Backend-agnostic Tree Evaluation for Genetic Programming

Overview of Mathematical Libraries for Symbolic Regression and their Energy Efficiency

Bogdan Burlacu
bogdan.burlacu@fh-ooe.at
University of Applied Sciences Upper Austria
Hagenberg, Upper Austria, Austria

ABSTRACT

The explicit vectorization of the mathematical operations required for fitness calculation can dramatically increase the efficiency of tree-based genetic programming for symbolic regression. In this paper, we introduce a modern software design for the seamless integration of vectorized math libraries with tree evaluation, and we benchmark each library in terms of runtime, solution quality and energy efficiency. The latter, in particular, is an aspect of increasing concern given the growing carbon footprint of AI. With this in mind, we introduce metrics for measuring the energy usage and power draw of the evolutionary algorithm. Our results show that an optimized math backend can decrease energy usage by as much as 35% (with a proportional decrease in runtime) without any negative effects in the quality of solutions.

CCS CONCEPTS

• **Software and its engineering** → *Software prototyping*; **Software performance**; • **Computing methodologies** → **Genetic programming**; *Supervised learning by regression.*

KEYWORDS

energy efficiency, genetic programming, symbolic regression

ACM Reference Format:
Bogdan Burlacu. 2024. Backend-agnostic Tree Evaluation for Genetic Programming: Overview of Mathematical Libraries for Symbolic Regression and their Energy Efficiency. In *Proceedings of The Genetic and Evolutionary Computation Conference 2024 (GECCO '24).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3638530.3664161

1 INTRODUCTION

Genetic Programming (GP) is a computationally-intensive approach for solving problems using the principles of natural evolution. Thanks to the flexibility of its genotype representation as variable-length computer programs, GP has found success application in a variety of different domains [29].

A particularly relevant task solved by GP is Symbolic Regression (SR), where the genotypes are represented by abstract syntax trees (AST) that encode mathematical expressions. Compared to other regression methods, one of the important advantages of SR is that it produces models that can be interpreted by humans.

A SR system starts with a given labeled set of data and coordinates the evolution of a population of individuals or *programs* who compete for survival according to the principles of natural selection. The programs are constructed by combining a limited set of mathematical primitives to produce many different tree-encoded mathematical expressions. A key ingredient in this process is the *fitness function*, which helps quantify how successful each individual is in solving the task, thus providing the basis for selection. Once fitness is computed via a domain-appropriate metric, individuals produce offspring via a recombination process (typically involving crossover and mutation) in which they participate with a probability proportional to their fitness. Finally, the offspring replace the parent population according to some predetermined replacement scheme and the process continues with a new iteration.

Depending on the size of the dataset, the size of the population and the complexity of the individuals, the computation of fitness may usually take a large proportion of the algorithm runtime. At the same time, due to the stochastic nature of GP, in order to obtain a reliable estimate of the empirical error rate, the GP algorithm must be run many times for each dataset and/or set of hyperparameters.

Approaches to mitigate the runtime attempt to take advantage of the "embarrassingly parallel" structure of the algorithm. We distinguish here between two complementary cases:

Thread parallelism. Since in a GP system each individual can be processed independently from the others, its fitness evaluation can take place in parallel, on a dedicated thread.

Data parallelism. As each tree individual will typically be evaluated on a large collection of data points, *vectorization* can be employed in order to take advantage of data parallelism on modern SIMD (single-instruction, multiple-data) processor architectures. Evidence suggests vectorization yields better energy efficiency than parallelism [2, 6].

In this paper, we focus on the data-parallel approach and propose an efficient and easy to extend tree interpreter that can take advantage of state-of-the-art vectorized math libraries. This approach is based on the Operon C++ library which already provides an efficient execution environment.

The paper is organized as follows: Section 2 discusses existing approaches for improving the efficiency of GP systems. Section 3 introduces a generic tree interpreter capable of interfacing with different C++ libraries for matrix and vector operations and gives examples of its usage. Section 4 presents empirical performance measurements across different mathematical backends and Section 5 is dedicated to conclusions.

ACM ISBN 979-8-4007-0495-6/24/07
https://doi.org/10.1145/3638530.3664161

2 EXISTING APPROACHES

A survey of existing literature reveals that GP performance has always been a matter of concern and SIMD evaluation represents a well-known approach for improving fitness function efficiency [15]. Modern frameworks outlined below make use of specialized math libraries on the CPU, specialized hardware (such as GPUs), or modern language features (such as just-in-time compilation) to ensure efficient evaluation. Each of these approaches comes with its own set of advantages and disadvantages:

CPU Evaluation. Perhaps the more versatile approach, as CPUs are suitable for executing a wide range of workloads. GP evaluation workflows on the CPU are implemented following standard programming paradigms within general-purpose programming languages. Data parallelism is possible using vector instruction sets, albeit on a much reduced scale compared to graphical processing units (GPUs). Multi-core parallelism is straightforward in GP populations.

GPU Evaluation. This is a data-centric approach which aims to take advantage of the massively-parallel processing capabilities of GPUs. Only parts of a GP system (typically the evaluation function) can be offloaded to the GPU, making it necessary to copy data back and forth between CPU and GPU. This approach is generally less accessible as it requires familiarity with special-purpose APIs (e.g., CUDA, OpenCL) and programming techniques. GP individuals have to be converted to corresponding GPU representations to be evaluated. Speedups are possible when these overheads (translation effort, data marshalling) become negligible relative to the amount of work, which typically involves large datasets (see e.g. [3, 26]).

Hybrid Approaches. Hybrid approaches may use both CPU and GPU to improve performance. Notably, bytecode-compatible languages (e.g. Python, Java, Julia, etc.) are able to compile various code constructs into optimized native code as they are encountered in the program. GP systems like PySR [7] (see below) use just-in-time (JIT) compilation to speed up evaluation of certain operator combinations. In certain cases, a JIT compiler can make use of runtime information to produce faster machine code, however, the amount of optimizations is limited by time constraints. Overall, bytecode translation and compilation overheads make JIT approaches advantageous when the compiled code is executed frequently and/or the workloads are large enough.

Considering the strength and weaknesses of the enumerated approaches, we argue that CPU-based solutions, in particular those based on compiled languages, still offer the best compromise between general-purpose performance, versatility and easy of use. We address these important characteristics in this paper, proposing a solution to improve versatility by supporting interchangeable mathematical backends for GP while maintaining a high level of runtime performance.

Symbolic Regression Libraries

GP-GOMEA. The Gene-pool Optimal Mixing Evolutionary Algorithm (GOMEA) [30] uses a tree-based representation where each individual is encoded as a perfect n-ary tree and leafs are all at maximum depth h (therefore, all trees in the population have the same number of nodes). GOMEA performs an additional model-learning phase in each GP generation, where the *linkage* between parts of the n-ary tree genotype is modeled. This linkage information is then used to propagate genetic patterns during recombination.

GOMEA has two C++ implementations, the canonical implementation[1] uses Armadillo [25] as a mathematical backend while the newer rewrite[2] employs the Eigen [11] library. These libraries offer vectorized evaluation of common mathematical functions and can provide significant runtime benefits.

GPQuick. GPQUICK [16] is a C++ library that uses a linear representation where each tree is encoded by a prefix list of symbols (with the root note at the start). This encoding is then evaluated recursively using AVX512 instructions which can process up to 16 floating point values simultaneously. In GPQUICK, individuals are limited to a length of 48 (as a multiple of the AVX512 bandwidth) and the algorithm only works with the basic arithmetic primitive set $(+, -, \div, \times)$.

Operon. The Operon [4] C++ library for symbolic regression has established itself as one of the fastest GP implementations [5] and has also been used as a performance baseline [8]. The framework uses a linear encoding scheme where each tree is represented as a postfix list of nodes, such that leaf nodes are evaluated before their parents during a natural iteration over the list. A specialized dispatch table class is used to map function tree nodes to their corresponding operation, which is resolved at runtime. This approach allows user-defined functions to be registered in the dispatch table and subsequently used as GP primitives. The operations are then evaluated with the Eigen [11] library in data-batches. The batched evaluation serves two purposes: 1) it reduces memory requirements when evaluating large datasets (as the memory is reused and the total usage is limited by $S \cdot$ tree nodes), and 2) it reduces overheads related to control flow within the interpreter.

PySR. PySR [7] is a Python frontend to SymbolicRegression.jl, a pure-Julia library for Symbolic Regression which achieves high evaluation speeds by just-in-time (JIT) compilation. This enables every combination of operators (up to a depth of two operators) to be fused together into a compiled SIMD kernel (e.g. fused multiply-add). The advantage of this approach is that not only preexisting operators in the library can be fused into SIMD kernels but user defined operators as well. PySR represents individuals as expression trees and implements a multi-population GP model with additional heuristics such as simulated annealing-based approach for rejecting unsuccessful mutations, tree simplification and optimization of tree coefficients.

GPU-based Frameworks. These approaches offload fitness computation to the Graphical Processing Unit (GPU) in order to take advantage of their massively-parallel architecture. The GP algorithm runs on the CPU where the individuals are evolved and during evaluation, these individuals are converted into SIMD kernels to be executed on the GPU. For instance, KarooGP [28] and TensorGP [3] use TensorFlow [1] as a computational backend while the GP system itself is implemented in Python. At evaluation time individuals are converged to directed acyclic graphs using the TensorFlow API

[1] https://github.com/marcovirgolin/GP-GOMEA
[2] https://github.com/marcovirgolin/gpg

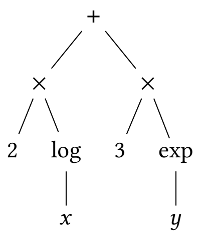


Figure 1: GP tree encoding the formula $2\ln(x) + 3\exp(y)$

and executed on the GPU. CUML[26] implements GP in C++. Individuals are stored as a prefix list of symbols which is subsequently converted into CUDA kernels for evaluation.

Summing up, modern GP approaches make use of vectorization and other speedup techniques in order to achieve efficient execution. However, existing implementations can be considered to be highly specialized, leading to a fragmented overall software landscape which leaves little possibility for extensibility and interoperability. In contrast, a backend-agnosting GP framework can be more easily integrated with existing software.

Despite many efforts to improve the runtime efficiency of GP, to our knowledge, so far there have been no efforts to quantify its energy efficiency. The relationship between runtime and energy efficiency is not necessarily linear [20]. Energy efficiency is becoming an increasingly important topic in AI and computational science in general[17, 18, 31].

3 EVALUATOR DESIGN

Since GP trees are dynamic and their subtrees can be exchanged or mutated during recombination, a common approach is to represent these trees as a collection of nodes, resulting in a representation similar to Figure 1. In Figure 1, leaf nodes x and y represent variables mapped to the corresponding feature values in the dataset, while 2 and 3 are ephemeral constants. The remaining function nodes log, exp, ×, ×, + will be executed by a tree interpreter.

Typically, a tree interpreter implements a recursive procedure that evaluates subtree values from the leafs towards the root node. That is, each training observation propagates through the tree via leaf nodes mapped to corresponding data features. However, evaluating data points one by one limits data parallelism and can lead to suboptimal throughput. Among different implementation strategies to address this issue, explicit vectorization is one of the most reliable [9]. Usually, the data is batched and processed by SIMD instructions that apply the same operation simultaneously on multiple data points.

Our approach is based on the Operon framework and uses its tree representation as a postfix array of nodes:

2	x	log	×	3	y	exp	×	+
0	1	2	3	4	5	6	7	8

We first detail how evaluation works in Operon. Operon makes use of a dispatch table class which defines which primitives are associated with each node type, also allowing library users to register their own primitives. In our proposed design, the dispatch table exposes a generic function interface which allows it to be coupled

```cpp
template<NodeType Type, typename T, std::size_t S>
requires Node::IsNary<Type>
static inline void NaryOp(std::vector<Node> const& nodes,
    Backend::View<T, S> data,
    size_t parentIndex, Operon::Range /*unused*/) {
    // calls backend generic function (see Listing 3)
}
```

Listing 1: N-ary operation dispatch to the math backend

$$\begin{tikzcd} \boxed{\text{Tree Individual}} \arrow[r] & \boxed{\text{Tree Interpreter}} \arrow[r] & \boxed{\text{Dispatch Table}} \arrow[r] & \boxed{\text{Math Backend}} \end{tikzcd}$$

Figure 2: Operon tree evaluation workflow

with different primitive implementations. The returned functions represent a thin layer of abstraction which forward the computation to the configured math backend, as illustrated in Figure 2. Note that the backend is configured at compile time and cannot be changed at runtime.

Inside the dispatch table, a dispatching function is used for gluing together function nodes and their leaf node arguments, as illustrated in Listing 1. This function calls into a lower-level generic forwarding function, which then calls the corresponding backend function.

During tree interpretation, the mechanism described above is executed in batches. The data is batched using a static batch size (known at compile time) equal to `S = 512 / sizeof(T)`, where `T` is the arithmetic scalar type used for data representation (either `float` or `double`). Therefore, for a tree composed of N nodes, a block of memory of size $N \cdot S$ will be allocated. This memory will be reused every S rows as the interpreter goes over the training data indices. After completing each batch, the last column of the $N \cdot S$ buffer will contain the tree output which will be copied in the the output buffer.

***Storage format*.** When the math backend is known, for example the Eigen library, the evaluation buffer can be easily allocated directly as an `Eigen::Matrix<T>` data type, but in the general case this memory needs to be owned independently. In order to promote interoperability, we allocate this memory using the C++17 language facility `std::aligned_alloc` that allows the *alignment* of the allocated storage to be specified as well. This is necessary because SIMD data types usually have higher alignment requirements.

However, interacting with a raw pointer to the allocated memory is problematic from a safety perspective and makes it more difficult to preserve semantics in the API, such as the dimensions N and S of the memory block. It is better to wrap this memory within a data object which contains this information and acts as a non-owning view of the data. Such objects known as *multidimensional arrays* are ubiquitous in high-performance computing (HPC). A reference implementation of multidimensional arrays for C++ is available in the `mdspan`[3] library [12], proposed for standardization in C++23[4].

With the help of `mdspan`, we can create a `View` over the allocated data as illustrated in Listing 2. Here, the batch size S for a scalar type T is available as the static variable `BatchSize<T>`.

[3]https://github.com/kokkos/mdspan
[4]https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p0009r18.html

```cpp
template<typename T>
static auto constexpr BatchSize = 512UL / sizeof(T);

template<typename T, std::size_t S = BatchSize<T>>
using View = std::mdspan<T,
  std::extents<int, S, std::dynamic_extent>,
  std::layout_left>;
```

Listing 2: Generic multidimensional array view

```cpp
// n-ary addition
template<typename T, Operon::NodeType N, std::size_t S>
struct Func {
    auto operator()(Backend::View<T, S> view,
    std::integral auto result,
    std::integral auto... args) {
        // apply operation associated with node type N
    }
};
```

Listing 3: Generic n-ary function signature

```cpp
// n-ary addition
template<typename T, std::size_t S>
struct Func<T, Operon::NodeType::Add, S> {
  auto operator()(Backend::View<T, S> view,
  std::integral auto result,
  std::integral auto... args) {
    auto* h = view.data_handle();
    Backend::Add<T, S>(h + result * S, (h + args * S)...);
  }
};
```

Listing 4: Template specialization for addition

The non-owning mdspan `View<T, S>` type (line 4) is defined as a two-dimensional view of scalar type `T`. Its dimensions are given by the `std::extents` object which specifies the index type as `int`, then a fixed first dimension S (number of rows) and a dynamic second dimension (a variable number of tree nodes). The last template parameter `std::layout_left` specifies that the data will be stored in column-major order.

Backend function interface. Based on the memory view defined above, the backend exposes an interface for function objects as expected by Operon's dispatch table, illustrated in Listing 3. Each function takes a `view`, an index to the column where the result of the operation should be stored, and a variadic pack of indices to the corresponding data columns of the child nodes of the current function node.

The functions are templated on the number type `T`, the node type `N` and the batch size `S`, where `T` can be either a scalar (e.g. `float`, `double`), a dual-number or potentially any other user-defined arithmetic type. Explicit template specialization on node type `N` is used to differentiate behavior based on the type of the node to be evaluated.

These functions serve as an abstraction layer to the concrete math library that Operon has been built against. Based on the provided indices, they extract the corresponding pointers to column-data and then forward the call to the library. This is straightforward knowing the storage layout and the fact that the first dimension S which will never change.

```cpp
// n-ary addition
template<typename T, std::size_t S>
auto Add(T* res, auto const*... args) {
  for (auto i = 0UL; i < S; ++i) {
    res[i] = (args[i] + ...);
  }
}
```

Listing 5: Backend n-ary function – plain implementation

```cpp
// type alias to an Eigen::Map used to wrap a C-style buffer
template<typename T, std::size_t S>
using Map = Eigen::Map<std::conditional_t<std::is_const_v<T>,
  Eigen::Array<std::remove_const_t<T>, S, 1> const,
  Eigen::Array<T, S, 1>>>;

// n-ary addition
template<typename T, std::size_t S>
auto Add(T* res, auto const*... args) {
  Map<T, S>(res, S, 1) = (Map<T const, S>(args, S, 1) + ...);
}
```

Listing 6: Backend n-ary function – Eigen implementation

Note that despite the longer call chain, this forwarding layer constitutes a zero-cost abstraction and does not impose any performance overhead. Listing 5 illustrates one possible implementation of the addition primitive whose interface is shown in Listing 3. This function simply iterates over the current batch and performs the addition using a fold expression. If one wanted to use the Eigen library instead, the implementation could look like Listing 6.

Similar implementations can be added for other backends in a straightforward manner. Backend selection (via inclusion of the corresponding header files) is done at compile time using preprocessor directives. The user must specify the desired backend when building the Operon library.

In this paper, we investigate popular backend math libraries and report their performance and energy efficiency.

Backend Libraries. In general, the advantage of C++ template libraries for linear algebra and dense math operations is that by implementing mathematical expressions using overloaded operators it is possible to build an expression tree of the entire computational graph at compile-time and perform certain optimizations such as elimination of temporaries via lazy-evaluation techniques, rewriting parts of the graph or changing loop ordering for more efficient execution. This allows to, on one hand, achieve similar performance levels as hand-crafted assembly, Fortran or C code while, on the other hand, maintain an elegant mathematical syntax[5].

We note that, although tree expressions in SR are dynamic, certain custom primitives implementing more complex chains of operations could still benefit from compile-time optimization of their corresponding computational graph.

A list of popular math libraries investigated in this paper are summarized in Table 1 and briefly described in the following.

Eigen. The Eigen [11] library offers an expressive API and a variety of dense and sparse algebra and optimization algorithms.

[5]It should be noted, however, that actual throughput depends a great deal on the used compiler and its ability to optimize and perform inlining.

Name		Description
Armadillo [25]	https://arma.sourceforge.net/docs.html	C++ template library for linear algebra and scientific computing
Blaze [13]	https://bitbucket.org/blaze-lib/blaze	C++ template library for dense and sparse arithmetic
Eigen [11]	https://gitlab.com/libeigen/eigen	C++ template library for linear algebra, numerical solvers and vectorized math operations
EVE [19]	https://jfalcou.github.io/eve/	C++ type based wrapper around SIMD extensions sets for most current architectures
Fastor [23]	https://github.com/romeric/Fastor	C++ template library for tensor algebra
VDT [22]	https://github.com/dpiparo/vdt	C++ library of vectorized math functions

Table 1: Open-source C++ math libraries usable as a backend for Operon

The library is implemented using the expression templates metaprogramming technique [14], building expression trees at compile time that are evaluated using a cost model of floating point operations. The library performs its own loop unrolling and vectorization.

Armadillo. Armadillo [25] is another very popular C++ algebra library for dense algebra operations, which additionally offers a high-level API very similar to Matlab syntax. It internally employs an expression evaluator which can fuse together various operations for increased efficiency, but does not support vectorization.

Blaze. Blaze [13] is a high-performance C++ library for dense and sparse algebra which uses a novel technique entitled "smart expression templates" which involves smart initialization and usage of optimized compute kernels. Although Blaze does not include vectorized elementary functions, this functionality can be included by compilation against the Sleef library[6] [27].

VDT. VDT [22] (VectoriseD maTh) is a C++ library of single and double precision highly optimized mathematical functions designed to be used in autovectorized loops. The library has a proven track record in LHC (Large Hadron Collider) experiments where it was shown to bring substantial runtime benefits.

EVE. The EVE [19] (Expressive Vector Engine) library exposes SIMD registers via wrapper types which allow developers to write architecture-agnostic code. Although it does not contain array or matrix types, the library offers a collection of optimized mathematical functions for SIMD types and can be used to implement backend functions in a very straightforward manner.

Fastor. Fastor [23] is a tensor contraction framework for the numerical analysis of coupled and multi-physics problems. It features efficient statically-sized tensor constructs and expressions which are optimized via a compile time depth-first search. It's vectorization capabilities are provided by external third-party libraries, of which we use `xsimd` library[7].

4 BENCHMARKS

We benchmark the implemented backends described in Table 1 on a Ryzen 5950X CPU running at 3.4Ghz, on a system with 64Gb of DDR4-3600 memory. Synthetic benchmarks are performed in single-precision using the `nanobench` microbenchmarking library[8]. The C++ code is compiled with the Clang compiler version 18.1.3 with optimizations turned on: `-O3 -march=x86-64-v3`. The following library versions are used:

- Armadillo v12.8.2
- Blaze v3.8.2 with XSimd v12.1.1
- Eigen v3.4
- Eve v2023.02.15
- Fastor v0.6.4 with Sleef v3.6
- Vdt v0.4.4

Source code and detailed benchmarking steps are available online[9]. We investigate three important aspects:

(1) Primitive set efficiency executed by Operon's interpreter with the selected backend
(2) Relative function accuracy measured against the C++ math library
(3) GP algorithm performance and energy efficiency

We also note however that the benchmark results should be interpreted purely in the context of our narrow use case and not as proof that any one math library is "better" than another.

Primitive set efficiency. We set up this experiment by generating expression trees of length two and three, corresponding to unary or binary functions from the primitive set. The arguments to these functions are floating point values uniformly distributed in the range $[-10, +10]$. As a baseline for performance, we introduce a simple backend called Stl, based purely on standard library math functions from the C++ `<cmath>` header. All functions in the Stl backend are implemented in the style of Listing 5, therefore their performance will depend on the compiler's ability to autovectorize the loops.

In order to benchmark each backend, we generate 1'000 trees and simulate a training partition of 10'000 rows (so that the interpreter's batch size is fully utilized).

The results displayed in Table 2 show evaluation performance expressed as a speedup factor compared to the Stl baseline. The vectorized backends (Blaze, Eigen, Eve, Fastor and Vdt) are able to

[6]https://sleef.org/
[7]https://github.com/xtensor-stack/xsimd
[8]https://nanobench.ankerl.com/
[9]https://github.com/foolnotion/gecco-evooss-workshop-2024

deliver significant speedups particularly in the case of trigonometric, inverse trigonometric, logarithmic, exponential and hyperbolic functions. However, they do not deliver a speedup in the case of a purely arithmetic primitive set composed of $+, -, \times, \div$ as those operations are trivial to autovectorize.

Based on these results, it seems more advantageous to use the plain Stl implementation for arithmetic functions, and rely on vectorized library versions for the others. In terms of absolute speed shown in Table 3, arithmetic functions are always the fastest. Protected division implemented as the analytical quotient:

$$AQ(x, y) = \frac{x}{\sqrt{1 + y^2}}$$

is approximately 30% slower than unprotected division, while the square root is also surprisingly cheap (approximately as fast as unprotected division).

We note that as the Armadillo library does not provide vectorized function implementations, its speed is comparable to the Stl backend, since the simple functions in the GP primitive set do not make use of its advanced expression evaluation capabilities.

Function accuracy. Here we compare the accuracy of the Stl backend against the other math library backend functions, which often include speed trade-offs (for example, performing less accurate approximations of some functions). We compute the *relative error* as

$$\eta = \left| \frac{x - y}{y + \varepsilon} \right|$$

where x is the backend value, y is the reference value (computed by Stl) and ε is the machine epsilon[10] added to prevent division by zero. Table 4 shows that all the tested libraries offer similar levels of accuracy up to 8-9 decimal places, except Vdt which delivers less precision for some functions: div, aq, sinh, cosh, tanh, sqrt.

Runtime and energy efficiency on synthetic benchmarks. For the synthetic benchmarks we use the Friedman-I and Friedman-II functions [10]:

$$F_1(\mathbf{x}) = 0.1 \exp(4x_1) + \frac{4}{1 + \exp -20(x_2 - 0.5)} + 3x_3 + 2x_4 + x_5 + \epsilon$$

$$F_2(\mathbf{x}) = 10 \sin(\pi x_1 x_2) + 20 \left(x_3 - \frac{1}{2} \right)^2 + 10x_4 + 5x_5 + \epsilon$$

with x_i sampled uniformly from the unit hypercube ($x \sim \mathcal{U}(0, 1)$) and ϵ generated from the standard normal distribution ($\epsilon \sim \mathcal{N}(0, 1)$).

We employ a canonical genetic programming algorithm configured with the parameters described in Table 5. Local search with 3 iterations is included in order to test the math backend function accuracy also when computing derivatives.

We investigate the relationship between dataset size and efficiency by varying the size of the training partition between 1000 and 5000, in increments of 1000 rows. For each training size, we perform 20 repetitions for each backend. The GP algorithm is allowed to use all threads on the machine (32 threads).

In terms of solution quality as well as other GP metrics (average population fitness, average tree length, number of residual and gradient evaluations), the results are indistinguishable across all the backends. This suggests that GP is much less sensitive to potential numerical precision issues than other optimization methods. Therefore, our analysis can be focused on runtime and energy usage.

Population size	1'000
Evaluation budget	1'000'000 evaluations
Maximum generations	1'000
Maximum tree depth	10
Maximum tree length	50
Selection operator	Tournament selection (tournament size = 5)
Crossover probability	100%
Mutation probability	25%
Local search	Levenberg-Marquardt algorithm (3 iterations per individual)
Primitive set	$\{+, -, \times, \div, \sin, \cos, \log, \exp\}$

Table 5: GP algorithm parameters

We measure energy efficiency for each GP run using the Linux `perf` tool, which has access to the RAPL (Run Average Power Limit) low-level interface on our AMD platform. Note that this measurement has a negligible impact on benchmark performance [24]. The measurements shown in Figure 3 suggest that in GP, energy consumption is directly proportional to the runtime of the algorithm, although this is not necessarily true in general [21]. We hypothesize that this relationship holds for GP due to its "embarrassingly parallel" problem structure. The runtime and energy differences between the Stl and Arma backends and the others can be attributed to lack of vectorization support.

Comparing baseline energy consumption by the Stl backend, we see that significant energy savings in the range of 20-35% can be achieved by switching to a more efficient computational backend, with potentially larger savings on larger datasets.

Knowing that power $P = \frac{dW}{dt}$, we can derive accurate values of a backend's "power rating" from our energy and runtime measurements:

- Arma $\sim 155.03 \pm 5.28$ watts
- Blaze $\sim 147.97 \pm 2.29$ watts
- Eigen $\sim 145.72 \pm 3.76$ watts
- Eve $\sim 145.76 \pm 3.35$ watts
- Fastor $\sim 147.19 \pm 2.41$ watts
- Stl $\sim 152.23 \pm 4.38$ watts
- Vdt $\sim 145.96 \pm 3.87$ watts

It is also possible to derive a custom efficiency metric as "node evaluations per watt" or "nodes per watt" as:

$$\text{NPW} = \frac{\text{generations} \cdot \text{training set size} \cdot \text{total nodes}}{\text{total energy consumption (Joule)}}$$

which is similar to "performance per watt" or "Flops per watt" metrics commonly used in energy efficiency benchmarks.

This metric shown in Figure 4 is particularly useful for comparing different GP algorithms and frameworks as it is completely agnostic of the internal implementation details and can be easily computed

[10]That is, the difference between 1.0 and the next representable value

	add	sub	mul	div	aq	pow	exp	log	sin	cos	tan	asin	acos	atan	tanh	sqrt	cbrt
Arma	0.95	0.90	0.96	0.97	0.84	1.03	1.04	1.03	1.04	1.02	1.03	1.12	1.04	1.06	1.04	1.02	1.04
Blaze	1.00	0.98	0.99	1.02	0.82	1.02	3.78	2.78	2.78	2.33	6.10	2.66	2.33	2.91	5.36	0.97	3.48
Eigen	1.01	0.98	1.00	1.04	0.87	0.51	4.18	2.81	4.40	4.08	1.04	1.06	0.98	1.00	22.48	0.89	1.03
Eve	1.02	1.00	1.00	1.00	0.99	1.63	4.96	3.17	4.37	4.08	12.86	6.28	3.71	7.48	10.83	0.99	6.23
Fastor	0.75	0.74	0.76	0.82	0.99	0.99	3.75	2.71	4.19	3.65	12.26	5.18	4.38	6.12	11.99	0.98	7.26
Vdt	1.03	1.06	1.06	0.81	1.10	1.45	3.26	2.21	4.10	3.91	12.01	5.69	5.47	9.86	1.09	0.96	1.04

Table 2: Relative primitive speedup compared to the Stl baseline implementation

	add	sub	mul	div	aq	pow	exp	log	sin	cos	tan	asin	acos	atan	tanh	sqrt	cbrt
Stl	3.50	3.48	3.41	3.12	2.22	0.24	0.37	0.44	0.34	0.37	0.10	0.33	0.33	0.22	0.09	2.94	0.12
Arma	3.30	3.11	3.26	3.03	1.87	0.24	0.38	0.45	0.35	0.38	0.10	0.37	0.35	0.23	0.09	2.99	0.12
Blaze	3.50	3.39	3.38	3.20	1.82	0.24	1.39	1.22	0.94	0.86	0.60	0.88	0.78	0.63	0.48	2.85	0.41
Eigen	3.52	3.40	3.41	3.23	1.94	0.12	1.54	1.23	1.49	1.50	0.10	0.35	0.33	0.22	2.01	2.63	0.12
Eve	3.56	3.46	3.40	3.13	2.21	0.39	1.83	1.39	1.48	1.51	1.27	2.07	1.24	1.63	0.97	2.91	0.74
Fastor	2.62	2.58	2.58	2.55	2.19	0.23	1.38	1.19	1.42	1.35	1.21	1.71	1.46	1.33	1.07	2.89	0.86
Vdt	3.60	3.69	3.62	2.53	2.44	0.34	1.20	0.97	1.39	1.44	1.19	1.88	1.82	2.15	0.10	2.82	0.12

Table 3: Absolute primitive speed measured in "billion operations per second"

	Arma	Blaze	Eigen	Eve	Fastor	Vdt
div	0.0000e+00	0.0000e+00	0.0000e+00	0.0000e+00	0.0000e+00	3.6905e-06
aq	2.2025e-09	0.0000e+00	3.1595e-08	2.2025e-09	2.2025e-09	1.7431e-06
pow	0.0000e+00	1.7387e-09	4.1903e-09	6.2787e-08	1.7387e-09	2.8929e-07
acos	0.0000e+00	7.6830e-09	0.0000e+00	2.6454e-08	2.6076e-08	4.3070e-08
asin	0.0000e+00	6.0957e-09	0.0000e+00	2.8938e-08	2.9738e-08	2.8933e-08
atan	0.0000e+00	5.7084e-09	0.0000e+00	5.3478e-09	3.0780e-08	2.9164e-08
cbrt	0.0000e+00	8.9872e-09	0.0000e+00	4.5418e-08	1.8654e-08	0.0000e+00
cos	0.0000e+00	3.6041e-09	1.4806e-08	1.4827e-08	2.2340e-08	1.4976e-08
cosh	0.0000e+00	1.8530e-08	0.0000e+00	1.0117e-08	7.6927e-09	3.7226e-06
exp	0.0000e+00	7.7464e-09	1.3913e-08	1.0613e-08	7.7464e-09	1.4067e-08
log	0.0000e+00	8.7051e-09	6.4153e-09	5.6286e-09	8.7051e-09	5.7876e-09
logabs	0.0000e+00	9.0657e-09	6.2292e-09	5.6071e-09	9.0657e-09	5.6541e-09
log1p	0.0000e+00	5.2067e-09	1.1616e-08	5.6531e-11	5.2067e-09	2.1537e-09
sin	0.0000e+00	3.6096e-09	1.2092e-08	1.1816e-08	1.8798e-08	1.2074e-08
sinh	0.0000e+00	2.1607e-08	0.0000e+00	1.8405e-08	3.3697e-08	3.7381e-06
sqrt	0.0000e+00	0.0000e+00	3.0827e-08	0.0000e+00	0.0000e+00	1.8487e-06
sqrtabs	0.0000e+00	0.0000e+00	3.0999e-08	0.0000e+00	0.0000e+00	1.8452e-06
tan	0.0000e+00	8.1600e-09	0.0000e+00	2.5846e-08	3.2115e-08	2.9201e-08
tanh	0.0000e+00	6.3581e-09	4.9498e-08	4.7615e-09	7.8041e-09	3.5453e-06

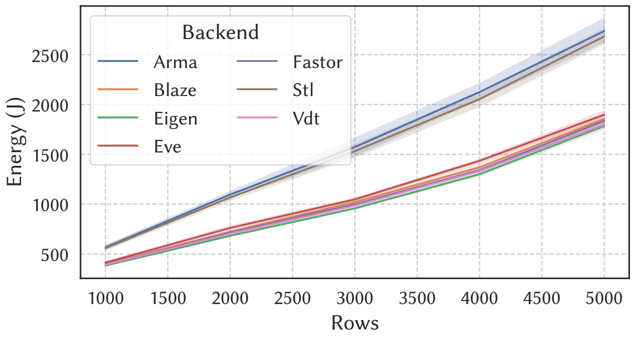
Table 4: Error relative to `<cmath>` functions as baseline

as long as basic evolutionary statistics are known at the end of the run.

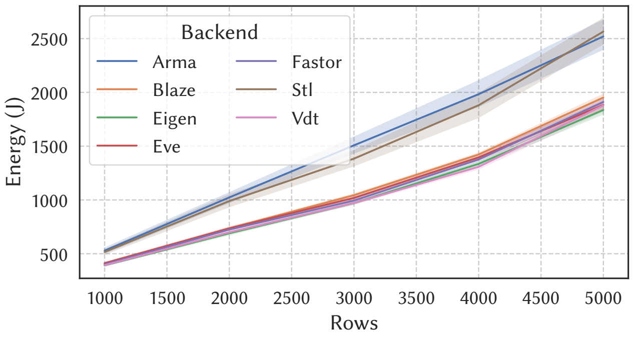
In summary, switching to a more efficient computational backend will not affect modeling results but may reduce energy consumption of GP frameworks by as much as one third (potentially even more for larger data).

5 CONCLUSION

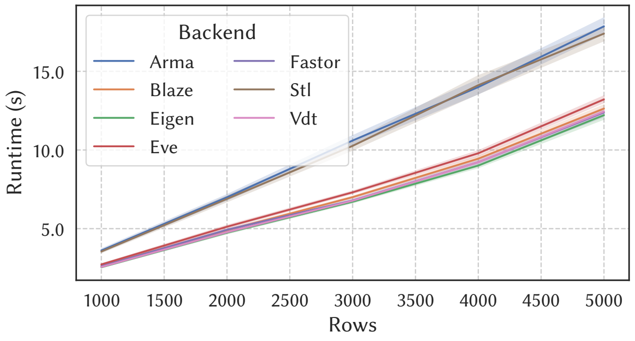
Optimizations such as just-in-time compilation or aggressive use of vectorization can significantly reduce the computational burden of tree evaluation in GP. In this paper, we described a design for tree evaluation that enables a clean separation between high-level evaluation logic associating node types with mathematical operations and low-level execution logic that computes the results. This enables optimizations at the lower level while maintaining a clean API and promoting extensibility and interoperability with other software.



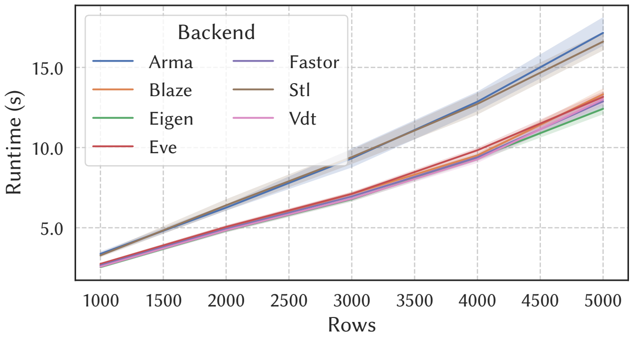
(a) Energy consumption on the Friedman-1 problem



(b) Energy consumption on the Friedman-2 problem



(c) Algorithm runtime on the Friedman-1 problem



(d) Algorithm runtime on the Friedman-2 problem

Figure 3: Runtime and energy efficiency with increasing problem size

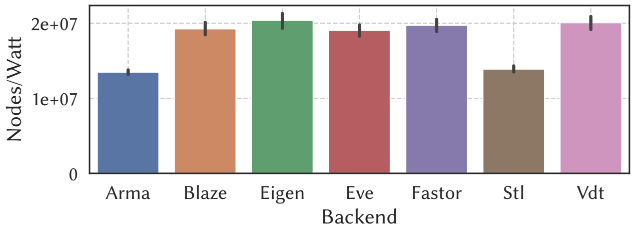


Figure 4: Overall nodes per watt (NPW) average

In order to achieve a clean interface, our approach employs modern C++ language features such as template specialization, variadic templates and fold-expressions. Through a very thin abstraction layer, we were able to integrate several state-of-the-art mathematical libraries as backends for evaluation. We have then shown through several benchmarks that, compared to a naive implementation, backends providing explicit vectorization of transcendental functions can significantly increase efficiency and reduce runtime.

At a deeper level, we have introduced measurements of energy efficiency, an aspect of growing concern in today's software engineering practice. Using the NPW metric, GP software can be compared in an implementation-agnostic manner. Our measurements suggest that large energy savings should be possible by using more efficient mathematical backends. Although our findings apply for symbolic regression, they can be easily generalized to other tasks.

Future work will focus on further improving efficiency, for example by fusing operations in the tree evaluation phase and improving memory usage. This should allow use of optimized evaluation routines provided by the tested math backends. From an energy standpoint, more detailed measurements (e.g. per generation energy usage instead of per entire run) should help gain additional insight on evolutionary dynamics and help developers design their algorithms with power consumption in mind.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.
[2] Abdullah Al Hasib, Lasse Natvig, Per Gunnar Kjeldsberg, and Juan M. Cebrián. 2017. Energy Efficiency Effects of Vectorization in Data Reuse Transformations for Many-Core Processors—A Case Study †. *Journal of Low Power Electronics and Applications* 7, 1 (2017). https://doi.org/10.3390/jlpea7010005
[3] Francisco Baeta, João Correia, Tiago Martins, and Penousal Machado. 2021. TensorGP – Genetic Programming Engine in TensorFlow. In *Applications of Evolutionary Computation*, Pedro A. Castillo and Juan Luis Jiménez Laredo (Eds.). Springer International Publishing, Cham, 763–778.
[4] Bogdan Burlacu, Gabriel Kronberger, and Michael Kommenda. 2020. Operon C++: An Efficient Genetic Programming Framework for Symbolic Regression.

In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion (GECCO '20)*, Richard Allmendinger et al. (Eds.). Association for Computing Machinery, internet, 1562–1570. https://doi.org/10.1145/3377929.3398099

[5] William La Cava, Patryk Orzechowski, Bogdan Burlacu, Fabrício Olivetti de França, Marco Virgolin, Ying Jin, Michael Kommenda, and Jason H. Moore. 2021. Contemporary Symbolic Regression Methods and their Relative Performance. arXiv:2107.14351 [cs.NE]

[6] Juan M. Cebrián, Lasse Natvig, and Jan Christian Meyer. 2014. Performance and energy impact of parallelization and vectorization techniques in modern microprocessors. *Computing* 96, 12 (dec 2014), 1179–1193. https://doi.org/10.1007/s00607-013-0366-5

[7] Miles Cranmer. 2023. Interpretable Machine Learning for Science with PySR and SymbolicRegression.jl. arXiv:2305.01582 [astro-ph.IM]

[8] Christopher Crary, Wesley Piard, Greg Stitt, Caleb Bean, and Benjamin Hicks. 2023. Using FPGA Devices to Accelerate Tree-Based Genetic Programming: A Preliminary Exploration with Recent Technologies. In *Genetic Programming*, Gisele Pappa, Mario Giacobini, and Zdenek Vasicek (Eds.). Springer Nature Switzerland, Cham, 182–197.

[9] Neil G. Dickson, Kamran Karimi, and Firas Hamze. 2011. Importance of explicit vectorization for CPU and GPU software performance. *J. Comput. Phys.* 230, 13 (June 2011), 5383–5398. https://doi.org/10.1016/j.jcp.2011.03.041

[10] Jerome H. Friedman. 1991. Multivariate Adaptive Regression Splines. *The Annals of Statistics* 19, 1 (1991), 1 – 67. https://doi.org/10.1214/aos/1176347963

[11] Gaël Guennebaud, Benoit Jacob, et al. 2010. Eigen. *URl: http://eigen. tuxfamily. org* 3, 1 (2010).

[12] David S. Hollman, Bryce Adelstein-Lelbach, H. Carter Edwards, Mark Hoemmen, Daniel Sunderland, and Christian R. Trott. 2020. mdspan in C++: A Case Study in the Integration of Performance Portable Features into International Language Standards. *CoRR* abs/2010.06474 (2020). arXiv:2010.06474 https://arxiv.org/abs/2010.06474

[13] Klaus Iglberger. 2012. Blaze C++ Linear Algebra Library. https://bitbucket.org/blaze-lib.

[14] Klaus Iglberger, Georg Hager, Jan Treibig, and Ulrich Rüde. 2012. Expression Templates Revisited: A Performance Analysis of Current Methodologies. *SIAM Journal on Scientific Computing* 34, 2 (Jan. 2012), C42–C69. https://doi.org/10.1137/110830125

[15] Hugues Juill. 2001. Parallel Genetic Programming on Fine-Grained SIMD Architectures. https://api.semanticscholar.org/CorpusID:209054810

[16] W. B. Langdon and W. Banzhaf. 2019. Faster Genetic Programming GPquick via multicore and Advanced Vector Extensions. arXiv:1902.09215 [cs.NE]

[17] Loïc Lannelongue, Jason Grealey, and Michael Inouye. 2021. Green Algorithms: Quantifying the Carbon Footprint of Computation. *Advanced Science* 8, 12 (2021), 2100707. https://doi.org/10.1002/advs.202100707 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/advs.202100707

[18] Pengfei Li, Jianyi Yang, Mohammad A. Islam, and Shaolei Ren. 2023. Making AI Less "Thirsty": Uncovering and Addressing the Secret Water Footprint of AI Models. arXiv:2304.03271 [cs.LG]

[19] Jules Penuchot, Joel Falcou, and Amal Khabou. 2018. Modern Generative Programming for Optimizing Small Matrix-Vector Multiplication. In *2018 International Conference on High Performance Computing & Simulation (HPCS)*. 508–514. https://doi.org/10.1109/HPCS.2018.00086

[20] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2017. Energy efficiency across programming languages: how do energy, time, and memory relate?. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering* (Vancouver, BC, Canada) *(SLE 2017)*. Association for Computing Machinery, New York, NY, USA, 256–267. https://doi.org/10.1145/3136014.3136031

[21] Gustavo Pinto and Fernando Castor. 2017. Energy efficiency: a new concern for application software developers. *Commun. ACM* 60, 12 (nov 2017), 68–75. https://doi.org/10.1145/3154384

[22] Danilo Piparo, Vincenzo Innocente, and Thomas Hauth. 2014. Speeding up HEP experiment software with a library of fast and auto-vectorisable mathematical functions. *Journal of Physics: Conference Series* 513, 5 (jun 2014), 052027. https://doi.org/10.1088/1742-6596/513/5/052027

[23] Roman Poya, Antonio J. Gil, and Rogelio Ortigosa. 2017. A high performance data parallel tensor contraction framework: Application to coupled electro-mechanics. *Computer Physics Communications* (2017). https://doi.org/10.1016/j.cpc.2017.02.016

[24] Guillaume Raffin and Denis Trystram. 2024. Dissecting the software-based measurement of CPU energy consumption: a comparative analysis. arXiv:2401.15985 [cs.DC]

[25] Conrad Sanderson and Ryan Curtin. 2016. Armadillo: a template-based C++ library for linear algebra. *Journal of Open Source Software* 1, 2 (2016), 26.

[26] Vimarsh Sathia, Venkataramana Ganesh, and Shankara Rao Thejaswi Nanditale. 2021. Accelerating Genetic Programming using GPUs. arXiv:2110.11226 [cs.NE]

[27] Naoki Shibata and Francesco Petrogalli. 2020. SLEEF: A Portable Vectorized Library of C Standard Mathematical Functions. *IEEE Trans. Parallel Distrib. Syst.* 31, 6 (jun 2020), 1316–1327. https://doi.org/10.1109/TPDS.2019.2960333

[28] Kai Staats, Edward Pantridge, Marco Cavaglia, Iurii Milovanov, and Arun Aniyan. 2017. TensorFlow Enabled Genetic Programming. arXiv:1708.03157 [cs.DC]

[29] Leonardo Vanneschi and Riccardo Poli. 2012. *Genetic Programming — Introduction, Applications, Theory and Open Issues*. Springer Berlin Heidelberg, Berlin, Heidelberg, 709–739. https://doi.org/10.1007/978-3-540-92910-9_24

[30] M. Virgolin, T. Alderliesten, C. Witteveen, and P. A. N. Bosman. 2021. Improving Model-Based Genetic Programming for Symbolic Regression of Small Expressions. *Evolutionary Computation* 29, 2 (06 2021), 211–237. https://doi.org/10.1162/evco_a_00278 arXiv:https://direct.mit.edu/evco/article-pdf/29/2/211/1921067/evco_a_00278.pdf

[31] Carole-Jean Wu, Ramya Raghavendra, Udit Gupta, Bilge Acun, Newsha Ardalani, Kiwan Maeng, Gloria Chang, Fiona Aga Behram, James Huang, Charles Bai, Michael Gschwind, Anurag Gupta, Myle Ott, Anastasia Melnikov, Salvatore Candido, David Brooks, Geeta Chauhan, Benjamin Lee, Hsien-Hsin S. Lee, Bugra Akyildiz, Maximilian Balandat, Joe Spisak, Ravi Jain, Mike Rabbat, and Kim Hazelwood. 2022. Sustainable AI: Environmental Implications, Challenges and Opportunities. arXiv:2111.00364 [cs.LG]

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009