# Speed Benchmarking of Genetic Programming Frameworks

Francisco Baeta
University of Coimbra, CISUC, DEI
Coimbra, Portugal
fjrbaeta@student.dei.uc.pt

João Correia
University of Coimbra, CISUC, DEI
Coimbra, Portugal
jncor@dei.uc.pt

Tiago Martins
University of Coimbra, CISUC, DEI
Coimbra, Portugal
tiagofm@dei.uc.pt

Penousal Machado
University of Coimbra, CISUC, DEI
Coimbra, Portugal
machado@dei.uc.pt

## ABSTRACT

Genetic Programming (GP) is known to suffer from the burden of being computationally expensive by design. While, over the years, many techniques have been developed to mitigate this issue, data vectorization, in particular, is arguably still the most attractive strategy due to the parallel nature of GP. In this work, we employ a series of benchmarks meant to compare both the performance and evolution capabilities of different vectorized and iterative implementation approaches across several existing frameworks. Namely, TensorGP, a novel open-source engine written in Python, is shown to greatly benefit from the TensorFlow library to accelerate the domain evaluation phase in GP. The presented performance benchmarks demonstrate that the TensorGP engine manages to pull ahead, with relative speedups above two orders of magnitude for problems with a higher number of fitness cases. Additionally, as a consequence of being able to compute larger domains, we argue that TensorGP performance gains aid the discovery of more accurate candidate solutions.

## CCS CONCEPTS

• **Computing methodologies** → **Discrete space search**; **Massively parallel algorithms**;

## KEYWORDS

Genetic Programming, Parallelization, Vectorization, TensorFlow, GPU Computing

**ACM Reference Format:**
Francisco Baeta, João Correia, Tiago Martins, and Penousal Machado. 2021. Speed Benchmarking of Genetic Programming Frameworks. In *Proceedings of the Genetic and Evolutionary Computation Conference 2021 (GECCO '21).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3449639.3459335

## 1 INTRODUCTION

Genetic Programming (GP) is a subfield of Evolutionary Computation (EC) that aims to evolve a set of computer programs in a process of continuous stochastic optimization. The fact that the candidate solutions themselves are represented through code makes GP an automatic problem-solving tool that does not require information about the structure or form of the optimal solution. On the other hand, GP is historically known for being rather computationally expensive, especially when it comes to the domain evaluation phase [8].

However, because the expressions to evaluate remains constant throughout the whole fitness domain, it is possible to vectorize the set of data points, providing an opportunity for parallelism. In his work, Keijzer [10] studied the benefits of such vectorized approach over the standard case-by-case evaluation for symbolic regressions. As pointed out, such a vectorized method effectively reduces the asymptotic time complexity of evaluating an individual to the number of nodes it contains.

A data vectorization approach is oftentimes coupled with the capabilities of parallel hardware such as Graphics Processing Units (GPUs) [13]. Whereas a Central Processing Unit (CPU) strives to minimize the latency of operations, a GPU mainly focuses on maximizing data throughput , *i.e.* the amount of information that gets processed in a given time unit. For GP, a higher data throughput translates to more GP operations per second (GPops) and consequently faster evaluation speeds.

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 concept applies to domain evaluation in GP, in the sense that in every generation one must run every program in the population on the same set of fitness cases. In recent years, we have witnessed a steady growth in the computing capability of GPUs.

Indeed, even considering that CPUs have seen a throughput increase in recent years, their floating point calculation performance is still nowhere near that of a standard GPU. In what concerns GP, a throughput oriented architecture is beneficial in the sense that we can always increase the amount of evaluated fitness cases, maximum generations, maximum allowed, etc., in the hopes of improving the fitness of resulting candidate solutions.

Considering how crucial evaluation speeds are for GP applications, in this work we aim to provide a comprehensive study on the performance differences amongst various frameworks using different iterative and parallel data implementations. In particular, we analyze TensorGP, a new GP engine written in Python that takes advantage of the TensorFlow library to vectorize the evaluation domain and perform fitness caching [3]. Other frameworks analyzed in this paper include KarooGP, ECJ, TinyGP (Java), DEAP, Evolving Objects (EO), and GPlearn.

The remainder of this paper is structured as follows. Section 2 summarizes the architecture and workflow of TensorGP. Section 3 introduces the remaining frameworks to test, detailing the experimental setup as well as the problems to benchmark. Section 4 analyses and discusses gathered results. Finally, Section 5 compiles the main conclusions taken from this study while suggesting several options for future work.

## 2 TENSORGP

TensorGP was developed to provide a robust system for seamlessly integrating domain vectorization in GP [3]. Furthermore, taking advantage of the TensorFlow library, TensorGP manages to heterogeneously distribute computational efforts across different processor types and architectures.

A detailed view of TensorGP's feature set is documented in [3]. The following list summarizes the current core features of TensorGP:

(1) Implementation of standard GP hyperparameters and functionalities: initialization methods, elitism, crossover
(2) Ability to stop and restart the evolutionary process;
(3) Support for custom initial populations in addition to randomly generated ones;
(4) Multiple stop criteria;
(5) Support for customized operators;

### 2.1 Domain Evaluation with Tensors

In TensorGP, the domain of fitness cases to be evaluated is defined by a tensor. Tensors are widely used in many fields of physics and mathematics not only as data structures but also to express multilinear mappings within a vector space [18]. However, in the context of Artificial Intelligence and Computer Science in general, they are often simply defined as a multidimensional array of elements. By this definition, a tensor is nothing more than a generalization of scalars (*i.e.* a tensor with 0 dimensions), vectors (a tensor with 1 dimension), matrices (a tensor with 2 dimensions), and so on. Therefore, for a problem domain containing $n$ dimensions, there will be $n$ variables in the terminal set that TensorGP creates to represent its candidate solutions.

The term "variables" is misleading in this scenario, as they in fact refer to constant tensors in TensorGP. In a serial approach, $x$, $y$, $z$, ... will actually be variables because they change according to which element of the domain we are evaluating. In a vectorized approach, however, we evaluate the entire domain in one vector operation and, as a result, $x$, $y$, $z$, ... will be tensors that contain coordinates within our problem domain. This means that, for instance, every element in the $x$ tensor will hold the coordinate value of a given fitness case along the $x$-axis. These tensors are initialized at the beginning of the application and remain unchanged throughout the evolutionary process.

In essence, TensorGP recursively applies tensor operations between these coordinate "variables" and other tensors of the terminal set to produce a final output tensor that represents the evaluated program. The following subsection details the intermediate representations used to efficiently calculate this output tensor.

### 2.2 Representation Overview

Starting with the traditional way of representing a GP individual as a mathematical expression, there are three translation steps (see Figure 1) employed by TensorGP to produce the phenotype for that individual:

(1) Convert the expression to a Tree Graph.
(2) Transform the Tree Graph into a Directed Acyclic Graph (DAG).
(3) Traverse the obtained graph, recursively performing operations to produce the phenotype.

The resulting phenotype from step 3 is represented as our result tensor that will be used in the next phase of fitness assessment. This resulting phenotype, shown in Figure 1, represents a resulting tensor of rank-3 relating to the height, width and color channels of an image.

It should be noted that step 2 is not implemented by TensorGP itself, but is rather an abstraction provided by TensorFlow to optimize the flow of tensor operations based on its execution mode. TensorFlow has two main modes of execution: *eager* and *graph*. As opposed to *graph* execution, *eager* mode does not explicitly build the intermediate DAG of operations. Although TensorGP was initially developed under the *graph* execution model, it was later upgraded to exploit *eager* execution as this was shown to yield better performance results [3].

## 3 FRAMEWORKS

This section presents the GP frameworks involved in the time comparison benchmarks and the necessary steps taken in the standardization of these systems to achieve commensurable results.

Aside from efficiency and wide-spread use, an essential criterion upon selecting these specific systems lies in the fact that they are free and open-source.

### 3.1 Implementations

Seven different frameworks were tested as shown in Table 1.

**Table 1: Considered GP frameworks along with their respective implementation languages and device support.**

| Framework | Language | Processor |
|---|---|---|
| TensorGP | Python | GPU/CPU |
| KarooGP | Python | GPU/CPU |
| ECJ | Java | CPU |
| EO | C++ | CPU |
| TinyGP | Java | CPU |
| GPlearn | Python | CPU |
| DEAP | Python | CPU |

We start by detailing TensorGP and KarooGP. TensorFlow's design philosophy builds upon the principles of heterogeneous computing [1] and as such, TensorGP also inherits this abstraction layer for hardware utilization. For this reason, benchmarks for both CPU and GPU devices will be provided in the next section, regarding

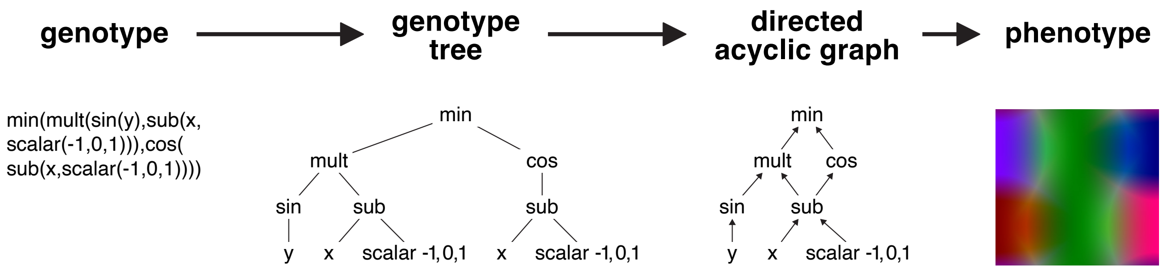

**Figure 1: Genotype to phenotype translation phases in TensorGP.**

this framework. Akin to TensorGP, KarooGP resorts to TensorFlow to vectorize operations over the domain of fitness cases [21], which makes it a good candidate for direct comparison with TensorGP running in GPU. Moreover, we chose KarooGP as there already exist several research projects that use this engine [4, 5, 20].

As far as performance is concerned, the main difference between TensorGP and KarooGP resides in the TensorFlow model of operation. Even though both engines represent individuals with a tree-based data structure, KarooGP uses TensorFlow's *graph* execution model, which means that every individual in the population has to be internally compiled into a DAG before the evaluation phase can happen. TensorGP, on the other hand, operates in an *eager* execution model that evaluates the tensor operations imperatively without building intermediate graphs. As a result, *eager* execution aims to eliminate the overhead associated with graph building without sacrificing the benefits provided by graphs [2]. In the field of EC in general, the individuals of a population tend to change throughout generations. While many memory and speed optimizations can be achieved by working with a compiled DAG structure, the evolutionary process in GP renders the compilation of an individuals' representation less useful [3].

Outside the realm of GPU computing, there are other prominent GP engines worth testing against. In particular, DEAP is a commonly used EC framework that represents the standard for iterative domain evaluation in GP research and literature [7]. Aside from its popularity, we have chosen to include DEAP as it allows for the quick prototyping of controlled evolutionary environments with little implementation efforts. Likewise, GPlearn [22] also capitalizes on the simplification of building GP models by extending the scikit-learn Python machine learning library to perform Symbolic Regression. Additionally, GPlearn supports running the evolutionary process in parallel. However, to the best of our knowledge, the phase that gets parallelized in this framework is the application of genetic operators themselves, not the evaluation of fitness cases. Because the experimental setup used is very evaluation intensive, we did not verify any positive performance gains from running GPlearn with multithreading and therefore this option was disabled.

Another well-established system for evolutionary computation is ECJ [14], written in Java. Because most of ECJ's functionality is determined by a hierarchy of parameter files provided by the user, this framework is suited for complex research projects. Despite ECJ not being primarily designed to run on parallel hardware, its ease of integration enabled Robilliard et al. [17] to develop and compare two GP parallelization schemes running on a NVIDIA GPU.

Similarly to ECJ we must consider EO, an object-oriented framework that provides a flexible set of classes to build EC applications [11]. The object-oriented design philosophy is identical to ECJ, with EO providing several classes and interfaces to abstract the evolutionary process. Although the initial learning curve for these last two frameworks might be steeper compared to other GP-based engines such as DEAP, their generalization ability, extensive documentation, and widespread use are the main factors for being included in this comparative performance study.

Lastly, we include TinyGP, a more minimalistic implementation of a GP system that uses a form of tree-based GP. This implementation is based on the Java implementation presented by R. Poli et al.[16], which in turn is based on the engine originally developed to meet the requirements set out in the TinyGP competition of the Genetic and Evolutionary Computation Conference (GECCO) 2004.

## 3.2 Standardization Efforts

In an effort to level the playing field and achieve comparable results, some modifications were made to the existing frameworks. Hereafter are presented the features that were the object of modifications as well as the reasoning behind them. The source code for all projects used, including modifications, is publicly available. [1]

GP, like any field of EC, entails stochastic processes related to evolution (*e.g.* primitive and operator selection, population initialization, genetic operators, etc) and as such, every GP framework needs to employ some form of random number generation. Because different implementations call their random generators with different frequencies and at different stages of the evolutionary pipeline, it is not possible to ensure the replication of a given evolutionary path in different engines using the same random seed (even if both initial populations are set to be equal). This means that experimental runs will inevitably follow distinct evolutionary paths and consequently perform a different number of GP operations, which can greatly affect both execution times and the evolution of candidate solutions.

Nevertheless, although a common evolutionary path cannot be set beforehand, we can make efforts to standardize the set of features that the evolutionary process uses, relying upon sufficient experimental runs to average out the inherent randomness. While

[1]Repository available at: https://github.com/AwardOfSky/GP-framework-comparison

it is outside the scope of this work to change existing implementations, because some frameworks do not provide out of the box support for all GP features considered in our experimentation, some implementation efforts were necessary, albeit avoided when possible.

Many of the implementation struggles faced stemmed from the nature of our benchmark: we are mainly interested in evolving domains with a large number of fitness cases (millions in some instances). The amount of data points to be evaluated made the approach of passing a file as input for each individual fitness case unfeasible as such files could easily exceed 1GB for some experiments. Therefore, engines like KarooGP that followed this design pattern were modified to programmatically evaluate over a linearly spaced domain of data points, which boundaries and granularity can be defined beforehand. Furthermore, we opted to implement Ephemeral Random Constants [12] across the board, either by generating the values within the tree representation or by having them pre-computed in the terminal set, as TinyGP does. Another implementation concern was the support for GP operations of variable arity, which meant further modifications in some cases.

## 4 EXPERIMENTATION

In this section, we compare the execution times of various GP systems within a controlled environment, with all results representing an average of 30 runs. The employed hardware and software is as follows:

**Table 2: Hardware and software specifications used for all experiments.**

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

Before proceeding with the experimental results, we must acknowledge that GP is not historically known for having a well-defined set of benchmarking problems and setups. As a result, some studies analyzed this issue and suggested useful pointers and problem sets aimed at organizing better benchmarking suites in GP [6, 15]. This work intends on putting these guidelines to use in the hopes of achieving a rigorous comparative study.

### 4.1 Experimental Setup

All tests employed in this section concern the approximation of the Pagie polynomial function defined by:

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

We chose this function as it is considered to be challenging to approximate and is therefore recommended by several GP benchmark articles [9, 15].

As mentioned, the main objective is to compare frameworks directly. In particular, we investigate how increasing the domain size of a problem impacts running times by employing a series of test sets that exponentially increase the number of fitness cases to calculate. Every test set refers to the evaluation of a square two-dimensional domain of values. In the first test set, each framework evaluates a grid of 64 by 64 (4,096 data points). Each subsequent test set then doubles the length of each side of the grid, effectively quadrupling the total number of evaluations (*i.e.* the second test set has a size of 128 by 128 or 16,384 data points, all the way up to 4,096 by 4.096 or over 16 million points).

Because the domain sizes considered in some test sets encompass a wide range of values, our best bet for graphical representation is to use a logarithmic scale as it would be otherwise impossible to distinguish between results. It is also worth noting that because the logarithmic nature of the scale clutters the visualization of the standard deviation values, a table is provided with the corresponding average and standard deviation results. Aside from domain size, the GP parameterization remains unchanged obeying the following setup:

**Table 3: Experimental GP parameters for the framework comparison experiment.**

| Parameter | Value |
|---|---|
| Runs | 30 |
| Generations | 50 |
| Population size | 50 |
| Elite size | 1 |
| Tournament size | 3 |
| Mutation probability | 0.1 |
| Mutation probability | 0.9 |
| Minimum initial depth | 1 |
| Maximum initial depth | 10 |
| Maximum allowed depth | 10 |
| Generation method | RHH (population) |
| Fitness metric | RMSE |
| Domain range | [-5, 5] |

As pointed out in Table 3, the initial population was generated with the Ramped Half-and-Half (RHH) method. Typically, there are two ways of implementing this method: one that works over a single tree, generating a root node and creating half of the tree with the grow method using full for the other half, and one that works over the entire population, dividing it into blocks of different depths splitting the number of trees in each block to use either the full or grow method. TensorGP uses the second approach to implement RHH. However, this is only but an implementation detail, other frameworks were given the freedom to employ their versions of the RHH algorithm.

Concerning fitness metrics, the Root Mean Squared Error (RMSE) was implemented across all engines.

No evolutionary methods were re-implemented within the analyzed frameworks to strictly comply with TensorGP's implementation. For this reason, we are not comparing the evolution of the same set of initial populations using the same genetic operator implementations. However, while it is true that a straightforward comparison can only be achieved over the same evolutionary path, as discussed in the last section, such a comparison is unattainable as

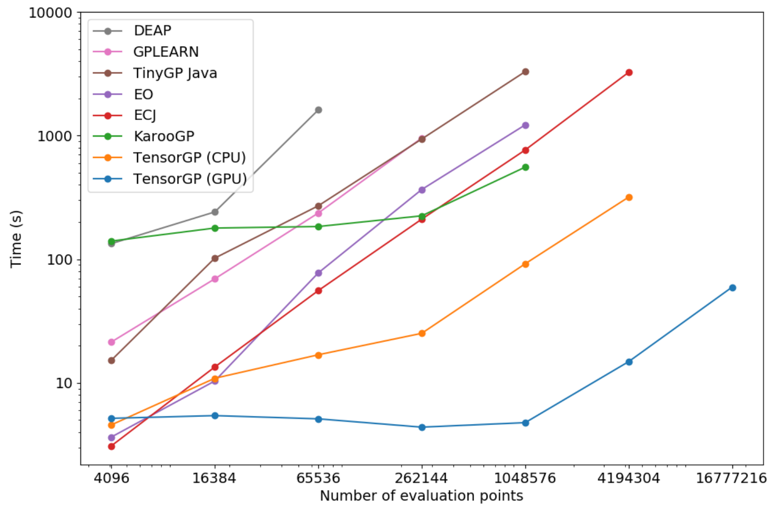

**Figure 2: 30 run average of total execution time (measured in seconds) across different approaches evaluating a two-dimensional domain ranging from 4,096 data points to over 16 million. Values not shown corresponds to test sets that did not finished either due to memory limitations or by failing to meet the time constraint threshold.**

it is impossible to reproduce the same evolutionary stochasticity in different frameworks. Besides, implementing the same algorithms across all engines (besides being too daunting of a task) would still not guarantee reproducibility. As a sidenote, GPlearn and KarooGP did not employ elitism. As far as we investigated, elitism is not directly included in the feature set of said frameworks. Still, we assume the possibility of having missed some implementation detail.

Disclaimers aside, we should see this comparison in terms of how much time will it take to execute the same setup across all frameworks considered.

## 4.2 Results

For the performance benchmark, a total of 1,230 tests were carried out. Some approaches that either took too long to finish on average (maximum of 1 hour per experimental run, on average) or crashed due to memory limitations were not considered. Because TengorGP was benchmarked twice on different hardware, we will refer to each run of the whole array of test sets as relating to an approach rather than a framework to avoid confusion. Figure 2 shows the total execution time for all approaches across different domain sizes, each value corresponding to an arithmetic average of 30 runs.

For larger domains, a clear preference towards data vectorization starts to become evident. Besides TensorGP, the only engine to internally employ data vectorization is KarooGP. Even though KarooGP is comparatively slower than most approaches for smaller domains, its running time remains relatively unchanged throughout the first few test sets, as seen in Table 4. This phenomenon is easily explained by the internal overhead of building the DAG of GP operations that KarooGP performs with the TensorFlow *graph* execution model. Up until the fourth test set (262,144 points), most of KarooGP execution is spent building the intermediate graphs rather than executing GP operations, hence the jump from last (on the first test set) to third fastest approach.

As demonstrated, the two fastest approaches both used TensorGP, only on different processors. As the name implies, "TensorGP (GPU)" was executed on the GPU described in Table 2 and "Tensor CPU" on the matching CPU. Due to TensorGP resorting to TensorFlow for domain evaluation, there is always data vectorization being exploited as long as the hardware supports it. We eould be inclined to assume that an approach running on GPU using TensorFlow such as KarooGP would not be slower than another running the same environment under the same library but using a CPU architecture. In reality, aside from the aforementioned graph building overhead specific to KarooGP, we must not underestimate the vectorization capabilities of modern CPUs. Indeed, TensorFlow is able to take advantage of the AVX2 [19] instruction set, available on the CPU used. Coupled with the absence of graph building overhead, this explains why TensorGP execution on CPU is the second-fastest approach according to our results.

Regarding TensorGP running in GPU, we can identify a clear performance lead throughout all test sets except the first (4,096 data points), where it is outperformed by its CPU counterpart as well as EO and ECJ. However, the lack of relative performance at the start can be easily explained by the fact that, while executing on the GPU, TensorFlow first needs to move the tensor data from the CPU, where it is initialized, to the GPU, and then back again. Although this memory management overhead limits the performance of TensorGP GPU for smaller domains, the modest performance hit pays off in the remaining test sets. As a matter of fact, by the second test set (16,384 points), TensorGP is already almost twice as fast as the approach running on CPU (see Table 4). TensorGP GPU was the only approach that made feasible the evaluation of the last test set containing over 16 million data points, with an average execution time of just under a minute. Apart from the parallelization power of our GPU, this would not be possible without a sufficiently large VRAM to hold all the tensor data.

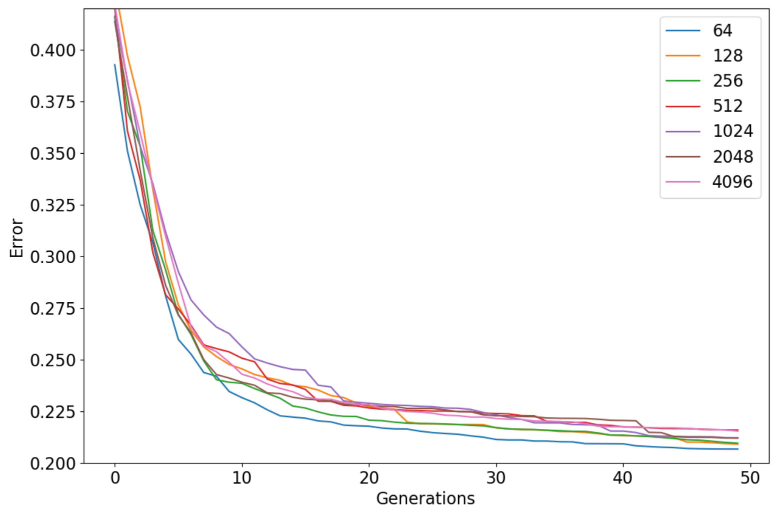

**Figure 3: 30 run average values regarding the error from the optimal solution of the best individuals across generations for test sets ranging from 4,096 points to over 16 million points. These results correspond to the TensorGP GPU approach, measured with the RMSE fitness metric.**

Moreover, it is shown that, for smaller domains, the overhead of moving tensor data is so overwhelming when compared with the time that it takes to execute GP operations, that TensorGP GPU maintains roughly the same overall engine time up to domains with more than 1 million fitness cases (fifth test set). After this point, TensorGP GPU starts to exhibit linear behaviour in proportion to the domain size, as shown by the last two test sets. This happens as we meet the throughput and Video Random Access Memory (VRAM) limitations of the GPU considered, resulting in more data transfers between processors. TensorGP CPU also shows a similar tendency, with a somewhat sublinear time behaviour up until the fourth test set (262,144 data points), a point from which it increases linearly with an increase in tensor side. In practice, this phenomenon of near-constant performance followed by a linear increase in computation time is characteristic of all approaches that use vectorization, especially those with higher parallelization potential such as TensorGP and KarooGP.

As opposed to vectorized approaches, the iterative ones display a linear-like behaviour from the start. ECJ, written in Java, is a performance winner when it comes to traditional evaluation, only giving its place to EO in the second test set (16,384 points) by a small margin. In turn, EO follows closely behind ECJ, being about 1.5 times slower than this framework for a tensor side of 2,048 (over 4 million data points).

Akin to ECJ, TinyGP is also implemented in Java but falls short of the evaluation speed achieved by ECJ and EO. Nevertheless, we must keep in min that TinyGP is mainly an academic exercise meant to show how easily one can set up a GP system, not necessarily a full-blown research tool. As a consequence, TinyGP has a more minimalistic design that compromises some performance for simplicity. We also showcase GPlearn, a framework that reveals similar execution times compared to TinyGP even though it is implemented in Python, which is inevitably slower due to the interpretation overheads. For this reason, GPLearn fell short of completing the fifth test case in the allocated time frame, even if by a slight margin.

Lastly, DEAP is the slowest considering evaluation speed alone, only completing the three first test sets. Still, in the same manner that TinyGP is not specifically tailored for performance, the main asset associated with DEAP is its ease of use and prototyping rather than speed. In fact, and despite being an EC framework, DEAP was one of the frameworks that required fewer implementation efforts to integrate our experimental setup, the other Python engines that are GP specific.

These tests explore populations with different average depths accentuates the tendency for some values to deviate from the norm, explaining some atypical behaviour in execution times, as demonstrated by the standard deviations in Table 4. This was verified in approaches such as DEAP and TensorGP GPU. In DEAP, the 256 test set took considerably longer than expected following a linear behaviour. On the contrary, TensorGP GPU saw a marginal reduction in execution times throughout the first few test sets, mainly due to large data transfer overheads and negligible tensor execution times.

Finally, an analysis of our experimental results would not be complete without looking at performance metrics regarding fitness evolution. Figure 3 shows the best fitness across generations for all considered test sets performed with TensorGP running on GPU. Because we perform elitism of size 1, the best-fitted individuals will automatically be promoted to the next generation, ensuring the monotonical decrease of obtained results. As shown, every test set roughly follows the same behaviour. The first test set displays, on average, slightly better performance across generations, with the largest domain size being marginally worst by the end of the experimental runs.

However, a disclaimer must be made about the direct comparison between test sets: because each set evaluates domains of increasing sizes, a lower score on the RMSE fitness metric for a larger domain does not necessarily mean that a worse solution was found. In fact, more data points condensed in the same domain range translates into less granularity, which in turn allows the assessment against a

**Table 4: Execution times, measured in seconds, of 30 run average (AVG) and standard deviation (STD) values across different approaches evaluating a two-dimensional domain which length ranges from 64 to 4,096 fitness cases. Cells corresponding to the fastest approach are highlighted for each test set. Values represented with DNF did not finished either due to memory limitations or time constraints.**

| | | TensorGP (GPU) | TensorGP (CPU) | KarooGP (GPU) | ECJ | EO | TinyGP (Java) | GPlearn | DEAP |
|---|---|---|---|---|---|---|---|---|---|
| **$64^2$** | **AVG** | 5.17 | 4.58 | 140.16 | 3.10 | 3.65 | 15.25 | 21.41 | 133.63 |
| 4,096 | **STD** | 2.66 | 2.19 | 47.35 | 1.70 | 4.31 | 30.96 | 12.32 | 185.49 |
| **$128^2$** | **AVG** | 5.44 | 10.89 | 178.64 | 13.45 | 10.36 | 102.06 | 69.43 | 241.12 |
| 16,384 | **STD** | 3.24 | 5.38 | 83.56 | 8.99 | 5.50 | 225.68 | 39.21 | 144.78 |
| **$256^2$** | **AVG** | 5.12 | 16.87 | 183.77 | 55.75 | 77.50 | 270.22 | 236.23 | 1610.46 |
| 65,536 | **STD** | 1.87 | 8.96 | 75.36 | 40.18 | 79.32 | 407.23 | 84.47 | 3088.09 |
| **$512^2$** | **AVG** | 4.39 | 25.16 | 224.22 | 210.77 | 365.95 | 937.06 | 954.02 | DNF |
| 262,144 | **STD** | 1.88 | 12.09 | 51.01 | 108.04 | 612.17 | 1513.51 | 545.43 | DNF |
| **$1,024^2$** | **AVG** | 4.77 | 91.85 | 555.33 | 764.34 | 1219.3 | 3287.50 | DNF | DNF |
| 1,048,576 | **STD** | 2.02 | 52.81 | 167.24 | 430.40 | 1375.67 | 4313.04 | DNF | DNF |
| **$2,048^2$** | **AVG** | 14.85 | 317.78 | DNF | 3244.16 | DNF | DNF | DNF | DNF |
| 4,194,304 | **STD** | 9.82 | 134.72 | DNF | 2058.81 | DNF | DNF | DNF | DNF |
| **$4,096^2$** | **AVG** | 59.46 | DNF | DNF | DNF | DNF | DNF | DNF | DNF |
| 16,777,216 | **STD** | 24.66 | DNF | DNF | DNF | DNF | DNF | DNF | DNF |

DNF stands for "Did Not Finish".

target that is closer to the optimal solution. Thereby, the fact that the fitness results are fairly close together only demonstrates that the Pagie Polynomial does not necessarily take advantage of the evaluation over domains with millions of fitness cases (at least not within the considered range).

## 5 CONCLUSION AND FUTURE WORK

In this work, we carried out a comparative performance study between an array of GP capable frameworks that are widely used amongst the research community. Additionally, in our study we include TensorGP, a recently developed GP engine written in Python that takes advantage of the TensorFlow library to perform data vectorization.

The experimental results demonstrate that TensorGP achieves faster execution times within the same controlled setup when compared to other systems. Specifically, speedups of up to 100 times were verified when compared to well-established frameworks that perform iterative evaluation such as ECJ or EO as well as a similar vectorization boosted framework (KarooGP). The aforementioned performance gains are shown to be even more pronounced for the evaluation of domains containing millions of data points and when executing on throughput oriented architectures where GPUs are included. This increase in the number of fitness cases to evaluate provides more information about the optimal solution increasing the potential for approximation and discovery of more accurate candidate solutions. However, analyzing fitness evolution across generations, we conclude that the benchmarked problem does not significantly benefit from being evaluated on larger domains, as obtained results are predominantly similar throughout all test sets.

More importantly, we must take these results in perspective, realizing that each framework has certain application specializations and that no approach is a perfect fit for all scenarios. Overall, the fast execution times provided by TensorGP reveal great potential to solve problems with larger domains that require more detailed solutions.

Regarding future work, several endeavors are to be considered. Primarily, the current study can be extended to include other problems, not only pertaining to symbolic regression but also in the areas of classification, predictive modeling, and other metrics. In particular, the study of problems with a higher pervasiveness of local extrema seems compelling as, theoretically, such problems would reap the benefits furnished by evaluating over high-resolution domains. Likewise, because TensorGP performance makes feasible the exploitation of GP parameters, investigating the influence of individuals with higher depths for certain problems is another possibility to consider. Furthermore, the inclusion of current real-world GP problems and experimental setups within future testing suites should not be neglected, as these provide irrefutable arguments to the usefulness of any GP framework.

## ACKNOWLEDGMENTS

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.

## REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

[2] Akshay Agrawal, Akshay Naresh Modi, Alexandre Passos, Allen Lavoie, Ashish Agarwal, Asim Shankar, Igor Ganichev, Josh Levenberg, Mingsheng Hong, Rajat Monga, et al. 2019. TensorFlow Eager: A multi-stage, Python-embedded DSL for machine learning. *arXiv preprint arXiv:1903.01855* (2019).

[3] Francisco Baeta, João Correia, Tiago Martins, and Penousal Machado. 2021. TensorGP – Genetic Programming Engine in TensorFlow. In *Applications of Evolutionary Computation*. Springer International Publishing, 763–778.

[4] Marco Cavaglià, Sergio Gaudio, Travis Hansen, Kai Staats, M Szczepańczyk, and Michele Zanolin. 2020. Improving the background of gravitational-wave searches for core collapse supernovae: A machine learning approach. *Machine Learning: Science and Technology* 1, 1 (2020), 015005.

[5] Marco Cavaglia, Kai Staats, and Teerth Gill. 2018. Finding the origin of noise transients in LIGO data with machine learning. *arXiv preprint arXiv:1812.05225* (2018).

[6] Robert Feldt, Michael O'Neill, Conor Rayn, Peter Nordin, and William B Langdon. 2000. GP-beagle: A benchmarking problem repository for the genetic programming community. *Late Breaking Papers at GECCO* (2000), 1–8.

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

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

[9] Robin Harper. 2012. Spatial co-evolution: quicker, fitter and less bloated. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*. 759–766.

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

[11] Maarten Keijzer, Juan J Merelo, Gustavo Romero, and Marc Schoenauer. 2001. Evolving objects: A general purpose evolutionary computation library. In *International Conference on Artificial Evolution (Evolution Artificielle)*. Springer, 231–242.

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

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

[14] Sean Luke, Liviu Panait, Gabriel Balan, Sean Paus, Zbigniew Skolicki, Jeff Bassett, Robert Hubley, and A Chircop. 2006. Ecj: A java-based evolutionary computation research system. *Downloadable versions and documentation can be found at the following url: http://cs. gmu. edu/eclab/projects/ecj* 880 (2006).

[15] James McDermott, David R White, Sean Luke, Luca Manzoni, Mauro Castelli, Leonardo Vanneschi, Wojciech Jaskowski, Krzysztof Krawiec, Robin Harper, Kenneth De Jong, et al. 2012. Genetic programming needs better benchmarks. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*. 791–798.

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

[17] Denis Robilliard, Virginie Marion, and Cyril Fonlupt. 2009. High performance genetic programming on GPU. In *Proceedings of the 2009 workshop on Bio-inspired algorithms for distributed systems*. 85–94.

[18] Todd Rowland and Eric W Weisstein. 2020. Tensor. From MathWorld—A Wolfram Web Resource. (2020). http://mathworld.wolfram.com/Tensor.html

[19] Daniel Smilkov, Nikhil Thorat, Yannick Assogba, Ann Yuan, Nick Kreeger, Ping Yu, Kangyi Zhang, Shanqing Cai, Eric Nielsen, David Soergel, et al. 2019. Tensorflow. js: Machine learning for the web and beyond. *arXiv preprint arXiv:1901.05350* (2019).

[20] Kai Staats. 2016. *Genetic programming applied to RFI mitigation in radio astronomy*. Master's thesis. University of Cape Town.

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

[22] Trevor Stephens. 2019. GPLearn (2015). *URL https://gplearn. readthedocs. io/en/stable/index. html* (2019).