Semantic Genetic Programming on the GPU

João David[1] and Ye Yang[1]

Departamento de Informática da Faculdade de Ciências da Universidade de Lisboa
{fc49448,fc49521}@alunos.fc.ul.pt

Abstract. Semantic programming is mainly a way to reach a target value, given some input, the way the input
The first sentence of an abstract should clearly introduce the topic of the paper so that readers can relate it to other work they are familiar with. However, an analysis of abstracts across a range of fields show that few follow this advice, nor do they take the opportunity to summarize previous work in their second sentence. A central issue is the lack of structure in standard advice on abstract writing, so most authors don't realize the third sentence should point out the deficiencies of this existing research. To solve this problem, we describe a technique that structures the entire abstract around a set of six sentences, each of which has a specific role, so that by the end of the first four sentences you have introduced the idea fully. This structure then allows you to use the fifth sentence to elaborate a little on the research, explain how it works, and talk about the various ways that you have applied it, for example to teach generations of new graduate students how to write clearly. This technique is helpful because it clarifies your thinking and leads to a final sentence that summarizes why your research matters.

Keywords: First keyword · Second keyword · Another keyword.

1 Introduction

The introduction should set the context for your project. Why is this topic relevant?

You should also define the scope of your project. You could design a software artifact that would end poverty and famine, but that is not realistic.

For example, this document describes the structure your paper should have. Despite using the LNCS LaTeX template [1], the formatting template is not relevant, only the content structure is relevant.

Finally, you should define the goals of your project. For instance,

- To propose a method for the parallelization of Genetic Algorithms
- An implementation of such algorithm
- The experimental evaluation of such method, with comparison with a sequential alternative.

[1] LNCS is the official template for Europar 2019, in case you are interested.

2 Background

To understand the underlying principles of the implementation we first have to introduce the main concepts.

- The semantic syntax our work will be based on will be presented as a tree structure, each node containing either a mathematical operator, a constant or a variable which can be swapped by a value from a given data set.
- To start off the program, we must introduce an initial data set. The data set will be read as a text file, each line of the text file containing randomly generated values and a final target value. These random data set values are to be added to trees that contain variables, each variable having its own index to determine which random value to swap the variable by. After all the necessary swaps are made, the final value is computed for each line of the data set and the fitness of the syntax tree is determined by calculating the Mean Squared Error (MSE) between the calculated values (O) and the target values (T):

$$\frac{1}{n}\sum_{i=1}^{n}(O_i - T_i)^2$$

3 Approach

The main genetic algorithm to crossover different trees of computation takes into account the following variables:

- $N \rightarrow$ the total number of trees
- $gen \rightarrow$ the current generation cycle
- $threadId \rightarrow$ the ID of a GPU thread

Given the initial data set, we first have to run the values through all the generated trees, after which we compute the respective fitness values for every single tree and store it in a matrix. We then place the indexes of each tree in the first row of a $N * gen$ sized matrix. Each column represents a concatenation of trees by summation, the trees are represented by their indexes from 0 to N-1.

The algorithm works as follows: for every even generation cycle, the GPU threads with even IDs will look for the best fitness only within the even numbered indexes on the matrix. The GPU threads with odd numbered IDs will do the same but on odd indexes. When the best fitness is found, the index of the column to which it belongs will be appended to that generation cycle (or row).

For every odd generation cycle, the GPU threads within the first half of the X axis in the tree index matrix will look for the best fitness within the first half of the matrix, the rest of the threads will search the other half. To each half of the row of the current generation cycle, the index of best fitness of the respective half will be appended.

Given this alternating crossover algorithm, we can achieve a less linear evolution from generation to generation of each sequence. At Algorithm 3 we can see a pseudo code representation of the algorithm.

Algorithm 1 Tree crossover

```
1: procedure TreeCrossover
2:     int max
3:     if gen % 2 == 0 then
4:         for i = threadId % 2; i < N; i+ = 2 do
5:             find highest fitness and store in max
6:     else
7:         if threadId < N/2 then
8:             for i = 0; i < N/2; i + + do
9:                 find highest fitness and store in max
10:        else
11:            for i = N/2; i < N; i + + do
12:                find highest fitness and store in max
13:    return max
```

4 Implementation Details

Before any values could be first calculated, we needed some form of input to compare values with. This lead to the implementation of a data set parser. The data set is a text file in which all the variable and target values are defined. Each line has an equal amount of values Y written, the last of which corresponds to the target value. Both variable values and target values are then extracted and stored in separate arrays which will be used to calculate the fitness for the first generation.

To store and calculate the values, a set number of random syntax trees have to be initially generated. Each node of the tree as previously mentioned can be a literal value, a variable or a mathematical operator. After generating the trees, each one of them must be run through the data set to evaluate its fitness by calculating the mathematical operations defined and comparing it with the target values. The computation is handled as a Reverse Polish Notation calculator (RPN) according to the following procedures:

- A seperate LIFO stack to store literal values and variables is created
- The tree is traversed in postorder so that the root operation or value is read last
- If either a literal or a variable are present, they are pushed into the stack
- If a mathematical operation is read, 2 values are popped out of the stack for computing, and the resulting value is pushed back into the stack
- At the end of the processing, the final value is extracted from the stack and stored in a result matrix of size

$$N_{trees} * dataset_{rows}$$

with each row representing the calculated values for a given data set row of a tree and each column the resulting values to compare with the target values.

The main parallelization applied to the algorithm is done by taking advantage of the much higher amount of threads on a GPU compared to a CPU. In order to better take advantage of all the computational power on the GPU we first have to copy all the data from CPU memory to GPU memory by first reserving memory for the array by using cudaMalloc(array,size) and then copying the data currently residing on CPU memory to the newly GPU memory allocated array using cudaMemcpy(`o_array,t_array,size,cudaMemcpyHostToDevice`) , the keyword cudaMemcpyHostToDevice signals that the array from which the contents are to be copied (`o_array`) is residing on CPU memory and the target array to which the values are to be copied to (`t_array`) on the GPU memory.

5 Evaluation

5.1 Experimental Setup

In this section you should describe the machine(s) in which you are going to evaluate your system. Select the information that is relevant.

5.2 Results

In this section you should present the results. Do not forget to explain where the data came from.

You should include (ideally vectorial) plots, with a descriptive caption. Make sure all the plots (Like Figure **??** are well identified and axis and metrics are defined.

5.3 Discussion

Here you should discuss the results on a high level. For instance, based on our results, the parallelization of the merge-sort is relevant as no other parallel work occurs at the same time, and the complexity $O(Nlog(N))$ can have a large impact when the number of individuals is high.

6 Related Work

This section can be either the second one, or the second-to-last. In the case where knowledge of other competing works is required, it could come before. But if you are confident on what you did, it should appear at the end, where you can compare existing works against yours. An example is below:

Chu and Beasley proposed a Genetic Algorithm for the Multidimensional Knapsack Problem [**?**]. This work introduces a heuristic as a repair operator. Our work makes no optimization with regards to the problem domain, making it more generic and supporting other problems.

When using BibTeX references, I suggest using DBLP[2], leaving Google Scholar as a backup, since DBLP has more correct and detailed information about research papers.

[2] https://dblp.org

7 Conclusions

Here you should resume the major conclusions taken from discussion. Ideally, these should align with the objectives introduced in the introduction.

You should also list the future work, i. e., tasks and challenges that were outside your scope, but are relevant.

Acknowledgements

First Author wrote the part of the program implemented the phasers. Second Author implemented the MergeSort in parallel.

Both authors wrote this paper, with First Author focusing on the introduction, related work and conclusions while the Second Author focused on approach and evaluation.

Each author spent around 30 hours on this project.