# An Open-Source Cloud-FPGA Gene Regulatory Accelerator*

**Lucas Bragança[1], Jeronimo Penha[1], Michael Canesche[2], Dener Ribeiro[1]**
**José Augusto M. Nacif[1], Ricardo Ferreira[1]**

[1]Universidade Federal de Viçosa (UFV)
Avenida Peter Henry Rolfs – 36.570-900 – Viçosa – MG – Brazil

[2]Universidade Federal de Minas Gerais (UFMG)
Av. Antônio Carlos – 6627 – 31270-901 – Belo Horizonte – MG – Brazil

{lucas.braganca,jeronimo.penha,jnacif,ricardo}@ufv.br

***Abstract.*** *FPGAs are suitable to speed up gene regulatory network (GRN) algorithms with high throughput and energy efficiency. In addition, virtualizing FPGA using hardware generators and cloud resources increases the computing ability to achieve on-demand accelerations across multiple users. Recently, Amazon AWS provides high-performance Cloud's FPGAs. This work proposes an open source[1] accelerator generator for Boolean gene regulatory networks. The generator automatically creates all hardware and software pieces from a high-level GRN description. We evaluate the accelerator performance and cost for CPU, GPU, and Cloud FPGA implementations by considering six GRN models proposed in the literature. As a result, the FPGA accelerator is at least* $12\times$ *faster than the best GPU accelerator. Furthermore, the FPGA reaches the best performance per dollar in cloud services, at least* $5\times$ *better than the best GPU accelerator.*

## 1. Introduction

Evolutionary models aim at understanding the origin and evolutionary dynamics of phenotypical traits [Chaos and *et al* 2006], where gene interactions are prevalent and critical during development. Genes form complex dynamical systems, named gene regulatory networks (GRN), that can reach several steady states (attractors) [Chaos and *et al* 2006]. Biologists use GRN to identify mechanisms of the complex diseases and therapeutic targets.

Regulatory network models are abstractions of multiple gene interactions. It is possible to represent GRN models using Boolean graph $G(V, E)$, where $V$ is the set of Boolean nodes or genes, and $E$ is the set of edges or gene interactions. GRN models are discrete, and experimental evidence suggests that gene expression is digital and stochastic at the individual cell level rather than continuous [Chaos and *et al* 2006]. One essential operation in GRN models is to compute the steady states or attractors, which has a practical implication: a cell type may correspond to an attractor [Guo et al. 2014]. However, the number of states grows exponentially $2^{|V|}$, where $n = |V|$ is the number of genes.

*Funding: FAPEMIG, National Council for Scientific and Technological Development -– CNPq (Grants #313312/2020-6 and #440087/2020-1 – CNPq/AWS 032/2019) Nvidia, Funarbe. We did this work with the support of the Coordination for the Improvement of Higher Education Personnel – Brazil (CAPES) – Financing Code 001. Support from the laboratories: Intel Academic Compute Environment and the Department of Informatics of the Federal University of Viçosa (UFV).

[1]https://github.com/lesc-ufv/grn_hw_accelerator

The computational problem of finding all the attractors is NP-hard [Akutsu et al. 1998]. Previous work proposes strategies based on BDD [Garg and *et al* 2007] and SAT using CPUs [Dubrova and Teslenko 2011] and multi-cores [Guo et al. 2014]. Moreover, there are also parallel approaches using GPUs [Mizera et al. 2019, Borelli and *et al* 2013], and FPGA-based hardware accelerators [Manica et al. 2020, Ferreira and Vendramini 2010]. Boolean networks can be efficiently mapped on hardware by using FPGA as shown in previous work [Manica et al. 2020, Bragança and *et al* 2017]. However, developing applications for FPGA is a challenging task, and even approaches that use high-level languages such as HLS (High-Level Synthesis) are not trivial for programmers or, in this case, for biologists who study these types of networks.

Amazon AWS cloud resource providers recently started offering FPGA hardware, which becomes widely available to academia and industry to develop accelerators [Braganca and *et al* 2021]. However, there are still many challenges to the widespread adoption of FPGA for the software community. We propose simplifying it by providing an accelerator generator tool to automatically create the hardware and software components from the GRN specification. The user should only write the GRN equations and the state space to explore. The generated code will be deployed and executed in Amazon AWS FPGA instances. The main contributions of this work are: (1) A complete framework that generates code for CPU, GPU, and FPGA to find attractors state in GRNs. (2) A software-hardware codesign for AWS F1 FPGA instance accelerator to find attractors state in GRNs. (3) A study on the cost-effectiveness of finding attractors in GNRs in different types of instances in the AWS cloud. (4) Performance analyses of real-life GRNs models in different platforms.

This paper is organized as following. Section 2 introduces the GRN basic concepts. Section 3 presents the AWS FPGA resources. Section 4 describes the architecture of the proposed accelerator. Section 5 evaluates the accelerator performance on six real-life biological GRNs. Finally, Sections 6 and 7 presents the related work and conclusions.

## 2. Background

### 2.1. Gene Regulatory Networks

A Boolean graph $G(V, E, F)$ models a GRN, where $V$ is the set of genes $\{g_1, g_2, \ldots, g_n\}$, $E$ is the set of edges that represents the gene interactions, and $F$ is the set of update functions. These functions will define the future state of each node based on the previous values of its correlated genes. In this work, we assume a discrete-time simulation $t$ and the synchronous mode, where every node is updated simultaneously.

Figure 1(a) depicts a simple GRN with two genes $a$ and $b$. The value of $a$ will be updated by the function $a(t+1) = f_a(t) = !b(t)$, and the value of $b$ by $b(t+1) = f_b(t) = a(t)\&b(t)$. Assume the initial state $a = 0, b = 0$, as shown in Figure 1(b). The simulation computes the state evolution until a steady state or a steady state cycle is found. The state (10) where $a = 1, b = 0$ is a steady state, which is referred as an attractor. All states should be visited to compute the entire state diagram, which models the GRN dynamics. Figure 1(c) depicts the state diagram for this example, where there are 4 states or $= 2^{|V|} = 2^2$, where $|V|$ is the number of GRN genes. This example has only one single attractor, and all states converge to this attractor.

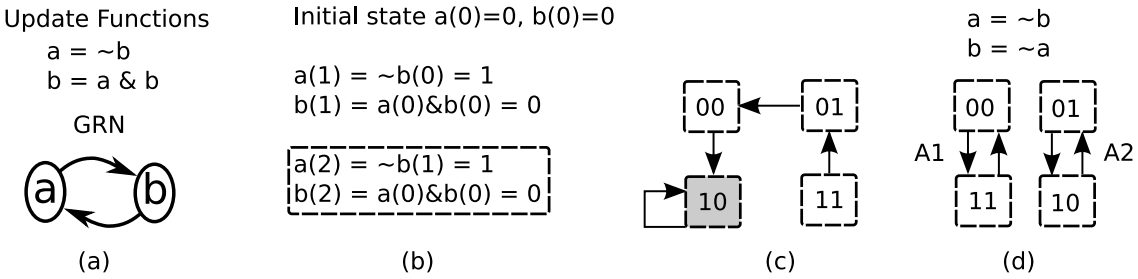


**Figure 1. (a) A simple GRN; (b) Simulation Computation; (c) State Diagram; (d) Another update function and a new state diagram with two attractor $A_1$ and $A_2$.**

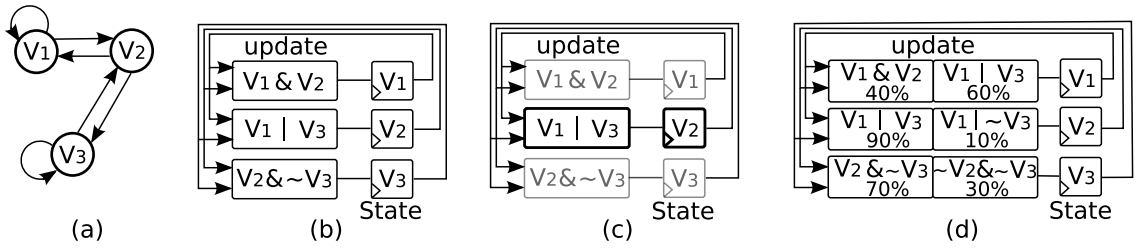


**Figure 2. (a) A three genes network; (b) Synchronous mode; (c) Asynchronous mode; (d) Probabilistic mode.**

## 2.2. Attractors and Update Mode

In general, a GRN has two or more attractors. Let us consider the same simple GRN with only 2 genes by using another update function of the gene $b$, where $b(t+1) = !a(t)$. The GRN dynamic is completely different, as shown in Figure 1(d), where the state diagram has two attractors $A_1$ and $A_2$. Each attractor has two states and a cycle size of 2. An attractor can be associated with a cell type or behavior.

There are three main models: synchronous, asynchronous, and probabilistic. Assume a three-gene network, as shown in Figure 2(a). Figure 2(b) depicts a hardware implementation of synchronous mode, where all genes are updated every clock cycle. Figure 2(c) shows the asynchronous mode where it is possible to update one or more genes at one clock cycle. In this example, the gene $v_2$ is updated, $v_1$ and $v_3$ do not change their value at time $t$. Finally, Figure 2(d) illustrates the probabilistic mode where each gene has one or more probabilistic update rules. In this example $v_1$ has 40% probability of being updated by $v_1 \& v_2$, and 60% probability by $v_1 \| v_2$.All three models have a direct mapping to a hardware implementation in FPGA. In this work, we have designed a synchronous mode generator. However, our design is extensible to handle the probabilistic and asynchronous modes.

As already mentioned, the problem of attractor computation is NP-hard [Akutsu et al. 1998], and several approaches have been presented by using BDD [Garg and *et al* 2007], SAT solver [Dubrova and Teslenko 2011], decompositions [Yuan et al. 2019], GPU [Mizera et al. 2019], and FPGAs [Bragança and *et al* 2017, Penha et al. 2019, Manica et al. 2020]. In this work, we propose to extend the FPGA-based GRN generator proposed in [Bragança and *et al* 2017] to AWS FPGA.

## 3. AWS Amazon FPGA F1 Instances

[Miskov-Zivanov and *et al* 2011] presents an FPGA GRN simulator. However, their I/O interface requires manual configuration. On the other hand, [Bragança and *et al* 2017], and [Manica et al. 2020] propose GRN frameworks that convert high-level inputs into Verilog descriptions, targeting, respectively, Intel Harp and an FPGA attached to a POWER8 processor with a coherent memory system. Nevertheless, these hardware generators include interfaces tailored to specific FPGA platforms, thus requiring physical access to these boards. In this regard, Amazon released compute instances equipped with Xilinx FPGA boards that enable sharing single or multiple FPGAs in the cloud as a service, efficiently scaling and accelerating HPC applications. Nevertheless, FPGA programming is still a complex and challenging task.

Although AWS includes an FPGA Developer AMI and Hardware Developer Kit (HDK), it requires design expertise to develop accelerators. Our generator simplifies these tasks, and the user should only register the accelerator as an Amazon FPGA Image (AFI) performing deployment in just a few clicks. In addition, AWS offers different instance sizes that include up to eight FPGAs per instance. Each FPGA includes local 64 GiB DDR4, approximately 2.5 million LUTs, and about 6,800 DSPs.

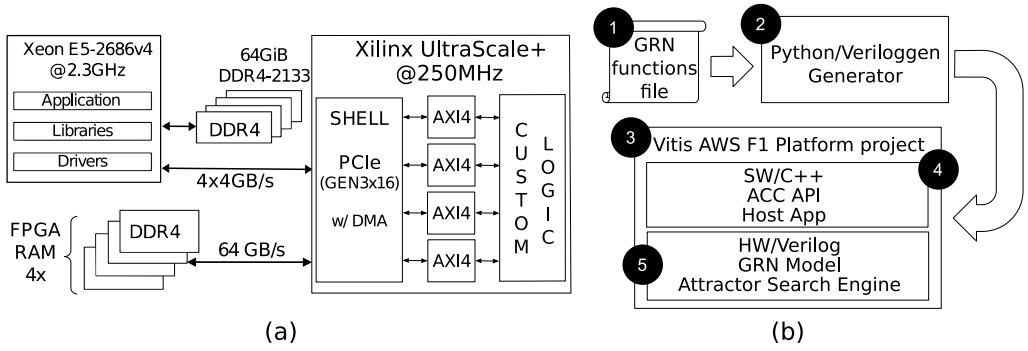


**Figure 3. (a) FPGA AWS F1 Platform; (b) Our Proposed HW/SW GRN Generator.**

Figure 3(a) depicts the AWS FPGA F1 instance platform. The host processor is a Xeon multi-core, and it has 64 GiB DDR memory space. Amazon provides libraries and drivers to communicate to the FPGA. Four PCI-buses implement the communication between the host and the FPGA device, where the maximum throughput is 16 GB/s. The FPGA has a split memory space of 64 GiB DDR memory and a throughput of up to 64 GB/s. AWS reserves a dedicated space (SHELL) inside the FPGA to provide the host/FPGA communication. The user application can only use the custom logic area.

## 4. Accelerator Architecture

The proposed GRN accelerator requires three main steps: generation, configuration, and execution. First, the user specifies the GRN set of update functions, as shown in the step ❶ of Figure 3(b). From this high-level GRN description, the generator creates the front-end C++ code to execute in the host processor and all Verilog code to build the FPGA accelerator. We use Python and Veriloggen [Takamaeda-Yamazaki 2015] to implement a flexible generator as shown in step ❷ of Figure 3(b). Then, the user should upload the accelerator code, a Vitis AWS F1 platform project ❸, to configure the F1 FPGA instances. The C++ host software ❹ and FPGA hardware code ❺ pieces are depicted

in Figure 3(b). Finally, the user specifies the set of states and executes the accelerator application. The following sections detail the accelerator architecture.

### 4.1. Generator

Figure 4(a) depicts the accelerator architecture structure. The generated Verilog code includes the AWS F1 interface, a software/hardware scheduling scheme in addition to the set of processing elements. The accelerator module has three main components: Reader, Arbiter, and Writer. The generator supports several parameters, where the user specifies the number of GRN copies ❼. For each copy, it is possible to create an entire process element unit (PE). The maximum number of copies depends on the size of the update function set and the input/output FIFOs. However, AWS libraries automatically generate the AXI interface, directly impacting the occupied area and throughput. Our generator creates clusters of PEs to share AWS interface resources better and overcome these constraints. For instance, assume it is possible to implement 128 GRN accelerator copies, and the cluster size is 32. The generator automatically creates four clusters or modules, as shown in Figure 4(a), where each cluster has an AXI interface and 32 PEs.

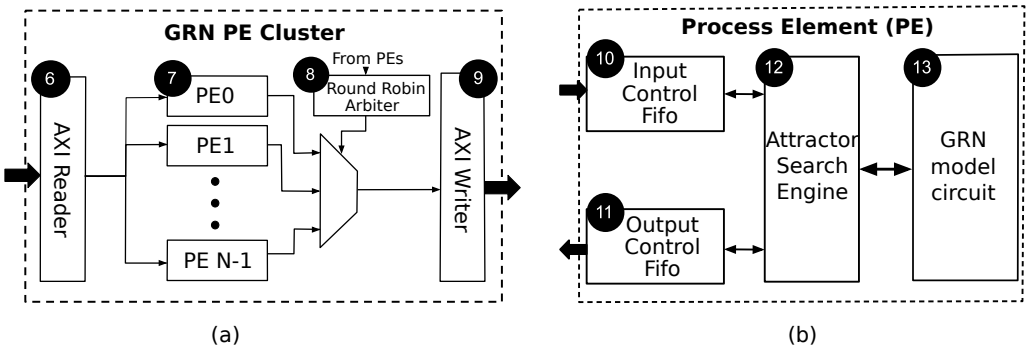


**Figure 4. AXI4 IP Accelerator: (a) A Cluster of PEs; (b) PE internal architecture.**

First, the AXI Reader ❻ receives a set of initial state packages. Each PE has an ID and a local task FIFO. We propose a distributed software and hardware scheduler where the host and the hardware accelerator explore the trade-offs to balance the computation workload across the PEs better. The size of the GRN parameterizes the Reader interface, and the input package has three fields: initial state, final state, and ID. The PE input FIFO interface verifies the ID, and if the ID matches, the PE enqueues the request. Assume a 70 genes GRN. The initial and final state field size is a multiple of 8 bits. For this example, each state requires 9 bytes (or 72 bits) package. The reader interface is portable, parametrized, and flexible to support new F1 devices or other FPGA cloud servers. Furthermore, as already mentioned, the cluster of PEs maximizes the I/O communication and reduces the resources required to implement the AXI interface. Although AWS provides a generic library to instantiate as many AXI interfaces as possible, the AWS F1 FPGA instance has four physical AXI4 channels, where each channel has a 512-bit width (64 bytes). The Reader interface and the PE clusterization optimize the package and the cluster size to better match the AXI4 bit width.

Each cluster or module has an arbiter ❽ which performs a round-robin policy to send the results back to the host CPU through the writer interface. The number of cycles to compute an attractor depends on the GRN dynamic and the initial state. Therefore

the PE workload is not homogeneous. In addition, a PE can receive a range of states to compute. For instance, the host sends the package (0,9,3), where 0 is the first state, 9 is the last state, and 3 is the PE ID. Therefore, the $PE_3$ computes the attractors for the states $0, 1, 2, \ldots 9$, and send back 10 output packages. Each output package has the attractor state ID, the number of cycles to reach this attractor (transient), and the size of the attractor. Therefore, the host software dynamically adjusts the PE workload for a given GRN without creating and instantiating a new hardware accelerator. When the PE finishes the computation, the attractor engine inserts the result in the output FIFO, which waits until the next round-robin time slot to send it back to the CPU. Finally, the AXI writer ❾ handles the output package to send back to the host CPU.

Figure 4(b) depicts an overview of the PE internal structure. First, the input ❿ and output ⓫ control FIFO implement an asynchronous PE interface to mitigate the unbalanced workload. One PE can receive more than one state package until fulfill the input FIFO. For each state in the assigned state ranging, the PE computes the attractor by using the one-two steps algorithm proposed in [Bhattacharjya and Liang 1996]. The attractor search engine ❿ has two registers T and L, to store the transient and the attractor lengths, similar to the PE proposed in [Bragança and *et al* 2017].

### 4.2. Execution

The generator also creates the host code in OpenCL. Although OpenCL provides a standard interface for parallel computing using task – and data-based parallelism, the code is verbose even for simple tasks. Our generator reduces the accelerator software complexity by automatically creating the FPGA OpenCL code and the specific AWS libraries. Thus, the user should only create an input file with a set of state ranges to explore. The accelerator then uses this input file to compute the attractors and their transients and outputs a CSV file with this information, which can be analyzed later.

## 5. Experimental Results

This section presents the experiments to evaluate the generator and compare the results with other AWS services. For this, we create three code generators targeting: (a) a sequential CPU, (b) a multi-core CPU by using openMP, and (c) GPU by using CUDA. Our goal is to compare the performance of heterogeneous cloud architectures. In addition, we also evaluated the cost to allocate these architectures in Amazon cloud services [Hashemipour and Ali 2020]. There is a wide range of devices in AWS. Therefore, we have selected low-cost as well as high-performance and instances for each platform.

### 5.1. Experimental setup

Table 1 shows the main features of each AWS instance used in the experimental setup. For FPGA, we select the *f1.2xlarge* instance, which is equipped with 8 vCPUs comprised of Intel Xeon E5-2686 v4 series processors (Broadwell) running at frequencies up to 2.2 GHz, Xilinx VU9P FPGAs and 122 GB of RAM [Hashemipour and Ali 2020]. We perform our experiments in this FPGA instance. However, the generated code can also execute in the *f1.4xlarge* and *f1.8xlarge*, which have four and eight FPGA boards, respectively. The instance price grows linearly with the number of FPGA devices.

For a multi-core, we select two configurations. The first one is the *c5a.16xlarge* instance, which is ideal for algorithms that require high-performance processors. This instance has 64 vCPUs composed of second-generation AMD EPYC 7002 series processors running at frequencies up to 3.3 GHz and 128 GB of RAM [Hashemipour and Ali 2020]. The code generator creates a parallel version of the attractor algorithm [Bhattacharjya and Liang 1996] using the OpenMP library to explore multithreading capabilities.

For GPU devices, we evaluate three AWS options to explore the cost/performance trade-offs. Since our application does not require specific resources such as tensor cores nor high memory throughput, it is interesting to compare the relative cost and performance trade-offs of a V100 7th generation GPU, low energy T4 8th generation GPU, and a K80 3rd generation GPU. The lowest-cost AWS instance with GPU is the **g4dn.xlarge**. On the other hand, the highest-cost GPU instance is the **p3.2xlarge** with a V100 GPU, 8 vCPUs Xeon E5-2686 v4, 2.2 GHz, and 61 GB of RAM.

**Table 1. AWS instance properties**

| Name | AWS Instance | CPU | Memory (GB) | Cores | Accelerator | Price On-Demand ($) |
|---|---|---|---|---|---|---|
| **T4G** | t4g.medium | AWS Graviton2 | 4 | 2 | - | 0.0336 |
| **T4 GPU** | g4dn.xlarge | Xeon Scalable v2 | 16 | 4 | NVIDIA T4 | 0.526 |
| **K80 GPU** | p2.xlarge | Xeon E5-2686 v4 | 61 | 4 | NVIDIA K80 | 0.9 |
| **FPGA** | **f1.2xlarge** | Xeon E5-2686 v4 | 122 | 8 | **Xilinx VU9P** | **1.65** |
| **C5A** | c5a.16xlarge | AMD EPYC 7R32 | 128 | 64 | - | 2.464 |
| **V100 GPU** | p3.2xlarge | Xeon E5-2686 v4 | 61 | 8 | NVIDIA V100 | 3.06 |

## 5.2. Real-life Biological Genetic Regulator Networks

We perform our experiments with real-life biological GRN. We have selected six GRN of the literature: (a) the B_bronch, a respiratory bacterium and a gastrointestinal helminth (BTC) [Thakar et al. 2012]; (b) the Colitis-Associated Colon cancer(CAC) [Lu and *et al* 2015]; (c) the epidermal growth factor receptor (EGFR)[Samaga and *et al* 2009]; (d) the CD4+ immune effector T-cell (CD4) [Conroy and *et al* 2014]; (e) the ErbB receptor signal transduction in human mammary epithelial cells (ERB) [Helikar et al. 2013]; and (f) the macrophase for the immunity system (SMA) [Raza and *et al* 2008]. Table 2 shows the main characteristics of each GRN. For these GRNs, the state space size ranges from $10^{15}$ up to $10^{96}$, which is higher than the number of atoms in the observable universe.

**Table 2. Real-life Biological Genetic Regulator Network properties**

| Full Name | Name | Nodes | Edges | Operations |
|---|---|---|---|---|
| B bronchiseptica and T retortaeformis coinfection | BTC | 53 | 160 | 139 |
| Colitis-Associated Colon cancer | CAC | 70 | 154 | 118 |
| EGFR & ErbB signaling | EGFR | 104 | 377 | 370 |
| CD4 T cell signaling | CD4 | 188 | 406 | 301 |
| ErbB (1-4) Receptor Signaling | ERB | 247 | 1,954 | 2,242 |
| Signaling in Macrophage Activation | SMA | 321 | 553 | 254 |

Table 2 sorts the GRN by the number of nodes. The execution time grows as a function of the number $N$ of nodes. However, it also depends on the network dynamic

**Table 3. FPGA resource usage**

| GRN | Copies | LUT | REG | LUT as Mem | Fmax MHz |
|---|---|---|---|---|---|
| BTC | 128 | 74.0K (8.3%) | 127.2K (6.5%) | 18.0K (3.3%) | 251 |
| CAC | 128 | 89.2K (9.98%) | 153.5K (7.86%) | 21.0K (3.83%) | 250.38 |
| EGFR | 128 | 110.0K (12.31%) | 199.1K (10.20%) | 28.2K (5.13%) | 251.13 |
| CD4 | 32 | 46.4K (4.65%) | 82.3K (3.86%) | 11.8K (2.05%) | 251.13 |
| ERB | 32 | 76.4K (7.68%) | 136.3K (6.42%) | 19.5K (3.40%) | 251.13 |
| SMA | 32 | 69.K (7.01%) | 107.5K (5.07%) | 15.6K (2.73%) | 239.87 |

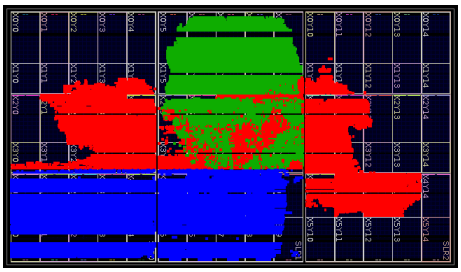

**Figure 5. Design utilization for EGRF GRN.**

controlled by the update functions. First, if the average transient and the attractor size are shorter, the accelerator process each state in a few cycles. Second, the edge complexity and the size of each update rule (column **operations**) also have a direct impact on the execution time as well as in the FPGA area occupied by the GRN circuit. For instance, the ERB GRN has 247 nodes, significantly smaller than SMA GRN, with 321 nodes. However, the ERB has several operations about 8.8 times greater than that of the SMA.

### 5.3. FPGA Resources

Table 3 summarizes the FPGA resources required for each GRN hardware accelerator. As already mentioned, it is possible to increase the number of accelerator copies for smaller networks to improve the performance by using spatial parallelism. For example, BTC, CAC, and EGFR GRN accelerators can allocate 128 copies, organized into four clusters of 32 copies each. For the more extensive networks, with more than 188 nodes, the FPGA place and routing tools have successfully mapped 32 copies. Column **LUT, REG,** and **LUTasMEM** depict the total number of FPGA lookup tables, registers, and memories for the accelerator module. These values do not include the interface resources and the AWS hardware APIs. The clock frequency was kept stable at the highest possible value allowed by the AWS FPGA platform at 250 MHz depicted in the last column.

Although the FPGA resource utilization is relatively low for kernel circuits, it is important to highlight that the input/output controller circuitry and the AWS shell consume a significant amount of FPGA resources. For EGFR, the total resources used is 30.6% of LUT, 10.3% of LUTAsRAM, and 24.3% of REG. Figure 5 depicts a graphical representation of the FPGA resource utilization. The blue area represents the AWS SHELL, and the accelerator can not allocate these quadrants to place the custom logic. The red area is the allocated resources for data inputs/outputs, and only the green area represents the accelerator kernel itself. The major challenge in increasing the number of copies was related to internal limitations in the FPGA used due to the maximum number of proper buses for clock signal propagation. Despite the limitations, the FPGA accelerator achieves lower execution time than multi-threaded executions on CPU and GPUs, as shown in the next sections.

### 5.4. Performance Analysis

To perform the performance analysis of the three platforms we generated the codes for the six real-world GRNs described in Table 2. For the smaller networks BTC, CAC,

EGFR, and CD4, we explore $2^{25}$ states, and for the larger networks, we explore $2^{24}$ states. As the number of evaluated states differs according to each network, we normalize the results. The execution time was measured by using *chrono::highresolutionclock* functions.

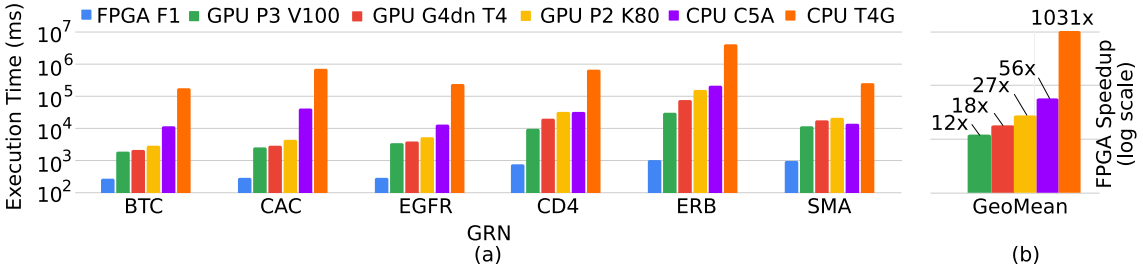


**Figure 6. (a) Execution time in milliseconds for the GRNs on each AWS instance. Lower is better; (b) FPGA Speedup.**

Figure 6(a) depicts the execution time in milliseconds for the six evaluated accelerator devices available in the AWS: an FPGA F1 instance, a low and a high-performance multi-core instances T4G and C5A, and three GPU instances (V100, T4, and K80). We sort the AWS instances in ascending order for the execution time. The FPGA accelerator shows a lower execution time for all evaluated scenarios. Figure 6(b) depicts the average speedup factor for the FPGA approach in comparison to multi-core and GPU devices.

FPGA accelerator explores spatial parallelism at two levels. As shown in Figure 2(a), the hardware implementation evaluates all update rules in parallel in one clock cycle, which provides a significant increase in the instruction-level parallelism (ILP). In addition, the generator creates as many GRN copies as possible, which increases the task parallelism (see Table 3). Considering the multi-core implementation, we used the maximum number of thread/core: 2 for the T4G AWS instances and 64 for the C5A AWS instances. Finally, each GPU thread executes an entire attractor computation for a given state by exploring the SIMT parallelism for GPU execution. Although the CPU and GPU operational frequency is more than $10\times$ faster than the FPGA, the ILP and task FPGA parallelisms overcome this drawback to provide better acceleration. The GPU takes advantage of the massive number of cores to provide better acceleration than the CPU devices. However, each GPU thread consumes a large number of registers due to the complexity of the GRN update rules, which reduces the number of scheduled execution threads in the GPU stream processors. A GPU V100 has 5,120 cores, and it is around $4.7\times$ faster than a 64-core CPU. However, it has $80\times$ more cores. It is important to highlight that it is a simple comparison since a GPU core cost and performance are not equivalent to the simple GPU cores. In summary (see Figure 6(b), the FPGA is $12\times$, $18\times$, $27\times$, $56\times$, and $1031\times$ faster than the V100 GPU, T4 GPU, K80 GPU, 64-core high performance CPU, 2-core low-cost CPU, respectively.

Cloud computing delivers computing resources as a service, where various pricing and tariff models create new cost metrics. In addition to execution time, we propose to evaluate the performance per Dollar by measuring the number of attractor computations per Dollar. Figure 7(a) depicts the number of Giga states processed per Dollar for the six GRN networks in the six AWS instances, where a high number means a better performance per Dollar. These results demonstrate one more advantage in the use of FPGAs when compared to other accelerator approaches. Although an FPGA cost is 1.65 US$ per

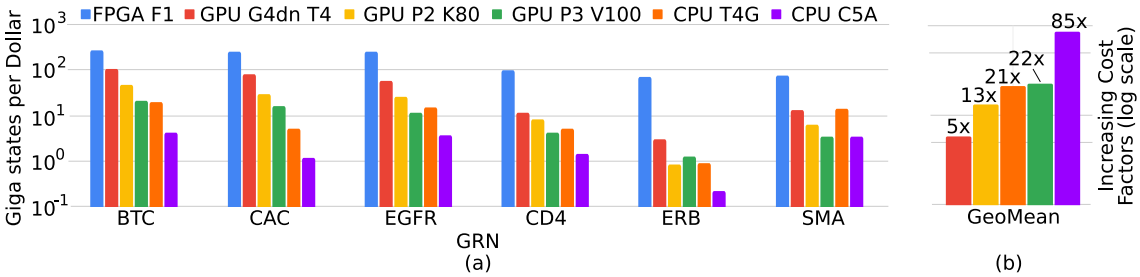


**Figure 7. Performance per dollar: (a) Giga processed states per Dollar in the six evaluated AWS instances. High is better. (b) Increasing cost factor, where lower is better.**

hour, which is $3.1\times$ more expensive than T4 GPU, the FPGA is $18.1\times$ faster than T4. On average, the increase in cost per state is approximately $5.7\times$. Moreover, expensive GPU resources such as tensor cores in V100 are useless to compute attractors. Even the T4 GPU is $1.5\times$ slower than the V100, and it is $6\times$ less expensive. Therefore, it is better to instantiate six $T4$ instead of one $V100$, which will be $4\times$ faster. Figure7(b) depicts the increase in cost factor to compute the attractor per Dollar relative to the FPGA performance per Dollar. The AWS instances pricing has been reported in Table 1. The 64-core instance has the worst performance per Dollar. It is $85\times$ worst than the FPGA, while the 2-core is only $21\times$. However, the 2-core is $1031\times$ and $18\times$ slower than the FPGA and the 64-core, respectively. Therefore, the FPGA achieves the best performance and the best performance per Dollar in all scenarios.

## 6. Related Work

GRN models involve expensive computation tasks, where HPC approaches using parallel accelerators and divide-and-conquer techniques can be applied. [Borelli and *et al* 2013] propose a GPU-based approach to perform the GRN inference from gene expression data. This work focus on the attractor computation of GRNs. The first challenge is the scalability, which was held in 2006 by [Irons 2006] to deal with Boolean networks up to 80 nodes. Then, [Guo et al. 2014] present a block decomposition approach to reduce the problem complexity by using strongly connected components and SAT solver to identify the attractors in the state transition graph of each block. Furthermore, a parallel multi-core approach is presented, which reaches an average speedup of 1.47, 2.06, and 2.64 on 2-core, 4-core, and 8-core [Guo et al. 2014].

[Yuan et al. 2019] deal with GRN sizes range from 100 to 500 nodes on synthetic random generated networks and real-life biological networks up to 100 nodes using a decomposition-based technique. However, this step is performed on the network structure and not on the network state transition [Guo et al. 2014]. Therefore, a block partially preserves the attractors of the original GRN. The block attractors are detected, and the dependency relationships between blocks reduce the state-space, thus accelerating execution time. Finally, the attractors of the entire GRN are computed from the block's attractors.

Recently, [Manica et al. 2020] achieves an order of magnitude speedup over a multi-threaded by using FPGA-based implementation, which first converts the model to Verilog to execute on an FPGA coherently attached to a POWER8 processor. The proposed FPGA-implementation is around $10^2 - 10^3\times$ faster than the Boolnet R pack-

age [Müssel et al. 2010] by computing on average 2M attractors per second. Considering the CAC GRN, their FPGA implementation requires 1.57s to visit $2^{25}$ states, while our approach spends 131 ms being $11\times$ faster. The authors in [Manica et al. 2020] argue that it is possible to explore spatial parallelism although not presenting any implementation.

## 7. Conclusions

This work proposes to explore the emergence of FPGAs in the cloud to develop a hardware accelerator for gene regulatory networks by deploying it in the AWS FPGA cloud. We are the first to report GRN hardware acceleration using Amazon AWS FPGAs to the best of our knowledge. Our GRN accelerator generator provides an easy interface for users to computer attractors without being responsible for the hardware design flow and FPGA resources management. Furthermore, we evaluate the performance per dollar, which is an important metric in the cloud services. The FPGA offers high instruction-level parallelism at the bit level and scalability to instantiated more accelerator copies by exploiting the spatial parallelism. Our accelerator is $12\times$ faster than the best solution on GPU, and it reaches a performance per dollar $5.7\times$ better than the best GPU solution. The generator code is open source[2] and modular to be reused to develop a new accelerator for GRN and AWS F1 instances. Future work will investigate the GRN algorithm for synchronous, asynchronous, and probabilistic models.

## References


Akutsu, T., Kuhara, S., Maruyama, O., and Miyano, S. (1998). A system for identifying genetic networks from gene expression patterns produced by gene disruptions and overexpressions. *Genome Informatics*, 9:151–160.

Bhattacharjya, A. and Liang, S. (1996). Median attractor and transients in random boolean nets. *Physica D: Nonlinear Phenomena*, 95(1):29–34.

Borelli, F. F. and *et al* (2013). Gene regulatory networks inference using a multi-gpu exhaustive search algorithm. *BMC bioinformatics*, 14(18):1–12.

Braganca, L. and *et al* (2021). An open source custom k-means generator for aws cloud fpga accelerators. In *Brazilian Symp on Computing Systems Engineering (SBESC)*.

Bragança, L. and *et al* (2017). Exploring the dynamics of large-scale gene regulatory networks using hardware acceleration on a heterogeneous cpu-fpga platform. In *IEEE International Conference on ReConFigurable Computing and FPGAs (ReConFig)*.

Chaos, A. and *et al* (2006). From genes to flower patterns and evolution: dynamic models of gene regulatory networks. *Journal of Plant Growth Regulation*, 25(4):278–289.

Conroy, B. and *et al* (2014). Design, assessment, and in vivo evaluation of a computational model illustrating the role of cav1 in cd4+ t-lymphocytes. *Frontiers in immunology*, 5.

Dubrova, E. and Teslenko, M. (2011). A sat-based algorithm for finding attractors in synchronous boolean networks. *IEEE Trans. on Comp. Biology and Bioinformatics*.

Ferreira, R. and Vendramini, J. (2010). Fpga-accelerated attractor computation of scale free gene regulatory networks. In *Field Programmable Logic and Applications (FPL)*.


[2] `https://github.com/lesc-ufv/grn_hw_accelerator`

Garg, A. and *et al* (2007). An efficient method for dynamic analysis of gene regulatory networks and in silico gene perturbation experiments. In *Int Conf on Research in Computational Molecular Biology*.

Guo, W., Yang, G., Wu, W., and Sun, M. (2014). A parallel attractor finding algorithm based on boolean satisfiability for genetic regulatory networks. *PloS one*, 9(4).

Hashemipour, S. and Ali, M. (2020). Amazon web services (aws)–an overview of the on-demand cloud computing platform. In *Int Conf for Emerging Technologies in Computing*. Springer.

Helikar, T., Kochi, N., Kowal, B., Dimri, M., Raja, S. M., Band, V., Band, H., and Rogers, J. A. (2013). A comprehensive, multi-scale dynamical model of erbb receptor signal transduction in human mammary epithelial cells. *PLoS One*, 8(4):e61757.

Irons, D. J. (2006). Improving the efficiency of attractor cycle identification in boolean networks. *Physica D: Nonlinear Phenomena*, 217(1):7–21.

Lu, J. and *et al* (2015). Network modelling reveals the mechanism underlying colitis-associated colon cancer and identifies novel combinatorial anti-cancer targets. *Scientific reports*, 5(1):1–15.

Manica, M., Polig, R., Purandare, M., Mathis, R., Hagleitner, C., and Martinez, M. R. (2020). Fpga accelerated analysis of boolean gene regulatory networks. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 17(6):2141–2147.

Miskov-Zivanov, N. and *et al* (2011). Emulation of biological networks in reconfigurable hardware. In *ACM Conference on Bioinformatics, Computational Biology and Biomedicine*, pages 536–540.

Mizera, A., Pang, J., and Yuan, Q. (2019). Gpu-accelerated steady-state computation of large probabilistic boolean networks. *Formal Aspects of Computing*, 31(1):27–46.

Müssel, C., Hopfensitz, M., and Kestler, H. A. (2010). Boolnet—an r package for generation, reconstruction and analysis of boolean networks. *Bioinformatics*, 26(10).

Penha, J., Braganca, L., Nacif, J., and Ferreira, R. (2019). Add: Accelerator design and deploy-a tool for fpga high-performance dataflow computing. *Concurrency and Computation: Practice and Experience*, 31(18):e5096.

Raza, S. and *et al* (2008). A logic-based diagram of signalling pathways central to macrophage activation. *BMC systems biology*, 2(1):36.

Samaga, R. and *et al* (2009). The logic of egfr/erbb signaling: theoretical properties and analysis of high-throughput data. *PLoS computational biology*, 5(8):e1000438.

Takamaeda-Yamazaki, S. (2015). Pyverilog: A python-based hardware design processing toolkit for verilog hdl. In *Applied Reconfigurable Computing*, volume 9040 of *Lecture Notes in Computer Science*, pages 451–460. Springer International Publishing.

Thakar, J., Pathak, A. K., Murphy, L., and Cattadori, I. M. (2012). Network model of immune responses reveals key effectors to single and co-infection dynamics by a respiratory bacterium and a gastrointestinal helminth. *PLoS computational biology*, 8(1).

Yuan, Q., Mizera, A., and Qu, H. (2019). A new decomposition-based method for detecting attractors in synchronous boolean networks. *Science of Computer Programming*.