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 Genetic programming aims to not only syntactically manipulate the data to promote a better fitness evaluation, but also ensuring that the resulting function is semantically correct. Traditional implementations of genetic programs have always been done on CPU, taking advantage of its multiple cores to do parallel work. We aim to improve on the parallelization of the genetic algorithm by implementing it on the Graphics Processing Unit (GPU), taking advantage of its vast amount of processing cores, which are multiple times that of the CPU. The implementation will be done on an NVidia GPU allowing the use of the Compute Unified Device Architecture (CUDA) API, which grants us access to directly manipulate the GPU memory and threads necessary to perform fitness evaluation and crossover.

Keywords: Semantic Genetic programming · GPU · CUDA.

1 Introduction

Traditional implementations of Semantic Genetic programming are on CPU, usually with syntax trees that contain elements of the domain in study, whether it be logical operations, mathematical operations, among others.

These trees are then processed given some input, and the output is compared to some target value to calculate its fitness. Fitness calculation algorithms are generally implemented on CPU, in this paper we aim to vastly improve the time required computing and evaluating fitness values by implementing an algorithm on GPU, taking advantage of its many cores and threads which allows for a vast amount of parallel computation to be done, when compared to a CPU.

While modern CPU's number of cores and threads round either the single or double digits, a GPU contains thousands of both of them. This paired with quick access, cache like, shared memory we can vastly improve execution times given large data set inputs.

In section 2 we explain some of the main concepts needed to understand the parallel implementation of fitness testing, in section 3 we discuss the general approach taken to parallelize fitness evaluation, which can be applied to any programming. In section 4 we explain in more detail our implementation which is written in C++ with a CUDA extension, allowing us to fully utilize all the functionality provided by the API. In section 5 we compare test results from both a parallel and sequential implementation of the fitness evaluation algorithm. In section 6 we discuss previous work done on the subject matter and compare it with our own implementation and finally we summarize all the findings in the paper and future work to be done in section 7.

With this paper we aim to:

- propose a parallel algorithm implemented on the GPU for fitness calculation and evaluation and genetic crossover
- present an implementation of the algorithm in CUDA
- compare execution times of the implemented algorithm with a sequential CPU version

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.
- After passing data from a data set through a tree, the output is then compared to the target value by calculating the Mean Squared Error (MSE) which determines the tree's fitness:

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

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 allocating memory for the array using cudaMalloc(`t_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 in CPU memory and the target array to which the values are copied to (`t_array`) is residing in the GPU memory. These are necessary steps since neither the GPU nor the CPU have direct access to each others memory region.

To execute a method on the GPU, we have to execute a kernel call. The kernel call can receive up to 3 parameters between the `<<< >>>` brackets. The first two parameters determine the number of blocks and threads per block respectively and the last one determines the amount of shared memory between threads available (threads can only access shared memory content within the same block) in bytes, when omitted it defaults to 0.

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

Each node on a syntax can hold a literal value, a variable or a mathematical operation. When processing a tree through a data set, each variable will be swapped by the corresponding value in the data set according to the variables index. In Figure 1 we can see the tree representation of the mathematical formula $x0 * (x1 + 4)$ which is then processed according to a data set line containing the values: $(0.5 \quad 25.3 \quad 6.2 \quad 12)$. The values within the data set that are not referenced by any variable in the tree are ignored, the final value computed is compared with the target value (12) which will evaluate the fitness of the tree.

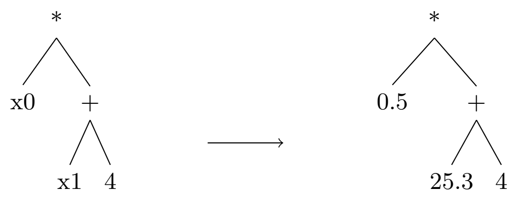

Fig. 1. Syntax tree value swap example

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 1 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 variables 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 of trees.

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 separate 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 popped 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 by a syntax tree and each column the resulting values to compare with the target values.

4.1 GPU Parallelization

After all the values are calculated and stored in the array, we then proceed to the computation of the fitness for all trees for the first generation. In Figure 2 we present an example of fitness calculation with a data set of 4 lines, meaning that each tree computes 4 different values which are compared to the 4 target values to determine the MSE and consequently the fitness of the tree. Each tree is processed by a block (an abstract representation of a group of executing threads) which will have its own ID that can be extracted with the built in keyword *blockIdx.x*. The *.x* termination means we are extracting the ID along the x axis, since the kernel is launched in 1 dimension, both the y and z values are 0.

$V_{i,j}$ is the value calculated by a tree of ID i on line j of the data set and T_j the respective target value. Each $(V_{i,j} - T_j)^2$ value is calculated by a single thread which is able to compute the correct index on the array by calculating its position relative to its own ID (*threadIdx.x*) and the block's ID in the following manner: array$[(blockIdx.x * num_rows) + threadIdx.x]$.

After all values are calculated we then proceed to a reduction phase, in this phase the amount of threads working on adding up all the values is reduced by half with each step until it reaches 1, which will be the final value to be divided by `num_rows` which will equal to the MSE of the computed value and the target value i.e. the fitness of the tree. Since all blocks work in parallel, the complexity of the reduction algorithm is $O(log(n))$.

0	1	2	3	4	5
2	2	2	2	2	2

5 Evaluation

5.1 Experimental Setup

Processor	CPU Cores	Threads	RAM
Intel Xeon X5670	12	24	24GB

Graphics Card	GPU Cores	Threads	Memory
NVidia GTX 960	1024	24	4GB

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

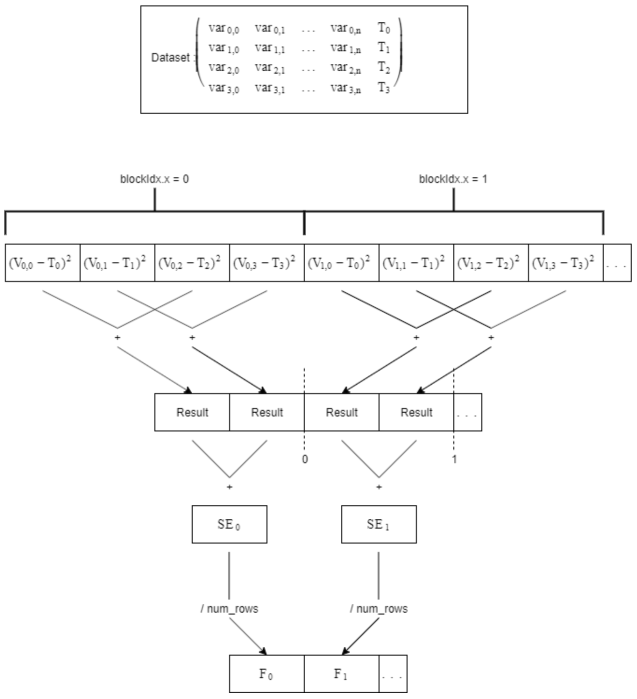

Fig. 2. Fitness computation for a data set of 4 lines

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

Alberto Moraglio, Krzysztof Krawiec, and Colin G. Johnson introduced a new way of genetic programming, Geometric Semantic Genetic Programming (GSGP) which consists in changing the semantics of the functions directly. Their work in regards to fitness evaluation, tree mutation and crossover is far more complex than ours given that we only accounted for mutation by concatenation with sums of trees. However the implementation of the genetic algorithm was done on CPU which makes performance considerably slower if it were implemented on GPU.

7 Conclusions

In this paper, we proposed a parallel GPU algorithm to both calculate and evaluate fitness on a genetic program, by applying 2 different crossover techniques. We concluded that GPU parallelization significantly benefits the execution time when passing from one generation to the next, by dynamically distributing work across the many GPU threads available. However, the size of the initial data set must be taken into account, as smaller data sets may be quicker to compute on the CPU due to the significant overhead when transitioning from CPU memory to GPU memory. Thus we must define a cutoff mechanism between CPU and GPU execution which can be determined in future tests.

We presented a simple crossover algorithm when passing from one generation to the next, in future work we would like to implement tree mutations and crossover between trees rather than the fitness calculated with each passing generation. The implementation of a CPU parallel version of the algorithm would also compliment the results obtained in this paper, so a comparison between the two parallel frameworks can be made.

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.