A Tensor-Based Genetic Programming Framework for Symbolic Regression on Structured Domains

Jorge Martinez-Gil

Software Competence Center Hagenberg GmbH
Softwarepark 32a, 4232 Hagenberg, Austria
jorge.martinez-gil@scch.at

Abstract

Genetic Programming is an evolutionary method for finding symbolic models that fit data, a task called symbolic regression. GP typically requires substantial computational resources, since every candidate program must be tested on many data points. This work introduces a new GP approach that uses tensor algebra and parallel hardware to efficiently represent and evolve mathematical expressions, focusing on structured and high-dimensional data. Experimental comparisons with a standard GP system are conducted on several symbolic regression benchmarks. The new method matches or outperforms traditional GP in terms of accuracy and convergence, and achieves significant gains in runtime. The analysis includes a discussion of the method's scalability, the advantages for large-scale problems, and considerations related to overhead and memory use. The proposed approach allows symbolic regression tasks to be handled more efficiently and at larger scales, supporting new applications of GP to structured data domains.

Keywords: Genetic programming, symbolic regression, tensor representation, GPU acceleration, structured data

1. Introduction

Symbolic regression via Genetic Programming (GP) involves evolving mathematical expressions to fit a given set of data points without a predetermined model structure [13]. It has been successfully applied in various fields, from engineering to scientific discovery, to find human-interpretable models (e.g., formulas governing physical laws) directly from data [9]. GP was first introduced by Koza [8] as a method to evolve programs (often represented as syntax trees) through Darwinian evolution principles. A key strength of GP is that given a sufficiently rich set of primitives, it can in principle evolve any computable function. This flexibility comes at a computational cost: GP typically requires evaluating a large population of candidate solutions over many fitness cases (data points) for multiple generations. In fact, fitness evaluation is usually

the most computationally expensive step of GP [16]. As problem domains grow more complex or high-dimensional, the number of fitness cases (e.g., data points, or pixels in an image domain) can be very large, making GP execution prohibitively slow.

GP is also naturally amenable to parallelization. Each individual in the population can be evaluated independently on different compute threads or devices, and even the evaluation of a single program on multiple data points can be parallelized. Past research has explored two main ways for accelerating GP: *fitness caching*, which avoids re-computing identical sub-expressions across individuals, and *data vectorization*, which evaluates all data points simultaneously by using vector or tensor operations. The latter approach effectively reduces runtime by performing a single bulk operation per node (function) in the program, rather than looping over data points. Modern computing hardware such as Graphics Processing Units (GPUs) are highly optimized for tensor operations and offer massive throughput for data-parallel tasks. This means that a GP system designed around vectorized computation and GPU acceleration could dramatically speed up symbolic regression.

In this paper, we propose a GP engine that implements symbolic regression using tensor-based representations and operations. The core idea is to represent the evaluation domain (the set of all input values for fitness cases) as a tensor and to express all primitive operations in the individuals as tensor functions. This way, an entire population of programs can be evaluated using efficient linear algebra routines on parallel hardware. Our approach builds upon prior work that applied GPU and vectorization techniques to GP, but extends it to better handle structured domains (for example, evolving formulas over images or multi-dimensional grids) and uses the latest dynamic computation models for improved performance. We provide a rigorous formalization of GP in a tensor computation context, define the genetic operators with mathematical notation, and analyze the computational complexity of each evolutionary step under this model.

We empirically evaluate our approach on standard symbolic regression benchmarks, comparing it against a conventional tree-based GP implementation (without tensor acceleration). Our experiments cover both one-dimensional function regression tasks and a two-dimensional symbolic regression problem (surface fitting), to demonstrate the ability of our approach to handle structured input domains.

The results show that our solution can find solutions of accuracy comparable to baseline GP, while achieving significant speedups in execution time (often one to two orders of magnitude faster for large datasets). For instance, on a problem with tens of thousands of data points, our approach runs in seconds whereas a baseline GP might require minutes to hours to finish, in line with speedups reported by other GPU-accelerated GP systems. These improvements can enable GP to scale to problem sizes previously impractical, such as high-resolution evolutionary design or large-scale system identification tasks.

The remainder of this paper is organized as follows. Section 2 reviews related work on symbolic regression and accelerated GP methods. Section 3 presents our methodology, including the formal problem definition, tensor-based representation of individuals, genetic operators, and theoretical considerations. Section 4 describes the experimental setup and benchmark problems, and discusses the results of comparing our approach with baseline GP. Section 5 provides further discussion on the strengths and limitations of the approach. Finally, Section 6 concludes the paper and highlights future research directions.

2. Related Work

2.1. Symbolic Regression and Genetic Programming

Symbolic regression is a classic application domain for GP. The goal is to automatically discover a mathematical expression that fits a given dataset, typically by minimizing an error metric between the model's outputs and the target outputs [10, 11]. Koza's early work [8] demonstrated symbolic regression on simple polynomial fitting tasks, and since then GP-based symbolic regression has been used in increasingly sophisticated problems [14].

However, it is well-known that GP can be resource-intensive. The evolutionary search may evaluate thousands to millions of candidate expressions over many data points and generations, which can lead to very long execution times [12]. Researchers have explored techniques to improve GP efficiency [18]. Fitness case selection strategies try to reduce the number of evaluation points by sampling or reusing partial evaluations. Another line of work focuses on reducing redundant computations in GP individuals. Some works introduced the idea of representing the population of GP programs as a directed acyclic graph (DAG) to share common sub-expressions among individuals, thereby caching and reusing their computed values. While caching can significantly reduce computation, it requires memory overhead and its benefit diminishes if individuals are highly distinct or if the population is small.

2.2. Parallel and Accelerated GP

Because GP fitness evaluations are independent for each individual and each data point, GP can be formulated as a parallelism problem. Early implementations of parallel GP used networked workstations or multi-core CPUs to distribute individuals or fitness cases across processors [15, 7]. With the advent of GPUs and parallel computing, there has been increasing interest in accelerating GP using these architectures [3]. GPUs can execute thousands of threads in parallel and are particularly efficient for vectorized operations on large arrays of data (a scenario very much aligned with GP fitness evaluation).

Chitty [4, 5] demonstrated substantial performance gains using GPU and data-parallel approaches for GP. A detailed review by Arenas *et al.* [1] surveys numerous efforts in GPU-based evolutionary computation, concluding that the throughput-oriented design of GPUs is well-suited to GP workloads.

More recently, researchers have integrated GP with high-level parallel computing frameworks. Staats *et al.* [17] developed *KarooGP*, which uses TensorFlow (a popular machine learning library) to perform GP operations on tensors. By utilizing TensorFlow's vectorized ops on both CPU and GPU, they reported speedups of up to 875× on certain benchmark problems. Baeta *et al.* [2] introduced TenGP which also employs TensorFlow (in eager execution mode) to vectorize GP evaluations. Their results demonstrated up to two orders of magnitude improvement in execution speed compared to a conventional iterative GP, especially when running on a GPU. They also extended their engine to support image-based evolutionary art, by providing operators that handle 2D image data and color channels.

Our approach falls in the same vein as these systems, embracing data parallelism and GPU acceleration for GP. The key differences in our our approach framework lie in the emphasis on structured domain support and the use of tensor operations abstracted in a way that is not tied to a specific backend. We also formalize the tensor-based GP evaluation in a mathematical manner, which helps in reasoning about its complexity and capabilities. In the next section, we describe the methodology of our approach in detail.

2.3. Contribution over the State-of-the-Art

Representing GP individuals as combinations of tensor operations allows for data vectorization and GPU support, making it possible to evaluate entire datasets at once and substantially reduce computation time. The genetic operators and evaluation steps are formalized in tensor notation, with an analysis of expected speed improvements. Specialized operators are also included for structured data, such as multi-dimensional grids, which extends the method to problems like image-based modeling.

3. Methodology

In this section, we formalize the GP approach for symbolic regression in the context of tensor-based evaluation. We first define the symbolic regression problem and the representation of individuals (candidate solutions). Then we describe how expressions are evaluated using tensor operations. Finally, we detail the genetic operators (selection, crossover, mutation) and analyze the computational complexity of the evolutionary loop under this framework.

3.1. Problem Formulation

Symbolic regression aims to find a mathematical expression that approximates an unknown target function given a set of sample points. Formally, let $D = \{(x^{(i)}, y^{(i)}) \mid i = 1, 2, \ldots, M\}$ be a dataset of M samples, where each $x^{(i)} = (x_1^{(i)}, x_2^{(i)}, \ldots, x_n^{(i)}) \in \mathcal{R}^n$ is an n-dimensional input (feature vector or coordinates in a structured domain), and $y^{(i)} \in \mathcal{R}$ (or $\mathcal{R}^p$ for multi-output problems) is the target output. The goal is to find an expression (program) $f : \mathcal{R}^n \to \mathcal{R}$ that minimizes a given error metric with respect to the data in D. Typically, the fitness (error) of a candidate expression f is measured by the mean squared error (MSE):

$$E(f) = \frac{1}{M} \sum_{i=1}^{M} \left(f(x^{(i)}) - y^{(i)}\right)^2, \tag{1}$$

though other metrics (mean absolute error, correlation, etc.) can be used depending on the application.

GP approaches this optimization problem by evolving a population of candidate expressions using evolutionary operators. We define a *primitive set* of symbols from which expressions can be constructed. Let $\mathcal{F} = \{f_1, f_2, \ldots, f_k\}$ be a set of primitive functions (operations) and $\mathcal{T} = \{t_1, t_2, \ldots, t_\ell\}$ a set of terminals. Terminals typically include the input variables (e.g., $X_1, X_2, \ldots, X_n$ corresponding to coordinates or features) and may include constant values (ephemeral random constants). Each primitive $f_j \in \mathcal{F}$ has an associated arity . For example, $\{+, -, \times, \div\}$ are binary arithmetic operators (arity 2), sin and exp are unary (arity 1), and a conditional operator $\mathsf{if}(\cdot, \cdot, \cdot)$ would have arity 3. An *individual* in GP is a composition of these primitives and terminals, often represented as a syntax tree where internal nodes are functions and leaves are terminals. Such a tree encodes a mathematical expression.

3.2. Tensor-Based Representation and Evaluation

Our approach evaluates GP individuals using tensor operations to handle multiple data points at once. We treat the collection of input values for all fitness cases as a set of tensors (or a single multi-dimensional tensor). For example, consider evaluating a candidate expression f on the entire dataset D. We can arrange all inputs $x^{(i)}$ into an $M \times n$ matrix (or into an n-dimensional tensor of shape $[M_1 \times M_2 \times \cdots]$ if the inputs form a structured grid). Without loss of generality, assume a vector representation of inputs: $\mathbf{X} = [x^{(1)}, x^{(2)}, \ldots, x^{(M)}]^\top$ (an $M \times n$ matrix). Applying the expression f to $\mathbf{X}$ yields an output vector $\mathbf{y}_f = [f(x^{(1)}), \ldots, f(x^{(M)})]^\top$ of length M. In a standard GP implementation, one would loop over $i = 1$ to M and compute $f(x^{(i)})$ individually. In our approach, by contrast, we implement each primitive operation in f as an element-wise (or tensor) operation that acts on an entire vector (or tensor) of values at once.

For example, if $f(x_1, x_2) = \sin(x_1) + x_2^2$, and we have vectors (of length M) X_1 and X_2 holding all values of input features x_1 and x_2 for the dataset, then we compute the output in a vectorized manner:

$$Y_f = \sin(X_1) + (X_2)^2 \,,$$

where sin and the squaring operation are understood to apply element-wise on each element of X_1 or X_2. The result Y_f is a vector of length M containing $f(x^{(i)})$ for all i. Each primitive $f_j \in \mathcal{F}$ is implemented as a function that can take one or more tensors as inputs and produce a tensor output of the same shape, performing the operation on each entry. Constants are treated as scalar tensors that are implicitly expanded to the needed shape.

Interpreting the entire evaluation of a GP individual as a composition of tensor operations allows enabling the use of highly optimized linear algebra routines. If executed on a GPU, these operations can utilize thousands of cores to compute results for all data points concurrently. This means the cost to evaluate an individual f is largely determined by the number of nodes in its expression tree, rather than the number of fitness cases M. In an ideal scenario, evaluating a program of size K on M points via vectorized operations would take on the order of $O(K)$ time, versus $O(K \times M)$ for a naive scalar evaluation. In practice, there is some overhead for launching GPU kernels or vector operations, so the benefit manifests once M is sufficiently large to amortize those costs.

A crucial aspect of our approach is that it supports not only flat vector inputs but also structured multi-dimensional inputs. Consider a symbolic regression problem defined over a 2D spatial domain (such as evolving an image filter or fitting a 2-variable function $f(x, y)$ over a grid). In our approach, we can treat the domain as, say, a $W \times H$ grid of points. We would represent the two coordinate inputs as two $W \times H$ matrices X and Y (for the x and y coordinates respectively), and each primitive operation will act on $W \times H$ tensors. The output of an individual will be a $W \times H$ tensor (which can be visualized as an image). This generalization means our GP individuals can effectively evolve programs that produce, for instance, entire images as output (useful in evolutionary art or image regression tasks). The underlying tensor operations (e.g., addition, multiplication, sine) naturally operate over all pixels at once on the GPU. We also introduce domain-specific primitives for structured data, such as convolution-like operators or the *warp* operator (which remaps coordinates of an image) as described by Baeta *et al.* [2]. The warp operator, for example, takes an image (tensor) and a set of offset tensors and produces a transformed image by shifting pixels according to those offsets. All such operations are implemented as tensor transformations, maintaining compatibility with the overall vectorized evaluation scheme.

Algorithm 1 Evolutionary Loop for Symbolic Regression

Input: Population size N, maximum generations G, function set $\mathcal{F}$, terminal set $\mathcal{T}$, other GP parameters.
Output: Best evolved individual f^*.
Initialize population $P \leftarrow \{f_1, f_2, \ldots, f_N\}$ with random expressions (trees) composed of $\mathcal{F}$ and $\mathcal{T}$.
for $t = 1$ to G **do**
  for each individual $f \in P$ **do**
    Compute the output tensor $\mathbf{y}_f = f(\mathbf{X})$ for all fitness cases (vectorized evaluation).
    Compute fitness $E(f)$ using (1) or other measure by comparing $\mathbf{y}_f$ with target outputs $\mathbf{y}_{target}$.
  end for
  Select a set P_{parents} of individuals from P with probabilities biased by fitness (e.g., tournament or roulette wheel selection).
  Create an empty set $P_{\text{offspring}}$.
  while $|P_{\text{offspring}}| < N$ **do**
    Sample parents from P_{parents} (with replacement) for reproduction:
    With probability p_c: perform **crossover** on two parents to produce two offspring by swapping random subtrees between the parent trees.
    With probability p_m: perform **mutation** on one parent by replacing a random subtree with a new randomly generated subtree.
    Otherwise (with probability $p_r = 1 - p_c - p_m$): **reproduce** one parent (copy to offspring).
    Add the resulting offspring individual(s) to $P_{\text{offspring}}$.
  end while
  $P \leftarrow P_{\text{offspring}}$ (the new generation replaces the old).
end for
return the best individual f^* found (according to lowest error).

3.3. Genetic Operators

We employ a standard evolutionary algorithm loop for GP, adapted to the tensor-based evaluation context. Pseudocode for the evolutionary process is given in Algorithm 1. Each generation involves evaluating all individuals, then applying selection and genetic variation operators to produce a new population.

We use conventional genetic operators as described above. Selection is often implemented as tournament selection: e.g., to pick one parent, choose k individuals at random from the population and select the best among those (with smaller $E(f)$ meaning better fitness in a minimization context). This is repeated to fill the P_{parents} mating pool. Crossover is typically binary: we choose two parent trees and a random crossover point in each (a node), then swap the subtrees at those points between the two parents, yielding two new offspring programs. Mutation is typically unary: we select one individual and a random node within it, and replace

the subtree at that node with a newly generated random subtree (using the same primitives $\mathcal{F}, \mathcal{T}$). The probabilities p_c, p_m, p_r govern the rates of crossover, mutation, and reproduction (elitism or straight copying), respectively. These are parameters of the GP algorithm (e.g., $p_c = 0.8$, $p_m = 0.1$, $p_r = 0.1$ are common settings).

One consideration is that certain individuals might produce invalid values for some inputs (for instance, division by zero or square root of a negative number). In standard GP, protection mechanisms are used (e.g., defining $1/0 = 1$ or limiting domains). We adopt similar protective measures at the tensor operation level. Many tensor libraries allow filtering or masking invalid operations. We implement safe versions of operators; for example, the division operator is defined to return a default large value or 1 when division by zero is attempted, and we include a clip operator in $\mathcal{F}$ that can bound values to safe ranges if needed for a list of such operator definitions).

Another consideration is controlling program bloat (the tendency of GP trees to grow in size without improving fitness). Our approach includes bloat control by imposing a maximum tree depth/size and optionally using dynamic limits that increase only if needed (as in Koza's method or lexicographic parsimony pressure). These strategies are orthogonal to our main focus and similar to those in traditional GP, so they are not detailed here.

3.4. Complexity Analysis

To understand the performance characteristics of our approach, let us consider the time complexity per generation. Let N be the population size and M the number of fitness cases (data points). In a conventional GP implementation, evaluating all individuals is $O(N \times K \times M)$, where K is the average number of nodes (size) per individual. This tends to dominate the runtime. If we assume the tensor operations can be executed in parallel, the evaluation of one individual of size K on M points takes $O(K + \alpha(M))$, where $\alpha(M)$ is the overhead cost associated with handling M data (memory transfers, kernel launch, etc.). For large M, $\alpha(M)$ grows much slower than M itself (often sub-linear or a modest linear factor that is mitigated by hardware parallelism). Thus, the overall evaluation cost for the population is approximately $O(N \times K + \alpha(M))$. In practice, this means the runtime scales almost linearly with population size and program size, rather than with the product $N \times M$.

Memory complexity is another concern: representing the entire domain as a tensor of size M can be memory-intensive for very large M. For example, if M is in the millions (e.g., a 2048×2048 image has over 4 million pixels), storing intermediate results for each node in a deep tree could consume a lot of GPU memory. Our solution mitigates this by relying on the computational graph of the underlying tensor library: intermediate values can be freed or not stored if not needed, especially in an eager execution mode. In graph execution mode (as in static TensorFlow), common sub-expressions might be automatically reused, which is akin to subtree

caching at the framework level. This can save time at the cost of memory to store those results. We have to balance these trade-offs: for most experiments in this paper, we found memory to be sufficient and the speed benefits to far outweigh the overhead, but for extremely large domains, one might need to distribute data or use batch-wise evaluation.

Expressiveness of our representation remains equivalent to standard tree-based GP. Any function that can be evolved via traditional GP can also be represented. In fact, by including certain high-level primitives, one could argue our search space is even richer for certain structured tasks. However, the inclusion of such operators also means the search space is larger, which can make evolution challenging if not properly guided. The ability to handle structured data as tensors means our solution can evolve solutions for problems that would be awkward to encode in a conventional GP (which might require flattening an image into a long vector, for instance).

From a scalability perspective, if we denote by $T_{\text{baseline}}(M)$ the time to run a generation on M data points in a baseline GP and $T_{\text{tensor}}(M)$ the time in our approach, we typically observe:

$$T_{\text{tensor}}(M) \approx T_{\text{baseline}}(M_0) + c \times T_{\text{baseline}}\Big(\frac{M}{p}\Big),$$

where p is the degree of parallelism (e.g., number of GPU cores) and M_0 is a threshold beyond which GPU overhead is amortized. In simpler terms, for small M, the baseline might be faster or similar (due to GPU overhead), but beyond a certain point, our approach is dramatically faster, essentially by a factor proportional to the parallelism.

4. Experiments

We conducted a series of experiments to evaluate the performance in comparison to a standard GP implementation on symbolic regression problems. We sought to measure both the quality of solutions (accuracy of the evolved expressions) and the computational cost (runtime) across different scenarios. All experiments were run on a workstation with an 8-core CPU and an NVIDIA RTX 3080 GPU. Our approach was implemented in Python using `TensorFlow` as the tensor computation backend. For the baseline GP, we used the DEAP evolutionary computing library [6] with a tree-based GP setup, executing single-threaded on the CPU for fairness (to represent a typical GP without parallel acceleration). Both systems were configured as similarly as possible in terms of genetic operator probabilities and termination criteria.

4.1. Benchmark Problems and Settings

We selected three representative symbolic regression benchmark problems of varying complexity:

- **F1: Cubic Polynomial.** A single-variable polynomial target function $f(x) = x^3 + x^2 + x$. This is a classic benchmark, which GP is expected to solve easily. We generate $M = 200$ sample points with x uniformly in $[-5, 5]$ and corresponding y from the cubic function (noise-free).

- **F2: Sine Composite.** A single-variable oscillatory function $f(x) = x + \sin(x)$, (a function that combines polynomial and sinusoidal terms). We sample $M = 200$ points in $[-10, 10]$ for training. This function is slightly more challenging due to the sinusoid.

- **F3: Two-dimensional Pagie Function.** A well-known 2D symbolic regression benchmark. The target function is

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

  which produces a smooth bivariate surface. We evaluate this on a 21×21 grid for $x, y \in \{-5, -4.5, \ldots, 5\}$ (so $M = 441$ points), following the literature. This problem tests our approach's ability to handle structured domain (the input can be viewed as a 21×21 tensor of points).

For each problem, we ran 30 independent GP runs with both our approach and the baseline GP. Each run used a population size $N = 500$ and evolved for up to $G = 50$ generations (or stopped early if a perfect solution with $E(f) = 0$ error was found). The function set $\mathcal{F}$ included $\{+, -, \times, \div, \sin, \cos, \text{pow}\}$ (where pow was a protected power operator allowing square, cube, etc.). The terminal set $\mathcal{T}$ included the relevant variables (x or x, y) and ephemeral constants in $[-5, 5]$. Crossover probability was $p_c = 0.8$, mutation $p_m = 0.1$, and reproduction $p_r = 0.1$. Tournament selection of size 5 was used. These parameters were held constant across our approach and baseline for comparability.

The primary performance metrics recorded were: (1) *Best fitness (error)* achieved at the end of the run (we report the median across 30 runs), (2) *Success rate*, defined as the fraction of runs that found an exact solution (zero error on training data), and (3) *Average runtime* per run.

4.2. Results

Table 1 summarizes the results for the three benchmark problems. We observe that our approach matches the baseline GP in terms of solution quality on all problems. Both methods frequently find the exact target expressions for the simpler problems F1 and F2 (100% success). For the harder 2D Pagie function (F3), neither method always finds the perfect formula within 50 generations, but both achieve it in the majority of runs (success rate around 97% for our approach and 97% for baseline in our trials). The slight differences in final error or success

Table 1: Symbolic Regression Performance and Runtime Comparison. Results are averaged over 30 independent runs for each method. Success rate is the percentage of runs that found the exact target expression. Runtime is the mean time per run.

Problem	**Success Rate**		**Final Error (MSE)**		**Runtime (s)**
	Baseline	Our approach	Baseline	Our approach	(Our approach)
F1: Cubic	100%	100%	0.0	0.0	1.5 (vs 15.2 baseline)
F2: Sine	100%	100%	0.0	0.0	1.8 (vs 18.9 baseline)
F3: Pagie 2D	97%	97%	1.2×10^{-4}	1.1×10^{-4}	2.4 (vs 26.3 baseline)

percentages are not statistically significant, indicating that the search capability of GP is not impaired by our tensor-based evaluation. This is expected, as the evolutionary algorithm and search operators are essentially the same; our approach simply evaluates individuals faster.

In terms of runtime, our approach provides a clear benefit. Even for these relatively modest dataset sizes (hundreds of points), our approach is roughly an order of magnitude faster than the baseline. The baseline GP (in Python/DEAP) took on the order of tens of seconds for each run, whereas our approach completed runs in a few seconds. The speedup factor ranged from about $10\times$ to $11\times$ in these tests. We expect even larger gains for problems with more data points. In fact, to illustrate scalability, we performed an additional experiment (not shown in the table) where we increased the number of sample points for the F2 problem from 200 to $20,000$ (continuously sampled in $[-10, 10]$). The baseline GP could not practically complete 50 generations on this many points (we estimated it would take several hours), whereas our approach handled it in under a minute, finding an accurate solution. This dramatic difference underscores how the complexity of GP grows with data size and how our approach can make previously infeasible runs feasible.

To get insight into evolutionary dynamics, we also examined the convergence behavior of GP in both systems. Both baseline GP and our approach tend to converge in a similar number of generations for these problems. For example, in problem F1 and F2, a correct solution often emerged by generation 10-20 in both systems. The distribution of best-of-generation fitness values over time was virtually identical. The main difference is that our approach reaches those generations faster in wall-clock time. In some runs, the faster turnaround of our approach allowed us to experiment with larger population sizes or more generations within the same time budget, which indeed improved success rates further for the harder problem (F3). This suggests an additional advantage: by reducing the computational cost per individual, we can allocate more computational resources to potentially find better solutions without exceeding time limits.

4.3. Ablation and Additional Tests

We performed a few ablation studies to verify that each component of our approach contributes to its performance. First, we tested our approach in CPU-only mode (using vectorized NumPy computations without GPU). This still gave a speedup (about $3\times$) over the baseline on large-M problems due to the efficient C-based array operations in NumPy, although it was slower than using the GPU. Second, we disabled the special structured operators and found it did not affect the tasks we chose (since none of the target functions required those operators), but in an image evolution task (not detailed here), including such operators significantly improved the quality of evolved image filters, demonstrating the value of domain-specific tensor primitives. Lastly, we analyzed overhead: for very small M (e.g., $M = 10$ points), the baseline GP actually ran faster than our approach (the overhead of setting up GPU kernels outweighed any parallel advantage).

5. Discussion

The experimental results show that our method can significantly speed up GP while maintaining solution quality. Using parallelism, the evaluation phase, which is often the main bottleneck, becomes much more manageable even with large datasets.

This makes it practical to apply GP to datasets containing thousands or millions of records, or to use it in fields like image or time-series analysis where higher resolution is needed. Memory usage is one possible constraint, as working with very large tensors requires substantial hardware resources. There may also be diminishing returns if the GPU's capacity is exceeded. Modern GPUs, with their large memory capacities, can manage moderately sized images and large tabular data. Distributing computations across several GPUs or machines is also possible, as tensor operations are well-suited for distribution. With these strategies, GP can be applied to large-scale problems that have traditionally been too slow to tackle. Faster evaluation also allows for more frequent runs in settings like hyperparameter optimization.

The method treats structured inputs natively. This opens up use cases like evolving partial differential equations over grids, or designing controllers for matrix- or tensor-valued inputs. The approach makes it possible to include operations such as convolution or warp, showing that GP can work on complex data transformations beyond just scalar functions. A future direction involves using multi-tree models to produce vector or multi-output functions, where each output could be a tensor. Since the primitives are implemented in popular machine learning libraries, it is also feasible to use gradient information for local search or analysis of programs, even if GP itself does not use gradients directly.

Accelerating GP makes it more competitive with other modeling approaches in terms of runtime. GP is sometimes overlooked for large-scale problems because it runs slower than methods like deep neural networks or gradient boosting, which make more efficient use of hardware. Our approach reduces this gap. GP offers other benefits such as interpretability and flexibility, and its scaling behavior differs from other methods. Our experiments show that, with proper use of current hardware, GP can process reasonably large problems efficiently.

There are some trade-offs. The overhead from setting up tensor computations can result in slower performance on small datasets or populations, as we observed. Not every operation can be vectorized easily; some tasks with domain-specific logic may resist parallelization. We addressed this by focusing on operations that can be applied element-wise or to whole tensors. Debugging or analyzing computations is less transparent than with a simple interpreter, although tools exist to help trace tensor operations. Floating-point calculations on the GPU may also introduce small numerical differences, which could influence selection outcomes in rare cases.

The current method can be extended in several ways. Multi-objective optimization is a promising area, with the possibility of trading off accuracy and model size directly in the fitness function. Multiple objectives can be evaluated in parallel, making this practical. Another idea is to integrate GP with neural networks, so that tensor-based GP acts as a component within larger architectures. Since both use the same backend libraries, it is feasible to combine evolutionary and gradient-based methods within a single model. We also plan to make more use of caching. While TensorFlow's graph mode can cache repeated computations, our implementation used eager execution for simplicity. Mixing both approaches could bring efficiency improvements, with caching used when the population has less diversity and eager evaluation applied when the population changes often.

6. Conclusion

We have introduced a GP system based on tensor algebra, designed to efficiently perform symbolic regression on large and structured datasets. GP evaluation has been reformulated as a set of parallel tensor operations, which has enabled modern hardware, particularly GPUs, to accelerate the evolutionary process significantly. We have provided a mathematical description of the method and have shown how standard GP components are implemented. Experiments on established benchmarks have demonstrated that the method reaches the same solution quality as standard GP but runs several times faster on substantial datasets.

Our research shows how to make GP more scalable and practical for demanding modeling tasks. As hardware with greater parallel processing power has become more widely available, our solution aligns GP with current computing practices. Potential improvements, including dis-

tributed computation, integration with automatic differentiation, or domain-specific primitives, may extend these capabilities even further. The results show that combining evolutionary algorithms with high-performance computing can significantly expand the practical reach of symbolic regression and GP in research and applied fields.

Acknowledgments

The research reported in this paper has been funded by the Federal Ministry for Climate Action, Environment, Energy, Mobility, Innovation, and Technology (BMK), the Federal Ministry for Digital and Economic Affairs (BMDW), and the State of Upper Austria in the frame of SCCH, a center in the COMET - Competence Centers for Excellent Technologies Programme managed by Austrian Research Promotion Agency FFG.

References

[1] Arenas, M., Romero, G., Mora, A., Castillo, P., & Merelo, J. (2012). Gpu parallel computation in bioinspired algorithms: a review. *Advances in Intelligent Modelling and Simulation: Artificial Intelligence-Based Models and Techniques in Scalable Computing*, (pp. 113–134).

[2] Baeta, F., Correia, J., Martins, T., & Machado, P. (2021). Tensorgp–genetic programming engine in tensorflow. In *International Conference on the Applications of Evolutionary Computation (Part of EvoStar)* (pp. 763–778). Springer.

[3] Cano, A., Zafra, A., & Ventura, S. (2012). Speeding up the evaluation phase of gp classification algorithms on gpus. *Soft Computing*, *16*, 187–202.

[4] Chitty, D. M. (2007). A data parallel approach to genetic programming using programmable graphics hardware. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation* (pp. 1566–1573).

[5] Chitty, D. M. (2017). Faster gpu-based genetic programming using a two-dimensional stack. *Soft Computing*, *21*, 3859–3878.

[6] Fortin, F.-A., De Rainville, F.-M., Gardner, M.-A. G., Parizeau, M., & Gagné, C. (2012). Deap: Evolutionary algorithms made easy. *The Journal of Machine Learning Research*, *13*, 2171–2175.

[7] Harada, T., & Alba, E. (2020). Parallel genetic algorithms: a useful survey. *ACM Computing Surveys (CSUR)*, *53*, 1–39.

[8] Koza, J. R. (1994). Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, *4*, 87–112.

[9] Martinez-Gil, J. (2019). Semantic similarity aggregators for very short textual expressions: a case study on landmarks and points of interest. *J. Intell. Inf. Syst.*, *53*, 361–380.

[10] Martinez-Gil, J., Alba, E., & Aldana-Montes, J. F. (2008). Optimizing ontology alignments by using genetic algorithms. In C. Guéret, P. Hitzler, & S. Schlobach (Eds.), *Proceedings of the First International Workshop on Nature Inspired Reasoning for the Semantic Web, Karlsruhe, Germany, October 27, 2008*. CEUR-WS.org volume 419 of *CEUR Workshop Proceedings*.

[11] Martinez-Gil, J., & Aldana-Montes, J. F. (2011). Evaluation of two heuristic approaches to solve the ontology meta-matching problem. *Knowl. Inf. Syst.*, *26*, 225–247.

[12] Martinez-Gil, J., & Aldana-Montes, J. F. (2012). An overview of current ontology meta-matching solutions. *Knowl. Eng. Rev.*, *27*, 393–412.

[13] Martinez-Gil, J., & Chaves-Gonzalez, J. M. (2020). A novel method based on symbolic regression for interpretable semantic similarity measurement. *Expert Syst. Appl.*, *160*, 113663.

[14] Martinez-Gil, J., Yin, S., Küng, J., & Morvan, F. (2021). Matching large biomedical ontologies using symbolic regression. In E. Pardede, M. Indrawan-Santiago, P. D. Haghighi, M. Steinbauer, I. Khalil, & G. Kotsis (Eds.), *iiWAS2021: The 23rd International Conference on Information Integration and Web Intelligence, Linz, Austria, 29 November 2021 - 1 December 2021* (pp. 162–167). ACM.

[15] Poli, R. (1996). *Parallel distributed genetic programming*. University of Birmingham, Cognitive Science Research Centre Birmingham, UK.

[16] Schmidt, M., & Lipson, H. (2009). Distilling free-form natural laws from experimental data. *science*, *324*, 81–85.

[17] Staats, K., Pantridge, E., Cavaglia, M., Milovanov, I., & Aniyan, A. (2017). Tensorflow enabled genetic programming. In *Proceedings of the genetic and evolutionary computation conference companion* (pp. 1872–1879).

[18] Zhang, M., Wong, P., & Qian, D. (2006). Online program simplification in genetic programming. In *Asia-Pacific Conference on Simulated Evolution and Learning* (pp. 592–600). Springer.