Genetic Programming using GPUs

A Project Report

submitted by

VIMARSH SATHIA

in partial fulfilment of the requirements
for the award of the degree of

BACHELOR OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, MADRAS.

June 2021

THESIS CERTIFICATE

This is to certify that the thesis entitled **Implementing Genetic Programming on GPUs**, submitted by **Vimarsh Sathia**, to the Indian Institute of Technology, Madras, for the award of the degree of **Bachelor of Technology**, is a bona fide record of the research work carried out by him under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. Rupesh Nasre
Research Guide
Professor
Dept. of CSE
IIT Madras, 600036

Place: Chennai

Date: June 12, 2021

ACKNOWLEDGEMENTS

I would like to thank Prof. Rupesh Nasre for giving me the opportunity to be able to work on this project.

ABSTRACT

KEYWORDS: Genetic Programming; Symbolic Regression; CUDA; GPGPU programming; Stack-based virtual machines;

Genetic Programming(GP) is a domain-independent technique for evolving computer programs on the principles of genetics and natural selection. In the context of machine learning, it is one example of an evolutionary algorithm, which iteratively finds solutions to optimization problems through fitness functions and repeated evolution. As such, GP has several inherently parallel steps, making it an ideal candidate for GPU based parallelization.

In this thesis, implementation details involving the parallelization of stack-based Genetic Programming algorithms (more specifically, symbolic regression and transformation) on GPUs are explored. To achieve higher performance and scalability, we focus on parallelizing the selection and evaluation steps of the generational GP algorithm, through parallel tournament-based candidate selections, candidate expression-tree evaluation using a fixed size stack machine in GPU memory, and batching of fitness computations for whole program populations. During program mutations, we restrict program depth by introducing a hoisted version of the crossover operation between a parent and a donor program.

Our work then profiles the performance of the new parallel GP algorithm on Airfoil Self-Noise, Airline on-time performance and YearPredictionMSD datasets, and compares the results with other standard symbolic regression libraries and naive linear regression.

Using nVidia Tesla V100 GPUs, average speedups of upto 10$\times$ were observed on the performance benchmarks of our algorithm against CPU-only implementations of the standard GP libraries.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

GP	Genetic Programming
EC	Evolutionary Computing
EA	Evolutionary Algorithm
GPU	Graphics Processing Unit

CHAPTER 1

INTRODUCTION

In recent years, there has been a widespread increase in the use of GPUs in the field of machine learning and deep learning, because of their ability to massively speedup the training of models. This is mainly due to the large parallel processing ability of GPUs, which can process multiple inputs using SIMD intrinsics. Genetic Programming(GP) is an class of a machine algorithms with several inherent parallel steps. As such, it is an ideal domain for GPU based parallelization.

1.1 The Problem

Genetic Programming is a technique which involves evolving a set of programs based on the principles of genetics and natural selection. It is a generalize heuristic search technique, which searches for a least suitable program optimizing a given fitness function. As such it has applications ranging from machine learning to code synthesis.[9] It also supports a meta-evolutionary framework, where a GP system itself can be evolved using GP.[17]

However, in practice, it is difficult to scale GP algorithms. Fitness evaluation of candidate programs in GP algorithms is a well-known bottleneck and there are multiple previous attempts to overcome this problem - either through parallelization of the evaluation step([8],[3],[6],[21],[20]), or by eliminating the need for fitness computations itself through careful initialization and controlled evolution.[4]

1.2 Our Contributions

In this thesis, we provide details about a parallelized version of the generational GP algorithm to the solve the problems of Symbolic Regression and Transformation. We use a stack-based GP algorithm for simplified program evaluation(inspired by [13]). Our algorithm can also be used to train symbolic binary-classifiers, where the classifier′s output corresponds to the estimated equation of the decision boundary.

Our main contributions are listed below:

- We parallelize the selection and evaluation step of the generational GP algorithm, by conducting parallel tounaments and fitness evaluation among candidate populations.
- We introduce a prefix-tree based Array of Structures (AoS) representation for candidate programs, along with a method to evaluate all programs on a dataset using a thread-local device side stack with constrained size.
- We also provide a new method to perform tournaments for program selection in parallel using random numbers generated inside the GPU.
- We then benchmark the time taken for program evaluation by our algorithm against other standard libraries like gplearn[21] and TensorGP[3]. We provide a study of the effect of population size, dataset size and program length on the final time taken for training. We also show the results of executing our algorithm on the Airline Regression, Airfoil-Self Noise and Year MSD datasets, all taken from the UCI Machine Learning repository[5].

The code for our algorithm is to be merged with the cuML library[15], which is a part of the RAPIDS suite of open source libraries maintained by Nvidia Corporation.

1.3 Outline

The rest of this thesis is structured in the following way.

- Chapter 2 introduces the generational GP algorithm in detail, with respect to the selection, mutation and evaluation steps. It also briefly talks about other existing GP libraries and the strategies used for program evaluation in them.
- Chapter 3 contains implementation details about our algorithm for performing symbolic regression. Here, we examine the new algorithm in detail, and also talk about a few challenges encountered when trying to implement it using CUDA
- Chapter 4 contains the results of various benchmarks run on our algorithm implementation and their speedups with respect to some standard GP libraries. It also contains information about the performance of our algorithm on 3 standard datasets.
- Chapter 5 talks about future extensions and possible optimizations to the current algorithm, like support for custom user-defined functions in addition to the pre-defined function set.

CHAPTER 2

BACKGROUND AND RELATED WORK

In this chapter, we describe in detail the generational GP algorithm([14],[19]), which serves as a base for our parallelization experiments in Chapter 3. Later on, we also give a brief description of existing work which also parallelizes this algorithm.

Note that throughout this chapter, the terms program and expression tree are interchangable, since we are using GP for symbolic regression.

2.1 The Generational GP Algorithm

The generational GP algorithm gives us a method to evolve candidate programs using the principles of natural selection. There are 3 main steps involved in the algorithm.

- Selection - In this step, we decide on a set of programs to evolve into the next generation, using a selection criterion.
- Mutation/Genetic operation - Before promoting the programs selected in the previous step to the next generation, we perform some genetic operations like on them.
- Evaluation - We again evaluate the mutated programs on the input to recompute fitness scores.

Algorithm 1 formalizes the above 3 steps. The initial set of programs is usually randomly generated. The termination criterion defined in line 9 of the algorithm is usually activated when the number of generations reaches a maximum limit or the an early convergence is reached on the given input dataset(governed by a user-specified threshold).

In section 2.1.1 to section 2.1.3, we dive deep into the selection, mutation and execution steps of the generational GP agorithm. In section 2.1.2, we define and implement 4 different possible types of mutation along with reproduction.

Algorithm 1 The Generational GP Algorithm

```
 1: procedure GP-FIT(dataset)                    ▷ Function to train a GP system
 2:     curr ←Initialize population      ▷ Stochastically initialized population
 3:     EVALUATE(curr, dataset)
 4:     repeat
 5:         next ← SELECT(curr)
 6:         next ← MUTATE(next)
 7:         EVALUATE(next, dataset)
 8:         curr ← next
 9:     until user defined termination criteria on curr not met
10:     return curr                           ▷ The final generation of programs
11: end procedure
```

Before talking about selection, we first talk about initialization methods for the first generation programs. The following 3 methods are some of the standard initialization methods [9],[11]

- **Full initialization** - All trees in the current generation are "dense", that is, only leaf nodes are terminals(variables or constants).
- **Grow initialization** - All trees grown in the current generation need not be "dense", and some of the non-leaf nodes can also be terminals.
- **Ramped half and half** - Half of the population trees are initialized using the Full method, and the other half is initialized using the Grow method. This is the most common method used for initializing programs.

In our code, we provide support for all 3 initialization methods.

2.1.1 Selection

Selection is the step where individual candidates from a given population are chosen for evolution into the next generation. Some commonly used selection schemes are as follows [7]:

- Tournament selection - Winning programs are determined by selecting the best programs from a subset of the whole population(a ”tournament”). Multiple tournaments are held until we have enough programs selected for the next generation.
- Proportionate selection - Probability of candidate program being selected for evolution is directly proportional to it's fitness value in the previous generation
- Ranking selection - The population is first ranked according to fitness values, and then a proportionate selection is performed according to the imputed ranks.
- Genitor(or ”steady state”) selection - This retains the programs with high fitness even in the next generation. However, the programs with low fitness are replaced with mutated versions of the ones with higher fitness.

In this thesis, we focus only on tournament selections, and impelement a parallelized tournaments using a CUDA kernel.

2.1.2 Mutation

As discussed earlier, the winning programs after selection are not directly a part of the next generation. Rather, we apply genetic operations on them to produce new offspring for the next generations. Some commonly used mutations are([21])

- Reproduction
- Point mutations
- Hoist mutations
- Subtree mutations

- Crossover mutations

In our code we provide support for all the above listed genetic operations, along with a modified version of the crossover operation to account for tree depth.

We now explain and visualize each mutation type in detail.

Reproduction

As the name suggests, the current winning program is cloned as a part of the next generation.

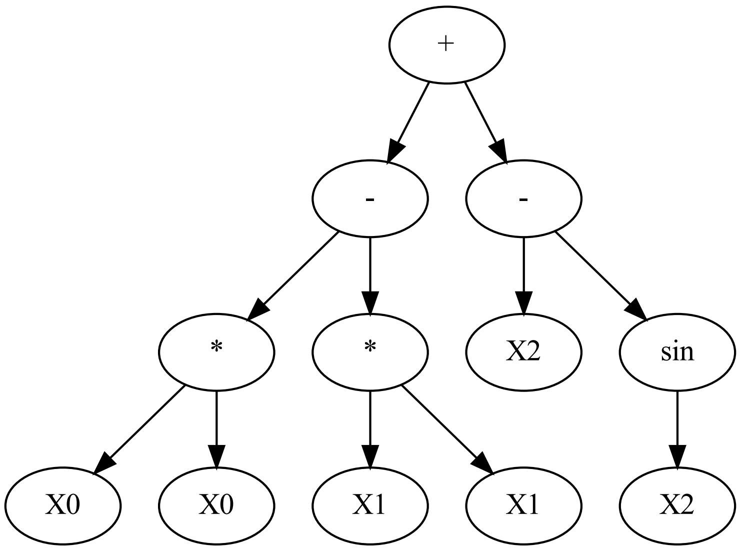
Point Mutation

In a point mutation, we take the winner of a tournament and replace random nodes from it. For symbolic mathematics, we replace terminals(variables or constants) with terminals and functions with another function of the same arity(number of inputs to the function).

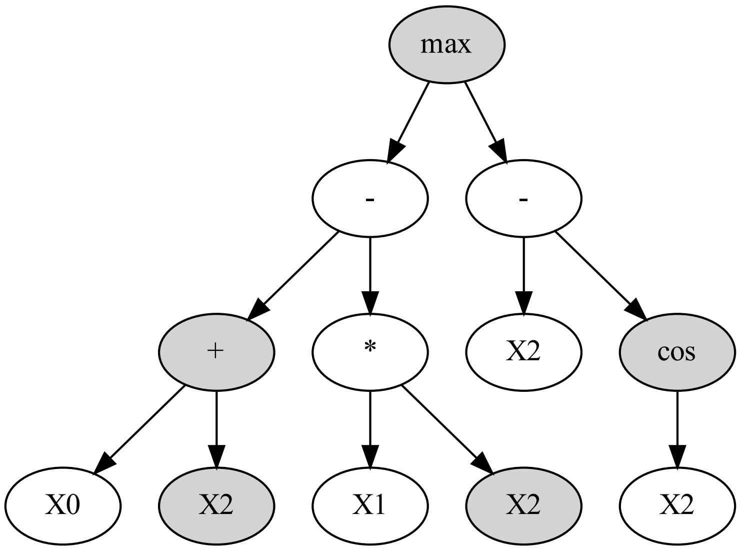
This mutation has the effect of reintroducing extinct functions and variables into the population and helps maintain diversity[21]. In our code, the amount of replacement to be done is governed by a per-node probability parameter for replacement. Figure 2.1 visualizes the effect of point mutations on a given program.

Hoist Mutations

In hoist mutations, we take the winner of a tournament and selects a random subtree from it. Another random subtree is selected from this subtree as a replacement for the subtree from the program.



(a) The original expression tree before performing a point mutation



(b) The same expression tree after a point mutation. The shaded nodes represent the changed terminals and functions in the final program.

Figure 2.1: Visualizing point mutations. Note that here, the program is the expression tree itself.

This mutation serves to reduce bloating of programs with increase in number of generations. Figure 2.2 visualizes the effect of hoist mutations on a given expression tree.

Crossover Mutations

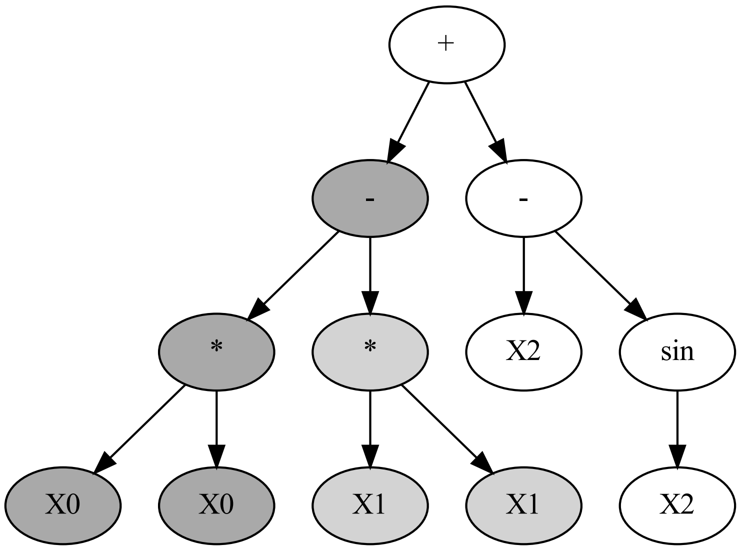
Crossover is a method of mixing genetic information between 2 programs from a given population. In crossover mutations, we first determine a parent and a donor program using 2 seperate tournaments. We then select a random subtree of the parent program. The child program for the next generation is generated by replacing this subtree with a random subtree from the donor program.

Crossover is considered as the principle method for evolution in our implementation, which is again inspired by [21]. We show a sample visualization of the crossover operation in fig. 2.3.

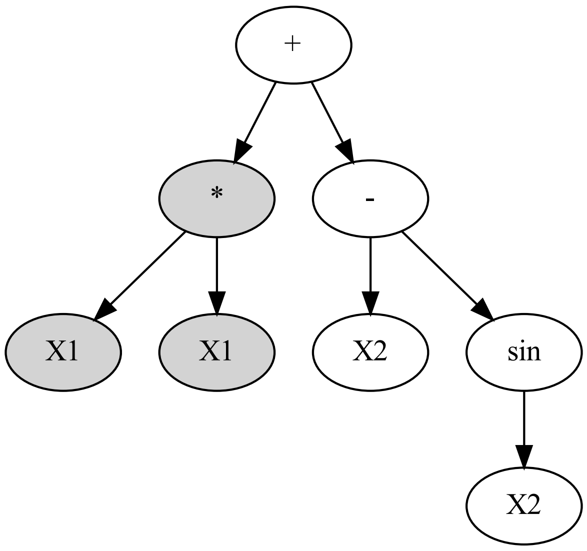
Subtree Mutations

In subtree mutations, we perform a crossover between the winning parent tree and a randomly generated program. It serves as a method to increase new terminals and extinct functions in the next generation of programs, and is more agressive compared to a crossover mutation.[21]

Figure 2.4 visualizes a sample subtree mutation in practice. We note that it is almost the same as the crossover visualization in Figure 2.3, with the exception being that the donor tree in fig. 2.3a is now a randomly generated tree in fig. 2.4a.

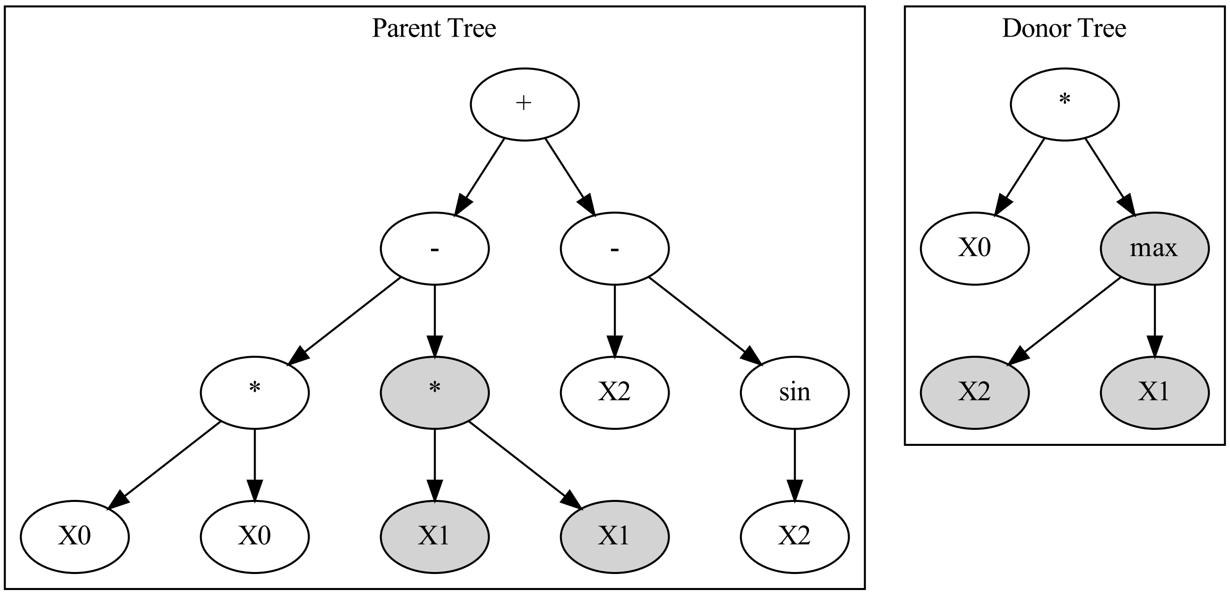


(a) The original expression tree before performing a hoist mutation. The dark grey nodes denote the selected subtree, and the grey nodes denote the hoist subtree.

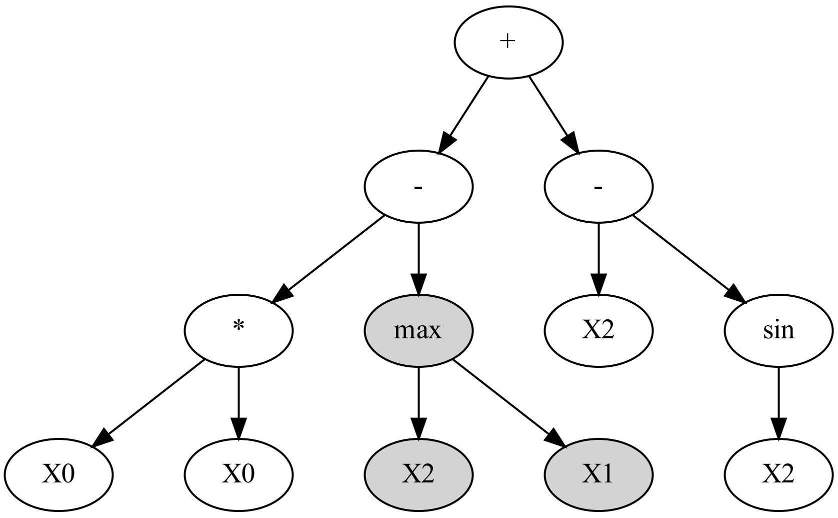


(b) The same expression tree after hoisting the sub-subtree onto it's parent's position.

Figure 2.2: Visualizing hoist mutations for an expression tree.

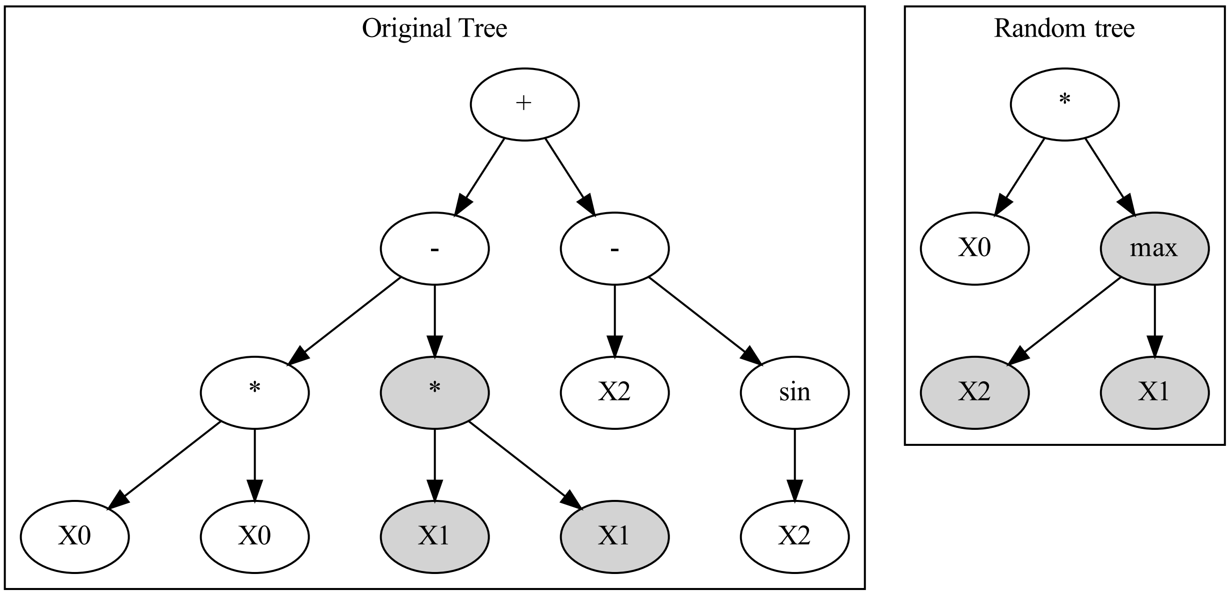


(a) The parent and donor expression trees, both selected through tournaments are shown here.

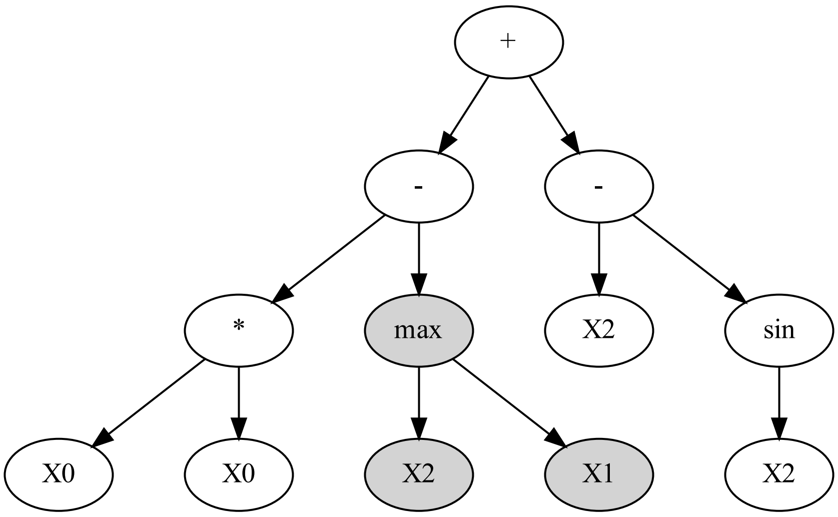


(b) The child expression tree after replacing a subtree of the parent with that of the donor.

Figure 2.3: Visualizing crossover mutations for a given parent and donor program.



(a) The parent expression tree and a randomly generated tree is shown here.



(b) The child expression tree after replacing a subtree of the parent with that of the random tree.

Figure 2.4: Visualizing subtree mutations for a given parent program.

2.1.3 Evaluation

Once the population for the next generation is decided after selection and mutation, the fitness of all programs in the new generation is recomputed. This step is the major bottleneck when scaling the generational GP algorithm to bigger datasets and bigger populations, as the evaluation is independent for every program and every row of the input dataset.

We perform the evaluation and computation of fitness in our implementation using a CUDA kernel and linear algebra primitives specified in the RAFT library([15]).

2.2 Existing Libraries and Their Coverage

The table below(mostly reproduced from [2]) lists some of the common libraries used for genetic programming.

Table 2.1: Existing GP frameworks along with language and device support

Framework	**Language**	**Compute Type**
KarooGP	python	CPU/GPU
TensorGP	python	CPU/GPU
DEAP	python	CPU
gplearn	python	CPU
ECJ	Java	CPU

Among the libraries listed in table 2.1, both TensorGP([3]) and KarooGP([20]) use the TensorFlow python framework to perform fitness evaluation the GPU. However, while KarooGP uses TensorFlow's *graph* execution model, where every program is compiled into a DAG wheras TensorGP uses TensorFlow's *eager* execution model[1], where expressions are evaluated immediately without the overhead of graph construction. The GPU parallelization here is through the use of TensorFlow-based vectorization, and not the explicit use of CUDA.

DEAP [6] is another commonly used Evolutionary Computing framework in Python implements a parallelized version of the GP framework. However, it offers only CPU based parallelization.

GPlearn[21] is another Python framework which provides a method to build GP models for symbolic regression, classification and transformation using an API which is compatible with scikit-learn[12]. It also provides support for running the evolutionary process in parallel. The base code that is parallelized on GPUs in this thesis is largely inspired by GPlearn.

The ECJ Evolutionary Computing Toolkit[10] is a Java library for many popular EC algorithms, with a emphasis towards genetic programming. Almost all aspects of algorithm 1 are governed by a hierarchy of user-provided parameter files and classes, and the framework itself is designed for large, heavyweight experimental needs. Using ECJ parameter files, it is also possible to define custom GP pipelines with user-defined evolution strategies.

In the next chapter, we will talk about our work on parallelizing the GP algorithm to perform symbolic regression.

CHAPTER 3

OUR WORK

This chapter contains the implementation details of our parallelized algorithm to perform genetic programming using CUDA. We first talk about a way to represent programs on the GPU. This is then followed by a description of the device side data structures used. We then give an overview of our modified GP algorithm, describing GPU-side optimizations for the selection, and evaluation step in detail. We also include details about the fitness computation step which comes after the evaluation step. Finally, we talk about the various challenges faced during the implementation of the modified algorithm, and the workarounds to avoid these problems.

In this implementation, we use a fixed list of functions with a maximum arity of 2. We also assume that the maximum depth of all expression trees is a constant. This assumption is needed in order to evaluate the trees in the GPU using a fixed size stack.

3.1 Program Representation

We define a struct for a program containing the following components.

- A array of operators and operands of the underlying expression tree stored in Polish(prefix) notation.
- A length parameter corresponding to the total number of nodes in the expression tree.

- A raw fitness parameter containing the score of the current tree on the input dataset.
- The depth of the current expression tree.

The entire population for a given generation is thus stored in the Array of Structures(AoS) format. The evaluation using a stack is almost similar to the way the Push3 system [18] evaluates GP programs, with the sole exception being the reverse iteration due to the prefix notation chosen for the trees.

Prefix notation was used for the representation of nodes to aid with the process of generating random programs, where we directly generate a valid prefix tree on the CPU.

3.1.1 Device Side Data Structures

In order to perform tournament selection and evaluation on the GPU, we use the following device side data-structures.

- **Philox Random Number Generator(RNG)** - We use a Philox counter-based RNG[16] implemented in the **raft** library [15] to generate random global indices for tournament selection inside the selection kernel.
- **Fixed size device stack** - We define a fixed size stack using a custom class, implementing the *push*, *pop* methods as inline **__host__ __device__** functions. To avoid global memory accesses and encourage register lookups for internal stack slots, the push and pop operation is implemented using an unrolled loop over all the available slots. This action is possible because the maximum size of the stack is fixed at 20 for now. We shall discuss more about this in section 3.3, where we examine how to avoid thread divergence in this setup.

Our kernels for both selection and program execution have been written in a way so as to eliminate any need for thread or memory synchronization.

3.1.2 Memory Footprint

We examine the memory footprint of our implementation with respect to the number of programs stored in both CPU and GPU memory below.

- **CPU** - Depending on a user-specified flag, we maintain a history of all the programs for all generations. If this is not desired, then we only store information about 2 generations, the current and the next generation until the end.
- **GPU** - Only the device memory corresponding to the current and the next generation of programs is stored through the run of the algorithm. During the course of the algorithm, the 2 memory locations are updated with new programs in a ping pong fashion.

We will now explain our code for implementing parallel GP in section 3.2.

3.2 Our Parallel GP Algorithm

In this section, we again outline the individual steps of Algorithm 1, defined in Section 2.1. However, each step contains details specific to our implementation. In this implementation, the selection and evaluation step is performed on the GPU, wheras mutations are carried out on the CPU.

Before performing any of the standard steps, we decide on the type of mutation through which the next generation program is produced. This mutation type selection is governed by user defined probabilities for the various types of mutations. This step is important, as we need to determine the exact number of selection tournaments to be run(as crossovers require 2 tournaments to decide the parent and donor trees).

Once exact number of tournaments has been decided, we move on to the selection step.

3.2.1 Selection

We perform selection by running parallel tournaments in a CUDA kernel. For a population of size n and tournament subset size k, we launch a CUDA kernel with $\lfloor n/256 \rfloor$ blocks and 256 threads.In each thread, we generate k random program indices using a Philox RNG(see section 3.1.1).

The index of the best program among the k indices (based on the raw fitness values of the device programs) is then chosen as the winning index for the given thread, and is recorded in a win_indices array.

3.2.2 Mutation

The mutation of programs takes place on the CPU itself. We implement all the mutations mentioned in section 2.1.2 with a slight modification to the crossover operation in order to constrain the depth of the output tree. We call this modification a hoisted crossover.

Hoisted Crossover

In this mutation, we initially perform a crossover between the parent tree and the donor tree. The selected stubtree of the donor is then repeatedly hoisted onto the parent tree until the depth of the resultant tree is less than the maximum evaluation stack size.

Note that the hoist mutation occurs only on the donor subtree, since that is the part of the tree which contributes to the depth violation. If this was not the case, then another subtree of the parent subtree would have contributed towards the max depth violation, making the parent program itself an invalid one.

This modification is necessary for our code since a stack of fixed size m can only evaluate a tree of depth $m - 1$(assuming the maximum function arity is 2)

At the end of mutations, we allocate and transfer the newly created programs onto GPU memory, in order to evaluate them on the input dataset. In order to save on device memory, we also deallocate the GPU memory of the previous population trees. Some of the challenges we faced due to the nested nature of programs during the **cudaMemcpy** operations between host and device memory are listed in section 3.3.

3.2.3 Evaluation

We divide the evaluation portion into 2 steps, namely execution and metric computation. In the execution step, all programs in the new population are evaluated on the given dataset, to produce a set of predictions. Once we have the predicted values for all programs, we compute the raw fitness value of the every program with respect to the expected output using the user-defined metric. These steps are explained in detail below.

Execution

For every program in the population and every sample in the input dataset, we produce predicted output values in this step. If n is the population size, and m is the number of samples in the input dataset, then we launch an execution kernel with a 2D grid of dimension $(\lceil m/256 \rceil, n)$ with 256 threads per block. Each thread has it's own device side stack which evaluates a program on a specific row.

We note here that to avoid thread divergence in the execution kernel, it is important to ensure that each block of threads execute on different rows of the same program. We provide more details about this in section 3.3.

Fitness Metric Computation

In the previous step, after running the execution kernel, for all programs in the population, we have a list of predicted values. We also have access to the list of actual outputs.

For every program, we compute fitness using a user-selected loss function. The inputs to the loss function are the program's output values(from the execution step) and the actual outputs.

Since the fitness computation is the same for all programs, we computed loss for all programs in parallel on the GPU. This was implemented using the matrix-vector linear algebra primitives present in the **raft** library[15], where the vector of actual output values is reused for all predicted values of different programs. We implement a weighted version of the following 6 standard loss functions:

- Mean Absolute Error (MAE)
- Mean Square Error (MSE)
- Root Mean Square Error (RMSE)
- Logistic Loss(binary loss only)
- Karl Pearson's Correlation Coefficient
- Spearman's Rank Correlation Coefficient

During the implementation of Spearman Rank Correlation, we used the **thrust** library from Nvidia to generate ranks for the given values. The default fitness function is set as MAE, in an effort to be consistent with **gplearn** [21]

3.3 Challenges Faced

In this section, we detail in brief some of the challenges faced during the implementation of our parallel GP algorithm.

3.3.1 Thread divergence and Global Memory Access

In the execution kernel, during the evaluation phase, when evaluating a stack node, we check for equality of the current function with 33 pre-defined functions, using **if−else** conditions on the node type.

Since CUDA executes statements using warps of 32 threads in parallel, when it encounters an **if−else** block inside a kernel which is triggered only for a subset of the warp, both the **if** and **else** block are executed by all threads. During the execution of the **if** block, the threads which don't trigger the **if** condition are masked, but still consume resources, with the same behaviour exhibited for the **else** block. This increases the total execution time as both blocks are effectively processed by all warp threads.

To avoid this behaviour in our code, we ensure that within every thread-block of the execution kernel, all threads execute the same program. This ensures that all threads in a warp will always parallely execute the same nodes, and thus avoid divergence during a single stack evaluation.

In the implementation of **push** and **pop** operations for the device side stack, we avoid trying a possible dereference using global memory index(the current number of elements in the stack) by using an unrolled loop for stack memory access. This is again safe from thread divergence because we ensure that within a thread-block, all threads evaluate the same program.

3.3.2 Data Transfers and Memory Allocation

Since we went with a AoS representation for the list of programs and each program has a nested pointer for the list of nodes in it, we are forced to perform atleast 2 **cudaMemcpy** operations per program in a loop spanning the population size. One of the copy operations is for the list of program nodes, and the other copy is to capture the metadata about the nodes, and the other copy is for the program struct itself.

However, during our experiments, we achieved a considerable speedup despite this step. Thus, the implementation was not altered.

In the next chapter, we will present the experiments and benchmarks run on the code, and present our results.

CHAPTER 4

EXPERIMENTS

CHAPTER 5

CONCLUSION AND FUTURE WORK

REFERENCES

[1] **Agrawal, A.**, **A. N. Modi**, **A. Passos**, **A. Lavoie**, **A. Agarwal**, **A. Shankar**, **I. Ganichev**, **J. Levenberg**, **M. Hong**, **R. Monga**, and **S. Cai** (2019). Tensorflow eager: A multi-stage, python-embedded dsl for machine learning.

[2] **Baeta, F.**, **J. Correia**, **T. Martins**, and **P. Machado** (2021). Speed benchmarking of genetic programming frameworks.

[3] **Baeta, F.**, **J. Correia**, **T. Martins**, and **P. Machado**, Tensorgp – genetic programming engine in tensorflow. *In Applications of Evolutionary Computation - 24th International Conference, EvoApplications 2021*. Springer, 2021.

[4] **Biles, J. A.**, Autonomous genjam: eliminating the fitness bottleneck by eliminating fitness. *In Proceedings of the GECCO-2001 workshop on non-routine design with evolutionary systems*, volume 7. Morgan Kaufmann San Francisco, CA,, USA, 2001.

[5] **Dua, D.** and **C. Graff** (2017). UCI machine learning repository. URL `http://archive.ics.uci.edu/ml`.

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

[7] **Goldberg, D. E.** and **K. Deb**, A comparative analysis of selection schemes used in genetic algorithms. volume 1 of *Foundations of Genetic Algorithms*. Elsevier, 1991, 69–93. URL `https://www.sciencedirect.com/science/article/pii/B9780080506845500082`.

[8] **Harding, S.** and **W. Banzhaf**, Fast genetic programming on gpus. *In* **M. Ebner**, **M. O'Neill**, **A. Ekárt**, **L. Vanneschi**, and **A. I. Esparcia-Alcázar** (eds.), *Genetic Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-71605-1.

[9] **Koza, J. R.**, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992. ISBN 0-262-11170-5.

[10] **Luke, S.** (1998). ECJ evolutionary computation library. Available for free at http://cs.gmu.edu/~eclab/projects/ecj/.

[11] **Luke, S.** (2000). Two fast tree-creation algorithms for genetic programming. *IEEE Transactions on Evolutionary Computation*, **4**(3), 274–283. URL `http://citeseer.ist.psu.edu/409667.html`.

[12] **Pedregosa, F.**, **G. Varoquaux**, **A. Gramfort**, **V. Michel**, **B. Thirion**, **O. Grisel**, **M. Blondel**, **P. Prettenhofer**, **R. Weiss**, **V. Dubourg**, **J. Vanderplas**, **A. Passos**, **D. Cournapeau**, **M. Brucher**, **M. Perrot**, and **E. Duchesnay** (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, **12**, 2825–2830.

[13] **Perkis, T.** (1994). Stack-based genetic programming. *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence.*

[14] **Poli, R.**, **W. B. Langdon**, and **N. F. McPhee**, *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. URL `http://www.gp-field-guide.org.uk`. (With contributions by J. R. Koza).

[15] **Raschka, S.**, **J. Patterson**, and **C. Nolet** (2020). Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence. *arXiv preprint arXiv:2002.04803*.

[16] **Salmon, J. K.**, **M. A. Moraes**, **R. O. Dror**, and **D. E. Shaw**, Parallel random numbers: As easy as 1, 2, 3. *In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11. Association for Computing Machinery, New York, NY, USA, 2011. ISBN 9781450307710. URL `https://doi.org/10.1145/2063384.2063405`.

[17] **Schaul, T.** and **J. Schmidhuber** (2010). Metalearning. *Scholarpedia*, **5**(6), 4650.

[18] **Spector, L.**, **J. Klein**, and **M. Keijzer**, The push3 execution stack and the evolution of control. *In Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, GECCO '05. Association for Computing Machinery, New York, NY, USA, 2005. ISBN 1595930108. URL `https://doi.org/10.1145/1068009.1068292`.

[19] **Staats, K.** (2016). *Genetic programming applied to RFI mitigation in radio astronomy*. Master's thesis, University of Cape Town.

[20] **Staats, K.**, **E. Pantridge**, **M. Cavaglia**, **I. Milovanov**, and **A. Aniyan** (2017). Tensorflow enabled genetic programming.

[21] **Stephens, T.** (2015). gplearn: Genetic programming in python, with a scikit-learn inspired api. URL `https://github.com/trevorstephens/gplearn`.