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, also known as Semantic GP, is a type of Evolutionary Algorithm (EA), a subset of machine learning. Aiming to not only syntactically manipulate the data and 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: Genetic Programming · GPU · CUDA · Machine Learning.

1 Introduction

Traditional implementations of Semantic GP are computationally heavy and are usually represented by 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. 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.

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 implementations of the 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:

- To propose a parallel algorithm implemented on the GPU using the CUDA API
- Compare execution times of the implemented algorithm with a sequential CPU version

2 Background

Genetic Algorithms are inspired by Nature and its biological mechanisms, take the following analogy as an example. Imagine that you are in the horse racing business, and you own a set of horses (population), some horses are better at different skills (speed, resistance, strength, etc.). When selecting horses to produce offspring (crossover and mutation), you must select them based on their fitness, which is calculated using a fitness function. In this case, it could be the average lap time from a set of tracks, better fitness values would mean lower average lap times.

The semantic syntax our work will be based on will be presented as a tree structure, each node containing either a mathematical operator, a literal or a variable which can be swapped by a value from a given data set. The root must always be an operator, and the leaf nodes a literal or a variable. When processing a tree through a data set row, each variable will be swapped by the corresponding value in the data set according to the variable's column 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 row 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.

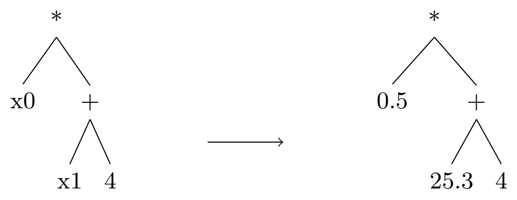

Fig. 1. Syntax tree value swap example

An NVIDIA GPU is composed of several SM (Stream Multiprocessor), each SM having a certain amount of cores, every core contains a number of warps (groups of 32 threads). The processing in an NVIDIA GPU is done through the use of blocks, which are composed of a set amount of threads. Each block will be scheduled to a single SM, and every SM can schedule different blocks at a given time if it has enough resources. When choosing the number of threads within a block, it is important that the number is a multiple of 32. Because if it isn't, there will be threads of a warp doing nothing.

Neither GPUs nor CPUs can directly access each others memory space. In order to cooperate, it is required to have copied memory from RAM to GPU memory (GDRR5 for instance) in order for GPUs to process data. Then the processed data must be copied back to RAM, so that the CPU may access it.

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 dynamic shared memory per block, in bytes, when omitted it is assumed a static amount of shared memory or none.

3 Approach

The algorithm starts by generating N trees, and then processing them based on the data set values. For each tree and for each row, there is an output value $V_{i,j}$, where i is the tree index, and j the row index (row number - 1) in the data set file. The Fitness (F_i) for each tree is then calculated using the Mean Squared Error (MSE) and stored in an array of size N indexed by the tree index. Y_j is the target value of row's number j.

$$F_i = \frac{1}{N}\sum_{i=1}^{N}(V_{i,j} - Y_j)^2$$

After that, a matrix A[G×N] is created, where G is the number of generations, $a_{i,j}$ is the tree index with which the tree j crossed over with at generation i.

$$A_{G,N} = \begin{bmatrix} 0 & 1 & \cdots N-1 \\ a_{2,1} & a_{2,2} & \cdots & a_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{G,1} & a_{G,2} & \cdots & a_{G,N} \end{bmatrix}$$

For every odd generation cycle, trees will be divided in two groups, one group with odd index and the other with even index, trees only crossover with trees within the same group.

For every even generation cycle, trees will be divided in two groups, one group with index < N/2 and the other with index ≥ N/2, trees only crossover with trees within the same group.

The fitness array obtained from the previous iteration will be split in two groups, (based on the current generation cycle) in the same way mentioned above, the index of the best fitness from each group will be the tree index that all the trees from the same group will crossover with.

When a tree a is crossing over with a tree b, the result of said crossover will be stored in a new fitness array at a's index. The result would be:

$$newFitness[a_{index}] = oldFitness[a_{index}] + sigmoid(oldFitness[b_{index}])$$

Given this alternating crossover algorithm, we can achieve a less linear evolution from generation to generation. Algorithm 1 is a pseudo code representation of the algorithm.

Algorithm 1 Tree crossover

```
 1: procedure TreeCrossover
 2:     int[G][N] $A$
 3:     float[N] $oldFitness$
 4:     float[N] $newFitness$
 5:     for $i = 0;\ i < N;\ i++$ do
 6:         A[0][i] = i
 7:     for $gen = 1;\ gen < G;\ gen++$ do
 8:         if $gen \,\%\, 2 \neq 0$ then
 9:             int $bestFitOdd$
10:             int $bestFitEven$
11:             for $i = 0;\ i < N;\ i++$ do
12:                 find best fitness with odd index and store it in $bestFitOdd$
13:                 find best fitness with even index and store it in $bestFitEven$
14:             A[gen][odd indexes] = bestFitOdd
15:             A[gen][even indexes] = bestFitEven
16:         else
17:             int $bestFitFirstHalf$
18:             int $bestFitSecondHalf$
19:             for $i = 0;\ i < N;\ i++$ do
20:                 find best fitness with index < N/2 and store it in $bestFitFirstHalf$
21:                 find best fitness with index ≥ N/2 and store it in $bestFitSecondHalf$
22:             A[gen][index < N/2] = bestFitFirstHalf
23:             A[gen][index ≥ N/2] = bestFitSecondHalf
24:         for $i = 0;\ i < N;\ i++$ do
25:             newFitness[i] = oldFitness[i] + sigmoid(oldFitness[A[gen][i])
```

4 Implementation Details

To compute the generated trees, the mathematical operations by them defined are 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 the following order: right child - left child and root
- 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 all data set rows by a syntax tree and each

$x_{i,j}$ entry (i being the row and j being the column of the value matrix) is compared to the respective targe value in row j od the data set.

4.1 GPU Parallelization

Just like mentioned in the Approach section, after processing the trees based on the data set values, we are left with a set of $V_{i,j}$ values. We will use the GPU to perform a reduction (Figure 2), obtaining the Mean Squared Error, which will be the initial fitness value.

In Figure 2 we present an example of initial fitness calculation with a data set of 4 lines, meaning that each tree computes 4 different $V_{i,j}$ values which are compared to the 4 target values Y_j to determine the MSE and consequently the fitness of the tree. Each tree is processed by a block with the number of threads being the same as the number of lines (4 in this case) 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 index i on row index j of the data set and Y_j the respective target value. Each $(V_{i,j} - Y_j)^2$ value is calculated by a single thread.

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 ($threadId.x < 1$), which will be the final value to be divided by `num_rows` resulting in the MSE, which will be the fitness value for the tree with index equal to *blockId.x*. Since all blocks work in parallel, the complexity of the reduction algorithm is $O(log(n))$.

After calculating the initial fitness for all trees, we can then proceed to apply the GPU version of the algorithm described in Algorithm 1. Due to the way the algorithm is implemented, the correct partitioning of the array in blocks can be achieved only if the number of trees is a power of 2 and higher than the default value of threads per block.

The default value for the number of threads per block is 512 (multiple of 32), the number of blocks to be launched is then calculated by dividing the number of trees by 512 (threads per block).

We then perform a reduction using the kernel call of the method `gpu_phase_one` Each block will find the best fitness and its corresponding index in the fitness array previously calculated within the following indexes of said array:

$$[blockDim.x * blockIdx.x + 0; blockDim.x * blockIdx.x + 512]$$

After this, we need to synchronize all blocks in order to find the best fitness and its index in the fitness array by comparing the results from each block. We do this by using another kernel call `gpu_phase_two` with the same blocks/threads per block as the previous one. This kernel call will look up the values written to the auxiliary arrays `dev_fitness_aux` and `dev_fitness_index_aux` by the `gpu_phase_one` kernel call. The values are written and read from said arrays based on whether the current generation is odd or even.

For odd generation cycles, every odd *threadIdx.x* from all blocks will find the best fitness value by only looking up odd indexes of the auxiliary arrays, while even *threadIdx.x* will do the same but for even indexes of said auxiliary arrays.

If a kernel call is launched with N blocks in 1 dimension, then *gridDim.x* represents the total amount of blocks.

For even generation cycles, blocks with $blockIdx.x < gridDim.x/2$ will find the best fitness value by only looking up indexes $< gridDim.x/2$ on the auxiliary arrays, while blocks with $blockIdx.x \geq gridDim.x/2$ will do the same but for indexes $\geq gridDim.x/2$.

After finding the best fitness x and the corresponding index i, every thread of every block will write to the $A_{G,N}$ at index $current\_gen * gridDim.x * blockDim.x + index\_fitness\_global$, the index of the tree with the best fitness, and to the `new_fitness` array, the fitness of crossing over the tree at index $blockDim.x * blockIdx.x + threadIdx.x$ and i, using the fitness value of the tree at index $blockDim.x * blockIdx.x + threadIdx.x$ from the previous fitness array plus the sigmoid of the best fitness value just discovered, x.

Then a new generation will start, where the `new_fitness` will now be the previous fitness array. Once all the generations have passed, the values in the matrix $A_{G,N}$ and `new_fitness` will be copied to RAM.

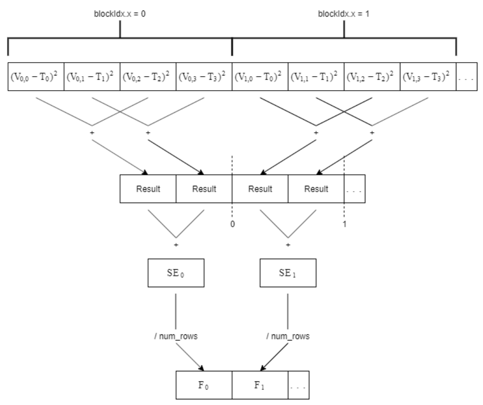

Fig. 2. Fitness computation for a data set of 4 lines

5 Evaluation

5.1 Experimental Setup

A machine with the following specifications was used to run all the tests.

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

Graphics Card	GPU Cores	CUDA Capability	GDDR5 Memory
NVidia GTX 960	1024	5.2	4GB

5.2 Results

The following time at Figure 3 values were outputted by running the program with a data set of 234 lines and 627 rows. The size of the trees was varied according to the X axis and the amount of generations was a fixed set of 5000.

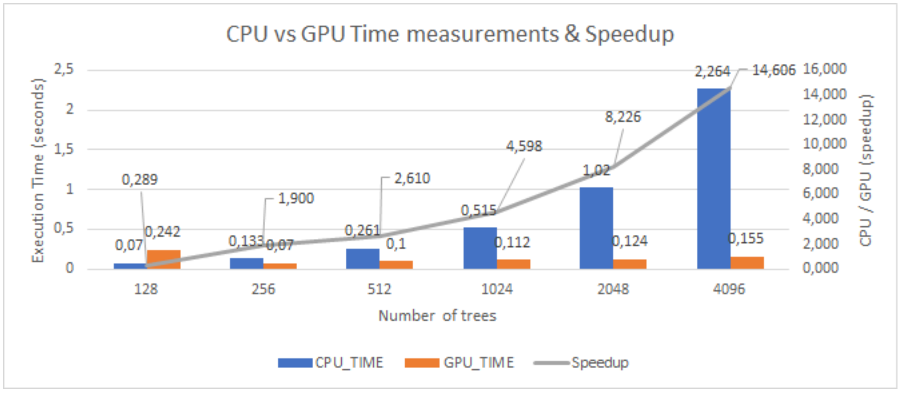

Fig. 3. CPU and GPU execution time comparison

5.3 Discussion

For a low amount of syntax trees, we can see only a small variance between CPU sequential and GPU parallel execution. In the case of 128 trees, the CPU finishes with a lower time compared to the GPU. This is largely due to the overhead when copying memory to and from host and device, as data must travel through the PCIe channel which is much slower compared to either's memory access. On larger amounts of trees, the GPU outperforms the CPU by a significant margin as expected, by parallelizing all fitness calculations the overhead due to memory copying is overshadowed by the performance gain.

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 evolution 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.

Simon Harding and Wolfgang Banzhaf were pioneers in implementing genetic programs on the GPU. Since the tools available at the time were limited, the implementation was done on a toolkit that relied on DirectX (a virtual machine interface that grants access to the GPU), their implementation was limited to Windows machines only. Given that CUDA is an API extended by both C and C++, the implementation presented in this paper is more portable and can be maintained on Windows, Linux or other supported operating systems.

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

João David wrote the sequential version of the algorithm and two GPU methods, while Ye Yang wrote the other GPU method, the data set parser and the tree processing.

Both authors wrote this paper, with João David focusing on the approach and introduction while Ye Yang focused on implementation details, related work and conclusion.

Each author spent around 40 hours on this project.

References

1. Carlos A. Coello Coello, Vincenzo Cutello, Kalyanmoy Deb, Stephanie Forrest, Giuseppe Nicosia, and Mario Pavone, editors. *Parallel Problem Solving from Nature - PPSN XII - 12th International Conference, Taormina, Italy, September 1-5, 2012, Proceedings, Part I*, volume 7491 of *Lecture Notes in Computer Science*. Springer, 2012.
2. Marc Ebner, Michael O'Neill, Anikó Ekárt, Leonardo Vanneschi, and Anna Esparcia-Alcázar, editors. *Genetic Programming, 10th European Conference, EuroGP 2007, Valencia, Spain, April 11-13, 2007, Proceedings*, volume 4445 of *Lecture Notes in Computer Science*. Springer, 2007.