Speeding up Genetic Programming Based Symbolic Regression Using GPUs

Rui Zhang[1], Andrew Lensen[2], and Yanan Sun[1] *

[1] Sichuan University, Chengdu 610000, China
zhang_ray@stu.scu.edu.cn; ysun@scu.edu.cn
[2] Victoria University of Wellington, Wellington 6140, New Zealand
andrew.lensen@ecs.vuw.ac.nz

Abstract. Symbolic regression has multiple applications in data mining and scientific computing. Genetic Programming (GP) is the mainstream method of solving symbolic regression problems, but its execution speed under large datasets has always been a bottleneck. This paper describes a CUDA-based parallel symbolic regression algorithm that leverages the parallelism of the GPU to speed up the fitness evaluation process in symbolic regression. We make the fitness evaluation step fully performed on the GPU and make use of various GPU hardware resources. We compare training time and regression accuracy between the proposed approach and existing symbolic regression frameworks including gplearn, TensorGP, and KarooGP. The proposed approach is the fastest among all the tested frameworks in both synthetic benchmarks and large-scale benchmarks.

Keywords: Symbolic regression · Genetic programming · Parallel algorithm · Graphics processing unit (GPU) · Compute unified device architecture (CUDA)

1 Introduction

Exploring and learning relationships from data is the central challenge of the sciences. Among various methods [33,34] for achieving this goal, symbolic regression [3] which can represents such relationships as a concise and interpretable function is the most popular [32]. It has a wider range of applications in curve fitting [14], data modeling [17], and material science [35].

Symbolic regression is achieved as an optimization problem. Given a dataset (X, y), symbolic regression is achieved by optimizing an interpretable function $f(X) : \mathbb{R}^n \rightarrow \mathbb{R}$ to minimize the loss $D(f(X), y)$. Achieving symbolic regression has two common approaches: Genetic Programming (GP) method [23] and neural network (NN) method [6, 24, 28]. As one of the Evolutionary Algorithms (EA), GP optimizes solutions by imitating the evolution procedure in nature and aims to find global optima. GP is a generalized heuristic search technique used to optimize a population of computer programs according to a fitness function that determines the program's ability to perform a task. Due to its flexible

* Corresponding author.

representation and good global search ability, GP is the mainstream method for solving symbolic regression problems. The advantage of GP-based symbolic regression compared to the recent neural network (NN) methods [6, 24, 28] is that: the black-box-like solutions provided by NNs are hard to explain and interpret by users. In GP-based symbolic regression, each candidate solution in the population is represented as an expression tree, and the evolutionary process of all participating programs is visible to the user. The user can intuitively discover the characteristics of the data by the features of the different participating programs. Therefore, GP can evolve programs with the potential for interpretability. On the other hand, GP can automatically evolve structures and parameters of programs, which can eliminate the need for the manual design of NN structures.

However, GP is known for its poor scalability. The main reason is that the fitness of each GP program is evaluated on the whole dataset in each generation, causing the GP algorithm to be computationally expensive and time-consuming. Thus, fitness evaluation is the bottleneck of GP in large-scale problems [8]. There are various previous works to optimize the fitness evaluation step of GP, such as caching fitness results of subtree [19], eliminating the need for fitness [7], and computational parallelization. In symbolic regression problems, using computational parallelization is the most effective way to speed up the fitness evaluation step, especially performing parallelizing through GPUs, which can execute thousands of threads in parallel and excel at processing multiple threads using Single Instruction Multiple Thread (SIMT) [11] intrinsic. The existing GPU approaches can be broadly grouped into these two categories:

1) Performing data vectorization and leveraging existing data vectorization interfaces. TensorGP [5] and KarooGP [30] are two common GPU-enabled GP frameworks that support symbolic regression. Both of them are based on the Tensorflow [1] interface. KarooGP adopts the Graph Execution Model [15] of Tensorflow and consequently has a slow execution speed. TensorGP requires a dataset in tensor type, and it does not support regression in real-world datasets well due to this limitation.
2) Directly leveraging GPU parallelization by involving more threads in the computation of the fitness evaluation phase. A SIMD interpreter [22] is developed to evaluate the whole population of GP in parallel. The interpreter computes the intermediate value of the current node each time the kernel is launched, which avoids the use of *switch-case* statements on the GPU to identify the type of the node. However, the frequent launching of kernel functions will cause the delay. Chitty [9] improves the stack structure and stores the prefix on the shared memory. Although better performance is obtained compared to that without memory access restrictions, they do not make greater use of the GPU hardware resources.

To better leverage the multi-threaded parallel computing capability of the GPU in GP-based symbolic regression, this paper proposes a GPU parallel approach to accelerate the fitness evaluation of GP-based symbolic regression. We use the constant memory for program storage, global memory for the stack that

records the temporary fitness results, and shared memory for the parallel metric reduction. The whole dataset and the evaluated program are stored in the device-side memory so that the fitness evaluation step of the proposed method can be performed entirely on the GPU. In the fitness evaluation process of a single GP program, the loss of the program in each fitness case will be executed simultaneously using the GPU parallelism. The experiment results demonstrate that the proposed approach outperforms other GP-based symbolic regression frameworks in execution speed without degradation in regression accuracy. The idea and the novel data structures of the proposed parallel algorithm can be used not only in symbolic regression but also can be generalized to similar stack-based GP methods. The contributions of this work are:

1) We create the GPU acceleration in the fitness evaluation step by using the CUDA C/C++ layer and the code is released at the following address: https://github.com/RayZhhh/SymbolicRegressionGPU.
2) We accelerate the fitness evaluation step by optimizing data structures for symbolic regression on the device side and performing a parallel metric reduction on the GPU, which fully leverages the GPU computational capability.
3) We evaluate the proposed approach against common CPU and GPU frameworks through synthetic datasets and real-world datasets. The proposed approach turns out to be the fastest among all regression tasks.

2 Background and Related Work

This section introduces the GP algorithm and its application in symbolic regression. We also introduce existing GP frameworks that support symbolic regression algorithms.

2.1 Genetic Programming

GP has four major steps: population initialization, selection, mutation, and evaluation. GP algorithm uses random mutation, crossover, a fitness function, and multiple generations of evolution to resolve a user-defined task. GP programs are often represented as syntax tree [29]. The structure of the syntax tree is defined by [29] as follows:

- The 'leaves' of the syntax tree are called terminals. They are variables, constants, and no-parameter functions in the program.
- The inner nodes of the tree are called functions.
- The depth of a node is the number of edges that need to be traversed to reach the node starting from the tree's root node (which is assumed to be at depth 0).
- The depth of a tree is the depth of its deepest leaf (terminal).

2.2 Existing GP Frameworks

The gplearn [31] is implemented based on the *scikit-learn* [27] machine learning framework. According to [4], gplearn can also perform parallelization, but the

parallelization can be used only on the mutation step. Our tests did not find that gplearn's multithreading parameters could effectively improve the computing speed. We disabled this parameter in our later benchmarks. DEAP [13] is another GP framework implemented by Python that provides CPU-based parallelization.

TensorGP and KarooGP are two GPU supported frameworks. Both frameworks are based on the interface of TensorFlow [1] for data vectorization. In the fitness evaluation step, TensorGP represents terminal and variable as the tensor with the same dimension as the input dataset. The metric is calculated by tensor operations (such as tensor multiplication, tensor addition, etc.) provided by Tensorflow. The required dataset of TensorGP is limited to a tensor for a set of points uniformly sampled in the problem domain. So it will be inapplicable when facing real-world problems since the required tensor can not be constructed. Different from TensorGP which leverages the tensor calculation interface, our work makes more intuitive use of GPU parallelism by having threads perform calculations on each data point. TensorGP adopts the Eager Execution Model [2] of TensorFlow, while KarooGP adopts the Graph Execution Model [15] of TensorFlow, which means that in KarooGP, each internal program has to be compiled into a DAG (Directed Acyclic Graph) before having fitness calculation. According to our experimental results and the conclusion in [4], TensorGP turns out to be much faster than KarooGP.

Several papers [4, 5] adopt Pagie polynomial [26] as the speed benchmark for GP-based symbolic regression frameworks. Pagie polynomial is considered to be challenging to approximate and it is recommended by several GP benchmark articles [16, 25]. According to the results in [4], TensorGP (GPU) is faster than other CPU and GPU frameworks including gplearn, KarooGP, and DEAP.

3 The Proposed Symbolic Regression Algorithm

In this section, we first explain the challenge in implementation. Then, we demonstrate the process of our algorithms in chronological order of execution.

3.1 Challenge Faced

Directly porting the CPU code logic to the GPU produces only limited performance improvement. This is because the warp divergence and unconstrained memory access will greatly influence the performance of the GPU. Warp divergence occurs when threads in a warp execute different code blocks. If they execute different *if-else* branches, all threads are blocked at the same time except the one that is executing, which affects performance. The proposed algorithm avoids warp divergence and also achieves coalesced data access by optimizing data structures. Modern GPU architectures provide various components (e.g., constant memory, global memory, and shared memory) with different features. Our work takes the advantage of different components according to specific computing tasks to make full use of the computing resources provided by the GPU.

[9] improves the stack structure and stores the prefix on the shared memory. The improvement made in our work is that all blocks in the grid evaluate the same program. Since there will not be a situation where each block evaluates a different program, we store the programs in constant memory and leverage the on-chip cache for memory access acceleration. This avoids the transfer from global memory to shared memory. Our modification may result in GPU computational resources not being fully used on small-size datasets, while larger datasets will ensure that the evaluation of a program will take up all GPU computational resources.

The flow chart of the proposed algorithm is shown in Fig. 1. In the proposed algorithm, the initialization, selection, and mutation steps are executed on the CPU; the fitness evaluation step, which is the most expensive component in most GP algorithms, is executed on the GPU.

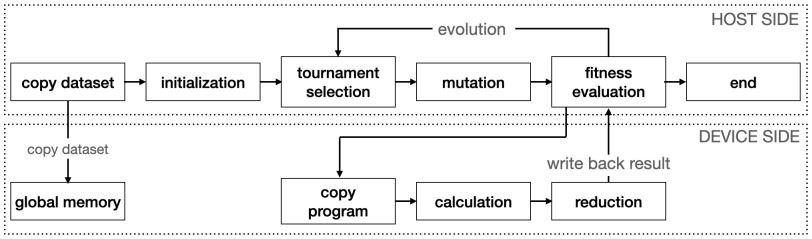

Fig. 1: Process of the proposed algorithm. Memory allocation and free parts on the GPU are ignored.

3.2 Memory Allocation and Dataset Transfer

This step is the initialization of the framework. Since the dataset will not be modified on the device memory, we transfer it to the device side at the beginning. This avoids the delay caused by memory transfer between the host side and device side during fitness evaluation. We also allocate stack memory space in the global memory and two arrays for a program in the constant memory, which can be reused for the fitness evaluation of each generation. On the device side, threads access memory in warp units. If threads of a warp read data with contiguous addresses, CUDA will coalesce their accesses, performing only one memory access request. Therefore, we design data structures for the stack and the dataset on the device side that support coalesced memory access.

We first allocate device-side memory space through *cudaMallocPitch()*, then the dataset is converted into the column-major type (shown in Fig. 2) and transferred to device-side memory through *cudaMemcpy2D()*. For column-major storage, each time when threads access variables, the entire row of the dataset is accessed. As the memory addresses of elements in a row are contiguous, coalesced memory access is available. To achieve coalesced memory access, we also do not

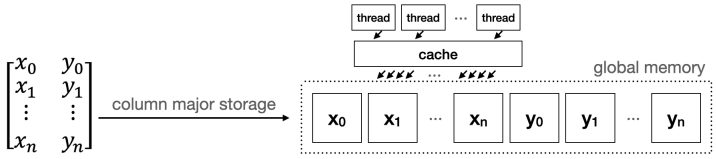

Fig. 2: The column-major storage of the dataset on the device side memory can achieve the coalesced access.

allow threads to allocate independent stack memory. Instead, we consolidate the stack memory space they need. In our implementation, the stack structure is essentially a *1D* array. Our stack structure is shown in Fig. 3, with 512 threads per block used in this work.

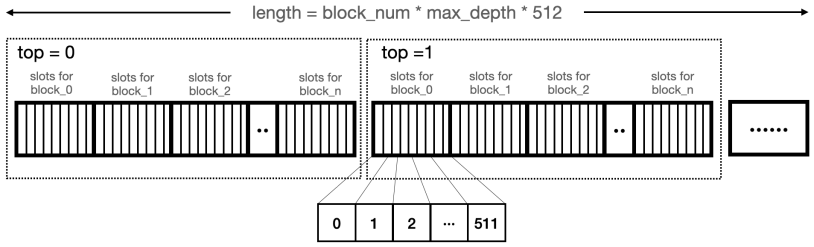

Fig. 3: Device-side stack allocation.

As shown in Fig. 3, if *stack_top* is zero currently, all the threads will access memory space in the box on the left side. And so on, if *stack_top* is one, threads will access memory in the box on the right side. Memory access like this can lead to a coalesced memory access that will greatly improve memory access efficiency. The program will not be modified on the GPU, and it will be accessed by all the threads. In our implementation, a program is stored in constant memory on the GPU, where a single memory-read request to constant memory can be broadcast to nearby threads, which saves memory-read-request times and speeds up the memory access efficiency. In addition, caches can save data of constant memory, so consecutive reads to the same address will not generate additional memory access.

3.3 Population Initialization and Selection

Both initialization and selection are carried out on the CPU. Although the GPU is based on the SIMT (Single Instruction Multiple Threads) architectures, the performance of the GPU cannot be effectively utilized because threads will perform different tasks and execute different instructions when initializing programs, which affects the performance. The proposed approach supports a user-defined function set, as well as the three initialization methods including full initialization, growth initialization, and ramped half-and-half [20]. In the selection step,

the proposed approach only provides tournament selection [10, 21]. The proposed approach also provides a *parsimony_coefficient* parameter inspired by [31] to prevent the bloating of programs.

In our implementation, we adopt an elitist preservation approach where the candidate program with the best fitness of each generation goes directly into the next generation. Elitist preservation can ensure that the fitness of the next generation population is not inferior to the current population.

3.4 Mutation

Mutations take place on the CPU because each program is different that using a GPU is not applicable. The proposed approach supports five mutation types:

- Crossover mutation: A random subtree of the parent tree is replaced by a random subtree of the donor tree.
- Subtree mutation: A random subtree of the parent tree is replaced by a random subtree of a randomly generated tree.
- Hoist mutation: Suppose A is a subtree of the parent tree, B is the subtree of A, hoist mutation replaces A with B.
- Point mutation: A node of the parent tree is replaced by a random node.
- Point replace mutation: Any given node will be mutated of the parent tree.

Note that hoist mutation can lead to a decrease in program depth, which can prevent the program from bloating. Since the stack structure we mentioned earlier limits the maximum depth of the program. To avoid overflow during fitness evaluation, if we find that the depth of a program exceeds the specified maximum depth after mutation, the hoist mutation will be repeatedly performed on the program until the depth of the program is less than the specified depth.

3.5 Fitness Evaluation

The fitness evaluation process is the most complicated part of our algorithm. In this step, the CPU and GPU need to work together. The CPU is responsible for data copy, and the GPU is responsible for data calculation. A program is first converted into a prefix expression. Then, it will be transferred to the constant memory allocated before. The process is illustrated in Fig. 4. The prefix is represented by two arrays that record the values and the types for nodes in the prefix. The element 'u' denotes that the node is a unary function; 'b' denotes a binary function; 'v' denotes a variable; 'c' denotes a constant. Nodes in different types will correspond to different stack operations in the kernel function.

The metric calculation and reduction steps are performed on the GPU. Each thread is responsible for calculating the predicted value for a data point with the help of our device-side stack. The reverse iteration begins from the last node to the first node of the program. For each node in the iteration, we make the corresponding operation according to the type of the node. If the node is a terminal, the thread simply pushes its value into the stack. If the node is a function, the thread calculates the value according to the function type and pushes the result into the stack.

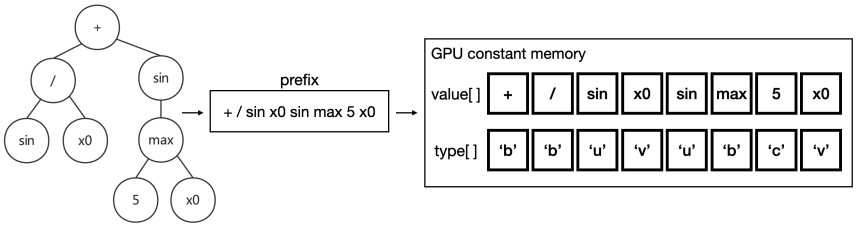

Fig. 4: Program transfers to the device-side memory. Each expression tree is represented by a prefix and each node of the prefix is identified by two tokens.

In the metric calculation step, each thread is responsible for the difference value calculating between the predicted value and its corresponding real value. The proposed approach supports three metric types, they are:

- MAE (Mean Absolute Error)
- MSE (Mean Squared Error)
- RMSE (Root Mean Squared Error)

The metric result computed by each thread will be stored in the shared memory, which is an on-chip memory that offers fast access speed.

In the reduction step, each block is responsible for the sum of losses calculated by its internel threads. The results of blocks are stored in a device-side array allocated in the global memory, which is then copied to the host side. Fig. 5 shows the reduction process on the device-side and host-side.

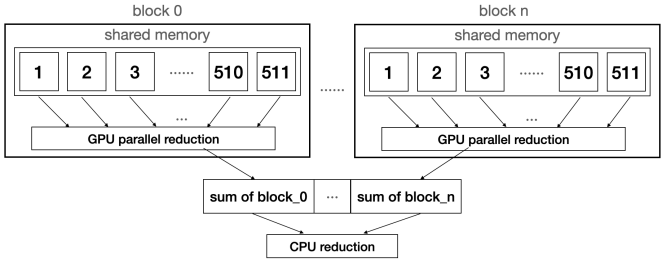

Fig. 5: Reduction on the GPU and the CPU.

After we get the sum of losses calculated by each thread, we will calculate the final loss result according to the specified loss function. The above procedures complete the evaluation of a single program, so these steps will be repeated until all programs in the population obtain fitness. Note that the bank conflict needs to be avoided in parallel reduction design, which occurs when multiple threads simultaneously access different addresses of the same bank. Our implementation ensures that threads in a warp are scattered across different banks during the shared memory access to avoid bank conflict. The kernel function of the proposed algorithm is shown in Algorithm 1.

4 Experiments and Results

This section presents our experimental results on synthetic datasets and large-scale real-world datasets.

Algorithm 1: Kernel function

```
Input: prefix, stack, dataset, realValue, result
Output: none
for node in prefix do
  | doStackOperation(node, stack, dataset);
end
metric = square(stack.top() - realValue);
sharedMem[threadID] = metric;
synchronize();
for i in [256, 128, 64, 32, 16 , 8, 4, 2, 1] do
  | // the loop is expanded in our inplementation
  | if threadID < i then
  |   | sharedMem[threadID] += sharedMem[threadID + i];
  | end
  | synchronize();
end
result[blockID] = sharedMem[0];
```

Table 1: Hardware and software specifications in synthetic benchmarks and large-scale benchmarks.

Component	Specification	Component	Specification
CPU	AMD Ryzen 5 5600H	CUDA Tool Kit Version	11.5
GPU	NVIDIA RTX 3050 Laptop	OS	Windows 11
GPU RAM	4.0 GB	Host RAM	16.0 GB

4.1 Benchmarks on Synthetic Datasets

We compare the average execution times between gplearn (CPU), TensorGP (GPU), KarooGP (GPU), and the proposed approach. We also test the best fitness after 50 iterations under different dataset sizes [4]. All tests employed in synthetic benchmark concern the approximation of the Pagie Polynomial [26] function defined by Equation 1, following the conventions of GP community [4, 16, 25].

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

We generate 7 datasets of different size from $64 \times 64 = 4,096$ data points to $4096 \times 4096 = 16,777,216$ by uniformly subsampling data points from the domain $(x, y) \in [-5, -5] \times [-5, -5]$. Framework parameters are listed in Tab. 2.

In our synthetic dataset experiment, we first compare the execution time of different frameworks in different dataset sizes. We ran each experiment 30 times and calculated the average execution time. The experimental results are shown in Tab. 3 and Fig. 6.

It can be seen from Tab. 3 that the proposed approach performs a faster training speed than other GPU and CPU frameworks for different sizes of datasets. Compared to gplearn which only supports CPU execution, the proposed approach achieves a maximum speedup of 170× acceleration on the fourth dataset

Table 2: Parameters for benchmarks.

Parameter	Value	Parameter	value
Population size	50	Generations	50
Tournament size	3	Fitness metric	RMSE
Maximum initial depth	10	Maximum allowed depth	10
Crossover probability	0.9	Function set	+, -, ×, ÷, sin, cos, tan
Mutation probability	0.08	Initialization method	Ramped Half and Half

Table 3: Average execution time of 30 runs on NVIDIA GeForce RTX 3050 Laptop GPU for various frameworks (lower is better). The symbol of "DNF" denotes that the test does not finish within three hours. The symbol of "MAF" denotes that the memory allocation failed on the GPU. The bold marks the minimum execution time for each test.

Framework	4,096	16,384	65,536	262,144	1,048, 576	4,194,304	16,777,216
Our Approach	**0.152**	**0.215**	**0.193**	**0.331**	**0.886**	**3.034**	**11.851**
TensorGP (GPU)	5.655	6.873	6.236	6.473	6.535	17.334	MAF
KarooGP (GPU)	27.42	47.92	60.08	123.97	367.21	DNF	DNF
gplearn (CPU)	1.731	2.936	8.897	53.006	174.228	DNF	DNF

(1,048,576 data points). The proposed approach is also faster than the two GPU-supported frameworks across each dataset. To discuss the influence of GPU models on the proposed algorithm, we also compare the execution speed on different GPUs. The hardware and software specifications are shown in Tab. 4. As shown in Tab. 5, the proposed approach is faster than TensorGP (GPU) in all GPU models.

Table 4: Hardware and software specifications in various GPUs tests.

Component	Specification	Component	Specification
CPU	Intel Xeon Gold 6310 @ 2.1GHz	RAM	32.0 GB
CUDA Tool Kit Version	11.0	OS	Ubuntu 18.04.5

From these tests, we notice that under different GPU models, TensorGP did not show an increasing trend in the dataset of 64^2 to 1024^2 data points. This may be because the GPU-based tensor calculating interface provided by Tensorflow works well for large-scale tensors, but there is little optimization for smaller tensors. For the proposed algorithm, the regression time in 64^2 to 512^2 data points are similar, and the regression time of 512^2 to 4096^2 dataset is close to linear growth, this is because datasets less than 512^2 data points in our test do not use up all the computing resources provided by the GPU, and the computing resources of the GPU have been exhausted for datasets in larger sizes that more computing tasks have to line up and show a linear growth of the regression time. We also notice that TensorGP has a much more memory consumption than the proposed method. The memory allocation for the 4096^2 dataset failed on the RTX 3050 Laptop GPU with four GB of device memory. This is because that tensor is a complex data structure, so the encapsulation of the dataset requires extra memory space.

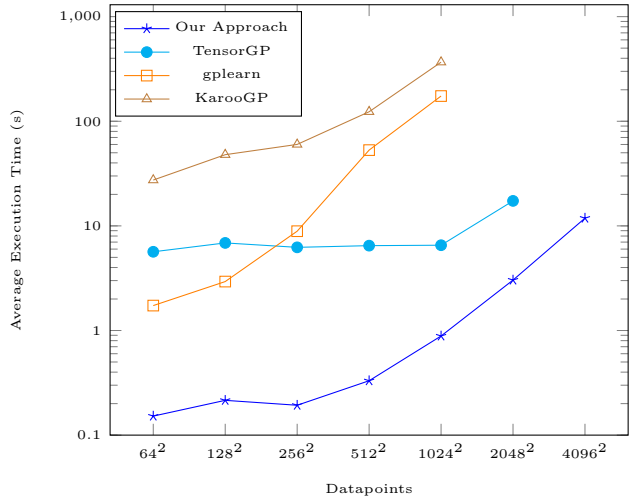

Fig. 6: Log-Log Plot of Execution Time for various frameworks on NVIDIA RTX 3050 Laptop GPU (Lower is better).

Table 5: Execution times on various GPUs in seconds (lower is better). The bold marks the minimum execution time for each test.

Framework	GPU	64^2	128^2	256^2	512^2	1024^2	2048^2	4096^2
TensorGP	RTX 2080 Ti	5.751	5.834	5.428	5.03	5.503	8.168	28.31
Our Approach		**0.086**	**0.085**	**0.114**	0.149	0.353	1.138	3.561
TensorGP	RTX 3090	9.014	8.769	8.551	9.338	9.618	8.7	14.482
Our Approach		0.099	0.094	0.155	**0.148**	**0.32**	**0.806**	**2.605**
TensorGP	NVIDIA A100	7.984	6.834	7.568	7.234	6.934	7.301	19.413
Our Approach		0.139	0.140	0.184	0.173	0.354	0.847	2.686
TensorGP	RTX A6000	8.602	8.454	9.290	7.593	8.528	8.133	22.072
Our Approach		0.154	0.138	0.167	0.234	0.470	1.225	3.486

We also analyze the regression accuracy of the proposed approach under different dataset sizes (shown in Tab. 6). Compare with the corresponding fitness results according to Tab. 6, the regression accuracy of the proposed approach on synthetic datasets is close to TensorGP. Therefore, on the premise of similar regression accuracy, the proposed approach is faster than TensorGP in execution speed.

4.2 Large-Scale Benchmarks

We run large-scale benchmarks on two datasets usually used to compare gradient boosting frameworks. In particular, we consider the Airline [18] and YearPredictionMSD [12] datasets with 115M and 515K rows respectively.

Since both of these two datasets are not able to transform to a *Tensor* form that TensorGP needs, experiments are carried out only on the proposed approach, KarooGP, and gplearn. Each framework will run three times for each

Table 6: Table showing the best RMS Error after 50 iterations.

Size	Our Approach	TensorGP	Size	Our Approach	TensorGP
4,096	0.233 ± 0.045	0.274 ± 0.048	1,048,576	0.242 ± 0.052	0.253 ± 0.066
16,384	0.258 ± 0.041	0.211 ± 0.065	4,194,304	0.246 ± 0.045	0.237 ± 0.078
65,536	0.246 ± 0.047	0.265 ± 0.058	16,777,216	0.247 ± 0.060	–
262,144	0.240 ± 0.052	0.239 ± 0.050			

dataset, and we record the execution time and best fitness after 50 generations of these experiments.

A total 18 runs were performed on gplearn, KarooGP, and the proposed approach. Karoo GP did not finish on the Airline dataset in less than an hour. So we only compared with gplearn on the Airline dataset. Tab. 7 lists the regression accuracy and average execution time in seconds after 50 iterations.

Table 7: Table containing mean execution time and best fitness across three runs for gplearn, KarooGP, and the proposed approach on Airline and Year datasets. The symbol of "DNF" denotes that the test do not finish within an hour.

	Airline Time	Airline Fitness	Year Time	Year Fitness
Our Approach	4.099	37.806	0.629	21.779
gplearn	251.178	37.757	150.119	22.045
KarooGP	DNF	DNF	772.822	22.063

We notice that the regression accuracy of different frameworks was similar across these two datasets, the proposed approach achieves a speedup of 200× acceleration compared to gplearn and a 1200× acceleration compared to KarooGP on the YearPredictionMSD dataset. Through these tests, we conclude that our algorithm can effectively improve the execution speed through GPU parallelization under the premise of achieving similar regression accuracy.

5 Summary

This paper introduces a GPU parallelization algorithm to accelerate the GP-based symbolic regression. We optimize memory access by using column-major storage for the dataset, and a stack space that supports coalesced access for threads. We also implement a GPU-side reduction that avoids bank conflict. After training, the proposed approach preserves the best program in the last generation and its corresponding metric. This program can be considered the optimal solution to the symbolic regression. Our experimental results show that the proposed approach performs faster execution speed than gplearn, TensorGP, and KarooGP. This indicates that the proposed algorithm can effectively improve the execution speed of symbolic regression through parallel computation in the fitness evaluation step. In particular, the fast execution speed on large datasets indicates that the proposed method has the potential to allow GP-based symbolic regression to be applied to large problems that it currently is not able to be.

References

1. Abadi, M., Agarwal, A., Barham, P., et al.: Tensorflow: Large-scale machine learning on heterogeneous distributed systems (2016)
2. Agrawal, A., Modi, A.N., Passos, A., et al.: Tensorflow eager: A multi-stage, python-embedded DSL for machine learning. CoRR **abs/1903.01855** (2019)
3. Awange, J.L., Paláncz, B.: Symbolic Regression, pp. 203–216. Springer International Publishing, Cham (2016)
4. Baeta, F., Correia, J.a., Martins, T., et al.: Speed benchmarking of genetic programming frameworks. In: Proceedings of the Genetic and Evolutionary Computation Conference. pp. 768–775. GECCO '21, Association for Computing Machinery, New York, NY, USA (2021)
5. Baeta, F., Correia, J., Martins, T., et al.: Tensorgp – genetic programming engine in tensorflow. In: Applications of Evolutionary Computation, pp. 763–778. Lecture Notes in Computer Science, Springer International Publishing, Cham (2021)
6. Biggio, L., Bendinelli, T., Neitz, A., Lucchi, A., Parascandolo, G.: Neural symbolic regression that scales. CoRR **abs/2106.06427** (2021), `https://arxiv.org/abs/2106.06427`
7. Biles, J.A.: Autonomous genjam: Eliminating the fitness bottleneck by eliminating fitness. In: In: Proceedings of the Genetic and Evolutionary Computation Conference Workshop Program. vol. 7 (2001)
8. Cano, A., Zafra, A., Ventura, S.: Speeding up the evaluation phase of gp classification algorithms on gpus. Soft Computing **16**, 187–202 (2012)
9. Chitty, D.M.: Improving the performance of gpu-based genetic programming through exploitation of on-chip memory. Soft Computing **20**, 661–680 (2016)
10. Chitty, D.M.: Exploiting tournament selection for efficient parallel genetic programming. In: Lotfi, A., Bouchachia, H., Gegov, A., Langensiepen, C., McGinnity, M. (eds.) Advances in Computational Intelligence Systems. pp. 41–53. Springer International Publishing, Cham (2019)
11. Cook, S.: CUDA Programming: A Developer's Guide to Parallel Computing with GPUs. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn. (2012)
12. Dua, D., Graff, C.: UCI machine learning repository (2017), `http://archive.ics.uci.edu/ml`
13. Fortin, F.A., De Rainville, F.M., Gardner, M., Parizeau, M., Gagné, C.: Deap: Evolutionary algorithms made easy. Journal of Machine Learning Research, Machine Learning Open Source Software **13**, 2171–2175 (2012)
14. H., L., S., Y.: Genetic programming approach to curve fitting of noisy data and its application in ship design. Transactions of the Society of CAD/CAM Engineers **9** (2004)
15. Handley, S.: On the use of a directed acyclic graph to represent a population of computer programs. In: Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence. pp. 154–159 vol.1 (1994)
16. Harper, R.: Spatial co-evolution: quicker, fitter and less bloated. In: Proceedings of the 14th annual conference on genetic and evolutionary computation. pp. 759–766. GECCO '12, ACM (2012)
17. Icke, I., Rosenberg, A.: Multi-objective genetic programming projection pursuit for exploratory data modeling (2010). https://doi.org/10.48550/ARXIV.1010.1888
18. Ikonomovska, E.: Airline dataset:for evaluation of machine learning algorithms on non-stationary streaming real-world problems (2009), `http://kt.ijs.si/elena_ikonomovska/data.html`

19. Keijzer, M.: Alternatives in subtree caching for genetic programming. In: Keijzer, M., O'Reilly, U.M., Lucas, S., Costa, E., Soule, T. (eds.) Genetic Programming. pp. 328–337. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
20. Koza, J.: Genetic programming: On the programming of computers by means of natural selection. Complex Adap. Syst. **1** (1992)
21. Koza, J.: Genetic programming as a means for programming computers by natural selection. Statistics and computing **4**(2), 87 (1994)
22. Langdon, W.B., Banzhaf, W.: A simd interpreter for genetic programming on gpu graphics cards. In: O'Neill, M., Vanneschi, L., Gustafson, S., Esparcia Alcázar, A.I., De Falco, I., Della Cioppa, A., Tarantino, E. (eds.) Genetic Programming. pp. 73–85. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
23. Langdon, W.B., Poli, R., McPhee, N.F., et al.: Genetic programming: An introduction and tutorial, with a survey of techniques and applications. In: Computational Intelligence: A Compendium, pp. 927–1028. Studies in Computational Intelligence, Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
24. Martius, G., Lampert, C.H.: Extrapolation and learning equations. CoRR **abs/1610.02995** (2016), http://arxiv.org/abs/1610.02995
25. McDermott, J., White, D.R., Luke, S., et al.: Genetic programming needs better benchmarks. In: Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation. pp. 791–798. GECCO '12, Association for Computing Machinery, New York, NY, USA (2012)
26. Pagie, L., Hogeweg, P.: Evolutionary Consequences of Coevolving Targets. Evolutionary Computation **5**(4), 401–418 (1997)
27. Pedregosa, F., Varoquaux, G., Gramfort, A., et al.: Scikit-learn: Machine learning in python. Journal of Machine Learning Research **12**, 2825–2830 (2012)
28. Petersen, B.K., Larma, M.L., Mundhenk, T.N., Kim, S., Kim, J.T., Santiago, C.P., Administration, U.N.N.S.: Deep symbolic regression, version 1.0 (12 2019). https://doi.org/10.11578/dc.20200220.1, https://www.osti.gov//servlets/purl/1600741
29. Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk (2008)
30. Staats, K., Pantridge, E., Cavaglia, M., et al.: Tensorflow enabled genetic programming. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion. pp. 1872–1879. GECCO '17, Association for Computing Machinery, New York, NY, USA (2017)
31. Stephens, T.: Genetic programming in python with a scikit-learn inspired api: Gplearn (2016), https://github.com/trevorstephens/gplearn
32. Tohme, T., Liu, D., Youcef-Toumi, K.: Gsr: A generalized symbolic regression approach (2022). https://doi.org/10.48550/ARXIV.2205.15569, https://arxiv.org/abs/2205.15569
33. Tohme, T., Vanslette, K., Youcef-Toumi, K.: A generalized bayesian approach to model calibration. Reliability Engineering & System Safety **204**, 107–141 (dec 2020). https://doi.org/10.1016/j.ress.2020.107141, https://doi.org/10.1016%2Fj.ress.2020.107141
34. Tohme, T., Vanslette, K., Youcef-Toumi, K.: Improving regression uncertainty estimation under statistical change. CoRR **abs/2109.08213** (2021), https://arxiv.org/abs/2109.08213
35. Wang, Y., Wagner, N., Rondinelli, J.M.: Symbolic regression in materials science. MRS Communications (2019). https://doi.org/10.48550/ARXIV.1901.04136