A SIMD Interpreter for Linear Genetic Programming

1st Tarek Ababsa
dept. Computer science
University of Biskra
Biskra, Algeria
tarek.ababsa@univ-biskra.dz

2022 International Symposium on iNnovative Informatics of Biskra (ISNIB) | 979-8-3503-2065-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/ISNIB57382.2022.10075819

Abstract—Genetic programming (GP) has been applied as an automatic programming tool to solve various kinds of problems by genetically breeding a population of computer programs using biologically inspired operations. However, it is well known as a computationally demanding approach with a significant potential of parallelization. In this paper, we emphasize parallelizing the evaluation of genetic programs on Graphics Processing Unit (GPU). We used a compact representation for genotypes. This representation is a memory-efficient method that allows efficient evaluation of programs. Our implementation clearly distinguishes between an individual's genotype and phenotype. Thus, the individuals are represented as linear entities (arrays of 32 bits integers) that are decoded and expressed just like nonlinear entities (trees).

Index Terms—Parallel Processing, GPGPU, linear genetic programming, symbolic regression

I. Introduction

In genetic programming as well as in evolutionary algorithms, the most time-consuming part is the fitness evaluation, because in every generation each one of the individuals must be evaluated at least one time over a single or many fitness cases according to the fitness function especially in the case of classification or regression tasks [16]. Due to the high number of fitness cases, this fact limits the kind of problems that can be handled by GP. There have been a big interest to achieve performance gains by simultaneously evaluate candidate GP programs using parallel computing architectures.

Newly introduced GPUs provide fast parallel hardware that achieve significant improvements in the execution speed of traditional parallel system and in particular genetic programming system as well.

The processing power of the GPUs has become accessible with the recent frameworks such as CUDA and OpenCl. GPUs are mainly designed to efficiently compute graphics primitives in parallel to produce pixels for the video screen, however they are able to handle general data-parallel computations.

In this paper we present a method for using the GPU to speed up the evaluation of a linear genetic programming interpreter. The evolved symbolic regression is used as a benchmark to evaluate the performance of the method.

II. Related work

In genetic programming, evaluating candidate GP programs in parallel is known as a "population parallel" approach and evaluating fitness cases in parallel is known as a "data parallel" approach. Improving the execution speed of GP has been extensively studied because of the high degree of computational complexity of this meta-heuristic.

Over the last years, there have been a variety of techniques for accelerating the evaluation of GP programs. For example: in [6] the others used a divide and conquer strategy in order to minimize the computational cost of the GP. Another strategy is to devise dedicated crossovers for reducing the bloat of individuals and thus the evaluation cost [7]. Distributing the computation of GP through computational nodes is a common strategy which is used in [7], [8]. Although, this distribution of GP is an efficient technique, it is quite expensive because distributed systems are not always available.

Taking advantage of the power of GPUs within the framework of evolutionary computation has been done first for genetic algorithms [9], [10].

Then, recently, implementation schemes for genetic programming on graphics processing units were published, depending on whether they were based on the dynamic compilation of GP individuals or on interpretation of the GP programs [11].

In the literature, the first data parallel approach implementations of GP to use the processing power of many-core GPUs were provided by Banzhaf [2], [3] and Chitty [4]. Both used GPU kernel for executing individual tree based candidate genetic programs. Inside the kernel, candidate GP programs were evaluated sequentially with the parallelization arising at the fitness case level. More recently, M. Chitty [1] used a two-dimensional stack for speeding up GPU-based genetic programming however, the interpreter uses also a non-linear representation of individuals. A nontraditional heterogeneous hardware such as XBox360 has been used in [5] to implement genetic programming with small population sizes, the author described the first instance of LGP implementation using GPGPU, however the experiments are based on a very

special commercial MiscoSoft console.

III. Linear Genetic Programming

Linear genetic programming (LGP) is a particular subset of GP where each individual in the population is expressed as a sequence of instructions from imperative programming language or machine language [15]. Encoding programs within sequences of imperative instructions has many advantages over encoding them within trees. The tree programs used in Koza-style genetic programming is based on a functional programming language. [14]. This conventional approach is called tree-based genetic

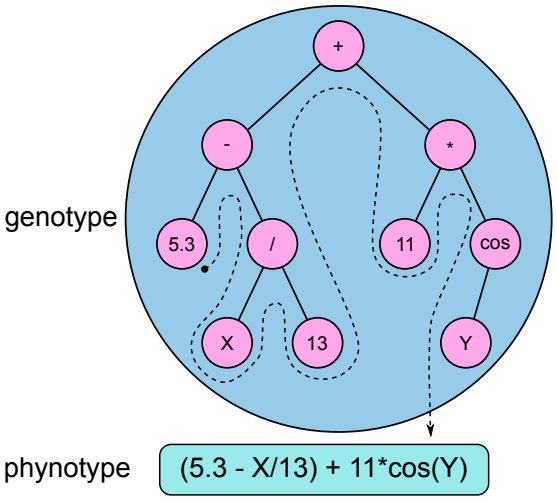

Fig. 1. Koza-style genetic programming representation

programming (TGP) "Fig. 1" where the inner nodes of the tree hold functions, while leaves of the tree hold input values or constants. In contrast, linear genetic programming evolves sequences of instructions from an imperative programming language. multiple registers are required for performing these instructions and allowing the partial results to be reused later [16].

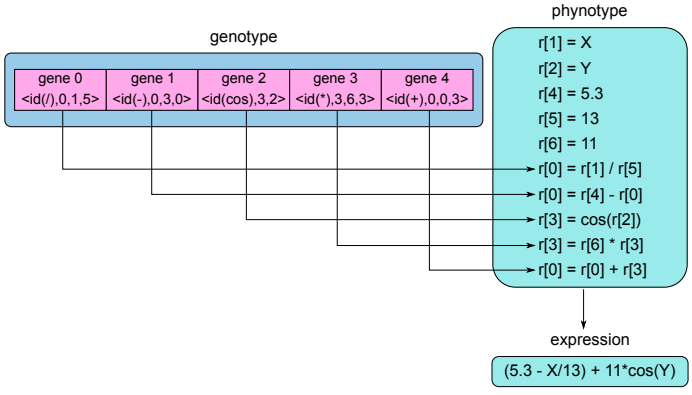

Fig. 2. Linear genetic programming representation

IV. Representation of Genotype

A linear genetic program is performed in a register machine based on von Neumann architecture. Such a machine is composed of multiple registers and operates instructions to manipulate their content in order to perform a computational task.

A genotype encodes a program composed of instructions whose execution has to be performed in sequence. To get a higher programming flexibility, a program with 3-register instructions is considered. This choice is justified by the fact that a program which consists of 3-register instructions is more compacted than a program which consists of 2-register instructions.

During program execution, constant and variable registers are addressed by indices in the internal program representation. They all hold floating-point values, where constants are stored in registers that are write-protected, so may not become destination registers. Constant registers are only initialized once at the beginning of a run with predefined values range. e.g. in "Fig. 2", variable registers are $r[0], r[1], r[2], r[3]$ where constant registers are $r[4], r[5], r[6]$.

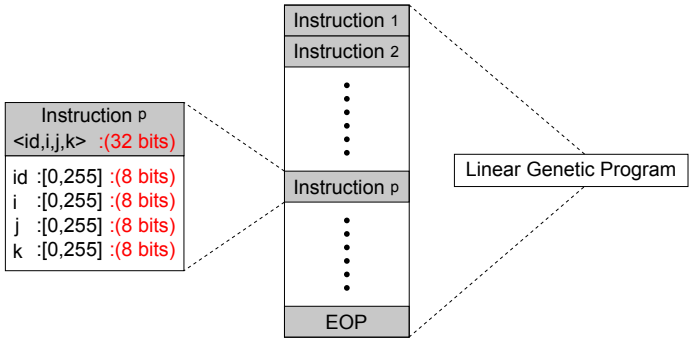

Fig. 3. Genome structure

In our implementation, an operation is held as a single 32-bit integer value as shown in "Fig. 3". An individual is then represented by an array of integers where each gene of the genome is encoded in 32 bits and it consists of the following four parts:

- id: the instruction identifier is incoded in 8 bits to specify the instruction type. (with 8 bits we can encode up to 256 functions).
- i: the destination register index is encoded in 8 bits.
- j: the first source register index is encoded in 8 bits.
- k: the second source register index is encoded in 8 bits.

In genetic programming, the crossover and mutation have to maintain the syntactic correctness of newly created programs. These operators must guarantee somehow that only valid programs are created.

In this work, we used LGP for solving polynomial regression problem. For that reason, we have defined a set of arithmetic, exponontial and geometric functions as shown in TABLE 1.

TABLE I
LGP instruction set

Operation	Id	General Notation	Codification
Add	0	$r_i = r_j + r_k$	<0,i,j,k>
Sub	1	$r_i = r_j - r_k$	<1,i,j,k>
Mul	2	$r_i = r_j * r_k$	<2,i,j,k>
Div	3	$r_i = r_j / r_k$	<3,i,j,k>
Pow	4	$r_i = (r_j)^{r_k}$	<4,i,j,k>
Exp	5	$r_i = e^{r_j}$	<5,i,j,-1>
Ln	6	$r_i = Ln(r_j)$	<6,i,j,-1>
Sqrt	7	$r_i = Sqrt(r_j)$	<7,i,j,-1>

In order to assure semantic correctness, the defined functions may be protected by returning a non-numeric value (MAX-DOUBLE-VALUE) for undefined input, e.g. when the function *Div* is being called to divide by zero it returns $1.8E^{308}$ for indicating an out-of-bounds.

V. Interpreting a LGP Expression

Typically interpreting a GP genome involves looking over the genome's array of genes subsequently "Algorithm 1". Each gene is a program step, where the interpreter must extract the encoded components (function identifier and registers indexes) in order to establish which instruction should be executed on which registers as shown in "Fig. 4".

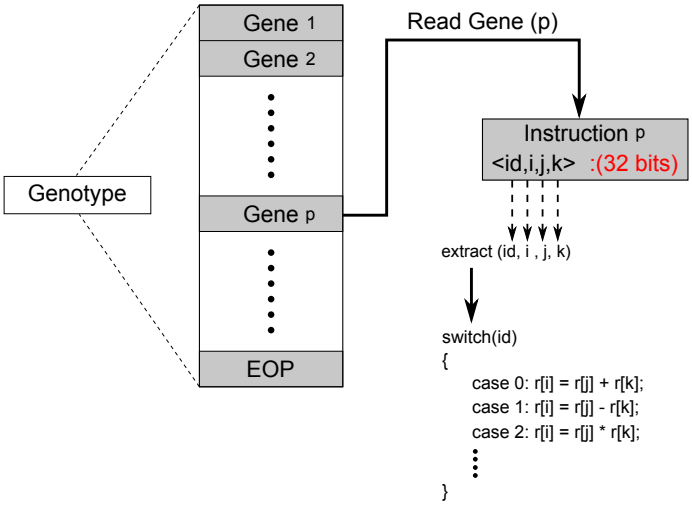

Fig. 4. LGP interpreter

In our implementation, we used a compact genome representation, an array of 32 bits integers. Every element of the array is a gene which encodes four 8 bits components as depicted in "Fig. 3". These components (id, i, j, k) are easily extracted by means of an appropriate binary mask and a logical operation (AND) as folow:

1) Extract the first component "id".
   - $id = gene$ AND 0xFF000000.
   - Shift_Right_Arithmetic(id, 24 positions).
2) Extract the second component "i".
   - $i = gene$ AND 0x00FF0000.
   - Shift_Right_Arithmetic(i, 16 positions).
3) Extract the third component "j".
   - $j = gene$ AND 0x0000FF00.
   - Shift_Right_Arithmetic(j, 8 positions).
4) Extract the fourth component "k".
   - $k = gene$ AND 0x000000FF.

Once the genome is evaluated in a given input, it must be re-evaluated for every input set in order to compute its quality (the fitness function). Hence, for n test cases the genome would be executed n times.

input : A *genome*: Array of n integer
for $idx \leftarrow 0$ to $n-1$ do
  $gene \leftarrow genome[idx]$;
  $id \leftarrow gene$ AND 0xFF000000;
  Shift_Right_Arithmetic(id, 24 positions);
  $i \leftarrow gene$ AND 0x00FF0000;
  Shift_Right_Arithmetic(i, 16 positions);
  $j \leftarrow gene$ AND 0x0000FF00;
  Shift_Right_Arithmetic(i, 8 positions);
  $k \leftarrow gene$ AND 0x000000FF;
  switch(id):
  case 0: $r[i] = r[j] + r[k]$;
  case 1: $r[i] = r[j] - r[k]$;
  case 2: $r[i] = r[j] * r[k]$;
  case 3: $r[i] = Protected(r[j]/r[k])$;
  case 4: $r[i] = Pow(r[j], r[k])$;
  case 5: $r[i] = Exp(r[j])$;
  case 6: $r[i] = Protected(Ln(r[j]))$;
  case 7: $r[i] = Protected(Sqrt(r[j]))$;
end

Algorithm 1: LGP interpreter

Using the GPU we are able to parallelize the evaluation of individuals, which means that in effect the genome only has to be parsed once with the function evaluation performed in parallel.

The CPU sends arrays of test cases to the GPU global memory and loads kernel program into the processors. The Accelerator performs each individuals GP program, and the resulting data is converted back in to an array. The fitness is determined as the sum of all errors from this output array.

VI. Graphics Processing Unit Architecture

In order to effectively benefits from GPUs parallelism, it is important to understand the GPU architecture and then design an efficient program accordingly. In this section we briefly summarize the GPU architecture for the reader to be able to follow the discussion on the parallelization of the algorithms.

Technically, GPUs are specialized stream processors which were originally designed for application of time-consuming graphics operations. They are able to per-

form graphics manipulations much faster than a general purpose CPU, as they are specifically designed to handle certain primitive operations. Hence, any non-graphic algorithm designed for GPU had to be written in terms of graphics APIs such as OpenGL and DirectX. This allowed the development of scalable applications for computationally expensive problems [12].

The GPU consists of a number of multiprocessors that are used for performing calculations on three-dimensional information. These processors cooperate with each other in parallel and in a pipeline fashion to solve the problem.

The GPU architecture is organized as a grid of highly threaded streaming multiprocessors, each one is organized as a set of SIMD processors. Each multiprocessor has a private memory space and can communicate with other multiprocessor through shared among its own processors.

The setup of the GPU is determined by its compute capability which is a parameter related to GPU micro-architecture that defines which hardware component will be accessible for CUDA development. Multiple phases of CUDA program are executed by either the CPU or the GPU. In order to take advantage of data parallelism in a SIMT (Single Instruction Multiple Thread) fashion, the GPU code is implemented as C++ functions called kernels that are launched in a compute grid using a CPU code. A two-level hierarchy is used to organize the threads in a grid, with the first level consisting of blocks that are arranged in three dimensions and each of them may hold up to 1024 threads. Each block is arranged in three dimensions at the second level as well. The programmer determines the dimensions of the grid, which must respect the compute capability limitations of the device [13].

It becomes clear that GPUs could be used to execute high performance general computing tasks for scientific computing due to their capacity to perform parallel operations using 3D data and graphics algorithms. Consequently, a typical technique entails three crucial steps:

- Converting data to graphics textures.
- Constructing a program segment.
- Executing the program segment on the texture.

VII. LGP Parallel Interpreter

The population in LGP is made up of programs written in an imperative programming language. Each program is made up of a number of lines of code that must be run in sequence. We use a register machine to evaluate each LGP individual program. All individuals are typically represented in imperative programming language just like C-style code. The code of each individual is a set of instructions, each of which is composed of three components: the operator, one or two arguments which are manipulated by the operator, and the destination register. The interpreter executes each instruction by applying the operator to the arguments and then stores the result in the destination register. The set of operators includes even simple standard arithmetic operators and some predefined functions for solving particular problems. An example of LGP program is depicted in "Fig. 2".

After executing a LGP program all destination register will hold a valued number. In this work, the state of the registers is represented by an array r. The values stored in r are the outputs of the LGP program which can be interpreted according to the problem at hand. The individual structures that undergo adaptation in LGP are represented by arrays of integers.

input : P: Array of virtual processors
$q \leftarrow threadIdx.x + blockIdx.x * blockDim.x$;
$idx \leftarrow 0$;
while $P[q].gene[idx] \neq EOP$ do
  $gene \leftarrow P[q].gene[idx]$;
  $id \leftarrow gene$ AND 0xFF000000;
  Shift_Right_Arithmetic(id, 24 positions);
  $i \leftarrow gene$ AND 0x00FF0000;
  Shift_Right_Arithmetic(i, 16 positions);
  $j \leftarrow gene$ AND 0x0000FF00;
  Shift_Right_Arithmetic(i, 8 positions);
  $k \leftarrow gene$ AND 0x000000FF;
  switch(id):
  case 0: $P[q].r[i] = P[q].r[j] + P[q].r[k]$;
  case 1: $P[q].r[i] = P[q].r[j] - P[q].r[k]$;
  case 2: $P[q].r[i] = P[q].r[j] * P[q].r[k]$;
  case 3: $P[q].r[i] = Safe(P[q].r[j]/P[q].r[k])$;
  case 4: $P[q].r[i] = Pow(P[q].r[j], P[q].r[k])$;
  case 5: $P[q].r[i] = Exp(P[q].r[j])$;
  case 6: $P[q].r[i] = Safe(Ln(P[q].r[j]))$;
  case 7: $P[q].r[i] = Safe(Sqrt(P[q].r[j]))$;
  $idx \leftarrow idx + 1$;
end

Algorithm 2: LGP parallel interpreter

In our implementation, each parallel element (PE) simulates a computing machine which is composed of the following elements:

- A memory segment to store the LGP program.
- Data registers to store constants and variables.
- A set of registers: the instruction pointer (IP), and general purpose registers: $A_1, A_2, ..., A_n$.

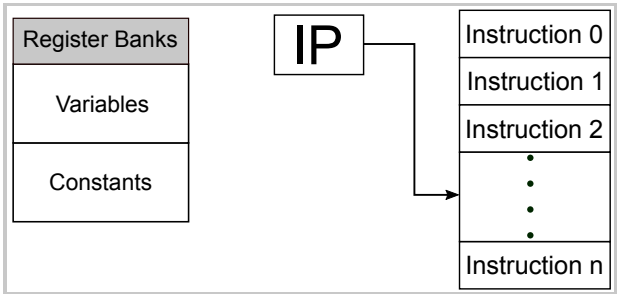

Fig. 5. Virtual processor scheme

"Fig. 5" presents the memory mapping and registers of the virtual processor. To evaluate the population, in our implementation, each PE is running a different genetic program. In this paper the instruction cycle refers to the process by which a virtual processor retrieves a genetic program instruction from the genome, determines what action is described by this instruction, and then carries out this action. The cycle is repeated continuously by the GP interpreter, from the first instruction until the entire genome is traversed.

In most cases of GP problems the fitness of an individual is computed by evaluating it through many inputs. Thus, it is possible to use data parallel approach for evaluating the expression of each individual in parallel [13].

VIII. Experimental Results

The results presented in this paper "Fig. 6" are generated using a 3.0GHz i7 processor with 16GB of memory. The GPU used for the experimentation was an NVidia GeForce 9700 which has 32 stream processors (SP). We compared the results of GP implemented on GPU with that implemented on CPU. All results were generated using the configuration shown in "TABLE II".

TABLE II
The parameters used for the experiments

Parameters	Values
Population size	500
Max genome size	200
Crossover rate	90%
Mutation rate	40%
Probability of mutating non terminals	10%
Probability of mutating terminals	50%
Total number of runs	5000

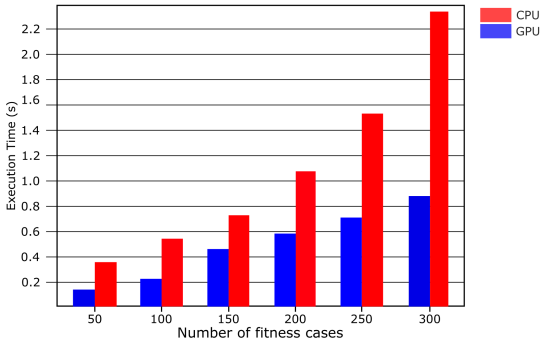

Fig. 6. Evaluation of 5OO genomes using CPU and GPU

As a benchmark, we have used the well known symbolic regression problem. The idea behind symbolic regression is to find a symbolic mathematical expression which maps a given set of input to output values. The target functions which we are looking for are described by (1) and (2).

$$f(x) = x^4 + x^3 + x^2 + 3 \quad \text{with} \quad x \in [-1, 1] \qquad (1)$$

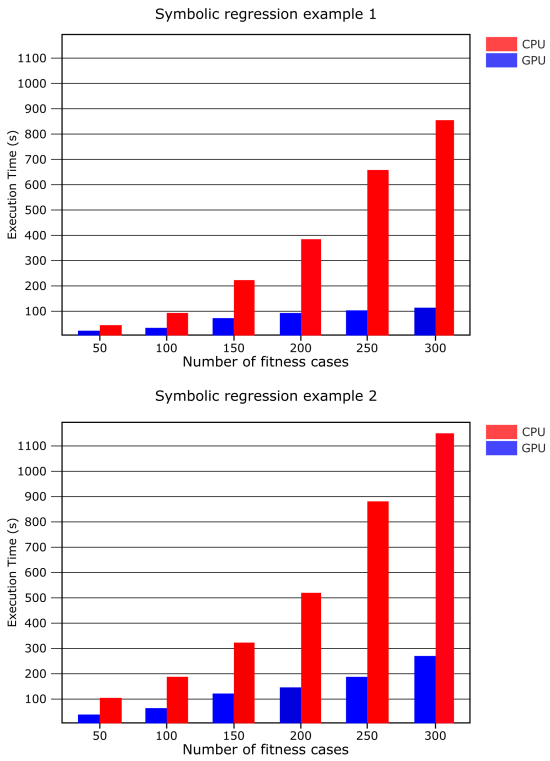

Fig. 7. Solving the regression problem using CPU and GPU

TABLE III
Parameters of example 1

Target function	$f(x) = x^4 + x^3 + x^2 + 3$
Terminals (variables)	one single variable x
Terminals (constants)	Constants randomly generated in [-50,+50]
Non Terminals (operations)	+, *, /, -, Ln, e,Pow, $Sqrt$
Fitness	The sum of errors

$$f(x) = \pi x^4 - x^3 + 2.5x^2 \quad \text{with} \quad x \in [-1, 1] \qquad (2)$$

Each of these benchmarks is tested 30 times with 50, 100, 150, 200, 250 and 300 fitness cases with the parameters which are described in "TABLE III" and "TABLE IV".

In the first experience "Fig. 6", the LGP interpreter described in "Algorithm 2" is implemented in CUDA and used to evaluate a population of 500 genomes, each of

TABLE IV
Parameters of example 2

Target function	$f(x) = \pi x^4 - x^3 + 2.5x^2$
Terminals (variables)	one single variable x
Terminals (constants)	Constants randomly generated in [-50,+50]
Non Terminals (operations)	+, *, /, -, Ln, e,Pow, $Sqrt$
Fitness	The sum of errors

them is represented by an array of 200 integers (this is the worst case according to "TABLE II" where the phenotype is a LGP of 200 instructions). The same population is evaluated using a CPU as well. In this experience, the CPU version of the interpreter had to perform each LGP program subsequently. In such a way, the evaluation of the population takes a very long time that influence the GP performance as shown in "Fig. 7". However, the GPU version of the interpreter had not to perform each LGP program subsequently, because every LGP is assigned to a separated thread to be performed. In such a way, all the genomes are performed at the same time which can yield a speed up of 2–8.

IX. Conclusion

In this paper we show that it is advantageous to use the graphics processor to parallelize the evaluations of linear genetic programming individuals. We have presented a data parallel approach for performing linear genetic programming as a general purpose computing on graphics processor to solve the problem of symbolic regression.

By using a linear rather than a non linear (tree based) representation, the recursive calls are replaced by a condition test to select which operation that must be performed in a given time step. This technique enabled us to run genetic programming with mega populations actually on the GPU. Thus, using this approach we believe that we can make efficient experimentation with a mega population of developmental systems since the linear representation of individuals is more compact and more efficient than tree based representation.

References

[1] M. Chitty, Faster GPU-based genetic programming using a two-dimensional stack, Soft Computing 21 (14) (2017) 3859–3878.

[2] Langdon and Wolfgang Banzhaf, A SIMD Interpreter for Genetic Programming on GPU Graphics Cards, Genetic Programming, 11 conf., EuroGP 2008(LNCS4971), Springer.

[3] Harding and W. Banzhaf. Fast genetic programming on GPUs. Proceedings of the 10th European Conference on Genetic Programming, volume 4445 of Lecture Notes in Computer Science, pages 90–101, Valencia, Spain, 11 - 13 April 2007. Springer. ISBN 3-540-71602-5.

[4] M. Chitty. A data parallel approach to genetic programming using programmable graphics hardware. GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation, volume 2, pages 1566– 1573, London, 7-11 July 2007. ACM Press.

[5] Wilson, G. and Banzhaf, W. (2008), Linear genetic programming GPGPU on Microsoft's Xbox 360, in 'Evolutionary Computation, 2008. CEC 2008. (IEEE World Congress on Computational Intelligence). IEEE Congress on', pp. 378–385.

[6] C. Fillon, A. Bartoli, A divide and conquer strategy for improving efficiency and probability of success in genetic programming, in Proceedings of the 9th European Conference on Genetic Programming, volume 3905 of Lecture Notes in Computer Science, (Springer, Berlin, 2006), pp. 13–23

[7] W.B. Langdon, Size fair and homologous tree genetic programming crossovers, in Proceedings of the Genetic and Evolutionary Computation Conference, (Morgan-Kaufmann, Los Altos, 1999), pp. 1092–1097

[8] F. Fernandez, M. Tomassini, L. Vanneschi, An empirical study of multipopulation genetic programming. Genet. Programm. Evolvable Mach., 4(1), 21–51, (2003)

[9] M.L. Wong, T.T. Wong, K.L. Fok, Parallel evolutionary algorithms on graphics processing unit, in Proceedings of IEEE Congress on Evolutionary Computation 2005 (CEC 2005), vol. 3. (Edinburgh, UK, 2005), pp. 2286–2293. IEEE

[10] Z. Luo, H. Liu, Cellular genetic algorithms and local search for 3-sat problem on graphic hardware, in IEEE Congress on Evolutionary Computation—CEC 2006., (2006), pp. 988–2992

[11] Wong, M., Wong, T., 2009. Implementation of parallel genetic algorithms on graphics processing units. In Intelligent and Evolutionary Systems, Springer, Berlin, Heidelberg, pp. 197–216.

[12] W. Banzhaf, S. Harding, W.B. Langdon, G. Wilson, Accelerating genetic programming through graphics processing units, in: Genetic Programming Theory and Practice, vol. VI, 2009, pp. 1–19.

[13] Coelho, I.M.; Coelho, V.N.; Luz, E.J.d.S.; Ochi, L.S.; Guimaraes, F.G.; Rios, E. A GPU deep learning metaheuristic based model for time series forecasting. Appl. Energy 2017, 201, 412–418

[14] Koza, J. R. (1992), Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT Press, Cambridge, MA, USA.

[15] M. Brameier and W. Banzhaf. Linear Genetic Programming. Number XVI in Genetic and Evolutionary Computation. Springer, 2007. ISBN 0-387-31029-0.

[16] A. Cano, A. Zafra, S. Ventura, Parallel evaluation of Pittsburgh rule-based classifiers on GPUs, Neurocomputing 126 (2014) 45–57.