Hybrid CPU/GPU Implementation of Symbolic Regression with Genetic Programming, Backpropagation and Bytecode VM

Doğu Kocatepe
Computer Engineering Department, Izmir Institute of Technology
İzmir, Türkiye
Email: dogukocatepe@std.iyte.edu.tr

Işıl Öz
Computer Engineering, Izmir Institute of Technology
Izmir, Turkey
Email: isiloz@iyte.edu.tr

***Abstract*—Symbolic regression, a method for discovering mathematical expressions from data, has traditionally faced challenges with efficient parallelization and the optimization of numerical constants. This report introduces QuickSR, a high-performance framework that addresses these challenges through a hybrid CPU/GPU implementation. QuickSR combines the global search capabilities of genetic programming (GP) with the local optimization power of gradient descent to simultaneously evolve the structure of expressions and fine-tune their embedded constants. To accelerate the computationally intensive fitness evaluation process, expressions are compiled into a custom bytecode format and executed on a stack-based virtual machine (VM). Several execution "runners" leverage both multi-core CPUs and GPUs in multiple parallel decompositions to optimize performance. We evaluate the performance of these different parallelization strategies in terms of loss, elapsed time, and scalability with respect to both population and dataset size, demonstrating the effectiveness of our hybrid CPU/GPU implementation.**

***Index Terms*—symbolic regression, gradient descent, syntax tree, vectorized bytecode evaluation**

I. Introduction

Symbolic regression is a type of regression analysis that searches for mathematical expressions that best fit a given set of data points. Unlike traditional regression methods that fit parameters to a predefined model, symbolic regression identifies the model structure itself from the data. It does this by searching the space of mathematical expressions to find the one that optimizes a certain fitness criterion. Genetic programming (GP) is a symbolic regression technique in which the evolution process of nature is simulated to evolve well-fitting expressions. The expressions are represented as trees (Fig. 1) in computer memory. At first, a population of random trees is generated, usually having expressions with varying depth and structure. At every generation, the expressions are evaluated in terms of loss to assign them a fitness value. Based on their fitness values, expressions with high fitness are selected to continue, while others are discarded in a process known as survivor selection. Among the surviving expressions, a process called parent selection is performed to choose candidates for reproduction. Those selected undergo crossover operations to generate new children, which are then mutated to form the new offspring. The procedure is repeated until a termination criterion (e.g., maximum number of generations), is reached.

The main performance bottleneck in genetic programming is the fitness evaluation step, in which every expression is evaluated for loss on the dataset. Since recursive evaluation of expressions' trees incurs a performance penalty, the use of stack-based virtual machine is a common approach in genetic programming. In this method, expressions are first converted to bytecode instructions, which take arguments from the stack and push results to the stack. The expression $3 \times x + 4$, for example, can be evaluated using the following bytecode instructions, where mul and add pop two arguments from stack and push one result to the stack.

```
push 3
push x
mul
push 4
add
```

Since every expression in the population is evaluated on all data points, it is possible to parallelize the execution of VM over both expressions (task parallelism) and data points (data parallelism). GPUs are well known to be good at data parallelism, especially when it comes to floating point calculations. For this reason, several frameworks utilize GPUs in stack-based VMs for genetic programming.

There are notable GPU-based frameworks which follow such a VM approach. One is cuML [3], which parallelizes over expressions by assigning each expression to a different warp, and also on data points by using each thread of a warp to calculate loss for a different data point. While cuML performs only the fitness evaluation step on GPU, another framework, evoGP [5] moves both fitness computation and genetic operations (e.g., initialization, mutation, crossover, selection) to GPU kernels, only leaving the main loop and configuration on the CPU side. Unlike previous frameworks, evoGP does not use a single type of parallel decomposition. Instead, it implements two modes that are automatically cho-

This work was supported in part by AMD under the Heterogeneous Accelerated Compute Clusters (HACC) program.

sen at runtime: (1) the hybrid mode which parallelizes over both data points and expressions and (2) the adaptive mode which parallelizes over only data points with a sequential loop over expressions. Although they also introduce a third mode named inter-individual mode (with parallelization only over expressions), they do not seem to be implementing it.

This project introduces QuickSR, a novel, stack-based VM framework that targets multi-core CPUs and AMD GPUs. The main strengths and contributions of the project are as follows:

- We design and build a novel symbolic regression framework applying several optimization techniques:
  - We implement backpropagation as part of stack-based expression evaluation, which integrates local search capabilities of gradient descent into symbolic regression.
  - We utilize the island model, which is a feature that is absent in existing GPU-based GP frameworks. With our algorithm, we have observed better loss evolution with respect to generation when the same set of genetic operators is used. We also parallelize islands over multiple CPU cores to obtain additional performance benefits.
  - We use warp/wave shuffle operations for reduction summation in our GPU implementation.
  - We include an optimization stage before expressions are converted to bytecode instructions, mimicking some of the fundamental optimization techniques supported by modern compilers.
- Our evaluation demonstrates that QuickSR outperforms the state-of-the-art GPU-based genetic programming frameworks for representative benchmarks.
- We target ROCm and HIP platform, which has not been targeted in existing GPU-based GP frameworks. While NVCC currently lacks support for QuickSR, it will also be possible to use our work on NVIDIA GPUs when the support comes, thanks to AMD having implemented HIP as a cross-platform language.
- We combine the project codebase into a Python package, which provides an easy-to-use API to users without requiring knowledge of project internals or C++.

II. Software and Dependencies

QuickSR is a hybrid system that combines a high-performance C++/HIP backend with a user-friendly Python interface. This architecture leverages the strengths of both languages: the raw computational power of C++ and HIP for demanding parallel processing tasks, and the flexibility and ease of use of Python for user interaction and workflow management. The interaction between Python and C++ is achieved through pybind11.

III. Genetic Programming Operators and Hyperparameters

A. Island Model

While most GP frameworks perform genetic operations on a single population, QuickSR implements the island model

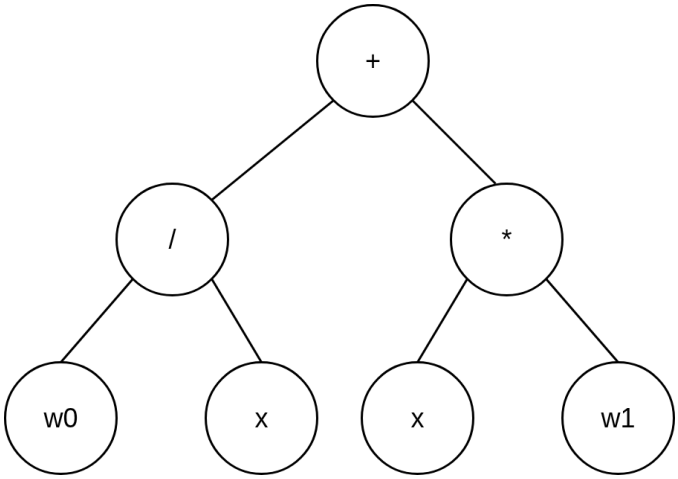

Fig. 1. Tree Representing $\frac{w0}{x} + x \times w1$

[4]. In this paradigm, the total population of expressions is partitioned into smaller, semi-isolated subpopulations called "islands." Each island evolves independently for a number of generations determined by the migration interval. In our project, we call the entirety of such generations a supergeneration. The number of generations within a supergeneration is set by the $ngenerations$ parameter whereas the number of supergenerations is configured by the $nsupergenerations$ parameter.

The parallel evolution of the islands is managed by assigning each island to a dedicated CPU thread. When the GPU implementation is used, the CPU thread is also associated with a distinct HIP stream.

After each super-generation, a migration step occurs to exchange individuals (expressions) between the islands. During migration, a small fraction (determined by a parameter named migration rate) of the best-performing individuals from each island is copied to the next island, replacing the worst-performing individuals in the destination population. As of now, only the unidirectional ring topology is implemented in QuickSR.

B. Initialization

QuickSR supports three distinct strategies for generating the initial population of expressions. Since islands are distributed to different CPU cores, the initialization step is also parallelized over islands.

- Grow Initialization: This method generates trees of varying shapes and sizes. The algorithm randomly selects functions and terminals, allowing branches to terminate before reaching the maximum depth.
- Full Initialization: In contrast to the grow method, this strategy creates "fully grown" trees where every branch extends to the maximum specified depth. This results in a population of expressions that are structurally dense and have a consistent depth.
- Ramped Half-and-Half Initialization: This strategy iterates over different tree depths, starting from one until the

maximum tree depth is reached. For each depth value, half of the expressions are generated using the grow method, whereas the other half is generated using the full method. In the end, the initial population contains a wide range of expression sizes and complexities, from very simple to fully branched trees. In addition to parallelization over islands, this initialization strategy also parallelizes the loop over tree depth by using nested parallelism of OpenMP.

C. Selection

In QuickSR, there are two types of selection. First, survivor selection is performed. This is implemented as sorting of the population according to fitness, followed by a truncation operation in which the proportion determined by $survival\_rate$ continues to live, while the rest dies. Second, parents are selected from the surviving population, in a process called the parent selection. As of now, there are two different parent selection strategies.

- Fitness Proportional Selection: Also known as roulette wheel selection, this strategy selects individuals based on their fitness values. The probability of an individual being selected is directly proportional to its fitness score relative to the total fitness of the entire population. Individuals with higher fitness are more likely to be chosen as parents, but even less fit individuals have a chance of being selected, which helps maintain genetic diversity.
- Rank Selection: This method selects individuals based on their rank within the population rather than their raw fitness score. Individuals are first sorted by fitness, and then selection probabilities are assigned based on their rank in this sorted list. This approach can prevent premature convergence by mitigating the effects of "super individuals" that have a disproportionately high fitness and might otherwise dominate the selection process. The selection pressure can be adjusted with the sp parameter to control the emphasis on higher-ranked individuals.

D. Mutation

Mutation introduces genetic variation into the population by randomly altering expressions. The $mutation\_probability$ parameter determines the likelihood that this operation will be applied to a child. QuickSR offers four distinct mutation operators.

- Subtree Mutation: This operator randomly selects a subtree within an expression and replaces it with a new, randomly generated subtree. The $max\_depth\_increment$ parameter controls the maximum depth of the new subtree to prevent uncontrolled growth in expression complexity.
- Hoist Mutation: This operator is designed to simplify expressions and combat the problem of "bloat" (the tendency of expressions to grow in size without a corresponding increase in fitness). It randomly selects a subtree and then replaces it with a smaller, randomly chosen subtree from within the originally selected subtree.
- Point Mutation: This mutation alters a single node in the expression tree. If a terminal node (a constant or variable) is selected, it is replaced with another random terminal. If a function node is selected, it is replaced with another function of the same arity (e.g., a binary operator is replaced by another binary operator), while its children (operands) are preserved.
- Distribution Mutation: This operator provides a flexible way to combine the other mutation strategies. It applies one of several mutation operators based on a specified probability distribution, allowing the user to control the frequency of different types of mutations.

When the number of islands is less than the number of CPU cores, remaining resources are utilized in this stage by executing a nested parallel for within each island's thread, meaning that the mutation operations are parallelized further. Dynamic scheduling is used in OpenMP to resolve load imbalance which arises when the expressions differ in complexity significantly.

E. Crossover

Crossover, also known as recombination, creates new offspring by combining genetic material from two parent expressions. QuickSR implements subtree crossover. In this process, a random subtree is selected from each of the two parent expressions. These two subtrees are then swapped to create two new offspring, each containing a mix of genetic material from both parents. The $crossover\_probability$ parameter determines the likelihood that this operation will be applied to a given pair of parents. Like in mutation, nested threads are created to further parallelize crossover operation with dynamic scheduling.

IV. Implementation Features

A. Bytecode Compiler and Optimizer

To accelerate fitness evaluation, QuickSR transforms the tree-based expressions into a linear sequence of bytecode instructions for a stack-based virtual machine. The compiler performs a post-order traversal of the expression tree to generate the bytecode. There is also an optimization step prior to compilation, in which expressions are algebraically simplified when possible. Fundamental optimizations such as constant folding (e.g., replacing (x + 2) + 3 with x + 5), are applied in this step to reduce the number of operations necessary during execution. These optimizations do not change the original expressions; they only affect the generated bytecode. Like in mutation and crossover, nested threads are created to further parallelize optimization and compilation on the CPU.

B. Runners and Modes

A "runner" in QuickSR is responsible for evaluating the MSE loss of expressions on the dataset. The framework provides four different runners to accommodate various hardware and problem types:

- CPU Runner: This runner evaluates and trains expressions entirely on the CPU. It leverages task parallelism by

assigning each island to a separate CPU core. Within an island, there is no parallelism over expressions or data points. Both are processed sequentially. The only parallelism comes from the distribution of islands to different cores.

- Inter-individual GPU Mode: This GPU mode parallelizes over the individuals (expressions) in the population. Each GPU thread processes the bytecode for a different individual, looping sequentially through the data points.
- Intra-individual GPU Mode: This mode parallelizes over the data points for a single individual at a time. A GPU kernel is launched to execute the same bytecode program in parallel across all data points, with the final loss calculated via a reduction summation. This mode is particularly effective for large datasets when compared against CPU (Fig. 2).
- Hybrid GPU Mode: As its name suggests, this mode combines the inter- and intra-individual approaches, parallelizing over both expressions and data points simultaneously. A single kernel is launched where each warp/wave processes a single expression, and each thread within a warp/wave processes a different data point for that expression. A warp shuffle is used to sum the losses calculated by different threads. While evoGP and cuML use a 2D grid in this mode, QuickSR uses a 1D grid of threads. Performance of this mode was observed to scale very well with increasing population size (Fig. 3), confirming the findings of [5].

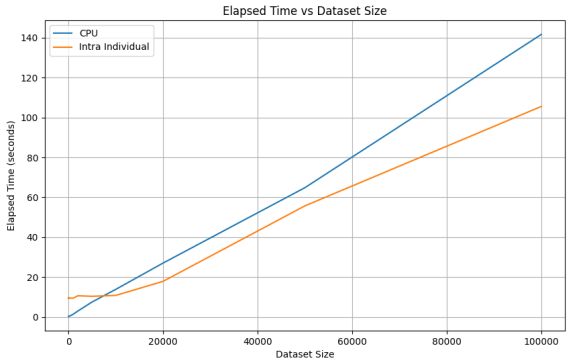

Fig. 2. Scalability of Intra-individual Mode for Dataset Size

C. Gradient Descent and Fitting Constants

Most GP frameworks use ephemeral random constants [1] and consecutive algebraic simplifications (e.g., $(x + x)/x = 2.0$) to discover constants in expressions. While these techniques are effective, QuickSR also has the ability to inject trainable constants which are then trained using gradient descent and backpropagation, a technique borrowed from deep learning. This is an optional feature, and the maximum number of distinct trainable parameters is configurable via the interface.

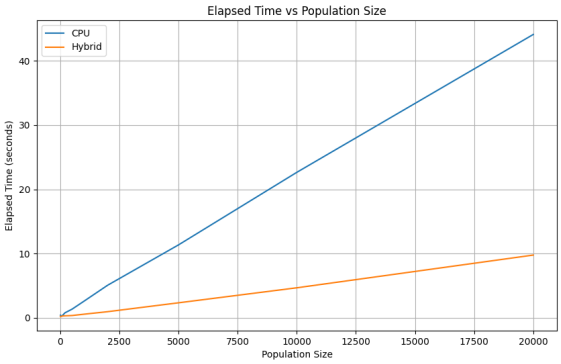

Fig. 3. Scalability of Hybrid Mode for Population Size

When the feature is enabled, the bytecode instructions generated by the compiler also include instructions for backpropagation, which efficiently calculate the gradient of the loss function with respect to each trainable parameter. These gradients are then used in a gradient descent algorithm to update the values of the parameters, iteratively minimizing the expression's Mean Squared Error (MSE) on the dataset.

Below are the bytecode instructions including backpropagation for the expression $w0 * cos(x) * x + x * x - w1$, where $w0$, $w1$ are trainable parameters:

```
 0 var 0
 1 param 0
 2 var 0
 3 cos
 4 mul
 5 mul
 6 var 0
 7 var 0
 8 mul
 9 add
10 param 1
11 sub
12 loss
13 grad_sub [9]
14 grad_add [7]
15 grad_mul [3]
16 grad_var 0 [0]
17 grad_mul [1]
18 grad_param 0 [0]
19 grad_cos [0]
20 grad_var 0 [0]
21 grad_mul [5]
22 grad_var 0 [5]
23 grad_var 0 [5]
24 grad_param 1 [9]
```

In addition to the stack memory used in traditional stack-based VM implementations for GP, QuickSR also uses an intermediate value array when backpropagation is used. This

array stores temporary calculation results obtained during the forward pass. Later, the backpropagation instructions (e.g., *grad_cos [0]*) use these values to compute outgoing gradients. The index in squared brackets denotes the intermediate value array location where the function's original input value was stored.

The use of gradient descent was observed to improve loss with respect to generation (Fig. 4, 200 epochs with learning rate 10^{-3}), but the increase in time was large enough to eliminate the benefits. Further work is needed.

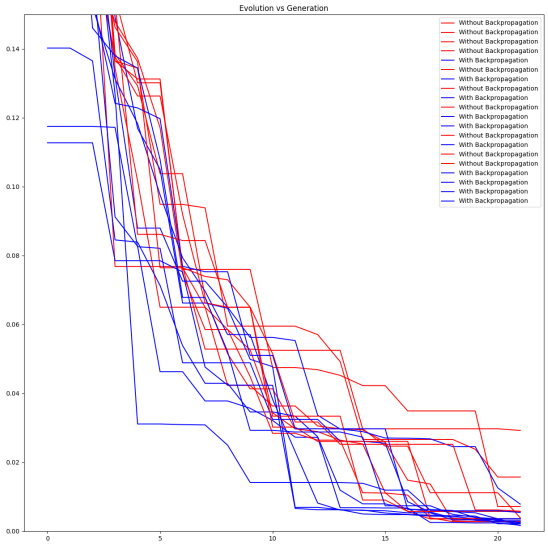

Fig. 4. Effect of gradient descent on loss with respect to generation

D. Fitness Caching

To further optimize performance, QuickSR implements a hash-based fitness caching mechanism. Since the fitness evaluation of an expression is a computationally expensive operation, the result is cached after the first time it is computed. In subsequent generations, if the same expression reappears, its fitness value can be retrieved directly from the cache, avoiding the need for a redundant re-evaluation. Currently, the hybrid and inter-individual modes can read from cache only if the expression was cached in a previous generation. It is currently not possible to cache expressions within the same generation. This constraint does not apply to the CPU mode and the intra-individual mode.

E. Floating Point Precision

QuickSR supports half precision (*half*), single precision (*float*) and double precision (*double*) for floating point computations in the stack-based VM. The only exception is the loss accumulation step, where loss is always accumulated in single precision due to the failure of half precision to store the total loss from all data points accurately. The computations outside the stack-based VM (e.g., calculation of selection probabilities from fitnesses in fitness proportional selection) are always performed in double precision. As of now, the precision that will be used in the VM code is configured at compile time by changing a single type alias in precision header.

V. Maintenance and Documentation

The codebase [1] uses class inheritance to organize genetic operators and runners. For example, adding a new runner to the project is possible by extending the BaseRunner class, even if the new runner uses a completely different technique to utilize GPU hardware. The only change necessary to make this new runner available on the Python side is to add the associated declaration to the source file where Python bindings are listed. The situation is the same for genetic operators, except that each type of genetic operator has its own base class (e.g., BaseMutation, BaseSelection). The information necessary to perform such changes is provided in Doxygen-style code comments, which can be used to generate a collection of offline HTML documents. The information here is thought to be sufficient for making changes to the code.

The documentation for end user [2], however, was prepared separately using Sphinx and readthedocs.io. We maintain the sources for this type of documentation in an independent code repository [3] at GitHub. Here, we provide a short tutorial on the usage of the framework in the form of a Jupyter notebook. We also give detailed explanation on runners and genetic operators, together with short code examples and drawings to describe tree operations.

A short presentation describing the project and its major innovations is uploaded to YouTube. [4].

VI. Comparative Benchmarks

Currently, ROCm/HIP supports C++23, whereas CUDA does not. Since QuickSR used features of this standard at many places in the code, it was challenging to assess QuickSR on both NVIDIA and AMD platforms. As a result, evoGP, a cutting-edge framework designed for NVIDIA GPUs with claimed state-of-the-art performance, was used to compare QuickSR's performance. QuickSR was run on AMD Radeon RX 9060 XT (16GB) whereas evoGP was run on NVIDIA RTX 4080 (16GB).

The Pagie Polynomial [2] (Fig. 5), a test function commonly used in genetic programming literature, was used to generate a synthetic dataset in all benchmarks. The dataset is two dimensional: 64 points along x dimension with range (-5, 5) and 64 points along y dimension with range (-5, 5). In total, the dataset contains 4096 data points.

In all benchmarks, subtree mutation and subtree crossover (always performed, corresponding to crossover probability of one in QuickSR) were used. To stay as close as possible to the

[1] https://github.com/kocatepedogu/quick-symbolic-regression
[2] https://quick-symbolic-regression-docs.readthedocs.io/en/latest/user/index.html
[3] https://github.com/kocatepedogu/quick-symbolic-regression-docs
[4] https://youtu.be/B5MfGlMG740

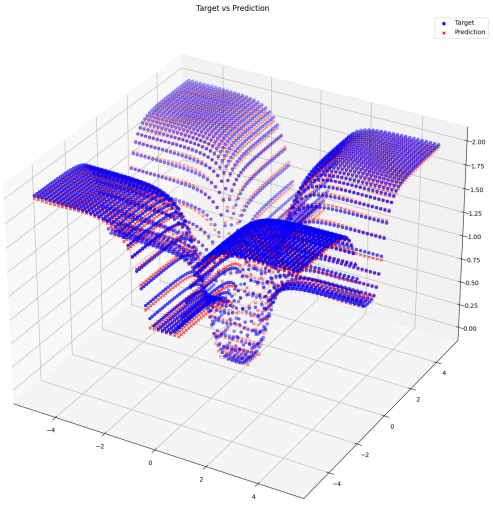

Fig. 5. Pagie Polynomial: Ground truth vs. QuickSR prediction

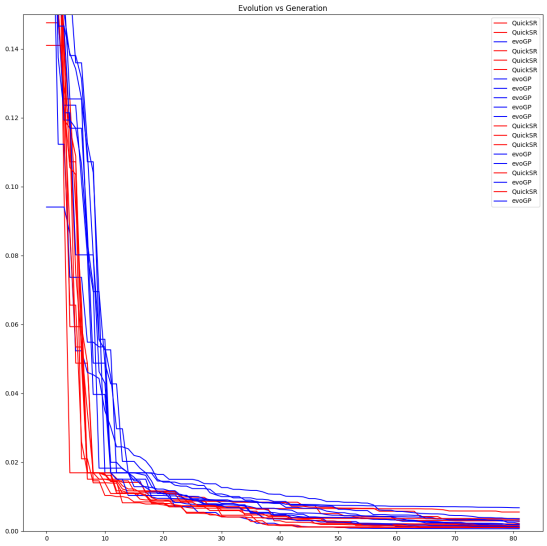

Fig. 6. Benchmark 1: Loss vs. Generation

configuration provided in evoGP's readme [5] in its repository, truncation selection was performed for survival selection and parents were selected uniformly randomly from the surviving population. Both frameworks were run for 80 generations and for a population size of 11200. The population was divided into 9 islands in the case of QuickSR.

In all cases, QuickSR was used with the hybrid runner. Since the dataset size (4096) was lower than the number of CUDA cores (7680), evoGP is also assumed to run in its hybrid parallelism mode, according to the automatic decision algorithm provided in the associated paper. Backpropagation was disabled.

The first benchmark was performed with a mutation rate of 0.2 and a survival rate of 0.3. Single precision floating point was used in QuickSR. In this case, QuickSR outperformed evoGP in terms of both loss with respect to generation (Fig. 6) and loss with respect to time (Fig. 7). A speedup of near 2x was observed when compared against time.

The second benchmark was performed with a mutation rate of 0.5 and a survival rate of 0.5. Half precision was used in QuickSR. Again, QuickSR outperformed evoGP in terms of both time and loss with respect to generation. However, it should be noted that evoGP only supports single precision. Due to the decrease in accuracy to represent individual losses from datapoints in QuickSR, the results in the benchmark should be treated with more caution. Nevertheless, we can at least claim similar performance.

The benchmarks provided in this report were selected to be close to the ones in evoGP's readme, while the population size was chosen higher to better saturate hardware resources.

[5]https://github.com/EMI-Group/evogp/blob/main/README.md

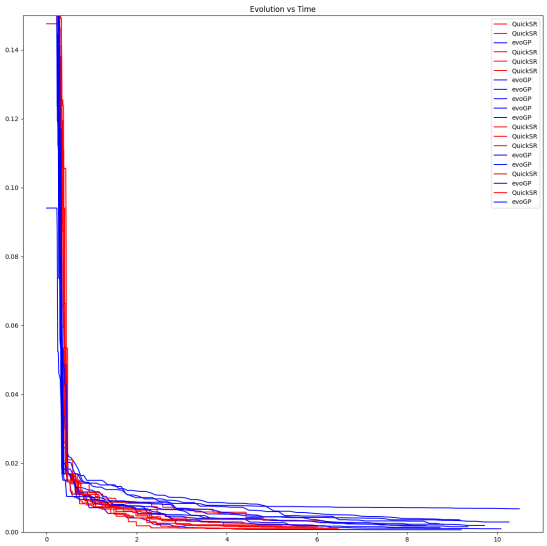

Fig. 7. Benchmark 1: Loss vs. Time

The infrastructure for more comprehensive and systematic benchmarks is available under *benchmarks/comparison* directory in the code repository [6]. Other frameworks or different hyperparameters can be evaluated by add new entries to the run_queue list in $benchmark\_runner.py$.

[6]https://github.com/kocatepedogu/quick-symbolic-regression/tree/main/benchmark/comparison

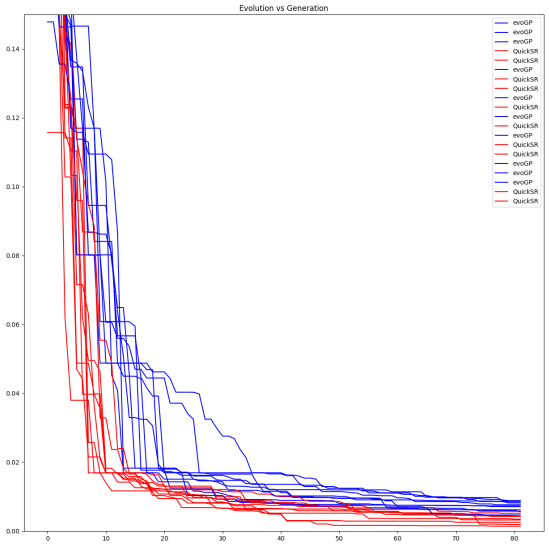

Fig. 8. Benchmark 2: Loss vs. Generation

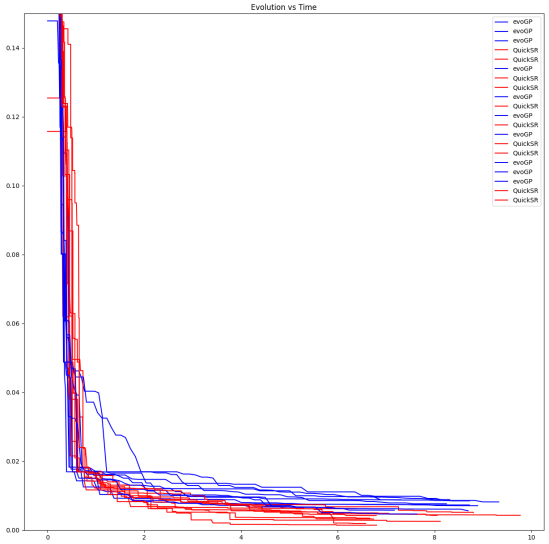

Fig. 9. Benchmark 2: Loss vs. Time

VII. Conclusion

In this project, we have presented QuickSR, a hybrid CPU/GPU framework for symbolic regression that integrates genetic programming with gradient-based optimization. The use of a bytecode virtual machine and a variety of parallel execution "runners" allows for fitness evaluations on both multi-core CPUs and AMD GPUs. Our comparative benchmarks demonstrate that QuickSR is a high-performance framework, achieving a speedup of nearly 2x against a state-of-the-art, NVIDIA-based framework in a single-precision floating-point configuration.

Various parallelization strategies have shown their distinct strengths: the intra-individual mode scales well with increasing dataset sizes, while the hybrid mode is highly effective for large populations, confirming the previous findings of evoGP. The optional gradient descent feature, enabled by backpropagation through the bytecode, has been shown to improve the convergence of loss with respect to generations, although further work is needed to optimize its runtime performance.

Future work could focus on several areas. First, the gradient descent implementation could be further optimized to reduce its time overhead, potentially making it a more consistently beneficial feature. Second, exploring other migration topologies could lead to further improvements. Finally, incorporating more sophisticated fitness caching mechanisms could improve the performance. Overall, QuickSR provides a high-performance platform for symbolic regression.

Acknowledgment

This work was supported in part by AMD under the Heterogeneous Accelerated Compute Clusters (HACC) program.

References

[1] John R. Koza. *Genetic programming : on the programming of computers by means of natural selection*. Cambridge, Mass. : MIT Press, 1992.

[2] Ludo Pagie and Paulien Hogeweg. Evolutionary Consequences of Coevolving Targets. *Evolutionary Computation*, 5(4):401–418, December 1997.

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

[4] Zbigniew Skolicki. An analysis of island models in evolutionary computation. In *Proceedings of the 7th annual workshop on Genetic and evolutionary computation*, pages 386–389, Washington D.C., June 2005. ACM.

[5] Zhihong Wu, Lishuang Wang, Kebin Sun, Zhuozhao Li, and Ran Cheng. Enabling Population-Level Parallelism in Tree-Based Genetic Programming for Comprehensive GPU Acceleration, July 2025. arXiv:2501.17168 [cs].