SN Computer Science (2022) 3:154
https://doi.org/10.1007/s42979-021-01006-8

ORIGINAL RESEARCH

Exploring Genetic Programming in TensorFlow with TensorGP

Francisco Baeta[1] · João Correia[1] · Tiago Martins[1] · Penousal Machado[1]

Received: 17 July 2021 / Accepted: 5 December 2021 / Published online: 4 February 2022
© The Author(s), under exclusive licence to Springer Nature Singapore Pte Ltd 2022

Abstract
In this paper, we resort to the TensorFlow framework to investigate the benefits of applying data vectorization and fitness caching methods to domain evaluation in Genetic Programming. For this purpose, an independent engine was developed, TensorGP, along with a testing suite to extract comparative timing results across different architectures and amongst both iterative and vectorized approaches. Our performance benchmarks further analyze the benefits of employing vectorization techniques and throughput-oriented hardware in several GP scenarios consisting of varying tree sizes and domain resolutions. In specific, it is shown that by applying the TensorFlow eager execution model to the evolutionary process, speedup gains of up to two orders of magnitude can be achieved on a parallel approach running on a GPU when compared to a standard iterative approach for a typical symbolic regression problem. Lastly, we also demonstrate the performance benefits of explicit operator definition when compared to operator composition in TensorGP.

Keywords Genetic programming · Parallel computing · TensorFlow · GPU computing

Introduction

Genetic Programming (GP), which targets the evolution of computer programs, requires large amounts of computational resources since all individuals in the population need to be executed and tested against the objective. As a result, fitness evaluation is generally regarded as the most computationally costly operation in GP for most practical applications [15]. Despite this, GP is beyond doubt a powerful evolutionary technique, capable of tackling every problem solvable by a computer program without the need for domain-specific knowledge [26]. Furthermore, although computationally intensive by nature, GP is also "embarrassingly parallel" [3].

Previous works on accelerating fitness evaluation in GP mainly focus on two techniques: the caching of intermediate fitness results and the vectorization of the evaluation domain. The first method aims to save the results of code execution from parts of a program to avoid re-executing this code when evaluating other individuals. On the other hand, the second method evaluates the full array of fitness cases simultaneously by performing a tensor operation for each function within an individual.

The last decade saw the exponential growth of computing power proposed by Gordon Moore back in 1965 [24] start to break down. As we start meeting the limits of physics, a paradigm shift towards multi-core computing and parallelization becomes inevitable. Namely, with the rise of parallel computing, devices such as the Graphics Processing Units (GPUs) have become ever more readily available [4]. 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 the data vectorization approach with such architectures.

In this work, we resort to the TensorFlow platform to investigate the benefits of applying the above-mentioned approaches to the fitness evaluation phase in GP and comparing performance results across different types of processors.

This article is part of the topical collection "Applications of bioinspired computing (to real world problems)" guest edited by Aniko Ekart, Pedro Castillo and Juanlu Jiménez-Laredo.

✉ Francisco Baeta
fjrbaeta@dei.uc.pt

João Correia
jncor@dei.uc.pt

Tiago Martins
tiagofm@dei.uc.pt

Penousal Machado
machado@dei.uc.pt

[1] Department of Informatics Engineering, Centre for Informatics and Systems of the University of Coimbra, University of Coimbra, Coimbra, Portugal

With this purpose, a novel and independent GP engine was developed: TensorGP. Other engines such as KarooGP [28] already take advantage of TensorFlow's capabilities to speed program execution. However, our engine exploits new TensorFlow execution models, which are shown to benefit the evolutionary process. Moreover, we intend on extending the application of TensorGP outside the realm of classical symbolic regression and classification problems by providing support for different types of functions, including image-specific operators.

The remainder of this paper is organized as follows. "Related Work" provides a compilation of related work. " TensorGP" presents the framework. "Benchmark Experimentation" lays the experimental setup and analyses benchmarking results. Finally, "Conclusions and Future Work" draws final conclusions and points towards future work.

Related Work

Because GP individuals usually share highly fit code with the rest of the population and not only within themselves [17], techniques to efficiently save and reuse the evaluation of such code have been of particular interest to research around GP. In specific, Handley [16] first implemented a method of fitness caching by saving the computed value by each subtree for each fitness case. 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.

However, because system memory is finite, the caching of intermediate results must obey certain memory constraints. In this regard, Keijzer [18] proposed two cache update and flush methods to deal with fixed size subtree caching: a first method using a postfix traversal of the tree to scan for nodes to be added to or deleted from the cache and a second method that implemented a variant of the DAG approach. Even if we rule out the amount of memory used, hit rates and search times are still a grave concern. Wong and Zhang [30] developed a caching mechanism based on hash tables to estimate algebraic equivalence between subtrees, which proved efficient in reducing the time taken to search for standard code by reducing the number of node evaluations. Besides, caching methods are instrumental in scenarios with larger evaluation domains, and code re-execution is more time-consuming. As an example, Machado and Cardoso [20] applied caching to the evolution of large-sized images (up to 512 by 512 pixels) in the NEvAr evolutionary art tool.

Another common way to accelerate GP is to take advantage of its potential for parallelization. Various works have explored the application of parallel hardware such as Central Processing Units (CPUs) with Single Instruction Multiple Data (SIMD) capabilities [7, 12, 23], GPU-based architectures [5, 8, 9, 11] and even Field Programmable Gate Arrays (FPGAs) [19] to fitness evaluation within the scope of GP. However, arguably the most promising speedups still come from GPUs as they are the most widely available throughput-oriented architectures. Namely, Cano et al. [9] verified speedups of up to 820 fold for specific classification problems versus a standard iterative approach by massively parallelizing the evaluation of individuals using the NVIDIA Compute Unified Device Architecture (CUDA) programming model.

One common way to abstract this parallelization process is to vectorize the set of operations performed over the fitness domain, effectively reducing the running time of a program to the number of nodes it contains [18]. Some interpreted languages such as Matlab, Python and Perl already support vectorized operations to reduce computational efforts. In particular, TensorFlow [1] is a numerical computation library written in Python that provides an abstraction layer to the integration of this vectorization process across different hardware. Staats et al. [28] demonstrated the benefits of using TensorFlow to vectorized GP fitness data in both CPU and GPU architectures, achieving performance increases of up to 875 fold for specific classification problems. The engine that the authors developed, KarooGP, is still used to tackle many symbolic regression and classification problems [10, 14, 21]. However, KarooGP does not take advantage of recent additions to TensorFlow execution models.

TensorGP

TensorGP takes the classical approach of most other GP applications and expands on it by using TensorFlow to vectorize operations, consequently speeding up the domain evaluation process through the use of parallel hardware. Moreover, TensorFlow allows for the caching of intermediate fitness results, which accelerates the evolutionary process by avoiding the re-execution of highly fit code. TensorGP is implemented in Python 3.7 using the TensorFlow 2.1 framework and is publicly available on GitHub.[1]

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

[1] TensorGP repository available at https://github.com/AwardOfSky/TensorGP.

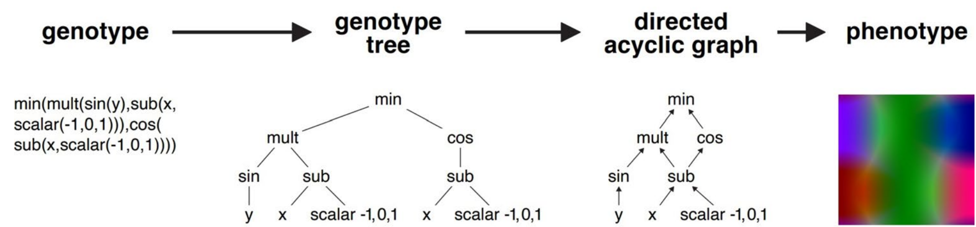


Fig. 1 Genotype to phenotype translation phases in TensorGP

Genotype to Phenotype

As the name implies, TensorGP works with tensors. In essence, a tensor is a generalization of scalars (that have no indices), vectors (that have precisely one index), and matrices (that have precisely two indices) to an arbitrary number of indices [27]. We start by describing the process of executing an individual in TensorGP. Figure 1 demonstrates our engine's translation pipeline from genotype to phenotype.

In its simplest form, each individual in GP can be represented as a mathematical expression. TensorGP follows a tree-based approach, internally representing individuals as a tree graph. It implies a first translation phase from string to tree representation, which is only performed at the beginning of the evolutionary process if the initial population is not randomly generated.

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 calculating any values. It allows the approach to cache potential intermediate results from subtrees, effectively generalizing our tree graph structure to a DAG. On the other hand, the eager execution model allows for the immediate execution of vectorized operations, eliminating the overhead of explicitly generating the intermediate DAG of operations.

Even though graph-oriented execution enables many memory and speed optimizations, there are heavy performance costs associated with graph building. TensorFlow eager execution mode aims to eliminate such overheads without sacrificing the benefits furnished by graphs [2]. Because the individuals we are evolving are constantly changing from generation to generation, we would think that eager mode would be a good fit for tensor execution. For this reason, in "Benchmark Experimentation", we include some performance comparisons between both these TensorFlow execution modes.

Finally, the last translation phase goes through the entire genotype data to produce a phenotype, the target of fitness assessment. Because the domain of fitness data points to be evaluated is fixed for all operations, the vectorization of this data is made trivial using a tensor representation. Generally speaking, our phenotype is a tensor, which can be visually represented as an image for a problem with three dimensions, as seen in the last stage of Fig. 1. In this example, the first two dimensions correspond to the width and height of the image, while the third dimension encodes information regarding the RGB color channels. The resulting tensor phenotype is obtained by chaining operators, variables and constants that make part of the individual. These variables and constants are also tensors, which occupy a range of $[-1, 1]$ for the example given. With the aid of TensorFlow primitives, we can apply an operation to all domain points at the same time while seamlessly distributing computational efforts amongst available hardware.

Primitive Set

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. Some image specific operators are also included to facilitate the application of TensorGP to image evolution domains (such as evolutionary art). One of such operators, and perhaps the most intriguing, is the *warp*. We will detail this operator in specific in "Explicit definition versus composition", where we compare several implementation approaches for this operator.

Operators must also be defined for all possible domain values, which means implementing protection mechanisms for certain cases. Table 1 enumerates the different types of operators and respective special cases. All operators are applied to tensors and integrated into TensorGP through the composition of existing TensorFlow functions. While the main math and logic operators can be implemented with a simple call to the corresponding TensorFlow function, more complex operators may imply chaining multiple functions. For instance, while TensorFlow possesses many 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.

Besides the specified protective mechanism, there are essential implementation details for some operators. For 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 $[-\pi, \pi]$

Table 1 Description of implemented TensorGP 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
		Minimum (min)	2	$min(x, y)$
		Maximum (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 mod y Remainder of division
		Fractional part[b] (frac)	1	$x - \lfloor * \rfloor x$
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 support operators

[c]Transformation to integer is needed

[d]More details in " Explicit definition versus composition"

range but are otherwise normalized to either [0, 1] or [− 1, 1]. As a matter of fact, [− 1, 1] this is the standard domain range used in TensorGP. 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.

Features

TensorGP was implemented with ease of use in mind. To demonstrate some of its functionality, the following paragraphs describe the main features of the presented engine.

When a GP run is initiated on TensorGP, a folder is created in the local file system with the aim of logging evolution data. In each generation, the engine keeps track of depth and fitness values for all individuals. When the run is over, a visualization for individuals' depth and fitness values across generations is automatically generated along with a CSV file with experimental data.

Besides, TensorGP keeps an updated state with all the important parameters and evolution data. With each new generation, the engine updates this file with information regarding evolution status. When it is time to resume the experiment, the engine loads the corresponding configurations from the file of that experiment, gathering the latest generational 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 specify an initial custom population by passing a text file containing string-based programs to the engine.

Currently, two stop criteria are implemented: the generation limit (which the engine defaults to) and acceptable error. In the acceptable error method, the experiment ends 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.

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

Benchmark Experimentation

This section describes the experimentation performed to investigate how TensorGP fares against other standard GP approaches in several scenarios. To achieve this goal, we perform the first batch of tests to benchmark TensorGP in a typical GP scenario where we define a domain of a constant range and fixed resolution (i.e. granularity) to test different depths within that domain. With the insights from the previous tests, we then perform two separate experiments meant to unveil the benefits of using different hardware to evaluate a fixed population of individuals and evolve that same initial population. Finally, we finish this section with the benchmark of several approaches to implementing a specific operator used by TensorGP as a proof of concept for possible future optimizations.

Experimental Setup

All experiments considered concern the symbolic regression problem of approximating the polynomial function defined by:

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

This function is also known as Pagie Polynomial and is commonly used in GP benchmarks due to its reputation for being challenging [25]. Because the domain of this problem is two-dimensional, we represent it using a rank two tensor. To ease the comparison across different domain resolutions in our experimentation, we let these tensors be square, with each side corresponding to another test case. The smallest test case considered evaluates over a 64 by 64 grid of fitness cases, thus involving 4096 evaluations. In each subsequent test case, the length of the grid doubles, effectively quadrupling the number of points to evaluate. This grid of values keeps increasing until the most extensive test case, where the tensor side is 4096 (over 16 million fitness cases). Overall, the range of the evaluation domain ranges from 64 by 64 to 4096 by 4096 point tensors, with every experiment considering a subset of these test cases, according to the objective of the comparison. Furthermore, all experiments minimize the Root Mean Squared Error (RMSE) metric as a fitness evaluation function.

Throughout the experimentation, a total of six approaches were considered. Four of these approaches concern TensorGP implementations, 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. DEAP is a commonly used EC framework written in Python and offers a powerful and straightforward interface for experimentation [13]. 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

Table 2 Default hardware and software specifications

Component	Specification
CPU	Intel Core i7–5930K (@3.7 GHz)
GPU	NVIDIA GTX TITAN X (12 GB)
RAM	2×16 GB @2.133 Mhz
Operative system	Ubuntu 16.04.5 LTS
Execution environment	Command line

prototyping of controlled environments within a few lines of code. 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 to run it. To eliminate the overhead of parsing code, *eval* is called only once by plugging in a lambda defined expression of the individual. For future reference, this approach will be referred to as EVAL.

Unless expressed otherwise, the software and processing hardware used for the execution of these experiments is defined in Table 2.

Because some of the experimental results presented hereafter encompass a wide range of values covering multiple orders of magnitude, our best bet for graphical representation in those scenarios is to use a logarithmic scale as it would be otherwise impossible to distinguish between timings.

Depth Experimentation

An important difference between academic exercises and real-world problems is that the latter are generally higher in complexity. When it comes to GP, this inherent complexity often translates to bigger and deeper individuals. It is therefore desirable to be able to evaluate larger individuals in a feasible amount of time.

This experiment aims to benchmark several GP approaches to determine which are more suited for evaluating increasingly larger individuals. Specifically, we compare all the considered approaches, each of which saw the evolution of 50 generations from fixed populations of increasing depth.

Having a tree graph representation as a reference, the size of an individual corresponds to the number of nodes of the individual. In contrast, the depth corresponds to the maximum distance in edges from the tree's root to any possible leaf node. As a disclaimer, it is worth noting that the only restriction imposed on the size of trees was the actual depth. TensorGP implements the Ramped Half-and-Half (RHH) method regarding the population, instead of generating half of one tree with the Grow method and the other half with Full. This way, TensorGP effectively divides the population into blocks of different depths, splitting the number of trees in each block to use either the Full or Grow methods. All individuals in the initial populations are generated with this type of RHH, and the depth values for a given population range from 4 to 26 with intervals of 2. Because we are generating the populations with the RHH method, the size of the trees in the initial population of depth 4, for instance, are in order of 10 s. However, this size significantly increases for trees on the populations of the last depths tested, with some Full trees having between 10,000 and 100,000 nodes.

All initial populations were generated in TensorGP and executed as-is on other approaches. Besides, it is worth emphasizing that all approaches started the evolutionary process from the same set of initial populations. In addition to this, individuals in populations of higher depths are extensions of the corresponding individuals in the populations with lower depths, meaning that they have the shallower nodes in common. This was done to prevent a deeper tree from having fewer nodes, which can happen otherwise, no matter the tree generation method used. Also, added to the RHH method, the fact that we are using a function set with multiple arities means that the increase in tree size is not exponential across subsequent depths.

In this manner, we start our experimentation by fixing the domain resolution of the considered problem with the 128 by 128 test case. In their efforts to standardize benchmarks in GP, McDermott et al. [22] point out a list of several GP scenarios considering different problems (including the Pagie Polynomial) and respective evaluation domains to use, the most stressing of which contains 10,000 fitness cases. We chose a domain resolution of 128 by 128 simply because it is the first test case with over 10,000 data points amongst the set of test cases that we considered.

In Fig. 2 we see the results from an average of 30 runs with the described experimental setup. Figure 2a shows the average execution time in seconds across different depths for all approaches, while 2b shows the same execution results but concerning evaluation speed, in the format of Genetic Programming operations per second (GPOps/s). Table 3 and 4 show the values represent in the referred figures. As a side note, the values corresponding to the last tested depths for the iterative frameworks (DEAP and EVAL) were excluded as they did not need to meet the average time threshold of 30 min per experimental run. To avoid confusion, we should also first clarify that we are considering calculating any element of our function set as a single operation, even if said operation is expressed as the composition of functions or other primitives (as is the case for image-type operators). In essence, this means that the number of GP of an individual is equal to the number of non-terminal nodes of its corresponding tree graph.

A quick look at Fig. 2b demonstrates that the evaluation speed of all approaches stays relatively constant across

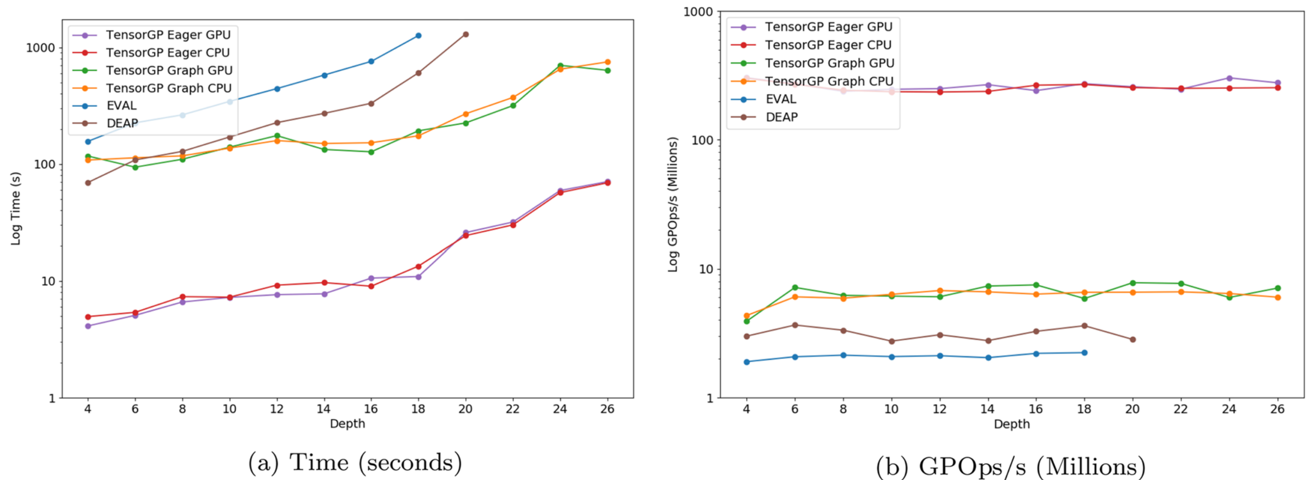


Fig. 2 Average execution timings and GPOp/s results in for all approaches for the evolution of initial populations generated with the RHH method for depths ranging from 4 to 26. Results not shown in the graph were removed for not meeting the defined time threshold

depths, meaning that these approaches manage to scale well across trees of different depths, and therefore different sizes. As observed from Fig. 2a, regarding the iterative approaches, DEAP proves to be about 2 times slower to execute the same setup. Nevertheless, we must remember that DEAP is a different framework that employs its own recombination and selection operators, and thus some degree of evolution bias towards trees of different sizes can be expected.

Additionally, as expected, the approaches that perform iterative domain evaluation are much slower than TensorGP running in either eager or graph execution modes, confirming that vectorization is indeed beneficial for the problem considered. Indeed, across all depths tested by both frameworks, TensorGP running on CPU in eager mode achieved an average of 253.4 Million GPOps/s, which turns out to be 123 times faster than DEAP, reaching on average 2.06 Million GPOps/s.

However, what is perhaps surprising is that TensorGP is not much faster on GPU than on CPU. The opposite is true for this experiment. We verify that for the eager execution mode, the GPU is 4.4% slower on average across all tested depths, while for graph execution, this percentage slightly raises to 4.9%. As we will verify in the subsequent experiments, this lack of performance on the GPU is due to the memory bottleneck of copying data from the CPU to the GPU and vice-versa. To further investigate the advantages of using the GPU, in the following subsection, we will study the impact of evaluating domains with a different number of fitness cases by changing the domain granularity instead of the depths of individuals.

Another exciting aspect of analyzing is the evolution of the computation speed through generations. In Fig. 3 it is shown the average and standard deviation values for the domain evaluation speed verified across generations for the experimental run of depth 18 in both EVAL (Fig. 3) and on TensorGP running on GPU (Fig. 3b).

One of the most glaring aspects of these figures is the high values for the standard deviation. This should, however, not come as a surprise because, within the same depth test, each initial population is generated using a different random seed, which in turn promotes different evolutionary paths that result in a notable variation of GPOps/s performed throughout an entire run.

Moreover, as demonstrated in both scenarios, there is an increase in evaluation speed with an increase in the number of generations evaluated. This behaviour is to be expected for the vectorized approaches because the caching of fitness results that TensorGP performs, avoids the re-execution of some tensor operations, theoretically reducing the calculation of such operations to a simple table lookup that can be completed in $O(1)$ time complexity, instead of the standard $O(n)$ time (n being the number of fitness cases of the evaluation domain). However, we see that EVAL also verifies an increase in computation speed when comparing the first and last generations of less than two times (Fig. 3a). In turn, the speed increase between the first and last generations for the TensorGP GPU approach is almost 3 times (Fig. 3b).

Because EVAL corresponds to our iterative baseline that is also executed on TensorGP, but without the aid of TensorFlow, we can conclude that the reason for the speed increase is not due to different engine implementations. In fact, upon analyzing the resulting nodes from individuals of the last generations, we verify a tendency towards more simplistic mathematical operations when compared to individuals in the first generations. This is most likely because the problem we are trying to approximate is itself a polynomial, composed of simple mathematical operations, which naturally guides the evolutionary process to prefer

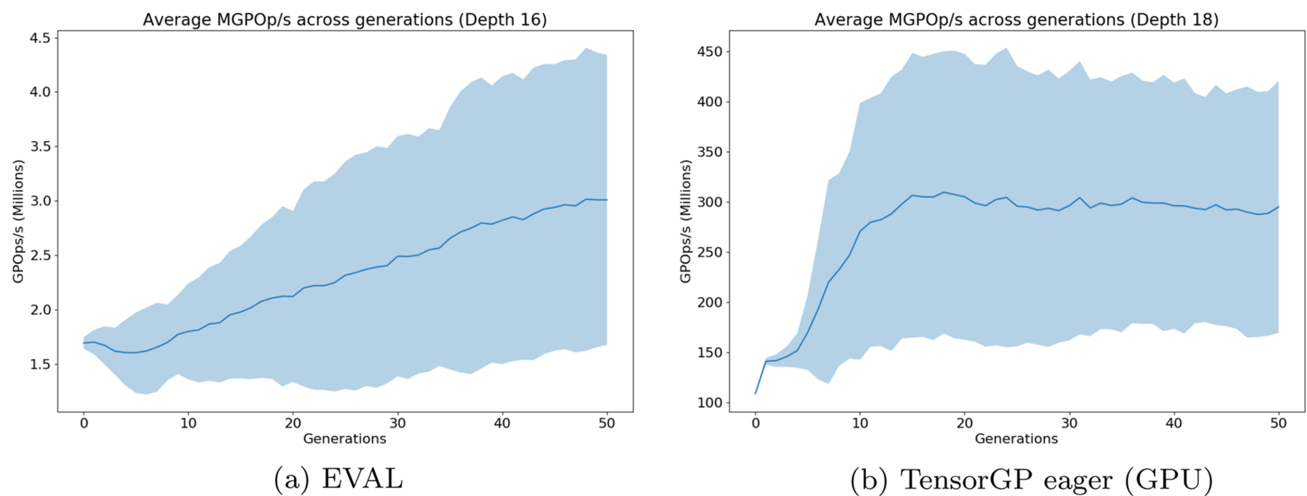


Fig. 3 Evolution of average (bold line) and standard deviation (filled area) GPOp/s results across generations for the TensorGP eager (GPU) and EVAL approaches, left and right respectively

Table 3 Average timing values (in seconds) for different approaches across domain resolutions

Depths	TF GPU EAGER	TF CPU EAGER	TF GPU GRAPH	TF CPU GRAPH	EVAL	DEAP
4	4.123*	4.957	117.297	108.420	156.688	69.554
6	5.083*	5.393	94.091	113.262	225.898	108.538
8	6.614*	7.351	110.086	117.744	264.352	128.510
10	7.242*	7.275	139.658	137.172	344.930	170.810
12	7.647*	9.222	175.486	159.076	443.584	226.658
14	7.775*	9.695	133.507	150.175	579.887	272.102
16	10.597	9.035*	127.474	152.320	759.211	332.497
18	10.932*	13.437	193.084	174.939	1265.037	606.335
20	25.897	24.390*	225.438	270.015	DNF	1305.290
22	31.844	30.113*	317.491	372.832	DNF	DNF
24	59.375	56.992*	702.615	651.214	DNF	DNF
26	70.728	69.122*	636.802	751.350	DNF	DNF

DNF stands for “Did Not Finish”. Best values marked with “*”

these kinds of operators. Individuals in the initial populations are generated by randomly drawing primitives from our primitive set, which includes more complex operators that are implemented as a composition of existing functions and python primitives. This way, on average, a GP operation takes longer to execute in the first generations due to operators' higher complexity, leading to decreased computation speeds. In addition to the fitness caching performed, TensorGP on GPU also benefits from the same perks of the evolutionary process, making the GPOps/s increase more pronounced as we verify by the curve in Fig. 3b.

Results from the experimentation made so far seem to make a case for using vectorized approaches in GP. In addition to this, all approaches tested seem to maintain a relatively constant evaluation seed across individuals of different sizes.

CPU Versus GPU Comparison

Even though the experimentation made on the previous subsection confirmed the benefits of vectorized approaches, it proved unsuccessful in identifying a meaningful performance different between a traditional CPU and GPU architecture. We hypothesize that this is due to the relatively reduce domain resolution used. Hence, this section will see the execution of two experiments to test the evaluation of an increasing number of fitness cases, first in an isolated environment, and then within an evolutionary context.

Table 4 Average GP operations per second (in millions) for different approaches across domain resolutions

Depths	TF GPU EAGER	TF CPU EAGER	TF GPU GRAPH	TF CPU GRAPH	EVAL	DEAP
4	302.943*	300.488	3.920	4.337	1.901	2.999
6	272.372*	269.817	7.167	6.059	2.075	3.658
8	237.534	242.412*	6.221	5.914	2.133	3.335
10	246.315*	236.062	6.135	6.336	2.081	2.745
12	249.359*	234.687	6.065	6.778	2.113	3.075
14	267.657*	237.207	7.350	6.625	2.041	2.769
16	240.695	265.189*	7.499	6.361	2.204	3.268
18	272.307*	268.539	5.876	6.570	2.232	3.613
20	257.728*	253.984	7.798	6.584	DNF	2.829
22	247.020	250.450*	7.704	6.625	DNF	DNF
24	302.417*	252.052	5.991	6.430	DNF	DNF
26	276.937*	253.435	7.083	6.024	DNF	DNF

DNF stands for "Did Not Finish". Best values marked with "*"

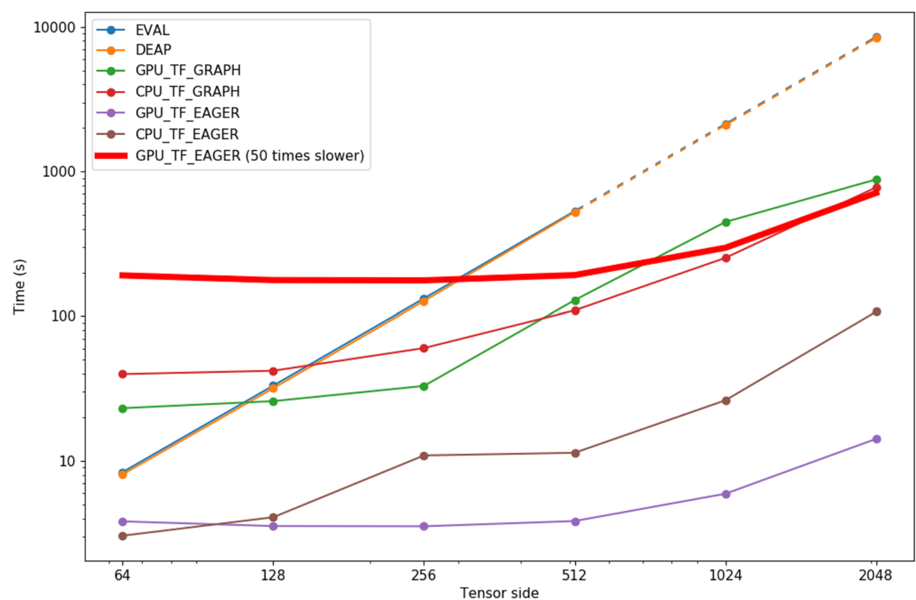


Fig. 4 Time (in seconds) comparison of different approaches for raw tree evaluation across domain resolutions

All experiments in this subsection consider the same array of 6 test cases, evaluating a two-dimensional domain that exponentially increases in the number of fitness cases. The subset of test cases considered in this experiment ranges from the smallest 64 by 64 test cases to a domain resolution of 2,048 by 2,048 (over 4 million fitness cases). As a disclaimer, all results are taken from an average of 5 runs. Larger domain resolutions were not tested mainly due to Video RAM (VRAM) limitations of the GPU used during the experiments. Moreover, the same set of populations were used for all test cases, where each population contains 50 individuals generated with the Ramped-Half-and-Half method and 12 for maximum allowed depth. While the first experiment only saw the execution of the population as mentioned above batch for all the considered approaches, in the evolutionary run, we let the individuals evolve for 50 generations, only leaving out the graph execution approaches of TensorGP.

The first experiment compares average execution times amongst all considered approaches. Figure 4 shows the average time taken for the evaluation of populations across 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.

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

Table 5 Average (bold) and standard deviation (non-bold) of timing values, in seconds, across domain resolutions for the tree evaluation experiment

	EVAL	DEAP	TF graph (GPU)	TF graph (CPU)	TF eager (GPU)	TF eager (CPU)
64^2	**8.35**	**8.05**	**23.14**	**39.81**	**3.82**	**3.04***
(4096)	±0.22	±0.20	±0.56	±0.84	±0.22	±0.08
128^2	**33.13**	**31.78**	**25.89**	**42.00**	**3.54***	**4.07**
(16,384)	±0.82	±0.89	±0.50	±1.07	±0.13	±0.11
256^2	**132.28**	**126.93**	**32.97**	**60.22**	**3.53***	**10.90**
(65,536)	±3.36	±3.50	±0.77	±1.66	±0.13	±0.11
512^2	**531.15**	**522.53**	**128.75**	**109.53**	**3.83***	**11.37**
(262,144)	±12.76	±13.97	±2.79	±2.92	±0.19	±0.92
$1,024^2$	DNF	DNF	**446.90**	**252.23**	**5.92***	**26.23**
(1,048,576)	–	–	±8.94	±21.27	±0.23	±1.06
$2,048^2$	DNF	DNF	**879.74**	**775.54**	**14.20***	**107.42**
(4,194,304)	–	–	±31.78	±58.38	±0.37	±4.74

DNF stands for “Did Not Finish”. Best values marked with “*”

Analyzing TensorFlow’s eager execution mode, we see that even though the CPU is faster for the smallest test case, this trend fades rapidly for larger problems domains. In fact, for 16,384 elements (128^2), the GPU is already marginally faster than the CPU. This margin widens with an increase in the tensor side, resulting in GPU evaluation over a 4 million point domain (2048^2) being almost 8 times faster as seen in Table 5. As a reminder, with TensorFlow, we are already providing operator vectorization; hence the eight-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. We observed that the slowest eager execution approach is about 10 times faster than the fastest graph execution one for the two smaller domain resolutions. This trend continues for larger domains, but the gap shortens to about 8 times (favouring eager execution).

Nevertheless, results gathered for graph execution show rather unexpected behaviour. As suggested by the previously analysed results, 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 65,536 (256^2), from which point the CPU takes over. The answer to this strange behaviour may lie in the graph implementation used. 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 3 GB compared to the available 16 GB for system memory) that did not even finish some test cases while including the entire population in a single graph.

For this reason, and to be consistent with all domain resolutions, we decided to test these graph execution approaches by opening a session graph for each instead of evaluating the entire population in a single graph. Still, we can safely conclude that both graph execution approaches are slower than their eager equivalents. The bold red line, in Fig. 4, is intended as a visualization aid that represents a 50 times performance threshold over our fastest approach (TensorFlow eager on GPU) for approaches above the line.

For the evolutionary experiment, however, only four of the considered approaches were used: the iterative ones (DEAP and EVAL) and the ones that concern TensorFlow execution in eager mode (both CPU and GPU). Graph execution was omitted as it was demonstrated to be systematically slower than their eager equivalents.

Figure 5 shows the total run time for all considered approaches. Here, the red line shows a threshold for approaches 100 times slower than TensorGP running in GPU. The most noticeable aspect of these results is that they appear to be less linear 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. Suppose the best-fitted individual happens to have a lower depth value. In that case, 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 computing time. This explains the relatively higher standard deviations presented in Table 6 and the non-linear behaviour across problem resolutions (e.g. the test case for size 65,536 (256^2) runs faster than the two smaller domains for TensorFlow running on GPU).

Still, regarding TensorFlow results, from the two first test cases, we can not identify a clear preference towards CPU or GPU.

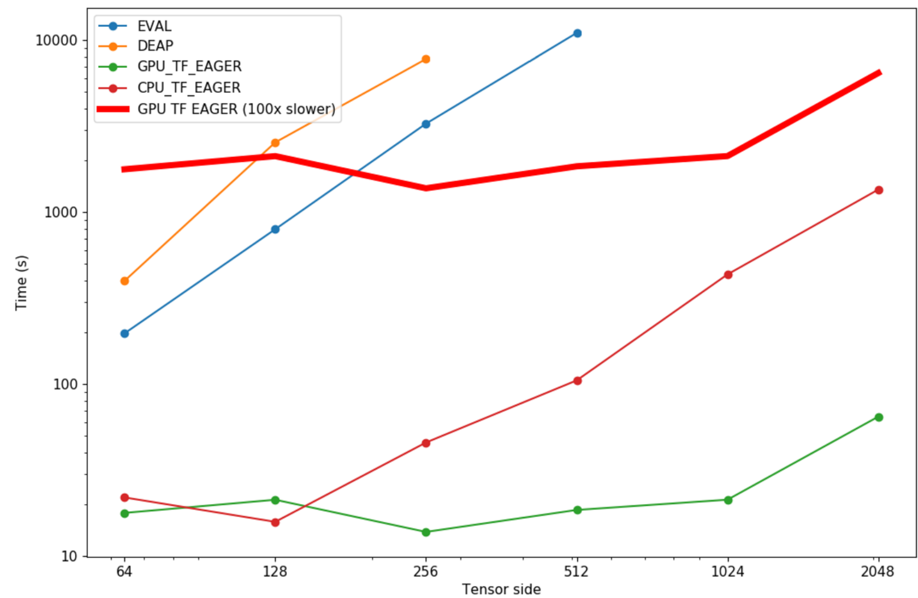


Fig. 5 Time (in seconds) comparison of different approaches for a full evolutionary run across domain resolutions

Table 6 Average (bold) and standard deviation (non-bold) timing values, in seconds, across domain resolutions for the evolutionary experiment

	EVAL	DEAP	TF eager (CPU)	TF eager (GPU)
64^2	**196.87**	**397.82**	**21.86**	**17.77***
(4,096)	±96.26	±339.50	±13.44	±8.29
128^2	**795.36**	**2546.83**	**15.76***	**21.19**
(16,384)	381.71	2127.38	7.85	11.99
256^2	**3274.13**	**7783.76**	**45.64**	**13.77***
(65,536)	±2482.256	±5824.78	±20.79	±7.39
512^2	**11052.16**	DNF	**104.96**	**18.49***
(262,144)	±3887.54	–	±30.21	±8.75
$1,024^2$	DNF	DNF	**434.37**	**21.21***
(1,048,576)	–	–	±160.05	±9.98
$2,048^2$	DNF	DNF	**1353.67**	**64.54***
(4,194,304)	–	–	±679.56	±32.51

DNF stands for "Did Not Finish". Best values marked with "*"

The GPU memory transfer overhead is on par with the lack of CPU parallelization power for these domain resolutions. Nonetheless, for test cases larger than 65,536 (256^2), a clear preference towards GPU starts to be evident, with an average speedup of over 21 times for a problem with 4 over million points (2048^2).

Perhaps the most unexpected results are the test cases for the DEAP framework, which are consistently slower than the EVAL baseline. In tree evaluation, we saw that domain calculation is slightly faster in DEAP than in our baseline. However, 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 would most likely reveal a more optimal set of genetic operators and parameters that could prove 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 be 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 512^2 test case (262,144 points).

We can take the red line in Fig. 5 line as a visualization aid for approaches two orders of magnitude slower. For both iterative methods, tests cases corresponding to larger problem sizes were not completed as they proved to be too time-consuming. Besides, based on previous results, performance margins would only maintain an increasing tendency.

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, as the genetic operators are run exclusively on the CPU. However, This seems not to be the case.

Speedups are higher when compared to tree evaluation experiments. Previous results with 512 tensor side (262,144) for TensorFlow GPU against EVAL regarding raw tree evaluation show a speedup of almost 140 times, which is a far cry from those mentioned above 600 times confirmed with evolution. This can be explained by the caching of intermediate results that TensorFlow performs, leading to a pronounced decrease in evaluation time after the first few initial

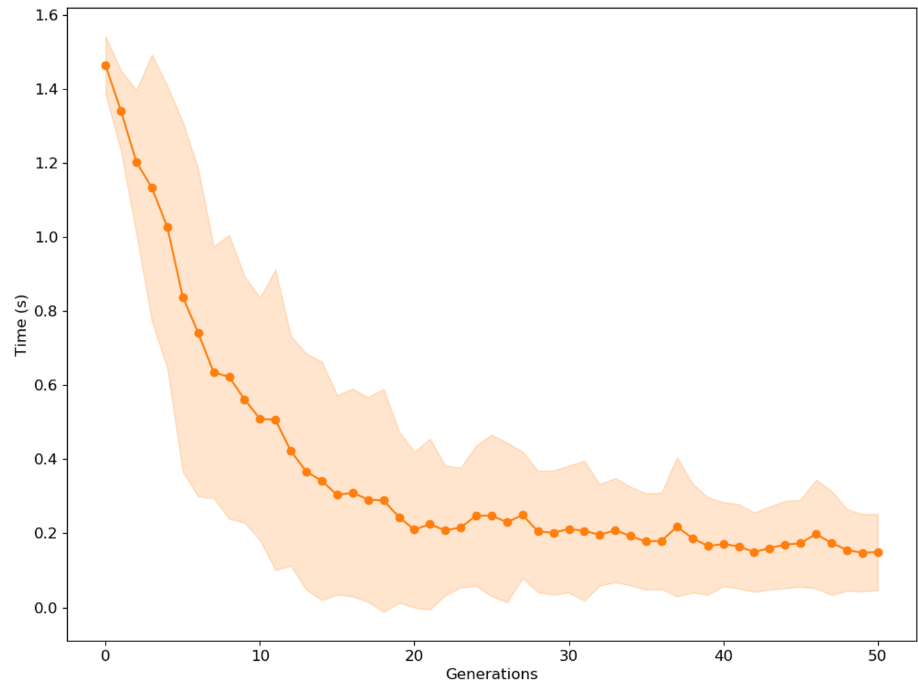


Fig. 6 Average evaluation time (in seconds) across generations for the 64 tensor side test case (4096 points) with the GPU_TF_EAGER approach regarding the evolutionary experiment. Painted regions above and below represent one standard deviation from the average

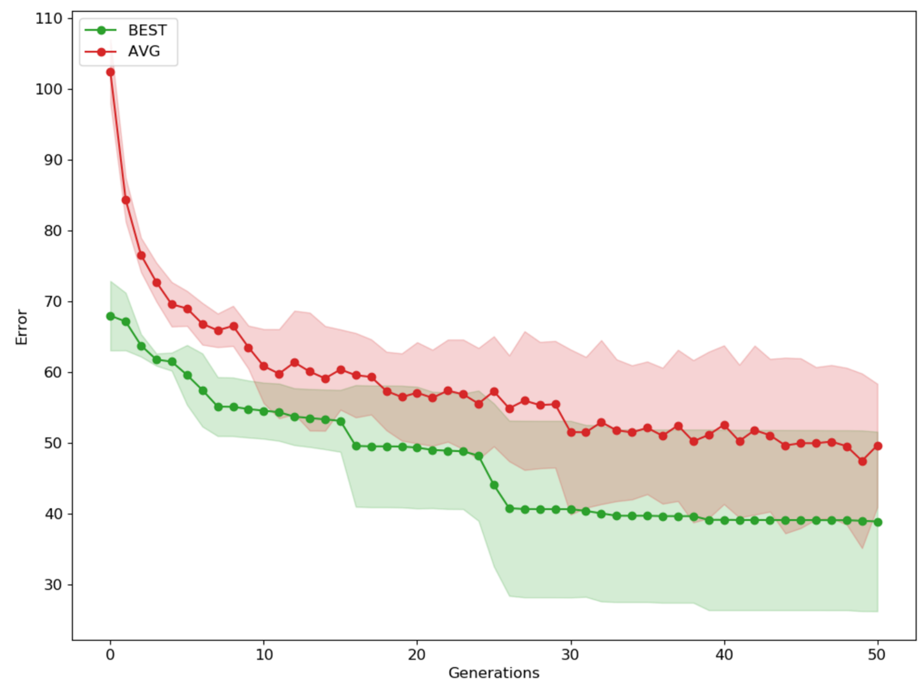


Fig. 7 Average and minimum error values for the GPU_TF_EAGER approach regarding the 2048^2 test case for the evolutionary experiment. Painted regions above and below represent one standard deviation from the average

generations, as observed in Fig. 6. This reduction in execution time reveals similar behaviour to the increase in execution speed already verified for the TensorGP GPU approach, demonstrated in "Depth Experimentation". These results further make a case for expression-based evolution with TensorFlow in eager mode. Finally, to demonstrate TensorGP's capability for the evolution of large domains, in Fig. 7 we showcase fitness progression across generations for the test case with over 4 million points (2048^2).

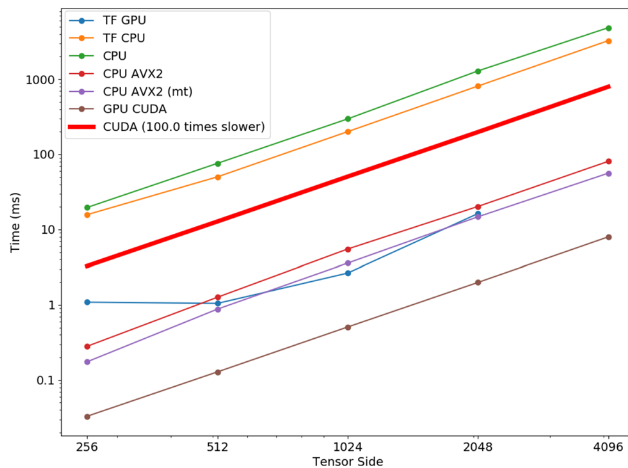
In this manner, we further confirmed the usefulness of vectorization approaches while also showcasing the benefits of using throughput-oriented architecture, such as the GPU, to evaluate domains with higher resolutions.

Table 7 Hardware and software specifications for the *warp* operator experiment

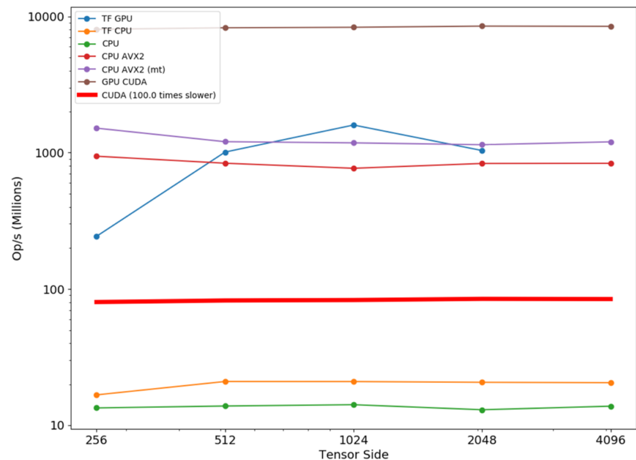
Component	Specification
CPU	Intel®Core™i7–6700 (3.40 GHz)
GPU	NVIDIA GeForce GTX 1060 3 GB
RAM	2 × 8 GB @2666 MHz
Operative System	Ubuntu 16.04
Execution Environment	Command Prompt

an array of n tensors of rank n, each with the new coordinate position for that dimension. The bi-dimensional version of this operator is often used to distort shapes in an image. The functionality of our *warp* is the same as implemented in the scikit-image image processing Python framework [29]. When it comes to performance, because we must perform a gather from the input for every element of the result tensor, the *warp* has demonstrated high parallelism potential.

To better understand the performance implications and verify the potential for parallelism of the *warp* operator, Fig. 8 compares six different approaches. The objective is to check how a TensorFlow implementation fares against more low-level implementations that explicitly define the operator and some other iterative baselines. All approaches were tested using rank 3 tensors with dimensions [x, x, 4], x varying from 64 to 4096. Results are computed from an average of 30 runs. As a side note, the setup described in this subsection was executed on a different machine, the specifications of which are described in Table 7.



(a) Execution Time



(b) Operations per second

Fig. 8 Execution timings in milliseconds (left) and operations per second (right) for several implementations of the *warp* operator across domains of increasing resolution

Explicit Definition Versus Composition

The performance numbers measured so far are promising for TensorGP. However, there are still some performance barriers in the current implementation of our engine.

As mentioned in "Primitive Set", TensorGP operators are defined as a function composition of existing TensorFlow primitives, meaning that each operator can perform multiple TensorFlow calls and even some computations from the native language used (Python in our case). This composition of operators automatically causes some overhead, which could be avoided with a direct definition of all primitives in our function set. Thus, this subsection serves as a proof of concept for the following steps to take in improving the performance of TensorGP. In specific, we compare several explicit and composite implementation approaches for the *warp* operator described in "Explicit Definition Versus Composition".

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 and

The objective of this experiment is not to compare the speed of different programming languages. This, coupled with the fact that TensorFlow allows for creating custom operators using C++ for the CPU code and Compute Unified Device Architecture (CUDA) for the GPU, leads us to implement the different approaches in these languages to get a better idea of the performance of an internal TensorFlow integration. Two approaches are GPU based: one uses TensorFlow composition (TF GPU) while the other implements explicit parallelization with an independent CUDA kernel (GPU CUDA). For the CPU, we implemented the traditional iterative approach (CPU), a

Table 8 Average timing values (in milliseconds) for different *warp* implementations across tensor sizes

Resolution	CUDA	TF (GPU)	TF (CPU)	CPU (AVX2 MT)	CPU (AVX2)	(CPU)
256^2	0.033*	1.077	15.631	0.173	0.278	19.486
512^2	0.127*	1.037	49.856	0.870	1.252	75.467
1024^2	0.503*	2.625	199.636	3.549	5.454	295.192
2048^2	1.973*	16.171	808.798	14.652	20.115	1286.380
4096^2	7.919*	DNF	3255.874	55.819	80.252	4843.459

DNF stands for "Did Not Finish". Best values marked with "*"

Table 9 Average operations per second values (in millions) for different *warp* implementations across tensor sizes

Resolution	CUDA	TF (GPU)	TF (CPU)	CPU (AVX2 MT)	CPU (AVX2)	(CPU)
256^2	8058.283*	243.372	16.771	1515.178	942.761	13.453
512^2	8270.179*	1011.006	21.032	1205.135	837.527	13.895
1024^2	8336.091*	1597.630	21.010	1181.827	768.962	14.209
2048^2	8503.404*	1037.515	20.743	1145.046	834.073	13.042
4096^2	8474.518*	DNF	20.612	1202.258	836.231	13.856

DNF stands for "Did Not Finish. Best values marked with "*"

Single Instruction Multiple Data (SIMD) version using the Advanced Vector Extensions (AVX) instruction set (CPU AVX2), another SIMD version with multithreading added (CPU AVX2 mt) and finally, the TensorFlow approach (TF CPU). Specifically, the SIMD instruction set used was AVX2 (also known as "Haswell New Instructions"). Moreover, all TensorFlow approaches were executed in eager mode.

Figure 8a 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 the 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 8).

In Fig. 8b we can more easily analyze the performance curve for this approach. Smaller domains display a lower operation per the second count caused by the initial overhead. In the meantime, results for domains of higher resolution seem to indicate some 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 VRAM (therefore not being present in the figures and tables). This test case completes in a basic CUDA application seems to indicate that TensorFlow stills perform some level of cashing for intermediate operator composition results, which occupies more memory (even when executing in eager mode).

Additionally, still looking at Fig. 8b, we see that the two SIMD approaches using AVX2 scale slightly better for smaller domains, a tendency that is inverted in the TensorFlow approach for CPU that achieves lower relative computation speed for tensors with size 256 when comparing with more significant test cases (see Table 9).

Overall, the most performant approach (CUDA) was around 300 times faster than the slowest (CPU, serialized), reaching almost 10,000 Million operations per second. The red line represents a threshold for approaches two orders of magnitude slower than CUDA in both figures. 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.

The implementation used in our GP engine was the second-fastest for the 1024 domain test case. Even when resorting to operator composition, this surprising performance might be explained by the TensorFlow internal XLA integration. XLA stands for "Accelerated Linear Algebra" and allows us to internally compile a sequence of operators into a single GPU kernel call, substantially reducing related overheads. Analyzing XLA performance in TensorFlow is, however, not the aim of this experiment.

Even though these results seem impressive, the performance differences shown are not indicative of real-world performance. Tree evaluation often chains dozens or even hundreds of different operators, leading to other problems in memory management between GPU and CPU, as we verified in previous experiments. Therefore, these performance

improvements of many orders of magnitude are not to be expected for full evolutionary runs.

Still, based on the results gathered, we believe that we can significantly increase TensorGPs' performance by explicitly defining the operators of the function set. The explicit definition of operators can be done through the incorporation of C++ and CUDA code directly with the TensorFlow library. The major disadvantage of this method lies in the fact that TensorFlow must be recompiled for operators to be called internally, resulting in less flexibility for the end-user.

Conclusions and Future Work

In this work, we propose different approaches to ease the computational burden of GP by taking advantage of its high potential for parallelism. Namely, we investigate the advantages 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. Additionally, The impact of tree size and domain granularity in the performance of several GP approaches was also analyzed. Lastly, we perform a series of comparisons to determine the benefits of explicitly defining GP operators versus expressing these as compositions of existing primitives.

Our experimental results confirm the benefits of vectorization in GP and further make the case for the use of GPUs for the domain evaluation phase. Specifically, we show that performance gains of up to 600 fold are attainable in TensorGP for the approximation of evaluation domains with high resolutions regarding the Pagie Polynomial function. Furthermore, we demonstrate the benefits of TensorFlow's eager execution model over graph execution for the caching of fitness results throughout generations. Nevertheless, our test results for smaller domains seem to make still the case for more latency-oriented programming models such as the CPU. Therefore, modern-day GP appears to be best suited for heterogeneous computing frameworks like TensorFlow that are device-independent. As a last remark, we verify that operator composition is advantageous, with possible speedup increases of one order of magnitude for the explicit CUDA implementation versus the composite approach currently implemented in TensorGP.

Upon completion of this work, some possibilities are to be considered for future endeavours. Aside from the implementation efforts implicit in the last subsection of our experimentation, we believe that implementing a preprocessing phase to simplify the mathematical expressions of individuals could also significantly improve TensorGP's performance. Moreover, with the incorporation of image-specific operators in our engine, the exploration of evolutionary art is an appealing work path. Finally, the time comparison study amongst different approaches could be extended by including other GP frameworks, possibly under less strict evolutionary setups.

Author Contributions This paper is an extended version of the work published in [6]. We first investigate the benefits of applying data vectorization and fitness caching methods to domain evaluation in Genetic Programming. For this purpose, an independent engine was developed, TensorGP, along with a testing suite to extract comparative timing results across different architectures and amongst both iterative and vectorized approaches. Our performance benchmarks demonstrate that by exploiting the TensorFlow eager execution model, performance gains of up to two orders of magnitude can be achieved on a parallel approach running on dedicated hardware when compared to a standard iterative approach. In summary, the contributions are as follows: (i) a new Genetic Programming Engine; (ii) a benchmark speedup study with both an off the shelf framework and direct implementation of a Genetic Programming algorithm; (iii) evaluation of different execution modes; (iv) showcase the implementation of a particular operator with the analysis on performance for different execution types.

Funding This work is funded by national funds through the FCT - Foundation for Science and Technology, I.P., within the scope of the project CISUC - UID/CEC/00326/2020 and by European Social Fund, through the Regional Operational Program Centro 2020 and by the project grant DSAIPA/DS/0022/2018 (GADgET). We also thank the NVIDIA Corporation for the hardware granted to this research. We also thank the NVIDIA Corporation for the hardware granted to this research.

Data Availability Statement All datasets used in the experiments are publicly available.

Code Availability Statement Part of the code can be found at https://github.com/AwardOfSky/TensorGP.

Declarations

Conflict of interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

Ethics approval Not applicable.

Consent to participate Not applicable.

Consent for publication Not applicable.

References

1. Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M, et al. Tensorflow: A system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16) 2016; 265–283 (2016).

2. Agrawal A, Modi AN, Passos A, Lavoie A, Agarwal A, Shankar A, Ganichev I, Levenberg J, Hong M, Monga R, et al. Tensorflow eager: A multi-stage, python-embedded dsl for machine learning. arXiv preprint 2019; arXiv:1903.01855
3. Andre D, Koza JR. Parallel genetic programming: a scalable implementation using the transputer network architecture. In: Advances in genetic programming. MIT Press; 1996. p. 317–37.
4. Arenas M, Romero G, Mora A, Castillo P, Merelo J. Gpu parallel computation in bioinspired algorithms: a review. In: Advances in intelligent modelling and simulation. Springer; 2013. p. 113–34.
5. Augusto DA, Barbosa HJ. Accelerated parallel genetic programming tree evaluation with opencl. J Parallel Distrib Comput. 2013;73(1):86–100.
6. Baeta F, Correia J, Martins T, Machado P. Tensorgp - genetic programming engine in tensorflow. In: P.A. Castillo, J.L.J. Laredo (eds.) Applications of Evolutionary Computation - 24th International Conference, EvoApplications 2021, Held as Part of EvoStar 2021, Virtual Event, Proceedings, Lecture Notes in Computer Science. 2021;12694: 763-778. Springer. https://doi.org/10.1007/978-3-030-72699-7_48.
7. Burlacu B, Kronberger G, Kommenda M. Operon c++ an efficient genetic programming framework for symbolic regression. In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion. 2020; 1562–1570.
8. Cano A, Ventura S. Gpu-parallel subtree interpreter for genetic programming. In: Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation. 2014; 887–894. ACM.
9. Cano A, Zafra A, Ventura S. Speeding up the evaluation phase of gp classification algorithms on gpus. Soft Comput. 2012;16(2):187–202.
10. Cavaglia M, Staats K, Gill T. Finding the origin of noise transients in ligo data with machine learning. 2018; arXiv preprint arXiv:1812.05225.
11. Chitty DM. A data parallel approach to genetic programming using programmable graphics hardware. In: Proceedings of the 9th annual conference on Genetic and evolutionary computation. 2007; 1566–1573. ACM.
12. Chitty DM. Fast parallel genetic programming: multi-core cpu versus many-core gpu. Soft Comput. 2012;16(10):1795–814.
13. Fortin FA, De Rainville FM, Gardner MAG, Parizeau M, Gagné C. Deap: evolutionary algorithms made easy. J Mach Learn Res. 2012;13(1):2171–5.
14. Fu X, Ren X, Mengshoel OJ, Wu X. Stochastic optimization for market return prediction using financial knowledge graph. In: 2018 IEEE International Conference on Big Knowledge (ICBK). 2018; 25–32. IEEE.
15. Giacobini M, Tomassini M, Vanneschi L. Limiting the number of fitness cases in genetic programming using statistics. In: International Conference on Parallel Problem Solving from Nature. Springer; 2002. p. 371–80.
16. Handley S. 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. 1994; 154–159. IEEE.
17. Keijzer M. Efficiently representing populations in genetic programming. In: Advances in genetic programming. MIT Press; 1996. p. 259–78.
18. Keijzer M. Alternatives in subtree caching for genetic programming. In: European Conference on Genetic Programming. Springer; 2004. p. 328–37.
19. Koza JR, Bennett F, Hutchings JL, Bade SL, Keane MA, Andre D. 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). 1997;1: 404–410. IEEE.
20. Machado P, Cardoso A. Speeding up genetic programming. Procs 2nd Int Symp AI Adapt Syst CIMAF. 1999;99:217–22.
21. Matousek R, Hulka T, Dobrovsky L, Kudela J. Sum epsilon-tube error fitness function design for gp symbolic regression: Preliminary study. In: 2019 International Conference on Control, Artificial Intelligence, Robotics & Optimization (ICCAIRO). 2019; 78–83. IEEE.
22. McDermott J, White DR, Luke S, Manzoni L, Castelli M, Vanneschi L, Jaskowski W, Krawiec K, Harper R, De Jong K, et al. Genetic programming needs better benchmarks. In: Proceedings of the 14th annual conference on Genetic and evolutionary computation. 2012; 791–798.
23. de Melo VV, Fazenda ÁL, Sotto LFDP, Iacca G. A mimd interpreter for genetic programming. In: International Conference on the Applications of Evolutionary Computation (Part of EvoStar). Springer; 2020. p. 645–58.
24. Moore, G.: Cramming more components onto integrated circuits. Proc IEEE **86**(1), 82–85 (1998). https://doi.org/10.1109/JPROC.1998.658762. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?tp=&arnumber=658762&isnumber=14340.
25. Pagie L, Hogeweg P. Evolutionary consequences of coevolving targets. Evol comput. 1997;5(4):401–18.
26. Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming. Lulu Enterprises, UK Ltd (2008).
27. Rowland T, Weisstein EW. Tensor. From MathWorld—A Wolfram Web Resource. http://mathworld.wolfram.com/Tensor.html. Accessed 11 June 2021.
28. Staats K, Pantridge E, Cavaglia M, Milovanov I, Aniyan A. Tensorflow enabled genetic programming. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion. 2017; 1872–1879. ACM.
29. Van der Walt S, Schönberger JL, Nunez-Iglesias J, Boulogne F, Warner JD, Yager N, Gouillart E, Yu T. scikit-image: image processing in python. PeerJ. 2014;2:e453.
30. Wong P, Zhang M. Scheme: Caching subtrees in genetic programming. In: 2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence). 2008; 2678–2685. IEEE.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.