Random Tree Generator for an FPGA-based Genetic Programming System

Carlos A. Goribar Jiménez, Yazmín Maldonado and Leonardo Trujillo
Instituto Tecnológico de Tijuana, Tijuana, Baja California.
cgoribar@tectijuana.edu.mx, yaz.maldonado@tectijuana.edu.mx,
leonardo.trujillo@tectijuana.edu.mx

ABSTRACT

The present work deals with the implementation of an automatic random tree generator on an FPGA, this implementation is intended to be part of a complete genetic programming embedded system. We propose two methods for a matrix implementations and one for a vector implementation. All trees in the population are created in concurrent processes leading to significant time savings. We present pseudocode and results of hardware consumption for the three implementations.

Keywords

FPGA; Genetic Programming; GSGP

1. INTRODUCTION

This work deals with the implementation of the an automatic generator of random syntax trees on FPGAs (Field Programmable Gate Array), to be used as the initial population in genetic programming (GP) algorithms although a similar approach can be used in any application that require tree data structures in its design. GP is an evolutionary computation technique designed to find solutions for search and design problems without requiring the user to provide prior information about the structure or form of the solution that is sought , John Koza expanded and popularized GP (1992) in [8], and was a pioneer in GP application field, becoming to be known as the father of GP. Trees are the most used data structure in GP literature to represent programs or mathematical functions, or more generally, any syntactical expression that defines some manner of computation.

This work is intended to be used with a GSGP (Geometric Semantic Genetic Programming) based system [9]. The GSGP representation takes into account the semantics (meaning) of the individuals rather than simply using the syntactic representation during the search process. Moreover, the genetic operators used by GSGP have the property of inducing a unimodal error surface for any supervised learning problem. On the other hand, one problem of GSGP is that their genetic operators generate individuals that are larger in size than their parents, turning this approach difficult to implement in practice. Nevertheless, the proposal in [2] only stores the individual trees form the initial population without change throughout the entire search process using tables of semantics.

Traditionally, most GP systems are executed as software on a desktop computer or workstation, but as the paradigm has matured, GP is used to solve increasingly more complex problems leading to longer run times (mainly due to fitness evaluation) and larger memory consumption, this is particularly problematic for GP systems that are affected by the bloat phenomena (a rapid increase in program size not accompanied by any significant corresponding increase in fitness) [12]. One method to reduce computational costs is to use multicore processors, this technique allows the whole program to run in concurrent threads that accelerate fitness computation (the most severe bottleneck in most GP algorithms).

However the number of cores found in commercial microprocessor are limited to about four. On the other hand, one can take advantage of the massive parallel processing power that a Graphics Processor Unit (GPU) exhibits; for instance, in [1] the authors propose a transcription of existing GP parallelization strategies to the OpenCL programming platform achieving $10x$ times the throughput of a twelve-core CPU and [5] explains the use of the GPU to accelerate the evaluation of individuals that allows to get speed increases of several hundred times over a typical CPU implementation, [4] which shows how to take advantage of GPUs for the evolution of an image filter. GPU may seem the natural choice when high parallel processing power is needed, however the strength of GPUs is also its own weakness because high parallelization works well when the very same operation needs to be performed on a vast amount of data, but it is not efficient when different operations on different data is required, the latter is a normal case when GP is used in real world applications. FPGAs on the other hand are a very flexible devices that allows for the implementation of complex digital systems in a small silicon chip or integrated circuit, this device considered to be at the intersection of software and hardware-oriented systems.

FPGAs are less expensive compared to GPUs or CPUs. Another attractive characteristic of FPGAs is the possibility to perform parallel computing, [13] is an example of using parallel processes in mapping of a population-based ant colony optimization algorithm that led to significant improvements in runtime.

GECCO'16 Companion, July 20 - 24, 2016, Denver, CO,USA

ACM ISBN 978-1-4503-4323-7/16/07…$15.00

DOI: http://dx.doi.org/10.1145/2908961.2931665

Examples of FPGA-based GP are the following: Cartesian GP is implemented in [7] using two Microblaze soft processors, the system is used to evolve an image filter and results show a significant speed-up of up to $58x$ in comparison with a highly optimized software implementation. A purpose of a register machine implemented on an FPGA is found in [6]. A linear GP structure is employed, hence individuals have the form of register-level transfer language instructions. The functional set contains opcodes while the terminal set contains operands, in this approach a population of (linearly structured) individuals is created off-line by the host. Finally, [14] deals with the implementation of a tree-based GP, exploiting the power of self-reconfigurable partitions on FPGAs, this system is used to evolve a circuit that exhibits the same behaviour as a digital multiplexer having 8 data inputs and 3 control inputs, as well as a regression problem. The results show a speed-up of three orders of magnitude for fitness evaluation (the stage that consumes up to 99% of execution time).

2. TREE GENERATION METHODS

The three original and most widely used initialization methods for the generation of the initial population [11]:

Grow: Nodes are selected randomly from the whole primitive set (functions and terminals) until the depth limit is reached. Once the depth limit is reached only nodes from the terminal set can be chosen.

Full:Nodes are chosen at random from the function set until the maximum tree depth is reached, then only terminals can be chosen.

Ramped half-and-half: In this method half of the initial population is constructed using the full method and half is constructed using the grow method. Also a range of depth limits is used during the population creation.

As we have mentioned before, pointers are not synthetizable on hardware using the VHDL programming language, thus our work consist of two proposal methods for matrix implementation and one for vector implementation.

Now, the first step in the implementation is to define the set of primitives, this set is composed by the terminal and function sets. Table 1 shows the primitive set that includes arithmetic functions, variables (x and y) and constants within range from 0 to 9. Each element in the primitive set will be associated with an ID, so the ID will represent only one element of the primitives in hardware implementation. This table can be modified to include the primitives of a particular problem domain.

For simplicity we use of the *full* method with all functions having the same arity of two, these kind of trees are enough to be used in the implementation of GSGP [2]. In other words, the use of the static full trees does not represent a problem when working with GSGP. More over, the use of arithmetic functions $(+, -, *, \div)$ and the absence of other functions like trigonometric functions are based on the fact that previous works on GSGP have shown that it is possible so solve complex real problems using only arithmetic functions.

Figure 1 shows how the implementation of one tree in a single matrix is done. In this case a 4x8 matrix is used to store a tree of depth 3 generated with the full method. In general, a tree with depth n can be stored in a matrix with

Table 1: Definition of primitives

Op/Var/Const	ID	Arity
+	0000	2
-	0001	2
*	0010	2
÷	0011	2
x	0100	0
y	0101	0
0	0110	0
1	0111	0
2	1000	0
3	1001	0
4	1010	0
5	1011	0
6	1100	0
7	1101	0
8	1110	0
9	1111	0

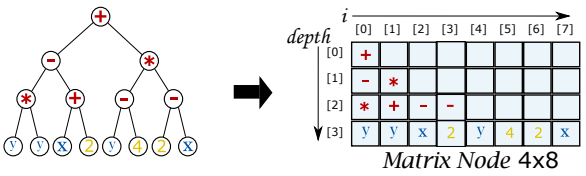

Figure 1: Tree implemented in a 4x8 matrix.

dimensions $n+1$ by 2^n provided that all functions exhibit an arity of 2. Note that elements in the first n rows will be filled with functions only, then the row $n+1$ will be filled entirely with terminals. In this implementation only $2^{n+1}-1$ elements of the matrix are used (one element of the matrix holds one tree node) from the $(n+1)\,2^n$ available in the $n+1$ by 2^n. A better implementation in terms of memory utilization might be to implement two trees in one matrix as follows.

In this case a 4x9 matrix is needed to store two trees of depth 3 generated with the full method. In general, a tree with depth n can be stored in a matrix with dimensions $n+1$ by 2^n+1 provided that all functions have an arity of 2. Figure 2 shows how the implementation of two trees in matrix are taking place. Note that in this implementation we make use of $2\left(2^{n+1}-1\right)$ to store the tree nodes from the $(n+1)\,2^n+1$ available in a $n+1$ by 2^n+1 matrix.

Figure 3 shows how to implement a full tree on a vector. TinyGP is a GP system implemented in Java, and employs a similar representation without pointers [11]. In this case not even one element of the vector will be wasted if the length of the vector is $2^{n+1}-1$ for a tree with depth n. In this approach functions are first generated until the element 2^n-1, then only terminals are selected from the terminal set and placed in consecutive order in the remaining spaces in the vector.

A Linear Feedback Shift Register (LFSR) [3] is used as pseudo-random number generator, this is a common method for population initialization and are specially easy to be used in hardware impementations [10].

A LFSR circuit with n outputs is able to generate a pseudo-random sequence of $2n-1$ patterns.

The pseudo-random number generator depends upon its initial condition or seed, so for a particular seed as input, the generator will generate a particular output which obviously depends directly on the number of registers in the LFSR.

The pseudo-code for the generator of GP trees is shown in Algorithm 1, the requirements are the maximum depth, the set of functions, and set of terminals, also a matrix of max_depth by 2^{max_depth} to hold the data is needed.

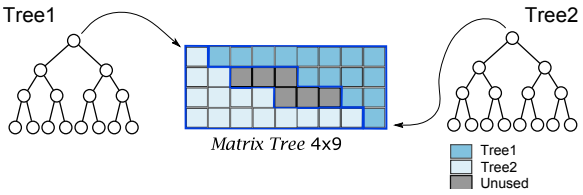

Figure 2: Tree implemented in a 4x9 matrix.

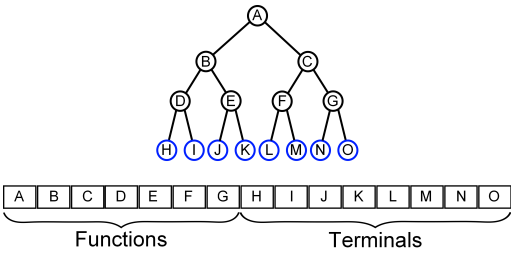

Figure 3: Tree implemented in a vector.

The random ID is generated with the LFSR circuit, this means that if a function is needed then the LFSR produces several outputs until an ID in the range of functions ($new\_random(functions)$) is found, and when a terminal ($new\_random(terminals)$) is needed the LFSR produces several outputs until an ID in the range of terminals is found. $Tree(depth, i)$ is a reference to the row number $depth$ and column i in the matrix $Tree$.

Steps 1 to 15 configure tree1, and steps 16 to 30 configure tree2, so in the implementation we follow steps 1 to 15 for the one tree in one matrix implementation while steps 1 to 30 were followed for the two trees in one matrix implementation method.

FPGAs are very flexible devices that allow for the implementation of concurrent processes [13].

In this case we just need to replicate the same exact circuit N times.

Note that each tree generator depends on its seed, obviously if all seeds are the same, each tree generated by each process will be exactly the same, thus different seeds are needed to obtain different trees.

3. RESULTS

Simulations were performed on *Xilinx Isim 13.4v (64-bit)* and implemented on the $XC3S700A$ chip, however, a different chip can be used to better suit a specific application and requirements of internal resources.

A segment of the simulation waveforms (last $56clk$ pulses) for the implementation of the two trees in one matrix are shown in Figure 4, *clk* ($50MHz$) and *seed* are input signals, while *qsal* and *ready* are output signals for debugging purposes. The pseudo-random generator is composed by a D-type Flip-Flop, so it needs to be inicialized with the desired seed, this is done using the *rst*, *set* and x internal signals.

A resulting tree with depth of 3 coded in *tree* which is a 4-by-9 array is shown in Figure 4. The explanation of that matrix is the following: the first row in *tree* is

$$tree[0] = [0010, 1010, 1101, 1110, 1111, 0111, 0011, 1000, 0100]$$

where the first element corresponds to the *root* of tree1, this node would be the function "$*$" per Table 1, this row also corresponds with depth 0 for the tree1, while is the depth n

Algorithm 1 Tree implemented on a matrix

Ensure: Input
$max\_depth \in \mathbb{N}^+$: maximum tree depth
$\mathcal{F}$: set of functions
$\mathcal{T}$: set of terminals

Ensure: Output
$Tree$: max_depth by 2^{max_depth} matrix

// Build *Tree1*.
1: $depth \leftarrow 0$ // $depth = 0$ at root node.
2: $i \leftarrow 0$ // i holds the i-column.
3: **while** $depth <= max\_depth$ **do**
4: **if** $depth < max\_depth$ **then**
5: $rand\_node \leftarrow new\_random(\mathcal{F})$
6: **else**
7: $rand\_node \leftarrow new\_random(\mathcal{T})$
8: **end if**
9: $Tree(depth, i) \leftarrow rand\_node$
10: $i \leftarrow i + 1$
11: **if** $i > 2^{depth}$ **then**
12: $i \leftarrow 0$
13: $depth \leftarrow depth + 1$
14: **end if**
15: **end while**
// Build *Tree2*.
16: $depth \leftarrow 0$ // $depth = 0$ at root node.
17: $i \leftarrow 0$ // i holds the i-column.
18: **while** $depth <= max\_depth$ **do**
19: **if** $depth < max\_depth$ **then**
20: $rand\_node \leftarrow new\_random(\mathcal{F})$
21: **else**
22: $rand\_node \leftarrow new\_random(\mathcal{T})$
23: **end if**
24: $Tree(max\_depth - depth, 2^{max\_depth} - i) \leftarrow rand\_node$
25: $i \leftarrow i + 1$
26: **if** $i > 2^{depth}$ **then**
27: $i \leftarrow 0$
28: $depth \leftarrow depth + 1$
29: **end if**
30: **end while**

for tree2. The last row in *tree* is

$$tree[3] = [0100, 1000, 0111, 1111, 1110, 1101, 1010, 0101, 0010]$$

that corresponds to $[x, 2, 1, 9, 8, 7, 4, y, *]$ in accordance to the definitions in Table 1. Note that the elements x to y are the terminals of tree1, while "$*$" is the *root* of the tree2. "$UUUU$" simply means that that element has not been initialized, however it does not cause any problems because we do not care about the state of those elements in the matrix.

Table 2 and Table 3 summarizes the consumption of hardware resources for the implementation of $1, 10, 50$, and 100 trees with a depth of 3, $1TM$ means *one tree on one matrix*, $2TM$ means *two trees on one matrix*, and VR means *vector representation*. Note that in the case of one tree in one matrix, resources are sufficient for the implementation of up to 100 trees except for the *Number of bonded IOBs*, this is because of the inputs needed for each tree *seed* ($4-bit$ input), and the *ready* ($5-bit$ output) signal. In the case of the two trees in one matrix implementation, although a reduction of hardware resources was expected due to a more efficient memory utilization, results show that less than 50 arrays (or less than 100 trees) can be implemented on the FPGA, this is because a more complex circuit is necessary to handle the matrix in order to place two trees in it.

4. CONCLUSIONS

Implementation of tree data structures in FPGA devices is not a trivial task, due to the impossibility of using pointers as it is normally done in software design, that is why

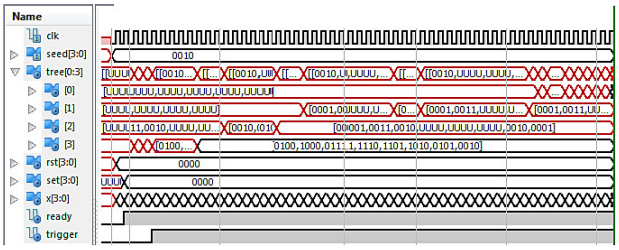

Figure 4: Simulation for method "two trees in one matrix".

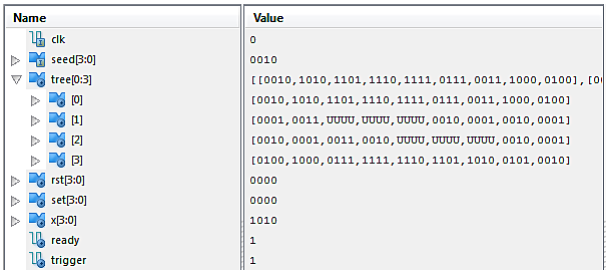

Figure 5: Tree with depth of 3 coded in a 4-by-9 array.

researchers avoid the use of these kind of structures in their hardware designs.

This work has proposed three methods to generate the initial population for a GP system on FPGAs.

Two methods using a matrix coded representation and one using a vector representation has been implemented using an FPGA.

We shown how to take advantage of the possibility of explode the parallelism that FPGAs offer to make the process more efficient, so trees can be configured concurrently independently of the number of trees. There are waste of memory elements that remains uninitialized in the matrix that holds the tree, but the vector representation does not present this problem.

We have shown that on the contrary of expected results, the two trees in one matrix approach consumes more internal resources compared with the one tree in one matrix implementation due to the internal circuitry to handle the matrix configuration, surprisingly, vector representation requires even more hardware resources, so we came to the conclusion that one tree in one matrix would be the best choice for FPGA implementation. This work is the first step in the implementation of a totally embedded Genetic Programming based system, so evaluation, selection and variation is matter of future work.

Table 2: Table that summarizes the consumption of resources in the FPGA for 1 and 10 Trees.

	1 Tree			10 Trees		
	1TM	2TM	VR	1TM	2TM	VR
Number of Slices	<1%	4%	1%	9%	35%	18%
Number of Slice Flip Flops	<1%	1%	<1%	3%	9%	6%
Number of 4 input LUTs	<1%	4%	1%	7%	34%	15%
Number of bounded IOBs	2%	3%	2%	24%	26%	26%
Number of GCLKs	4%	5%	8%	4%	45%	45%

Table 3: Table that summarizes the consumption of resources in the FPGA for 50 and 100 Trees.

	50 Trees			100 Trees		
	1TM	2TM	VR	1TM	2TM	VR
Number of Slices	47%	180%	92%	99%	360%	188%
Number of Slice Flip Flops	18%	49%	33%	36%	98%	67%
Number of 4 input LUTs	36%	170%	78%	71%	341%	156%
Number of bounded IOBs	121%	128%	128%	242%	257%	257%
Number of GCLKs	4%	100%	100%	4%	100%	100%

Acknowledgments

Funding for this work was provided by CONACYT basic science research project No. 178323, TecNM(México) Research projects 5861.16P, Prodep(México) ITTIJ-PTC-007, and by FP7-Marie Curie-IRSES 2013 European Commission program through project ACoBSEC with contract No. 612689.

5. REFERENCES

[1] D. A. Augusto and H. J. Barbosa. Accelerated parallel genetic programming tree evaluation with OpenCL. *Journal of Parallel and Distributed Computing*, 73(1):86–100, 2013.

[2] M. Castelli, S. Silva, and L. Vanneschi. A C++ framework for geometric semantic genetic programming. *Genetic Programming and Evolvable Machines*, 16(1):73–81, 2015.

[3] S. Golomb. *Shift Register Sequences*. Holden-Day, Inc., 1967.

[4] S. Harding. Evolution of image filters on graphics processor units using cartesian genetic programming. *Evolutionary Computation, IEEE World Congress on Computational Intelligence*, pages 1921–1928, 2008.

[5] S. Harding and W. Banzhaf. Fast genetic programming on GPUs. *Proceedings of the 10th European Conference on Genetic Programming (EuroGP'07)*, pages 90–101, 2007.

[6] M. I. Heywood and A. N. Zincir-Heywood. Register based genetic programming on FPGA computing platforms. *Genetic Programming, Proceedings of EuroGP'2000*, pages 44–59, 2000.

[7] R. Hrbacek and M. Sikulova. Coevolutionary cartesian genetic programming in FPGA. *12th European Conference on Artificial Life Proceedings*, pages 431–438, 2013.

[8] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[9] A. Moraglio, K. Krawiec, and C. G. Johnson. Geometric semantic genetic programming. *Proceedings of the 12th International Conference on Parallel Problem Solving from Nature*, 1:21–31, 2012.

[10] D. M. Munoz, C. H. Llanos, L. dos S. Coelho, and M. Ayala-Rincon. Hardware opposition-based PSO applied to mobile robot controllers. *Engineering Applications of Artificial Intelligence*, 28:64–77, 2014.

[11] R. Poli, W. B. Langdon, and N. F. McPhee. *A Field Guide to Genetic Programming*. Lulu Enterprises, 2008.

[12] R. Poli, N. F. McPhee, and L. Vanneschi. Analysis of the effects of elitism on bloat in linear and tree-based genetic programming. *Genetic Programming Theory and Practice*, VI(7):91–111, 2008.

[13] B. Scheuermann, K. So, and M. Guntsch. FPGA implementation of population-based ant colony optimization. *Applied Soft Computing*, 4(3):9303 – 9322, 2014.

[14] R. P. Sidhu, A. Mei, and V. K. Prasanna. Genetic programming using self-reconfigurable. *FPGAs, in 9th International Workshop on Field Programmable Logic and Applications*, pages 301–312, 1998.