Rushing the CGP Logic Optimization Flow

Bryan M. Lima, Naiara Sachetti, Augusto Berndt, Cristina Meinhardt and Jonata Tyska Carvalho
Department of Informatics and Statistics, Federal University of Santa Catarina - UFSC
Florianópolis, Brazil
bryan.l@grad.ufsc.br, naiara.sachetti@grad.ufsc.br, augusto.berndt@posgrad.ufsc.br,
cristina.meinhardt@ufsc.br, jonata.tyska@ufsc.br

***Abstract*—With Moore's law slowing down, the process of logic synthesis is struggling to maintain the same technological growth rate. In this scenario, alternative methods started to appear. With the ascension of Machine Learning, these algorithms became more attractive, taking another approach to the concept of large integrated circuits. While the best circuit implementation for a problem involves a trade-off between power, area, and delay, these alternative methods are more attractive in error-tolerant applications when they can have a better overall circuit to the detriment of accuracy. One of these methods is Cartesian Genetic Programming (CGP), a subclass of Evolutionary Algorithms that uses concepts from biological evolution, such as recombination, mutation, and fitness, applied in electronic design automation (EDA). The main challenge of CGP-based flows is reducing the extensive runtime necessary to learn the circuit compared to other logic synthesis strategies. This paper investigates strategies for lowering the CGP runtime while finding logic circuits with even or better accuracy. We specifically study how different ways of evaluating the candidate solutions, using partial or complete truth tables, impact the runtime and accuracy of the evolved solutions. We show that, in some cases, by using only portions of the truth table, the CGP achieves better accuracy in less time than using all of the truth table lines.**

***Index Terms*—Logic synthesis, Cartesian Genetic Programming (CGP), Evolutionary algorithms**

I. Introduction

Logic optimization is an initial task in the synthesis flow where the main target is reducing the number of elements, including nodes and logic depth. These optimizations will influence the next steps in the synthesis flow, impacting area, delay, and power results. Traditional logic optimization methods simplify a Boolean function, exploring exact logic minimization techniques like the Algebraic method, the Karnaugh map technique [1], or the Quine-McCluskey method [2]. However, the main limitation of the traditional logic optimization methods is the number of inputs that they can deal with. For instance, the Quine-McCluskey method is limited to functions with up to 15 variables [3]. For real world applications Espresso is used for simplification of circuits with many inputs reaching faster results by exploring sub-optimal heuristic methods [4] [5]. These fast simplification methods provide a trade-off for computing performance at the cost of output quality.

Some new logical optimization flows address fast logic optimization based on machine learning approaches like decision trees [6]. However, many of them fail to deal with scaling the logic functions' complexity. For example, decision tree solutions, in general, must expand all the input combinations, which becomes prohibitive for large inputs. To deal with the expensive rising of complexity on logic functions, neural networks [7], or evolutionary algorithms as Cartesian Genetic Programming [8], [9] have been recently studied. An important drawback of these approaches is the large runtime compared to the traditional logic synthesis approaches.

Cartesian Genetic Programming (CGP) is a form of genetic programming that uses a graph representation to encode computer programs. It is called 'Cartesian' because it represents a program using a two-dimensional grid of nodes [10]. In [8], [9], a CGP-based flow that seeks to optimize both accuracy and size is proposed. The results of this flow are two-fold: (i) it allows to improve further the solutions found by the other techniques used for bootstrapping the evolutionary process, and (ii) seek optimal circuits starting the CGP search from random (unbiased) individuals [9]. However, the runtime for learning a Boolean function is considerably large. This is due to the number of generations needed to improve the accuracy. Overall, the proposed CGP-flow presents a good trade-off between accuracy and nodes reduction, mainly when explored to improve an initial solution.

Approximate Computing is a paradigm to speed up the design flow and produce power-efficient solutions for error-tolerant applications. As these applications have less strict accuracy requirements of the implemented functions, it is possible to focus on a smaller circuit, which can improve delay and power [11]. In these integrated circuits (IC) power efficiency is as important as its accuracy. We can use approximate Computing to generalize a circuit based on a few selected samples in this context. This was one of the goals of the International Workshop on Logic and Synthesis (IWLS) Contest in 2020 [12], in which multiple teams of different countries competed to generalize logic functions. The teams used multiple strategies, including Espresso [4], multi-layer perceptrons (MLP), random forests, lookup-table (LUT) networks as well as CGP [7]. Overall, the presented results confirmed that sacrificing some accuracy was possible to achieve a significantly smaller circuit.

In this context, our work investigates strategies for decreasing the runtime required for the CGP flow and finding good solutions. We investigate how different ways of evaluating candidate solutions affect the runtime and the CGP generalization capacity, seeking to optimize it. We investigate whether and how it is possible to improve runtime synthesis and with reasonable accuracy by using only portions of the truth table

to evaluate individuals.

This paper is organized as follows. Section II presents the core definitions for a better understanding of the CGP algorithm and the means to collect the data. In section III there is a discussion on the data collected. Finally, section IV concludes this work and presents future work further to improve the CGP flow runtime for approximate Computing.

II. Methodology

In our implementation we explore the utilization of AND-Inverter Graphs (AIGs), which are the state-of-the-art data structure for technology-independent optimizations during logic synthesis [13]. An AIG is a directed acyclic graph composed of nodes representing AND gates and edges representing inverted or directed connections. An AIG node is composed of exactly two inputs and an arbitrary number of outputs. An AIG may represent any logic function. In our CGP implementation an AIG is represented as an individual from the CGP population, and CGP mutations concern modifying the AIG connections and inversions.

The CGP evaluates all the individuals every generation, after they are mutated. The evaluation process for each of them requires processing all the training batch inputs. This is done with a depth first search along the AIG. In other words, the evaluation has a time complexity of $O(i * n * b)$, with i being the number of individuals in the CGP population, n the size of each individual, and b the batch size (BS) used for training.

Our work is an improvement upon the CGP C++ implementation used in [9]. The main difference in our version is when to evaluate the nodes accuracy, as this process is computationally costly. For this , we used the proposed strategy in [14] which is to evaluate only the functional nodes. This change was not only better for performance, but it was necessary to be possible to evenly compare the multiples hyperparameters used in CGP.

A. Definitions

Definition 1 (Batch size): For approximate computing, the CGP can evaluate candidate solutions using less lines than the whole truth table. The number of lines used is called *batch size* (BS).

Definition 2 (Change each): When BS is lesser than the whole truth table the *change each* (CE) represents for how many generations the algorithm will use the same batch for evaluating the candidate circuits. After that number of generations, a different batch will be sampled and used for evaluation.

Definition 3 (Mutation): Is the process in which a circuit is changed so as to explore the search space. These mutations occur in the AIG nodes based on upon the $1/5^{th}$ rule described in [15].

Definition 4 (Fitness): In this work, fitness means the capability of a circuit to produce the correct outputs given a set of inputs (accuracy). However, it can be defined as a different optimization goal in mind such as size, for example.

Definition 5 (Search algorithm): The search algorithm used is the $(1+4)-ES$ [16] strategy. Therefore, a so-called parent circuit will generate 4 mutated (child) circuits. The circuit with the best fitness, including the parent, will be selected as the parent of the next generation.

B. CGP hyperparameters

Since we want to investigate how different ways of evaluating individuals impact on runtime and synthesized solutions accuracy, we mainly explore two CGP hyper-parameters: (i) the *batch sizes* used by that run; and (ii) the *change each*. We provide an example for illustrating how these parameters affects the CGP flow in practice. For instance, in a CGP run with 10.000 generations, a batch size of 64, and a change each equal to a 1.000, the mini-batches will change 10 times. These parameters are related to the train set used in the CGP program. Therefore, with a truth table with 1024 lines, a batch size of 64 is only 6.25% of the complete circuit.

As the main benefit of CGP in logic synthesis is to generate a complete circuit based upon an incomplete truth table, our main objective was to identify some evidences to improve the accuracy, and/or the convergence time to generate the circuit when compared to running CGP with the complete truth table. For this, we ran the algorithm with the only completely specified function of the IWLS Contest 2020, the exemplar 41. Therefore, it is possible to measure our implementation performance with a number of variations of the investigated hyperparameters. The exemplar 41 is an arithmetic function representing the least significant and middle bit of an *n-bit* square-root.

III. Results

In this Section we will present the results obtained with the experiments performed on the previously discussed logic function. Table I shows the hyperparameters used in our experiments. We used 50.000 generations, which is not much, to identify which of parameters impacted the initial learning of the circuit the most. The number of generations directly impact the time to learn the logic function, which varied between 60 and 340 seconds per *seed* depending on the BS value. Using an interval of 128 between each BS provides us with the information of to what extent the CGP can produce circuits able to generalize for each BS value on this particular function. We used a number of nodes of 250, which means that the CGP could use a maximum of 250 functional nodes. Moreover, we used 10 seeds per run, i.e. the CGP was executed with 10 different seeds for its random initialization. As this algorithm is impacted by its initial circuits, 10 runs provided enough information to fairly compare the results. Furthermore, we designed experiments aiming to discover how the CE could affect the performance achieved using each BS value. Therefore, considering the combinations of BS, CE and the number of seeds, we ran a total of 630 experiments for the exemplar *ex41*. It is noteworthy that the number of generations and number of seeds does not provide directed useful information as to analyze the CGP algorithm

TABLE I
HYPERPARAMETERS TESTED

Parameter	Value
Number of Generations	50.000
Batch size	128, 256, 384, 512, 640, 768, 896
Change each	1000, 2000, 3000, 4000, 5000, 10000, 15000, 20000, 25000
Number of nodes	250
Number of seeds	10

TABLE II
COMPARISON OF THE BEST CE (CONSIDERING ACCURACY) FOR EACH BS

BS	Best CE
128	4000
256	2000
384	2000
512	4000
640	4000
768	3000
896	1000

performance. Therefore, the number of generations was not a set of values, as we intended to investigate the two primary hyperparameters, the CE and BS. For this, we kept this value as a constant. Similarly, the value for the number of seeds was used as we could gather enough information for these two primary hyperparameters so we could actually confirm any impact to the algorithm performance. As the algorithm is sensitive to its seed, using a small value of the number of seeds could drive us to wrong conclusions. It will be possible to notice the variations on the accuracy on the box plots in this section.

Figure 1 presents the accuracy of the best CE value for each BS. It is noteworthy that the BS of 1024 is using the complete truth table. As we can see, 512 and 786 values have a better median compared to the complete batch. Furthermore, an evolutionary process using a BS of 256 could generate circuits able to generalize and achieve comparable accuracy to the complete version with only a quarter of the truth table. Table II shows the values of the CE which composes figure 1. It is possible to notice that for all BS the best values of CE were relatively small, between 2% and 8% of the whole evolutionary process. Furthermore, figure 2 presents the number of functional nodes of the 250 maximum for the given batch sizes. The data on the BS of 512 was particularly interesting considering its size, as it was possible to achieve a similar accuracy compared to the complete version, while having a 40% decrease on the functional nodes. This indicates that the CGP could not only generalize, but optimize the logic function using half the lines of the truth table.

Figure 3 shows a comparison of the learning accuracy for exemplar *ex41*. The vertical axis presents the accuracy improvement along the generations on the horizontal axis. We present three BS options used during learning. Both 128 and 512 BS presents decreases in accuracy every 4.000 generations, this is due to the CE value which modifies the minterms composing that batch. The complete batch includes

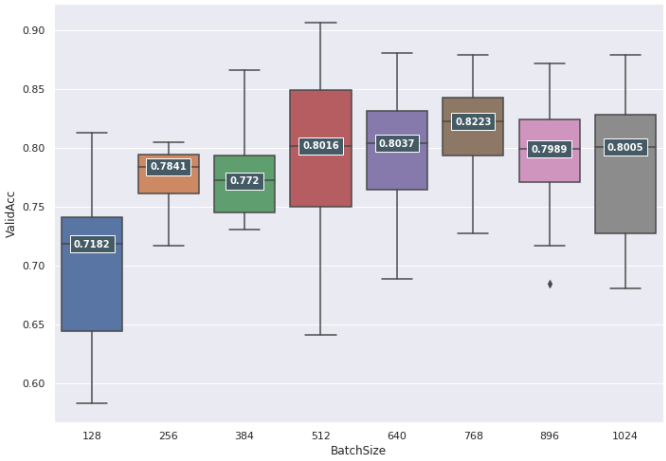

Fig. 1. Comparison of Accuracy considering different BS

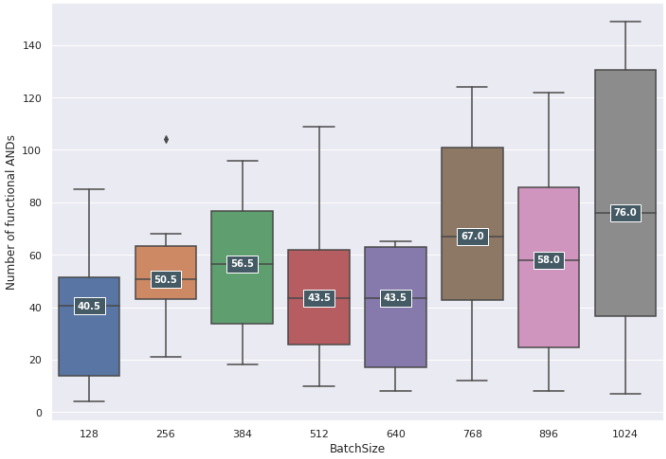

Fig. 2. Comparison of the number of functional nodes considering different BS

all combinations, for this reason there is no decrease in accuracy, since all the lines are available since the beginning. It is noteworthy, that the accuracy displayed is relative to the size of the BS. For that reason, the BS of 128 has an easier task, compared to batches with higher values, as it only needs to learn 128 lines of the complete truth table. These results indicate that could be worthy to investigate whether gradually increasing the CE value could improve the CGP flow in regarding accuracy and/or runtime. Furthermore, we can see how drastically the accuracy is lost when a batch change occurs. A possible strategy that could be investigated to mitigate this effect is to partially change the batch, i.e. changing 50% of the batch when a CE occurs.

Finally, figure 4 presents the comparison of the runtime in seconds between all the BS tested as well as the complete version. The 512 BS had a median of 20.96 seconds compared to the 28.33 seconds of the complete version. This roughly represents a 26% improvement in time. Moreover, the BS of 512 had a better accuracy as shown in figure II, which

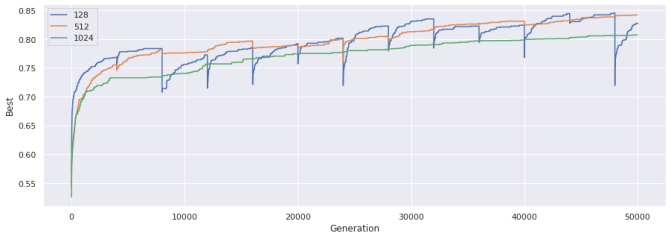

Fig. 3. Comparison of Learning on two different BS with exemplar ex41

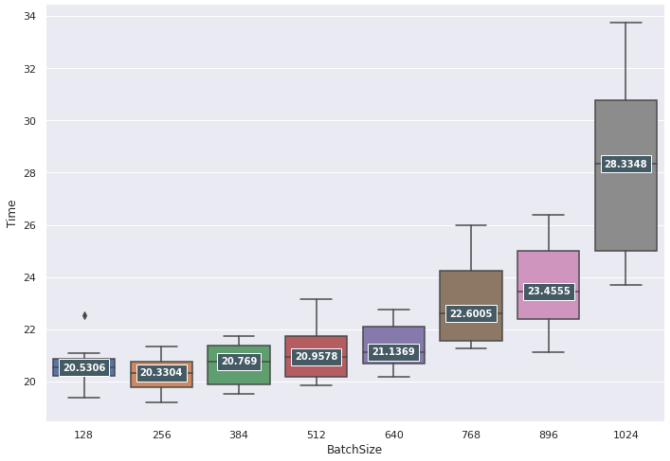

Fig. 4. Comparison of runtime between each BS

further confirms our initial hypothesis. Overall, the results indicate that using only a portion of the the truth table can indeed improve both the accuracy and the runtime of the CGP algorithm, as well as the circuit size.

IV. Conclusion

This work explored whether different evaluations strategies can lead to improvements into the CGP search algorithm regarding runtime and accuracy of the synthesized solutions. Our results confirmed that by optimizing the batch size and the frequency it is changed along the evolution (change each) we can obtain significant improvements on the CGP flow speed and quality. Moreover, the experiments presented in Section III, indicate a promising future research direction of gradually increasing the BS. It seems that by doing it, CGP could improve even further by balancing fast circuits evaluations at the beginning of the evolutionary process and accuracy estimation at the end. Furthermore, we intend to develop an automated flow that dynamically tests these hyperparameters according to the function at hand. As our results are only for one arithmetic function, these parameters may not be the optimal ones if used in another one.

Another future work, already in progress, is investigating how partially changing the BS every CE may further accelerate the CGP.

Acknowledgment

This work was financed in part by National Council for Scientific and Technological Development – CNPq and the Propesq/UFSC.

References

[1] M. Karnaugh. The map method for synthesis of combinational logic circuits. *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*, 72(5):593–599, 1953.

[2] W. V. Quine. A way to simplify truth functions. *The American Mathematical Monthly*, 62(9):627–631, 1955.

[3] Olivier Coudert and Tsutomu Sasao. *Two-Level Logic Minimization*, pages 1–27. Springer US, Boston, MA, 2002.

[4] R. L. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued minimization for pla optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(5):727–750, 1987.

[5] Robert K. Brayton, Gary D. Hachtel, Curtis T. Mcmullen, and Alberto L. Sangiovanni-Vincentelli. Logic Minimization Algorithms for VLSI Synthesis". *The Kluwer International Series in Engineering and Computer Science*, 2:1–194, 1984.

[6] Brunno A. de Abreu, Augusto Berndt, Isac S. Campos, Cristina Meinhardt, Jonata T. Carvalho, Mateus Grellert, and Sergio Bampi. Fast logic optimization using decision trees. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2021.

[7] Yukio Miyasaka, Xinpei Zhang, Mingfei Yu, Qingyang Yi, and Masahiro Fujita. Logic Synthesis for Generalization and Learning Addition. In *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1032–1037, 2021.

[8] Augusto Berndt, Brunno A. de Abreu, Isac S. Campos, Bryan Lima, Mateus Grellert, Jonata T. Carvalho, and Cristina Meinhardt. Accuracy and size trade-off of a cartesian genetic programming flow for logic optimization. In *Proceedings of the 34th Symposium on Integrated Circuits and Systems Design*, SBCCI '21, 2021.

[9] Augusto André Souza Berndt, Brunno Abreu, Isac S Campos, Bryan Lima, Mateus Grellert, Jonata T Carvalho, and Cristina Meinhardt. A cgp-based logic flow: Optimizing accuracy and size of approximate circuits. *Journal of Integrated Circuits and Systems*, 17(1):1–12, 2022.

[10] Julian Francis Miller. Cartesian genetic programming: its status and future. *Genetic Programming and Evolvable Machines*, pages 1–40, 2019.

[11] Hrishav Bakul Barua and Kartick Chandra Mondal. Approximate computing: A survey of recent trends—bringing greenness to computing and communication. *Journal of The Institution of Engineers (India): Series B*, pages 1–8, 2019.

[12] Shubham Rai and et al. Logic synthesis meets machine learning: Trading exactness for generalization. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021.

[13] H. Riener, W. Haaswijk, A. Mishchenko, G. De Micheli, and M. Soeken. On-the-fly and DAG-aware: Rewriting Boolean Networks with Exact Synthesis. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1649–1654, 2019.

[14] Julian F. Miller and Lukas Sekanina. Cartesian genetic programming and its applications, 2021.

[15] Benjamin Doerr and Carola Doerr. Optimal parameter choices through self-adjustment: Applying the 1/5-th rule in discrete settings. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1335–1342, 2015.

[16] Nikolaus Hansen, Dirk V Arnold, and Anne Auger. Evolution strategies. In *Springer handbook of computational intelligence*, pages 871–898. Springer, 2015.