engine performance but rather to validate the ability for evolution of a population towards an objective.

6.1.1 JPEG compression

This first experiment aims to maximize the size of a JPEG image. In this scenario, fitness assessment was performed by generating and saving the image corresponding to the individual's expression in JPEG format. The size of the produced JPEG was then read and attributed to the fitness value for that individual.

Figure 6.1 shows the image of an individual produced during evolution. This individual takes advantage of a self-repeating pattern and modifies it with trigonometric operators, just enough to trick the JPEG compression algorithm. To put sizes in perspective, Figure 6.1 was originally nearly 2MB whereas the initial population revealed an average size of a few KB.

This pattern seems to indicate some degree of evolution. Nevertheless, in an effort to demonstrate evolution in a more traditional context, fitness and timing values were then analysed for a symbolic regression problem.

Parameter	***Value***
Runs	10 (average)
Maximum Tree Depth	10
Population Size	50
Generations	20
Objective	Maximize JPEG file size

Table 6.1: Experimental setup for the JPEG evolution experiment.

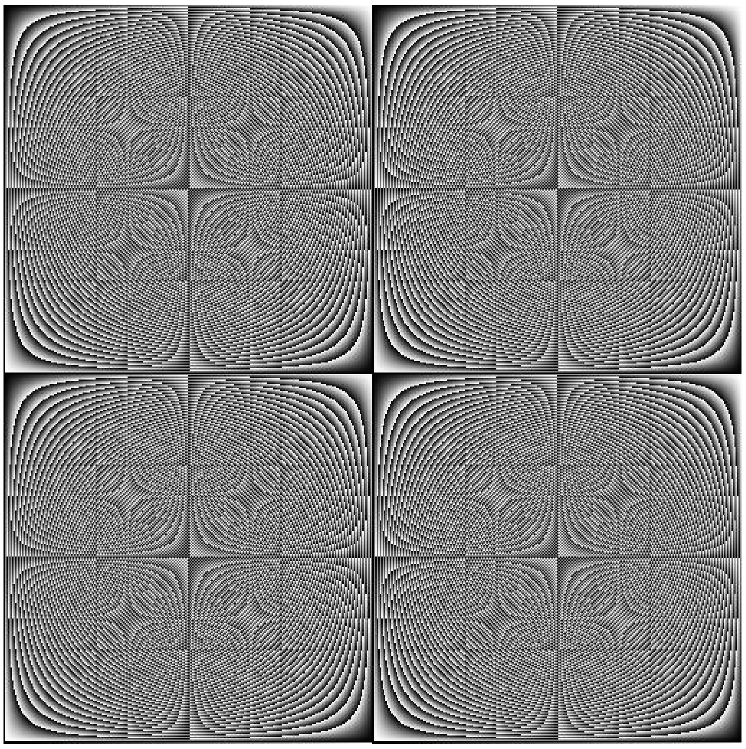


Figure 6.1: Artifact resulting from the task of maximizing JPEG size. Best of generation 5 out of 50 individuals.

6.1.2 Symbolic Regression - Paige Polynomial

To further verify the integrity of the evolution process, the engine was then tested on a classical symbolic regression task. The objective is to approximate the polynomial function defined by:

$$f(x, y) = \frac{1}{1 + x^{-4}} + \frac{1}{1 + y^{-4}} \tag{6.1}$$

This function is also known as Pagie Polynomial and is often regarded as difficult to approximate. As proposed by Lourenço et al. [30, p. 13], the domain considered is defined in the $[-5, 5]$ interval, with a step of $s = 0.4$, totalling 676 data evaluation points (26 points in each dimension).

In Figure 6.2, we see the results of running this setup with evolution for a population of 100 individuals for 100 generations. We conclude that there is in fact evolution, as the error from the target function decreases with passing generations. As shown, the MSE score of the best individual decreases from almost 0.2 to slightly more than 0.14 in the final generation.

Next, we benchmarked this task for GPU and CPU, but now measuring fitness evaluation time. The configuration is the same apart from the population (now 20) and the number of generations, which was modified to meet the specified number of evaluations.

As shown in Figure 6.3, the GPU takes slightly more time on average than the CPU. This happens because the number of programs to evaluate is reduced (a few hundred) and the evaluation domain is also rather small. This translates into a performance penalty when performing the vectorization process on the GPU instead. In fact, the timing difference

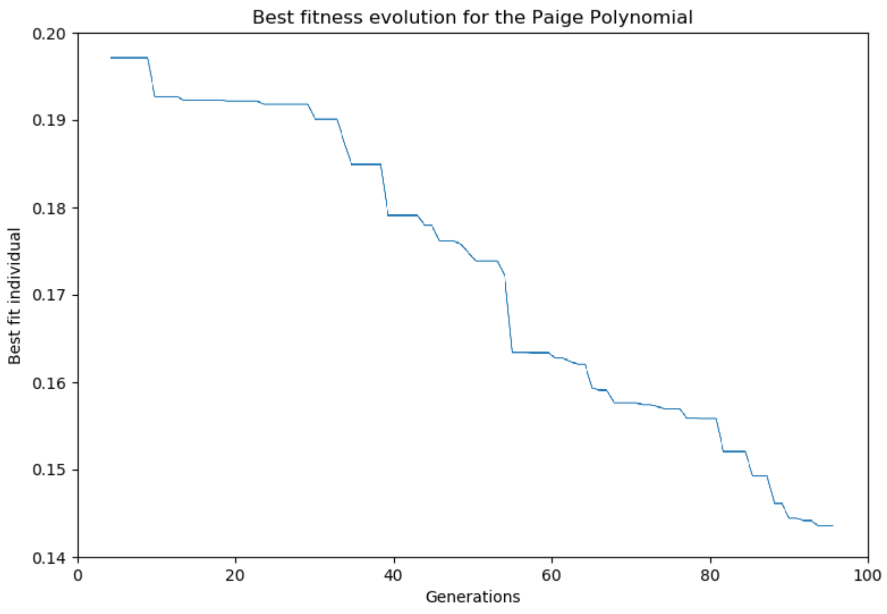


Figure 6.2: Evolution of the MSE metric for the best fit individual over 100 generations for the Pagie polynomial, 676 fitness points.

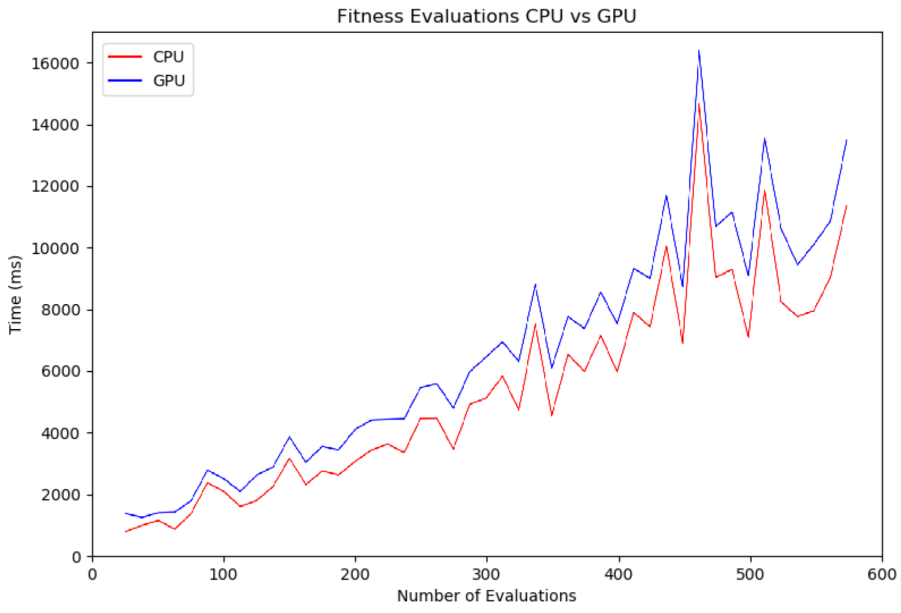


Figure 6.3: Benchmark of time taken to perform n expression evaluations in CPU (red) and GPU (blue).

seen in the graph corresponds to the overhead of copying data from the CPU to the GPU and then back to the CPU when needed.

In upcoming experiments, we will analyse how different approaches (iterative and vectorized) compare against each other for larger problem domains.

Parameter	***Value***
Runs	5 (average)
Maximum Tree Depth	10
Population Size	100 (fitness evolution); 20 (CPU vs GPU benchmark)
Generations	100 (fitness evolution); dynamic (CPU vs GPU benchmark)
Objective	Minimize error from Pagie Polynomial

Table 6.2: Experimental setup for Symbolic Regression - Pagie Polynomial.

6.1.3 Image Evolution

So far, outside of early JPEG evolution and other functionality testing, we have been only experimenting with a symbolic regression problem within a more traditional setting. This experiment aims to extend upon the previous symbolic regression results while also serving as a revalidation tool for new features in TensorGP. Results in this section and from this point on were obtained using TensorFlow 2 running in eager execution mode, unless expressed otherwise.

Indeed, any image can be represented through a mathematical expression, no matter how complex it might be. This means that we can effectively generalize symbolic regression to the evolution of any initial set of images towards any other target image. For instance, the Pagie Polynomial function when taken in the $[-5, 5]$ domain for both x and y can be represented with the image in Figure 6.4.

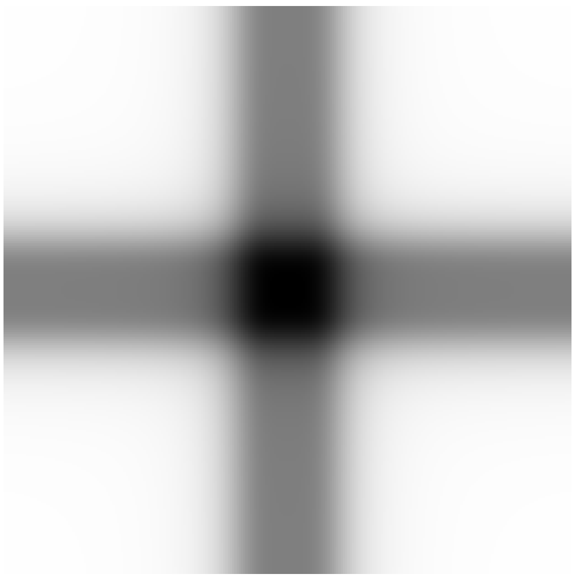


Figure 6.4: Phenotype representation of the Pagie polynomial function 6.2 for all (x, y) in $[-5, 5]$.

Akin to other regression problems, the initial batch of images should evolve to visually approximate whatever image is defined as target. In the area of GP, this problem of image evolution towards a target is not a novelty. However, the success of research tackling this problem is relative to the problem complexity.

Generally speaking, the main issue is that for highly complex target images (images with complex mathematical representations), the evolutionary process might require a large number of individuals and generations to attain meaningful results. This amount of individuals and generations leads to a large number of evaluations and an equally large experimental run time. In their work, DiPaola et al. [13] attempted to evolve a rather complex picture of Darwin's gaze, letting the experiment run for about 50 days. Impressively, even with limited computational resources, the best fitted individuals started resembling the target image for certain features by the end of the run. Even so, the fact that such a large amount of time was needed to achieve some degree of feature resemblance really puts in perspective how complex this task can be.

The reason for this picture of Darwin being so inherently complex is because it was originally a painting created by a human, which adds a difficult layer of detail for a computer to simulate. Nevertheless, no matter how unfathomably complex an image might be, we can always represent it in a digital format, even if we have to individually define a value for every pixel with a conditional operator.

Commonly in GP, assessing fitness for symbolic regression problems is done through measures like Root Mean Square error (RMSE) or MSE (which was used for the previous experiment). However, a known shortcoming of the RMSE metric when applied to symbolic regression is that it may end up over benefiting solutions that are constant on the average domain value. In the context of an image this means preferring individuals only containing a single color close to the average of the target image. As a countermeasure, the Structure Similarity Index Measure (SSIM) metric was implemented and benchmarked against the more common RMSE.

$$SSIM(x,y) = \frac{(2\mu_x\mu_y + C_1) + (2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)} \tag{6.2}$$

where:

μ_x, μ_y	= mean of x and y, respectively
σ_x^2, σ_y^2	= variance of x and y , respectively
σ_{xy}	= co-variance between x and y level
C_1, C_2	= $(k_1 L)^2$, $(k_2 L)^2$
L	= dynamic range of color channels

SSIM has the advantage of taking into account image structure and gradients, therefore penalising images containing different shapes from the target. This metrics accepts two sets of data (x and y in Equation 6.2) and determines the visual impact of shifts in image luminance, changes in contrast, as well as any other remaining errors which are collectively referred to as structural changes [43]. The internal TensorFlow implementation of SSIM follows a default of $k_1 = 0.01$ and $k_2 = 0.03$ for division stabilization. In our case, each color channel is 8 bits, which translates to a dynamic range of $L = 255$.

As a side note, SSIM is a maximization metric that outputs values in the $[-1, 1]$ range (1 representing equality amongst two sets of data). RMSE on the other hand, is a minimization metric that only outputs positive values (0 representing perfect similarity amongst two sets of data). Due to the fact that RMSE ranges depend on the data to be analysed and the scale used is different from SSIM, numerical comparisons between both metrics are of little help. In fact, there is not an obvious way to directly compare these two metrics because RMSE is an absolute error metric while SSIM represents perceptual and saliency-based errors [43]. Nonetheless, we will make an attempt to interpret the test results critically from a visual standpoint.

In total, 3 images with an increasing level of complexity were tested. These images were generated using the norBErT evolutionary art tool [48]. The first test concerned the evolution towards a gray-scale image composed of simple shapes with a size of 128 by 128 pixels. A population composed of 50 individuals was evolved for 200 generations.

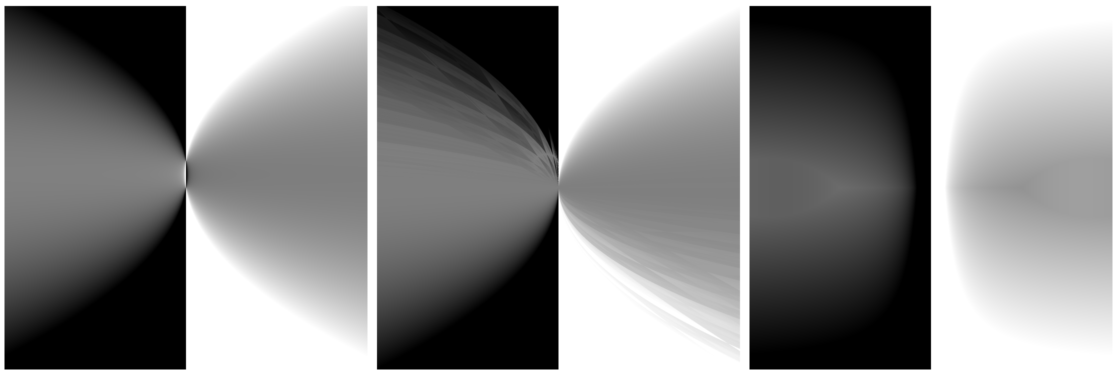


Figure 6.5: Target image (left) with best individuals generated with RMSE (middle) and SSIM (right) - Black and white image.

The results verify evolution, with both metrics managing to visually approximate the target image. RMSE seems to achieve a more accurate solution in this particular test, as shown in Figure 6.5. Despite this, the final SSIM solution is both significantly simpler (Figure 2) while still reaching a SSIM index of 0.9 (Figure 1). RMSE was faster with a total engine time of 263 seconds compared to 311 seconds for SSIM.

Even though these results might seem encouraging, tests on slightly more complex colored images were conducted instead of jumping straight to evolution with photorealistic images. In the next two experiments, the number of generation was increased to 5,000 and the size of corresponding images decreased to 64 by 64 pixels.

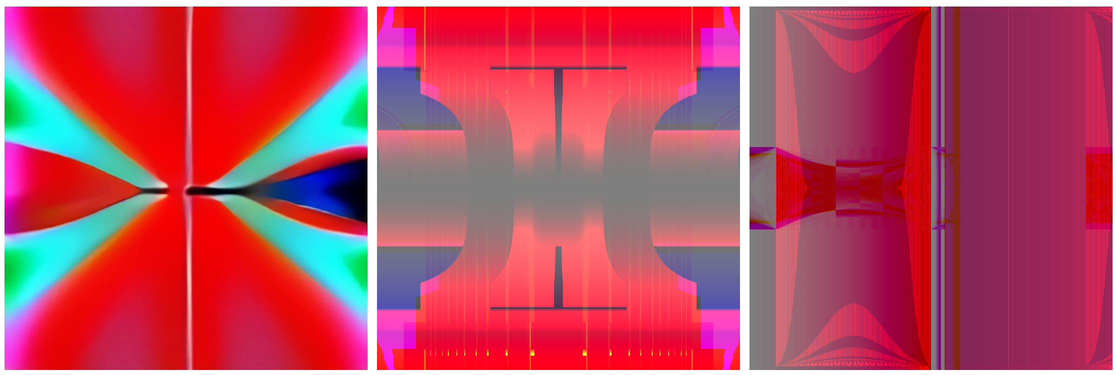


Figure 6.6: Target image (left) with best individuals generated with RMSE (middle) and SSIM (right) - First colored image.

For this run, we observe that RMSE manages to attain a slightly more similar image in relation to the target when compared to SSIM (Figure 6.6). Even though both metrics max out tree depth (Figure 4), fitness seems to reach a stagnation point faster with RMSE (no major improvements past the 600 generation mark - Figure 3).

Despite the increase in generations, overall fitness results are worse than the ones verified for the first image, with a best of 68.6 for RMSE and almost 0.62 for SSIM. Experiments also took significantly longer with around three and a half hours of total engine time for RMSE and 10 more minutes for SSIM.

For the second colored image, it is unclear which metric manages to better approximate the target from a visual perspective (Figure 6.7). However, RMSE results for the overall

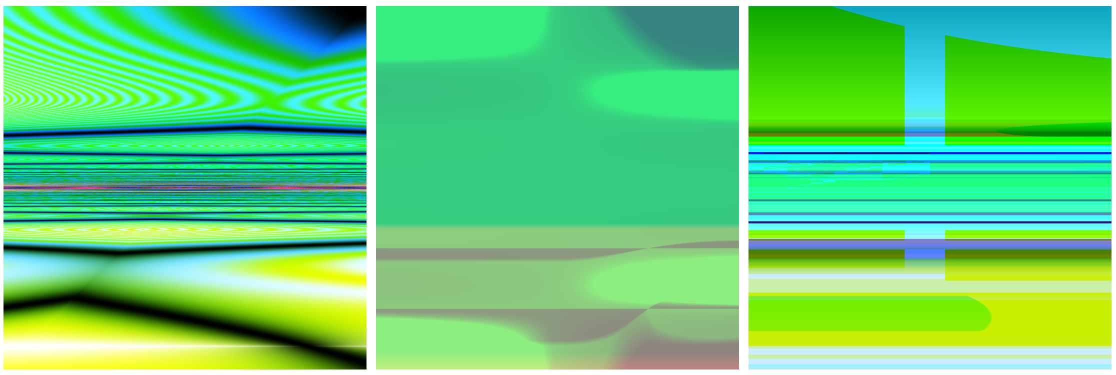


Figure 6.7: Target image (left) with best individuals generated with RMSE (middle) and SSIM (right) - Second colored image.

best individual are marginally worse (Figure 5) when compared with those of the previous image (68.6 and 72.8 respectively), whilst the SSIM metric shows a small improvement (0.62 against 0.72 over last image).

Once again, both metrics max out available tree depth after a couple of generations, indicating that evolution towards more complex targets might benefit from larger and deeper trees (Figure 6). For this run, RMSE was significantly faster, with a modest 6 hours of engine time versus 17 hours and 14 minutes with SSIM.

Preliminary tests for photorealistic images proved not to be largely insightful due to regular fitness plateaus for small tree depths. Such experiments would most likely require evolving much bigger individuals throughout thousands of generations in order to obtain noteworthy results as seen in [13]. Therefore, these tests were deemed too computationally demanding to be performed inside the scope of mere validation testing.

Parameter	***Value***
Runs	Single
Maximum Tree Depth	10
Population Size	50
Generations	200 (first image); 5,000 (last two images)
Objective	Minimize error from target image

Table 6.3: Experimental setup for Symbolic Regression in the context of Image Evolution.

6.2 GPU versus CPU

Whereas last section mainly focused on making sure the engine is able to evolve suitable solutions and that operator functionality is as intended, this section on the other hand, is concerned with the time comparison of different GP approaches both in a more controlled environment and within a real-world scenario that features a complete evolutionary run.

Akin to the tests performed for the *warp* operator, the large time differences amongst approaches benchmarked in this section warrants the use of a logarithmic scale for graphical representation, as a linear scale would fail to discern between different results.

6.2.1 Tree evaluation

As mentioned previously, TensorFlow evolved since the early testing phase, introducing the eager execution mode which allows for immediate tensor execution without the need to build the intermediate DAG. The graphs are different for each individual and amongst generations, so it would make sense for eager execution to have an advantage, at least while the evolutionary process does not reach the exploitation of one specific type of individual.

To further understand the performance implications of both execution modes when compared to more traditional GP methods, the evaluation of a fixed batch of populations was benchmarked. This experiment intends to both compare TensorFlow execution modes and showcase the raw evaluation power of the GPU in comparison to the CPU.

In total, six approaches were compared. Four of these approaches concern the GP engine implementation: testing both graph and eager execution modes when running in the GPU versus CPU. The other two approaches implement serial GP evaluation methods: one resorting to the DEAP framework and another one using a modified version of the engine that evaluates individuals with the *eval* Python function instead of TensorFlow. The next two paragraphs describe the last two approaches.

Perhaps one of the most commonly used GP tools, DEAP is an Evolutionary Computation (EC) framework written in Python that offers a powerful and simple interface for Experimentation [16]. We have chosen to include comparisons to this framework because it represents the standard for iterative domain evaluation in research and literature around GP. Moreover, DEAP is easy to install and allows for the prototyping of controlled environments within a few lines of code.

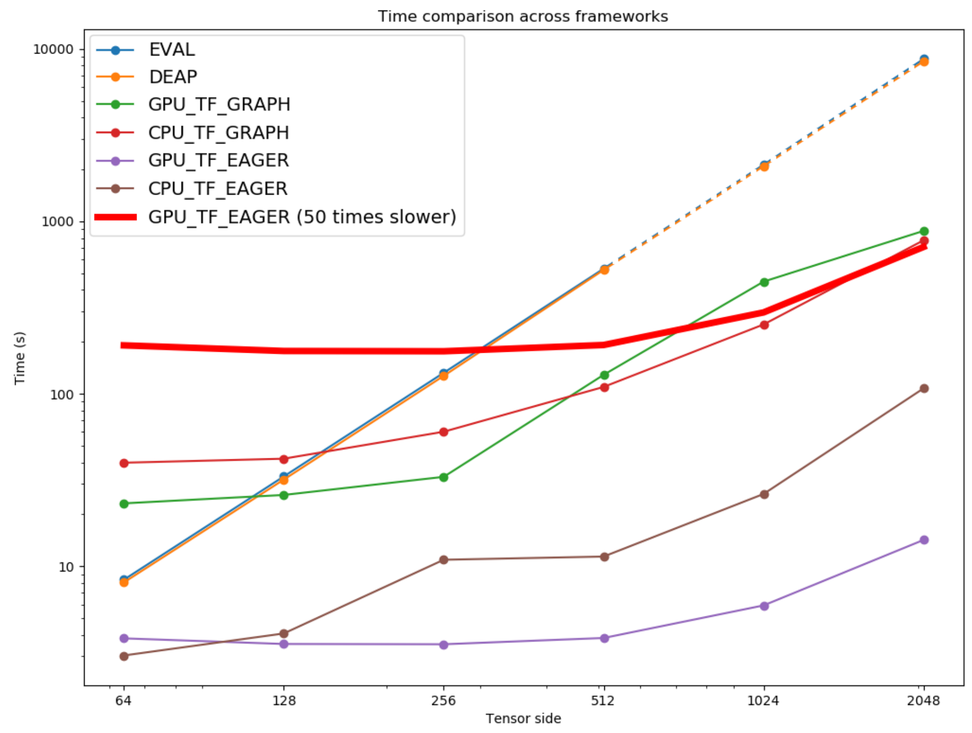


Figure 6.8: Time comparison of different approaches for tree evaluation.

Furthermore, we also include our own serial baseline. The purpose is to compare achievable timings for an iterative approach that does not use third-party software. We do this by passing the expression of an individual to the Python *eval* method so we can run it. However, this method proves to be slow because it parses the input and then runs the produced Python code. Therefore, calling *eval* for all domain values may not be feasible. Instead, we only call *eval* once by plugging in a lambda defined expression of the individual. This will effectively return a function that we can call for any value, eliminating the overhead of parsing code. This method turns out to be way faster as we are basically "compiling" the expression/program before running it. For future reference, this approach will be referred to as EVAL.

In this experiment, we run a static batch of 5 populations for each test case. Every population consists of 50 individuals and was generated using the RHH method with a maximum tree depth of 12 as defined in Table 6.6. For all six described approaches we run the population batch for rank 2 tensors ranging from 64 by 64 (4,096 elements) all the way to 2,048 by 2,048 (over 4 million evaluation points). Larger domains were not tested mainly due to VRAM limitations of the GPU.

Figure 6.8 shows the average time taken for the evaluation of all 5 populations in all test cases. We can conclude that both EVAL and DEAP results are similar, following a linear increase in evaluation time with an increase in evaluation points.

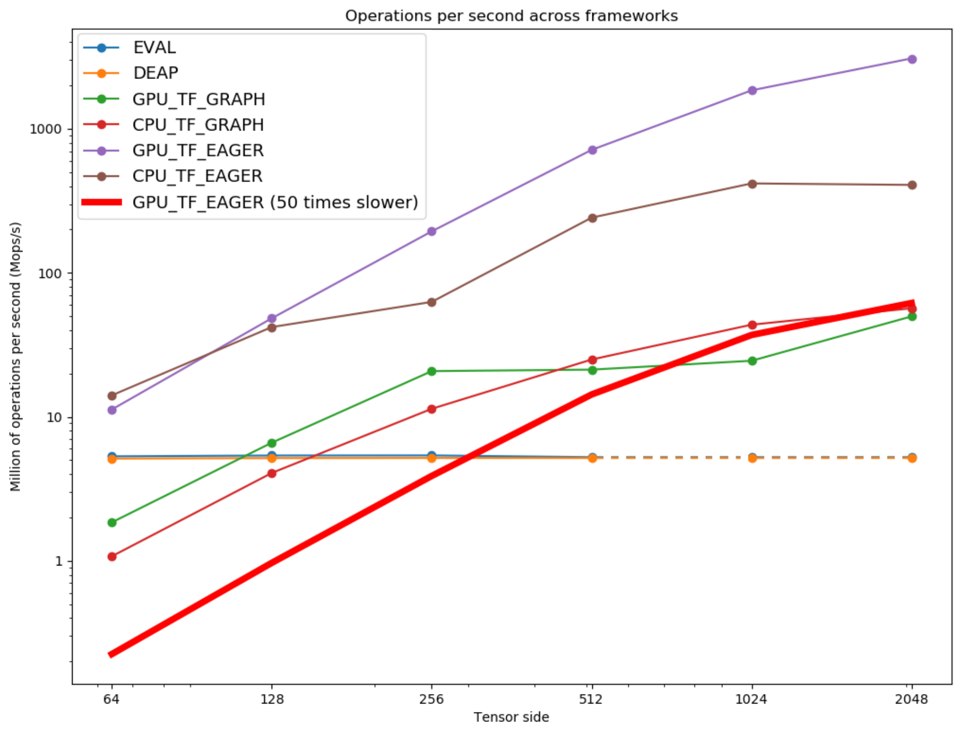


Figure 6.9: Operations per second comparison of different approaches for tree evaluation.

Because there is no domain vectorization, the direct relation between elements and time taken comes as no surprise. It is worth noting that because of time constraints, results corresponding to the dashed lines in the two largest tensor sizes were not run but instead predicted by following the linear behaviour from previous values.

		EVAL	DEAP	TF graph (GPU)	TF graph (CPU)	TF eager (GPU)	TF eager (CPU)
64	**AVG**	8.346	8.050	23.138	39.813	3.815	3.036
	STD	0.219	0.201	0.564	0.842	0.217	0.076
128	**AVG**	33.132	31.779	25.885	42.003	3.540	4.073
	STD	0.818	0.885	0.497	1.067	0.128	0.112
256	**AVG**	132.277	126.930	32.971	60.224	3.527	10.898
	STD	3.362	3.499	0.773	1.661	0.131	0.112
512	**AVG**	531.153	522.531	128.748	109.530	3.834	11.373
	STD	12.756	13.973	2.792	2.920	0.186	0.920
1024	**AVG**	DNF	DNF	446.900	252.232	5.924	26.233
	STD	DNF	DNF	8.940	21.268	0.225	1.056
2048	**AVG**	DNF	DNF	879.737	775.540	14.197	107.416
	STD	DNF	DNF	31.784	58.375	0.370	4.739

DNF stands for "Did Not Finish"

Table 6.4: Standard deviation (STD) and average (AVG) of timing values for the tree evaluation experiment (in seconds).

Analysing results for TensorFlow eager execution mode, we see that even though the CPU is faster for smaller domains (64 by 64), this trend fades rapidly for larger problems. In fact, for 128 by 128 (16,384) elements, the GPU is already marginally faster than the CPU. This margin widens with an increase in tensor side, resulting in GPU evaluations over a 2,048 by 2,048 domain being almost 8 times faster. As a reminder, with TensorFlow we are already providing operator vectorization, hence the 8 fold increase is merely a product of running the same approach on dedicated hardware.

Moreover, we can also confirm the hypothesis that the evolutionary process in GP benefits from eager execution. For the two smaller domain sizes, we observed that the slowest eager execution approach is about 8 times faster than the fastest graph execution one. This trend continues for larger domains.

		EVAL	DEAP	TF graph (GPU)	TF graph (CPU)	TF eager (GPU)	TF eager (CPU)
64	**AVG**	5.123	5.311	1.848	1.074	11.233	14.086
	STD	0.013	0.029	0.007	0.013	0.631	0.126
128	**AVG**	5.162	5.382	6.607	4.072	48.336	41.997
	STD	0.011	0.039	0.066	0.039	0.695	0.243
256	**AVG**	5.172	5.390	20.751	11.361	194.072	62.777
	STD	0.013	0.028	0.420	0.162	3.287	1.335
512	**AVG**	5.172	5.237	21.255	24.986	714.443	241.604
	STD	0.033	0.029	0.287	0.326	18.340	15.924
1024	**AVG**	DNF	DNF	24.493	43.617	1850.492	417.695
	STD	DNF	DNF	0.361	3.444	19.868	17.234
2048	**AVG**	DNF	DNF	49.788	56.708	3084.161	407.904
	STD	DNF	DNF	0.824	4.476	23.578	8.984

DNF stands for "Did Not Finish"

Table 6.5: Standard deviation (STD) and average (AVG) of operation per second values for the tree evaluation experiment (in Mops/s).

However, results gathered for graph execution show rather unexpected behaviour. As suggested by past experiments, it would be safe to assume that the CPU would be faster for small domains with the GPU taking over for larger ones.

In reality, the opposite is happening: the GPU is faster for domains up to 256 in size, from which point on the CPU takes over. The answer to this strange behaviour may lie in the graph implementation that TensorFlow uses. Fitting every individual of a population in one session graph proved to take too much memory for larger domains, making these approaches even slower both in GPU and CPU. This need for memory is especially taxing for the GPU VRAM (which is only 3GB versus the available 16GB for system memory) that did not even finish some test cases trying to include the entire population in a single graph. For this reason, and to be consistent with all domain sizes, we decided to test these graph execution approaches by opening a session graph for each individual instead of evaluating the entire population in one graph. Still, we can safely conclude that both graph execution approaches are slower than their eager equivalents.

Similarly to the warp operator experiments in Section 5.5.1 of the last chapter, the red bold line represents a 50 times performance threshold over our fastest approach (TensorFlow eager on GPU), which reached a peak of over 3,000 Mops/s as shown in Figure 6.9.

This turns out to be a rather steep performance drop when compared to the amount of operations per second demonstrated with the warp operator. The main reasons being more complex memory management (the chaining of operations fills the GPU VRAM which needs to free space by sending unnecessary data back to the CPU) and operator safety methods which include both the protection mechanisms described in Table 5.1 as well as Not a Number (NaN) checks.

Parameter	***Value***
Runs	5 (average)
Maximum Tree Depth	12 (all 5 populations only had this depth)
Population Size	50
Generations	1
Objective	- (no evolution)
Test cases	6 (2^{2n}, for n in $[6, 11]$)

Table 6.6: Experimental setup for Tree Evaluation.

6.2.2 Evolutionary run

Last experiment results only concern the tensor evaluation phase, stripped down of both the fitness evaluation phase and the remaining evolutionary process. This setup certainly does not constitute a typical real-world scenario within the context of GP research.

For that reason, in this subsection, we once again take the previous Paige polynomial symbolic regression problem and let the same initial populations evolve for 50 generations. Although the main purpose of this experiment is to benchmark timings, we will also analyse fitness progression across generations to make sure that evolution is verifiable for all approaches.

In this experiment, only four out of the six previously analysed approaches were compared. TensorFlow graph execution was excluded due to the fact that both its GPU and CPU approaches proved to be slower than any other running in eager execution mode. Because our aim is to confirm speedup between our fastest TensorFlow approaches and the more standardized iterative ones, we chose to compare TensorFlow eager execution against the baseline (EVAL) and DEAP.

Regarding our baseline approach, fitness was assessed by reverting the tree representation of an individual back to a string and then following the previous algorithm of compiling a function with Python's *eval* method. For DEAP, the *eaSimple* evolutionary algorithm was used along with a *PrimitiveTree* object to represent each individual. There is an additional struggle with this framework because the number of assessed individuals does not remain constant throughout generations. Instead, the population size depends on the offspring from past populations, which further obfuscates direct comparisons.

Figure 6.10 shows total run times for all considered approaches. Probably the most noticeable aspect of these results is that they appear to be less linear in nature when compared to those regarding raw tree evaluation. This happens because, even though we are using a fixed population batch for each test case, evolution might be guided towards different depths for different initial populations. If the best fitted individual happens to have a lower depth value, the rest of the population will eventually lean towards that trend, lowering the overall average population depth and thus rendering the tensor evaluation phase less computationally expensive. The opposite happens if the best fitted individual is deeper, resulting in more computation time. This explains the relatively higher standard deviations presented in Table 6.7.

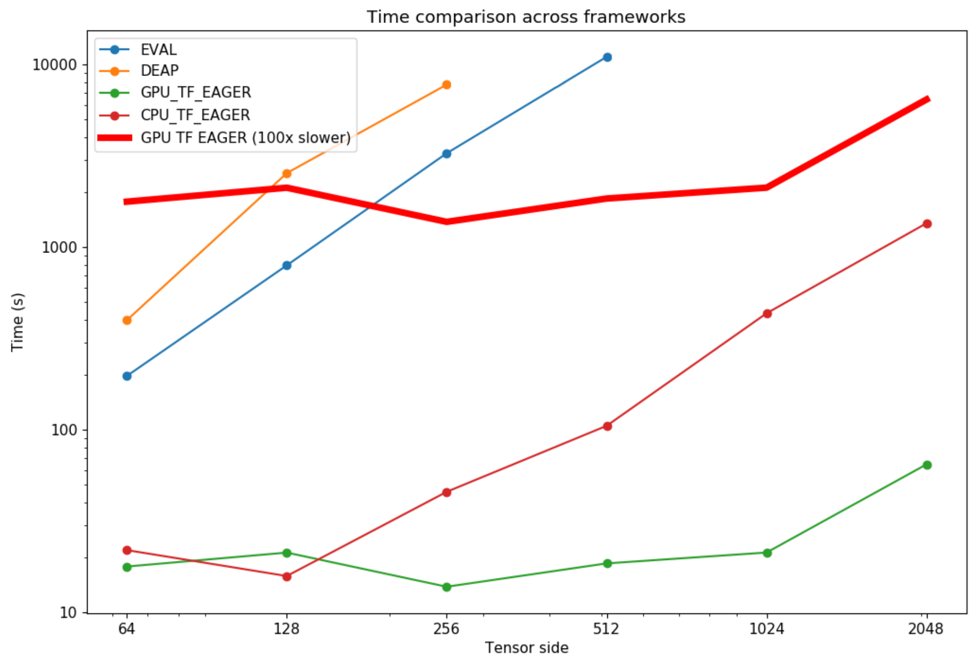


Figure 6.10: Time comparison of different approaches for a full evolutionary run.

In reality, even for the same initial population, the evolutionary process might benefit different programs for different problem sizes. An expression that roughly approximates a 4 by 4 domain might fail to approximate a larger one, and even though the contrary might not be true, evolution rarely finds the most efficient solution. Basically, this means that given the exact same populations and parameters (other than domain size), it is possible for the best fitted individual achieved for a small domain to be more complex than another one reached for a bigger domain. In this scenario, if the size discrepancy is big enough, the overhead incurred by evaluating bigger individuals might very well outweigh the higher number of evaluation cases, making the evolution process slower for the smaller domain.

This also explains the non-linear behaviour across problem sizes. In specific, we observe that for the TensorFlow GPU approach (GPU_TF_EAGER), the test case for size 256 runs faster than the two smaller domains. This happens because, as verified by analysing previous results for this approach (Tables 6.4 and 6.5), domains up to a certain size roughly take the same amount of time to evaluate in the GPU due to the massive overhead of copying data to and from the device. It just so happens that the 256 test case saw a low enough average depth of individuals to verify such a decrease in computation time (Figure 6.11).

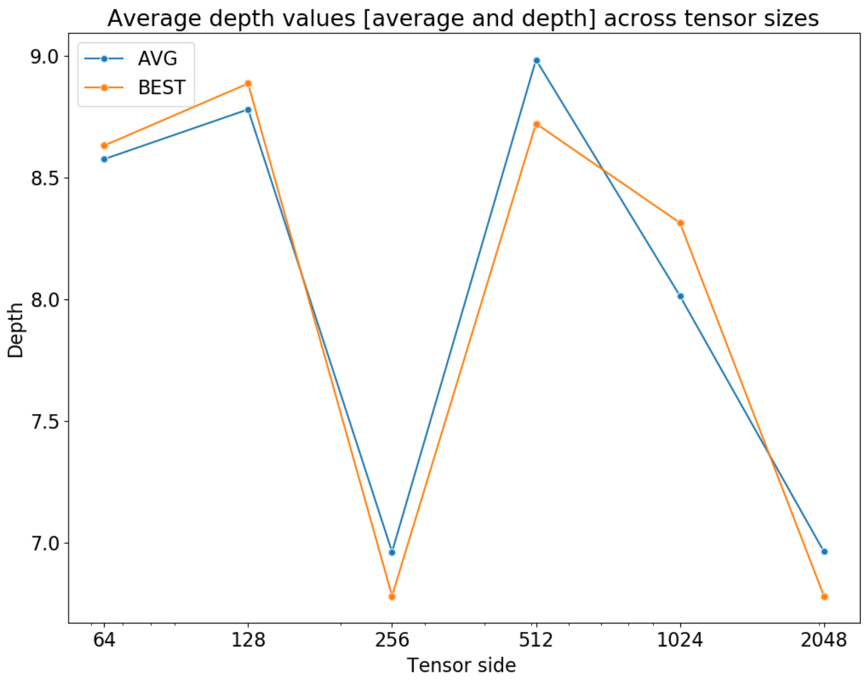


Figure 6.11: Average depth values across test cases for the GPU_TF_EAGER approach.

It is worth noting that in this experiment the tensor side doubles for each consequent test case, making the evaluation of a given expression 4 times slower. Assuming we use the full method for tree generation and a simplification of our primitive set to have arity 2 for all operators, it would only take an increase of 2 average depth levels to make evolution more than 4 times slower (and therefore more taxing than doubling the tensor side).

Still regarding TensorFlow results, from the two first test cases, we can not identify a clear preference towards CPU or GPU as for these domain sizes the GPU memory transfer overhead is on par with the lack of CPU parallelization power. Nonetheless, for domains larger than 256 in tensor side, a clear preference towards GPU starts to be evident, with an average speedup of over 21 times for a 2,048 by 2,048 problem size (64 seconds average for the GPU against 1,353 seconds for the CPU approach).

Perhaps the most unexpected results are the test cases for the DEAP framework, which are consistently slower than even the EVAL baseline. In tree evaluation we saw that domain calculation is slightly faster in DEAP than in our baseline. However, as mentioned, DEAP uses dynamic population sizes during evolution which might slow down the run. It is also worth mentioning that only the basic genetic operators and algorithms were used for DEAP. More extensive experimentation with the evolutionary capabilities of this framework would most likely reveal a more optimal set of genetic operators and parameters that could be faster than EVAL. Even so, that is not the aim of this work and so we shall compare TensorFlow timings against our baseline, which follows the same iterative principle.

In turn, EVAL proves to slower than any of the TensorFlow approaches for all considered test cases, with an average verified speedup of almost 600 times over GPU_TF_EAGER for the 512 domain test case (18 and a half seconds average for the GPU in TensorFlow against 11,052 seconds or slightly over 3 hours for EVAL). For both iterative approaches (DEAP and EVAL), tests that on average took longer than 5 hours for a single population were not completed as they would steal computation time from other experiments (to put these times in perspective, EVAL would take almost two weeks to complete if it were to follow the displayed behaviour for smaller test cases).

Clearly, the results shown for the GPU in TensorFlow are fast, maybe even too fast. Indeed, with the evolutionary process thrown in the mix, it would be safe to assume that the speed up between iterative and vectorized approaches would shorten, even if marginally. This seems to not be the case, in fact, speedups are higher when compared to tree evaluation experiments. Previous results with 512 tensor side for TensorFlow GPU against EVAL regarding raw tree evaluation shows a speedup of almost 140 times, which is a far cry from the aforementioned nearly 600 times confirmed with evolution.

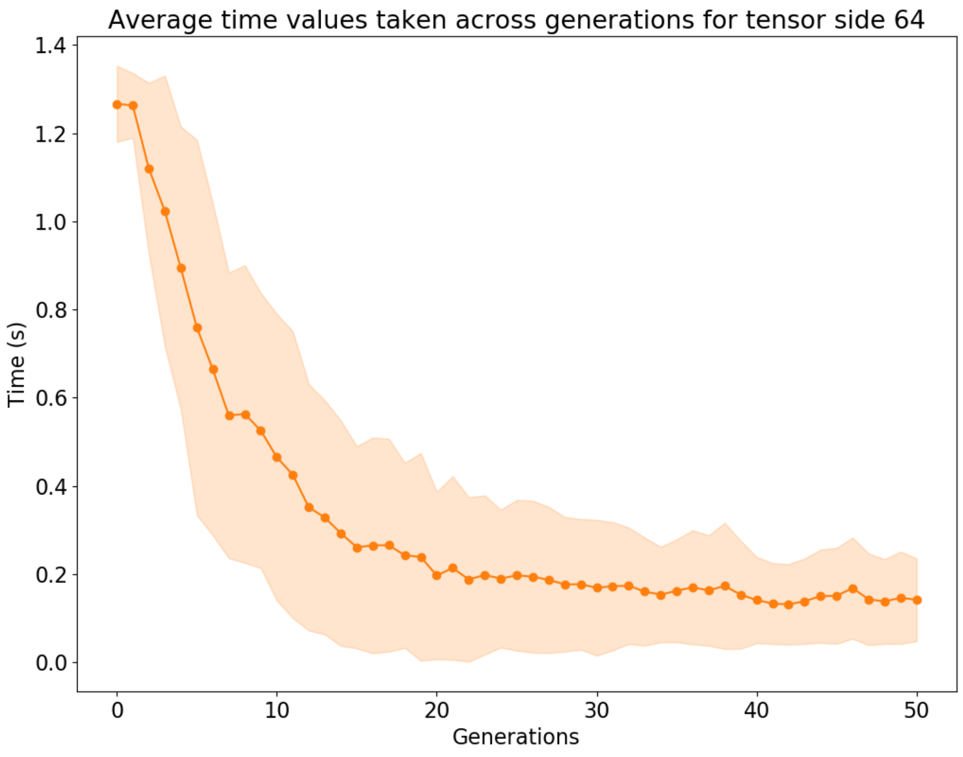


Figure 6.12: Average evaluation time across generations for test case 64 in the GPU _TF_EAGER approach.

This can be explained by the caching of intermediate results that TensorFlow performs even when executing in eager mode. While neither DEAP or EVAL use any memory during expression calculations, TensorFlow can store results concerning parts of individuals that remain unchanged through the generations. This leads to a pronounced decrease in evaluation time after the first few initial generations, as observed in Figure 6.12. These results further make the case for expression evolution with TensorFlow.

Finally, with the aim of confirming evolution in this experiment, Appendix B contains graphs regarding fitness evolution across generations for all studied approaches.

		EVAL	**DEAP**	**TF eager (CPU)**	**TF eager (GPU)**
64	**AVG**	196.866	397.821	21.859	17.766
	STD	96.264	339.495	13.439	8.287
128	**AVG**	795.363	2546.834	15.761	21.189
	STD	381.714	2127.378	7.852	11.986
256	**AVG**	3274.131	7783.763	45.640	13.769
	STD	2482.257	5824.779	20.788	7.390
512	**AVG**	11052.164	DNF	104.960	18.486
	STD	3887.540	DNF	30.212	8.750
1024	**AVG**	DNF	DNF	434.372	21.205
	STD	DNF	DNF	160.051	9.979
2048	**AVG**	DNF	DNF	1353.666	64.535
	STD	DNF	DNF	679.557	32.505

DNF stands for "Did Not Finish"

Table 6.7: Standard deviation (STD) and average (AVG) timing values for the evolutionary run experiment (in seconds).

Parameter	***Value***
Runs	5 (average)
Maximum Tree Depth	12 (all 5 populations only had this depth)
Population Size	50
Generations	50
Objective	Minimize error from Pagie Polynomial
Test cases	6 (2^{2n}, for n in $[6, 11]$)

Table 6.8: Experimental setup for Evolutionary Run.

6.3 Evolutionary art with NIMA

With the increase in tensor evaluation speed and consequently of the evolutionary process made possible by our engine, the potential for computer evolutionary art seems enticing. Specifically, in this section, we will apply TensorGP to the problem of producing visually appealing imagery with the aid of an image quality assessment classifier. We will first provide a comparative benchmark between DEAP and our GPU approach using TensorFlow in order to further corroborate the advantages of GPU in this type of experimentation. Afterwards, relying on the GPU parallelization capabilities, we will relax specific GP parameters and analyse the resulting effects on image generation.

For this task, we need a way to rank individuals according to how visually appealing they seem to humans. As explained at the end of Chapter 2, this fitness assessment process is non-trivial when it comes to evolutionary art.

Because the images need to be appealing from a human perspective, one obvious way to deal with fitness assessment would be to resort to human supervision. However, this option would not only slow down the evolutionary process which is the main enabler for the exploration of this theme but would also be rather time consuming for the human expert. For this reason, we decided to make use of the Neural Image Assessment (NIMA) classifier for fitness ranking. NIMA tackles the problem of quantification of image quality and aesthetics with a deep Convolutional Neural Network (CNN) that is trained to predict which images a typical user would rate as visually appealing, either from a technical or aesthetic standpoint [47].

NIMA makes use of Keras, which is a Python module that provides a set of deep learning models through TensorFlow's backend. The implementation of NIMA used is publicly available at [28] and allows the use of pre-trained CNNs in Keras. For this experimentation in particular, we are interested in the NIMA aesthetic network, trained with the Atomic Visual Action (AVA) image datasets specifically for assessing image aesthetics [37].

The classifier internally uses images with a resolution of 224 by 224 pixels and therefore all the domains considered for this section have dimensions $[224, 224, 3]$, to accommodate image resolution and the RGB color channels as previously done with experiments in targeted image evolution (Section 6.1.3). Besides, NIMA outputs two values for each image: the main score and an associated error. For simplicity, fitness in the following tests is calculated using only the main score of an image.

Moreover, apart from total engine time, values for fitness and tensor evaluation will be analysed to better understand the contribution of each evolution phase in overall system performance. "Fitness" timings refer to the time taken for classification with NIMA, "Tensor" corresponds to expression evaluation time and "Engine" indicates total time taken for the whole run.

6.3.1 GPU vs CPU

In this experiment, we show performance differences between the DEAP framework and our eager GPU approach in TensorFlow using NIMA (simply referred to as TF GPU in subsequent results). For this purpose, the aesthetic classifier was integrated with both approaches and applied to the same initial population batch used for previous experimentations in the context of CPU versus GPU (Section 6.2).

	Engine		**Tensor**		**Fitness**	
	Avg.	Std.	Avg.	Std.	Avg.	Std.
DEAP	20828.926	22126.456	20808.229	22123.844	17.928	0.659
TF GPU	40.860	8.862	21.508	6.022	10.857	0.817

Table 6.9: Average and standard deviation timings values (in seconds) for GPU versus DEAP evolution with NIMA.

The preceding table shows average and standard deviation values for the considered metrics amongst all 5 runs. As expected, the TF GPU approach is significantly faster than DEAP, with an average speedup of over 500 times for total engine time and almost 1,000 times for tensor evaluation. Again, even taking into account the power of the GPU, these results should be taken with a grain of salt as we are dealing with different engines that follow different evolutionary pathways. Indeed, the higher standard deviations values imply a wide range of timing values, especially with DEAP (with results ranging from 2,700 to 60,000 seconds for minimum and maximum run times, respectively). Although deviations are not so pronounced for GPU, we should remember that the classifier resolution (224 by 224 pixels) translates to a domain that still entails some memory copying overhead for the GPU, as seen in Section 6.1.3. Furthermore, the caching of intermediate results greatly favors TF GPU timings.

Regarding DEAP, expression evaluation is shown to take the bulk of all computation time, while for TensorFlow the percentage of time spent on classification significantly increases.

As a sidenote, the slightly increased fitness evaluation time for DEAP derives from the fact that fitness is assessed on an individual basis, therefore forcing one call to NIMA per individual, whereas in TensorFlow we measure fitness for the whole population at once, which allows passing all corresponding tensors as a batch with a single classifier call. Besides, the gap between the sum of tensor plus fitness times and total engine time in our GPU approach is mainly due to TensorFlow initialization, generation of log files and some genetic operator overhead.

These experiments provide a rough estimate of achievable speedups in this scenario. However, to obtain more accurate overall results would mean performing more runs, which has proven unfeasible with the DEAP approach.

Parameter	***Value***
Runs	5 (average)
Maximum Tree Depth	12 (all 5 populations only had this depth)
Population Size	50
Generations	50
Objective	Maximize score of NIMA aesthetic classifier.

Table 6.10: Exprimental setup for TF GPU vs DEAP classification with NIMA.

6.3.2 Parameter Exploitation

The experimental setup used for benchmarking in the previous subsection proved to be a relatively light load for our engine, which allows some leeway to exploit desirable GP parameters in the context of evolutionary art. Mainly, we are interested in harnessing latent GPU potential to study how increasing depths, population size and number of generations affect NIMA guided evolution.

For this purpose, three different experimental setups were executed where we analyse system run times and display the most aesthetic individuals both according to classifier scores and from a human perspective.

	Engine		**Tensor**		**Fitness**	
	Avg.	Std.	Avg.	Std.	Avg.	Std.
SETUP 1	154.060	53.340	76.389	44.113	56.320	4.187
SETUP 2	1549.142	851.596	1020.004	678.013	237.839	27.847
SETUP 3	3893.523	595.160	1546.332	466.752	1982.522	20.945

Table 6.11: Average and standard deviation timings values (in seconds) for different setups for NIMA evolution.

	Runs	**Generations**	**Population size**	**Max Tree Depth**
SETUP 1	30	200	50	12
SETUP 2	30	500	100	15
SETUP 3	30	200	200	20

Table 6.12: Experimental setups for exploratory tests in evolutionary art with NIMA.

Apart from the experiment specific parameters in Table 6.12 and defaults defined in Table 5.5, every one of the following experiments were run with a crossover probability of 90% to further promote diversity. [32, p. 145].

The first setup is based upon the standard batch of 50 individuals that we have been using. Setup 2 is based upon [48] and intends to emulate another recent experiment in the realm of evolutionary art. For the last and most computationally intensive setup, we increase max tree depth to 20 to broaden the search space and revert the number of generations back to 200.

The first conclusion we can take from the timing results in Table 6.11 is that setup 3 is in fact, the most taxing on system resources. The drastic increase in NIMA evaluation time verified for setup 3 is mainly explained by an implementation detail that forced execution exclusively to the CPU for the classifier in this setup, or TensorFlow would otherwise crash due to memory overflow of the GPU VRAM. It should be noted that setup 3 does indeed entail the highest memory footprint because of population size. As expected, the smaller standard deviations values are detected for fitness assessment as the work done by the classifier is constant (always evaluate the same number of individuals per generation) whereas tensor times vary according to average population depth, types of operation favored by evolution and opportunities for caching of intermediate results.

Lastly, in the hopes of demonstrating some of TensorGP's potential for artistic generation, we compile a small gallery with the best individuals according to both the NIMA aesthetic classifier (Figure 6.13) and a human (and therefore subjective) perspective (Figure 6.14).

Perhaps the most noticeable aspect of these images is that the individuals with the highest fitness values are not always the most visually appealing. Nonetheless, this should not come as surprise. In reality, NIMA is trained using the AVA dataset which primarily uses images related to human activities, not necessarily works of art. Besides, there is always the underlying subjectiveness of sorting any kind of art, which undermines, even if marginally, the effectiveness of using a classifier for that purpose.

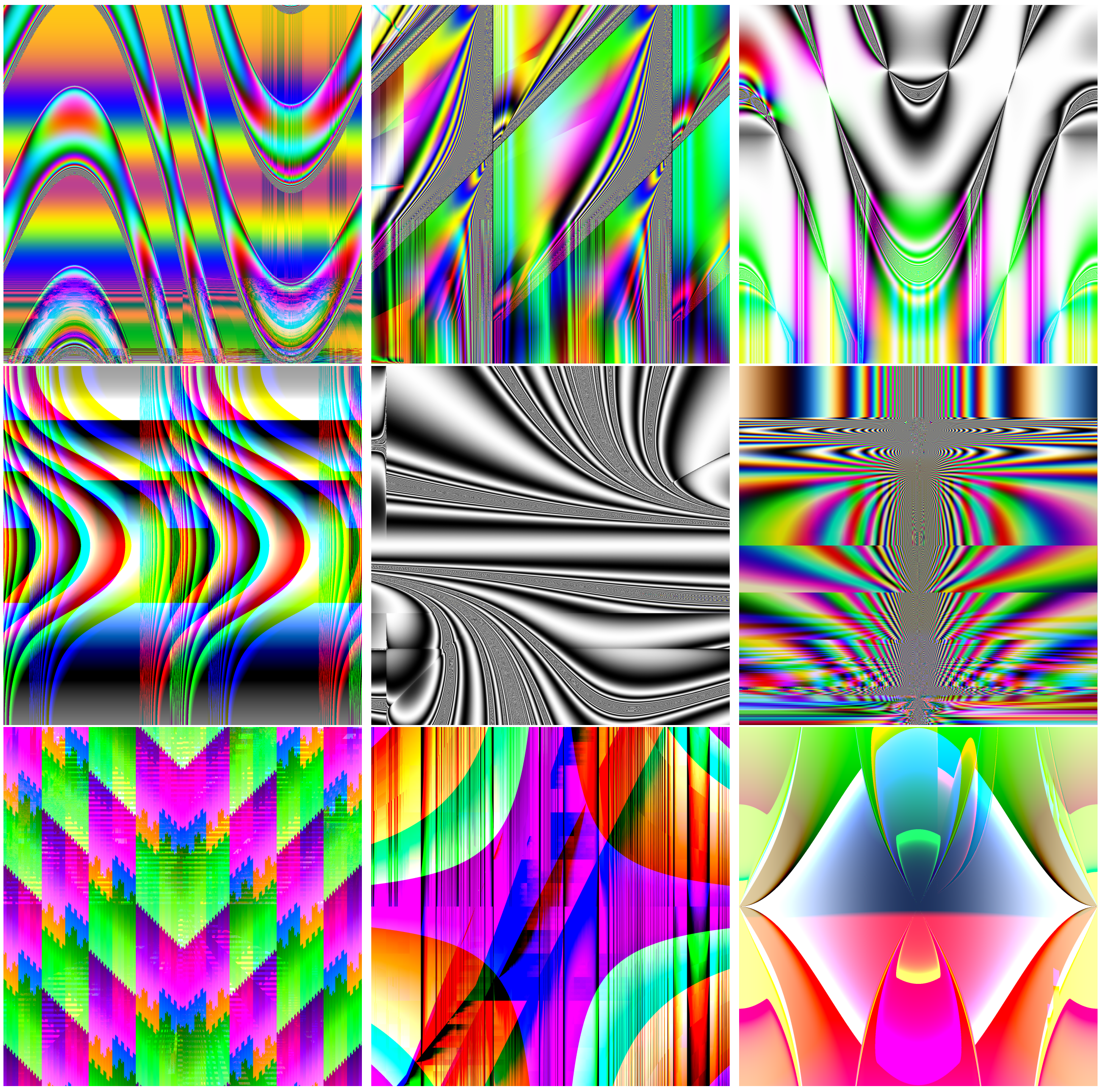


Figure 6.13: Gallery of 3 best images according to NIMA score, generated with experimental setup 1 (left), 2 (middle) and 3 (right).

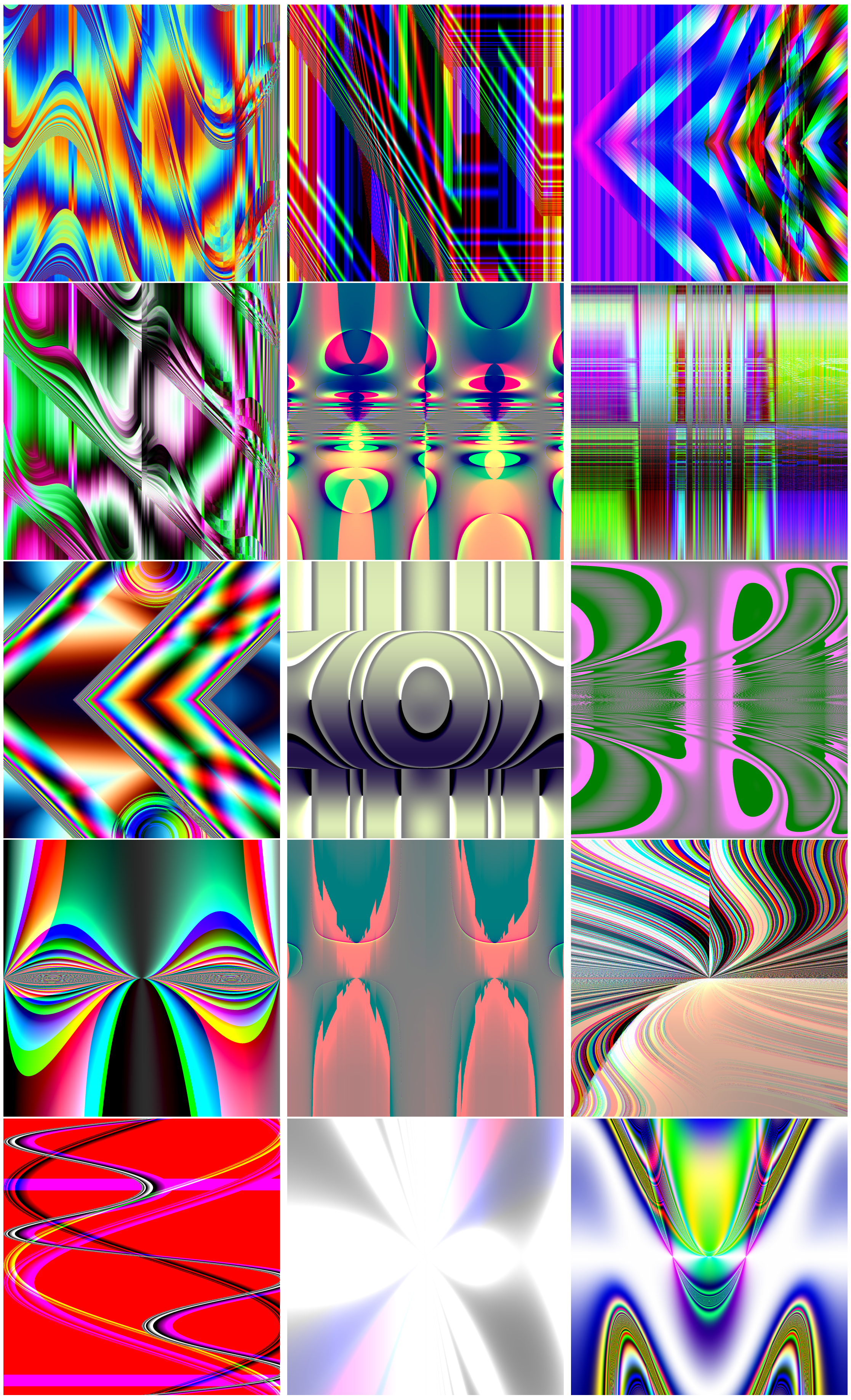


Figure 6.14: Gallery of 15 visually appealing images from a human perspective, generated with experimental setup 1 (left), 2 (middle) and 3 (right).

Chapter 7

Conclusion

The usefulness of GP goes undisputed due to its ability to approximate the optimal solution of a given problem without domain-specific knowledge. However, because all candidate solutions need to be evaluated against the entire fitness domain for each generation of the evolutionary process, GP is often regarded as very computationally expensive.

In this work, we propose an approach to ease this computational burden by taking advantage of the high potential for parallelism in GP. Namely, we investigate the benefits of applying data vectorization to the fitness evaluation phase using throughput-oriented architectures such as the GPU.

To accomplish this we employed the TensorFlow numerical computation library written in Python, to develop a general-purpose GP engine capable of catering to our vectorization needs, TensorGP. TensorFlow provides seamless integration across different computing devices thus simplifying the process of performance comparison amongst various architectures.

Regarding experimentation, the functional aspect of evolution was first validated by maximizing image file size when compressing to the JPEG format. Additionally, and still within the validation phase, TensorGP was used to approximate both the Pagie Polynomial function and a selection of target images in the context of classical symbolic regression. Expression based image evolution was then performed in a benchmarking phase where performance differences amongst different GPU and CPU approaches were analysed.

Our results show that, by using TensorFlow 2 to perform vectorization of large fitness case domains in a GPU, speed gains in excess of two orders of magnitude over a standard iterative approach can be reached under controlled environments that exclusively target raw tree evaluation rates. Furthermore, experiments comprising full fledge evolutionary tasks suggest that TensorFlow manages to save intermediate results from expression calculation, avoiding the re-execution of highly shared code, consequently accentuating speedups in relation to more traditional iterative GP approaches.

Finally, we applied our engine to an evolutionary art problem using an image quality assessment classifier to perform automatic fitness evaluation. Time comparisons in this area further demonstrate how advantageous parallel GP solutions can be even outside the scope of typical application scenarios. Moreover, our TensorFlow approach is shown to make feasible the exploitation of certain GP parameters such as depth and population size, which are desirable in evolutionary art research.

These test results provide an experimental demonstration of how larger domains benefit from the higher computational throughput of massively parallel hardware such as the GPU [46]. In fact, as the complexity of problems attempted by GP increases, a natural demand for higher computational resources follows. Namely, with GPU computing on the rise and the potential to perform trillions of floating point operations per second, a shift towards the applications of these devices to GP is likely to occur. However, our test results for smaller domains seem to still make the case for more latency-oriented programming models such as the CPU. Therefore, modern day GP seems to be best suited for heterogeneous computing frameworks like TensorFlow that are device independent.

Upon completion of this work, several possibilities are to be considered for future endeavours. Some engine improvements and application scenarios that were not furthered or contemplated are subsequently listed.

Firstly, TensorGP performance would benefit from the elimination of overhead incurred by operator composition and protection mechanisms through the integration of device specific code (dedicated CPU functions and CUDA kernels) with TensorFlow. However, this would require the integration of our custom operators with the TensorFlow libraries, possibly in a future release of the framework.

Another appealing idea would be to integrate a pre-processing phase into the engine with the purpose of simplifying expressions before tensor evaluation. This would help eliminate redundant computations, effectively making the fitness evaluation phase even faster.

On the other hand, the study of image-based symbolic regression could be extended by comparing and analysing the impact of different fitness metrics and classifiers in the evolution process. Specifically, based on the data collected from Chapter 6, evolution towards photorealistic images is tempting. Even though this task seems to require deeper trees to cope with the inherently higher complexity, the accelerated evaluation process provided by GPUs coupled with the aforementioned improvements could prove fruitful in this scenario.

Evolutionary art also opens up a plethora of applications for TensorGP. For instance, because the engine is prepared to deal with an arbitrary number of dimensions, evolution of video is therefore possible, despite not attempted. This can be achieved by simply introducing another dimension (corresponding to time) in the tensor input data.

Lastly, the time comparison study amongst different approaches could be extended by including different GP frameworks (*e.g.* GPLearn, TinyGP, ECJ, . . .). However, this work path was discarded early on mainly because neither of these tools are implemented in Python, which would defeat the purpose of providing a direct comparison between different approaches and devices.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] David Andre and John R Koza. Parallel genetic programming: A scalable implementation using the transputer network architecture. In *Advances in genetic programming*, pages 317–337. MIT Press, 1996.

[3] 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.

[4] Thomas Bäck, DB Fogel, and Z Michalewicz. Introduction to evolutionary algorithms. *Evolutionary computation*, 1:59–63, 2000.

[5] Francisco Baeta, João Correia, and Tiago Martins. Tensorgp. `https://github.com/AwardOfSky/TensorGP`, 2020.

[6] Walter Bohm. Exact uniform initialization for genetic programming. *Foundations of Genetic Algorithms*, pages 379–407, 1996.

[7] Alberto Cano and Sebastian Ventura. Gpu-parallel subtree interpreter for genetic programming. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 887–894. ACM, 2014.

[8] 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.

[9] Ran Cheng, Cheng He, Yaochu Jin, and Xin Yao. Model-based evolutionary algorithms: a short survey. *Complex & Intelligent Systems*, 4(4):283–292, 2018.

[10] Sarit Chicotay, Omid E David, and Nathan S Netanyahu. Image registration of very large images via genetic programming. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 323–328, 2014.

[11] Darren M Chitty. A data parallel approach to genetic programming using programmable graphics hardware. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1566–1573. ACM, 2007.

[12] Darren M Chitty. Fast parallel genetic programming: multi-core cpu versus many-core gpu. *Soft Computing*, 16(10):1795–1814, 2012.

[13] Steve DiPaola and Liane Gabora. Incorporating characteristics of human creativity into an evolutionary art algorithm. *Genetic Programming and Evolvable Machines*, 10(2):97–110, 2009.

[14] Yongsheng Fang and Jun Li. A review of tournament selection in genetic programming. In *International Symposium on Intelligence Computation and Applications*, pages 181–192. Springer, 2010.

[15] David B Fogel. Introduction to evolutionary computation. *Evolutionary computation*, 1, 2000.

[16] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner Gardner, Marc Parizeau, and Christian Gagné. Deap: Evolutionary algorithms made easy. *The Journal of Machine Learning Research*, 13(1):2171–2175, 2012.

[17] Philip Galanter. What is generative art? complexity theory as a context for art theory. In *In GA2003 – 6th Generative Art Conference*, 2003.

[18] Mario Giacobini, Marco Tomassini, and Leonardo Vanneschi. Limiting the number of fitness cases in genetic programming using statistics. In *International Conference on Parallel Problem Solving from Nature*, pages 371–380. Springer, 2002.

[19] Simon Handley. On the use of a directed acyclic graph to represent a population of computer programs. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, pages 154–159. IEEE, 1994.

[20] Simon Harding and Wolfgang Banzhaf. Fast genetic programming and artificial developmental systems on gpus. In *21st International Symposium on High Performance Computing Systems and Applications (HPCS'07)*, pages 2–2. IEEE, 2007.

[21] Simon Harding and Wolfgang Banzhaf. Fast genetic programming on gpus. In *European Conference on Genetic Programming*, pages 90–101. Springer, 2007.

[22] Maarten Keijzer. Efficiently representing populations in genetic programming. In *Advances in genetic programming*, pages 259–278. MIT Press, 1996.

[23] Maarten Keijzer. Alternatives in subtree caching for genetic programming. In *European Conference on Genetic Programming*, pages 328–337. Springer, 2004.

[24] John R Koza, FH Bennett, Jeffrey L Hutchings, Stephen L Bade, Martin A Keane, and David Andre. Evolving sorting networks using genetic programming and the rapidly reconfigurable xilinx 6216 field-programmable gate array. In *Conference Record of the Thirty-First Asilomar Conference on Signals, Systems and Computers (Cat. No. 97CB36136)*, volume 1, pages 404–410. IEEE, 1997.

[25] John R Koza and John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[26] William B Langdon and Wolfgang Banzhaf. A simd interpreter for genetic programming on gpu graphics cards. In *European Conference on Genetic Programming*, pages 73–85. Springer, 2008.

[27] William Benjamin Langdon. *Genetic programming and data structures*. PhD thesis, University of London 1996., 1996.

[28] Christopher Lennan, Hao Nguyen, and Dat Tran. Image quality assessment. `https://github.com/idealo/image-quality-assessment`, 2018.

[29] Ying Liu, Hongyuan Cui, and Renliang Zhao. Fast acquisition of spread spectrum signals using multiple gpus. *IEEE Transactions on Aerospace and Electronic Systems*, 55(6):3117–3125, 2019.

[30] Nuno Lourenço, Filipe Assunção, Francisco B Pereira, Ernesto Costa, and Penousal Machado. Structured grammatical evolution: a dynamic approach. In *Handbook of Grammatical Evolution*, pages 137–161. Springer, 2018.

[31] Henri Luchian, Andrei Băutu, and Elena Băutu. Genetic programming techniques with applications in the oil and gas industry. In *Artificial Intelligent Approaches in Petroleum Geosciences*, pages 101–126. Springer, 2015.

[32] Fernando Jorge Penousal Martins Machado. *Inteligencia artificial e arte.* PhD thesis, University of Coimbra, 2007.

[33] Penousal Machado and Amilcar Cardoso. Speeding up genetic programming. In *Procs. 2nd Int. Symp. AI and Adaptive Systems, CIMAF*, volume 99, pages 217–222, 1999.

[34] Julian F Miller. An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation-Volume 2*, pages 1135–1142. Morgan Kaufmann Publishers Inc., 1999.

[35] Julian F Miller and Stephen L Smith. Redundancy and computational efficiency in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 10(2):167–174, 2006.

[36] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965.

[37] Naila Murray, Luca Marchesotti, and Florent Perronnin. Ava: A large-scale database for aesthetic visual analysis. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2408–2415. IEEE, 2012.

[38] Riccardo Poli et al. Evolution of graph-like programs with parallel distributed genetic programming. In *ICGA*, pages 346–353. Citeseer, 1997.

[39] Riccardo Poli, William B Langdon, Nicholas F McPhee, and John R Koza. *A field guide to genetic programming.* Lulu. com, 2008.

[40] 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*, pages 85–94. ACM, 2009.

[41] Juan J Romero. *The art of artificial evolution: A handbook on evolutionary art and music.* Springer Science & Business Media, 2008.

[42] Todd Rowland and Eric W Weisstein. Tensor. From MathWorld—A Wolfram Web Resource.

[43] Umme Sara, Morium Akter, and Mohammad Shorif Uddin. Image quality assessment through fsim, ssim, mse and psnr—a comparative study. *Journal of Computer and Communications*, 7(3):8–18, 2019.

[44] Hans-Paul Schwefel. Advantages (and disadvantages) of evolutionary computation over other approaches. *Evolutionary computation*, 1:20–22, 2000.

[45] Pramod Singh and Avinash Manure. Introduction to tensorflow 2.0. In *Learn TensorFlow 2.0*, pages 1–24. Springer, 2020.

[46] Kai Staats, Edward Pantridge, Marco Cavaglia, Iurii Milovanov, and Arun Aniyan. Tensorflow enabled genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1872–1879. ACM, 2017.

[47] Hossein Talebi and Peyman Milanfar. Nima: Neural image assessment. *IEEE Transactions on Image Processing*, 27(8):3998–4011, 2018.

[48] Adriano Vinhas, Filipe Assunção, João Correia, Aniko Ekárt, and Penousal Machado. Fitness and novelty in evolutionary art. In *International Conference on Computational Intelligence in Music, Sound, Art and Design*, pages 225–240. Springer, 2016.

[49] Matthew Walker. Introduction to genetic programming. *Tech. Np: University of Montana*, 2001.

[50] David H Wolpert, William G Macready, et al. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

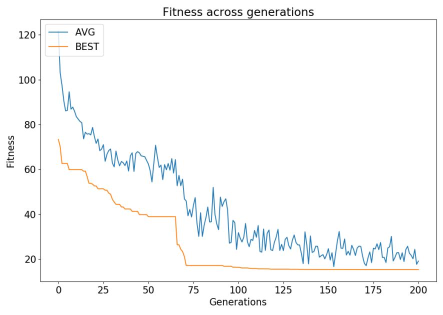
[51] Phillip Wong and Mengjie Zhang. Scheme: Caching subtrees in genetic programming. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pages 2678–2685. IEEE, 2008.

[52] Qizhi Yu, Chongcheng Chen, and Zhigeng Pan. Parallel genetic algorithms on programmable graphics hardware. In *International Conference on Natural Computation*, pages 1051–1059. Springer, 2005.

Appendices

Appendix A

This appendix contains the graphics corresponding to the depth and fitness evolution mentioned in the experimental results for the symbolic regression of images (see Section 6.1.3).



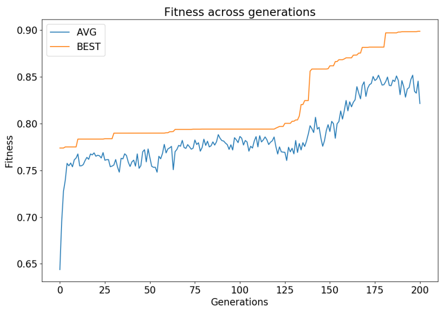
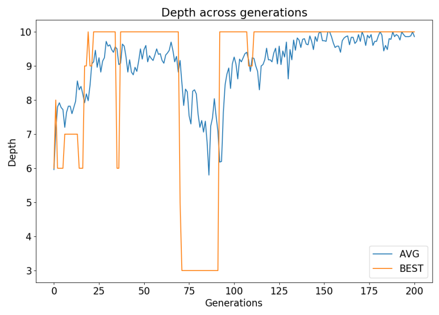


Figure 1: Fitness values across generations for RMSE (left) and SSIM (right) - Black and white image.



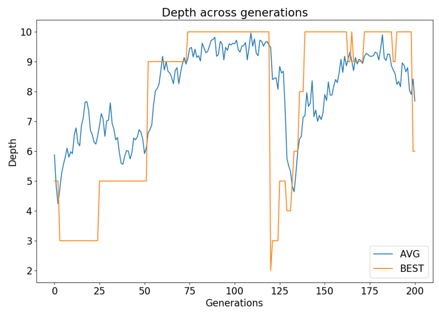
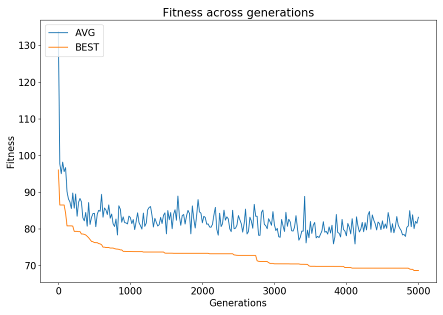


Figure 2: Depth values across generations for RMSE (left) and SSIM (right) - Black and white image.



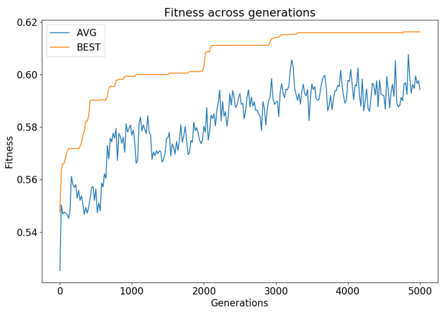


Figure 3: Fitness values across generations for RMSE (left) and SSIM (right) - First colored image.

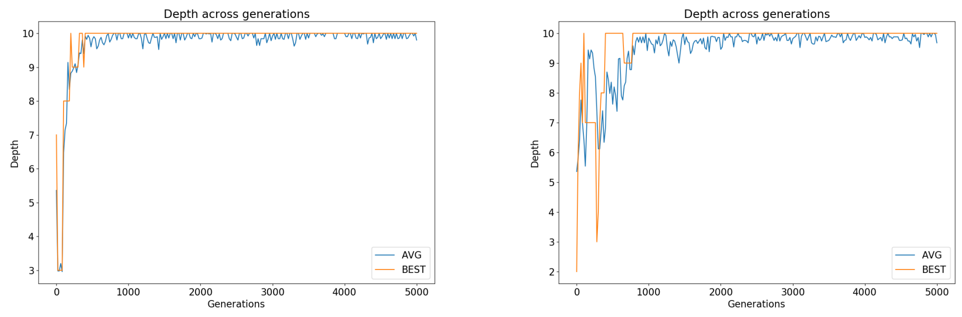


Figure 4: Depth values across generations for RMSE (left) and SSIM (right) - First colored image.

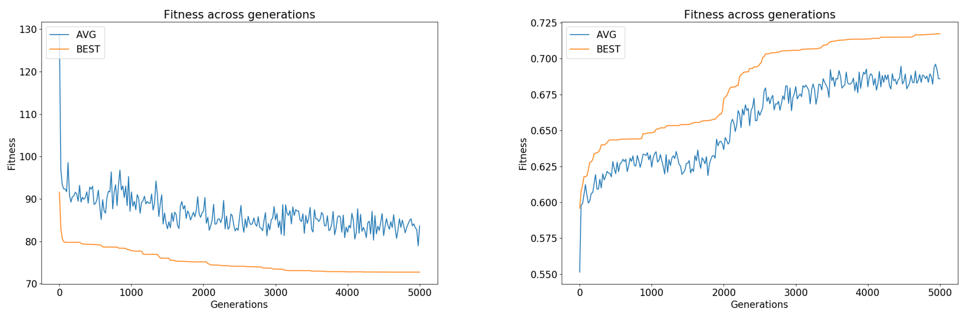


Figure 5: Fitness values across generations for RMSE (left) and SSIM (right) - Second colored image.

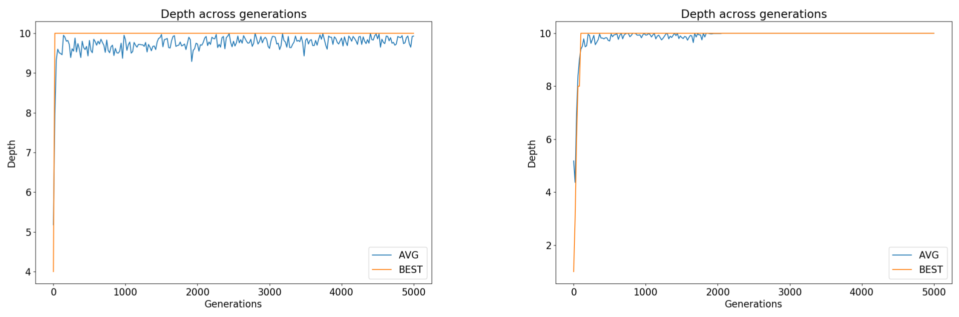


Figure 6: Depth values across generations for RMSE (left) and SSIM (right) - Second colored image.

This page is intentionally left blank.

Appendix B

The graphics in this appendix correspond to the fitness evolution of the approaches compared in Section 6.2.2.

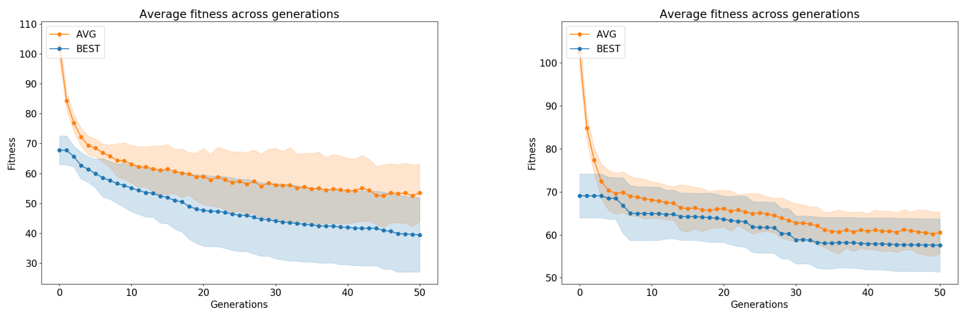


Figure 7: Average fitness values (generation average and best) for the EVAL (right) and DEAP (left) approaches.

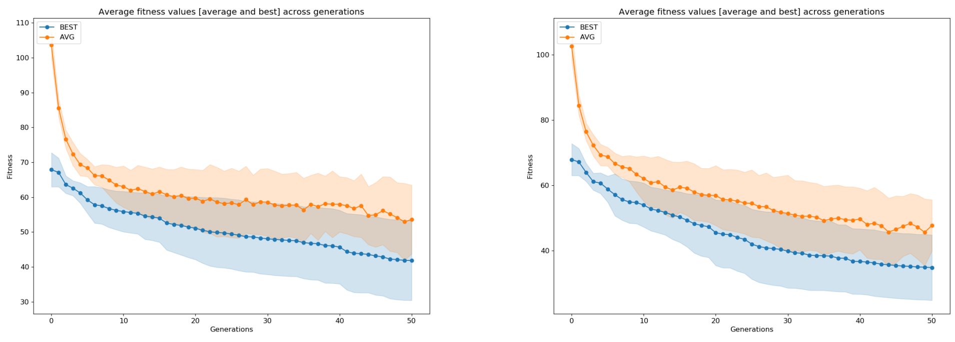


Figure 8: Average fitness values (generation average and best) for the eager CPU (right) and GPU (left) approaches.