Large Scale Cellular Automata on FPGAs: A New Generic Architecture and a Framework

NIKOLAOS KYPARISSAS* and APOSTOLOS DOLLAS, Technical University of Crete, Greece

Cellular Automata (CA) are discrete mathematical models discovered in the 1940s by John von Neumann and Stanislaw Ulam, and used extensively in many scientific disciplines ever since. The present work evolved from a Field Programmable Gate Array (FPGA)-based design to simulate urban growth into a generic architecture which is automatically generated by a framework to efficiently compute complex cellular automata with large 29×29 neighborhoods in Cartesian or toroidal grids, with 16- or 256-states per cell. The new architecture and the framework are presented in detail, including results in terms of modeling capabilities and performance. Large neighborhoods greatly enhance CA modeling capabilities, such as the implementation of anisotropic rules. Performance-wise, the proposed architecture runs on a medium-size FPGA up to 51 times faster vs. a CPU running highly optimized C code. Compared to GPUs the speedup is harder to quantify, because CA results have been reported on GPU implementations with neighborhoods up to 11×11, in which case FPGA performance is roughly on par with GPU; however, based on published GPU trends, for 29×29 neighborhoods the proposed architecture is expected to have better performance vs. a GPU, at one-tenth the energy requirements. The architecture and sample designs are open source available under the creative commons license.

CCS Concepts: • **Computer systems organization → Real-time system architecture**; **Special purpose systems**; • **Computing methodologies → Real-time simulation**; • **Hardware → Reconfigurable logic applications**; **Hardware accelerators**.

Additional Key Words and Phrases: cellular automata, generic architecture, framework, FPGA accelerator

ACM Reference Format:
Nikolaos Kyparissas and Apostolos Dollas. 2020. Large Scale Cellular Automata on FPGAs: A New Generic Architecture and a Framework. *ACM Trans. Reconfig. Technol. Syst.* 14, 1, Article 5 (December 2020), 32 pages. https://doi.org/10.1145/3423185

1 INTRODUCTION

Cellular automata (CA) are Turing complete, discrete mathematical models discovered in the 1940s by John von Neumann and Stanislaw Ulam [56, 58]. Through time, these highly parallelizable mathematical models have been used to study and model physical processes. Ever since the early days of FPGA-based computing, such CA architectures have been demonstrated to offer excellent performance vs. general-purpose ones. Every decade or so novel FPGA-based architectures have been developed with new architectural approaches rather than existing architectures mapped on new FPGA technologies. Within the last few years, extending typical CA rules to large-scale rules has provided new aspects of modeling physical processes with realistic features and results. Our work is a new architecture for CA execution on FPGA technology (the term used by the CA community is "CA simulation"), together with a framework to generate the entire architecture, including the necessary timing constraints for graphics display.

*Currently with the APT Group, Department of Computer Science, University of Manchester, UK.

Authors' address: Nikolaos Kyparissas, nikolaos.kyparissas@postgrad.manchester.ac.uk; Apostolos Dollas, dollas@ece.tuc.gr, School of ECE, Technical University of Crete, Chania, 73100, Greece.

1.1 Motivation

A myriad physical processes can be modeled with CA, from chemistry and molecular dynamics [26, 46, 49, 50] to large ecosystems [23] and artificial brains [16]. In cosmology, digital physics suggests that a universal CA computes the evolution of the universe [62]. Apart from modeling physical phenomena, CA are also used as abstract computational systems in various applications. The *Universal Constructor*, the first CA designed by John von Neumann in the 1940s, is an abstract machine which demonstrates the logical requirements for machine self-replication [4, 58, 59]. Since then, the mathematical properties and modeling potential of CA have been meticulously studied, with universality and reversibility being essential properties of numerous CA rules [2, 38, 47, 53]. The potential hidden in their inherent parallelism and modeling capabilities has greatly motivated computer scientists and computer engineers to advance the field, and it would be an omission to not mention John Horton Conway's (1937 - 2020) 1970's "Game of Life" which brought CA to the general audience [14].

1.2 Key Ideas, Contributions, and Project Timeline

The main idea behind this work is that with judicious use of the FPGA internal memory (Block RAM - BRAM) and a well-dimensioned architecture we can exploit the immense internal data transfer bandwidth and available parallelism of a present-day FPGA in order to parallelize computations in large CA neighborhoods. The DDR external memory - a bottleneck in CA execution on CPUs and GPUs - is used once per cell read and once per cell write during each CA iteration. Thus, FPGAs are an ideal technology for high complexity CA, because a CPU cannot have such fine grain parallelism and at best will operate at L1 cache speeds, whereas a GPU will have many of its resources underutilized. With large neighborhoods (in our case, up to 29×29), for each datum entered into our accelerator, there will be $O(29^2)$ operations, as long as the BRAM can hold buffers of 30 *rows* (for 30 rows a typical BRAM can store tens of thousands of cells per row). In addition, the customizable datapath of FPGAs allows for the implementation of many states per cell and complex CA rules, leading to CA models which to date have been too expensive to compute.

The specific contributions of this paper are:

- Brief presentation of the original (non-generic) architecture and its development.
- Generic architecture presentation in detail (datapath, custom buffer structure and data forwarding mechanism, pipelining, graphics - processing synchronization, memory subsystem).
- Detailed description of the framework and how it generates the design.
- Results from actual runs, not only highlighting performance vs. CPUs and GPUs, but also how the ability to implement CA with large neighborhoods is a "game changer": it reveals patterns which do not appear otherwise, and it allows for new kinds of CA algorithms to be developed (we demonstrate how anisotropic rules can be executed on the proposed architecture).
- The entire architecture is available as open-source under the Creative Commons license, and it can be found in: `https://github.com/nkyparissas/Cellular_Automata_FPGA`

The current work started as a project to model urban development. Over a period of five years it evolved into a generic architecture for CA with many states (up to 256), large neighborhoods (up to 29×29), virtually unconstrained Cartesian or toroidal grids, a VGA interface to display the execution of the model in real time, and a framework to generate FPGA designs for new rules easily. This work achieved twice top-12 distinction in the Xilinx Open Hardware design competition (in 2015 and 2018). In 2019, applications of large-scale CA and the key elements of the architecture were published in the international conferences HPCS [31] and FPL [32], respectively. The current paper is a complete in-depth presentation of the new architecture and framework.

1.3 Paper Outline

This paper is divided into seven sections. Following the introductory section:

- Section 2 comprises an overview of related works and approaches that have been proposed to accelerate CA simulations with the use of FPGAs.
- Section 3 contains the theoretical background and terminology necessary for one to understand the basic concepts of CA. It also provides the timeline of the project.
- Section 4 describes in full detail the new CA architecture. In addition, key architectural decisions and system dimensioning are presented.
- Section 5 presents the framework by describing the scalable architecture design's ability to automatically perform the process of resource dimensioning, allocation, interconnection and synchronization, and generate a ready-to-go system.
- Section 6 uses examples to showcase the design's features and advantages over conventional methods as well as some interesting results obtained with it. Furthermore, comparative performance results are presented and discussed in this section.
- Finally, section 7 provides a summary of the present work and conclusions drawn from the experimental results, and it discusses potential future extensions.

2 RELATED WORK

This section has an overview of prior FPGA-based CA accelerators. The first part presents a brief description of significant prior results and architectures. A short summary of other noteworthy developments follows, leading to the rationale and approach followed in the present work. The prevailing approaches to design hardware for CA simulations in space are "template over a grid area" and "grid of processors", whereas in time the prevailing approaches are "completion of an iteration before the next one" and "multiple iterations over an area in order to reduce memory requirements". A more detailed coverage of the subject can be found in [52].

2.1 Toffoli and Margolus' Cellular Automata Machines (1984–2000)

Toffoli and Margolus' *Cellular Automata Machines* (*CAM*) were the first special-purpose computers designed to accelerate CA simulations. Tommaso Toffoli, who invented the universal reversible logic gate named after him, has been working on reversible CA and reversible computing since the 1970s. Early efforts on a programmable, high-performance TTL- and memory-built CA engine were published in 1984 [54]. Toffoli explains that in a truly parallel implementation of CA each cell would have to be implemented as an independent element having access to its own copy of the rule's transition function, its own state variables and its neighborhood values. However, this approach is in practice restricted by various technological constraints and is not suitable to accelerate large-sized simulations. The alternative approach followed by Toffoli consists of a single, static RAM (SRAM)-based look-up table (LUT) to implement the transition function. This unit is "time-shared" between cells. In other words, *CAM*'s architecture processes a stream of cells and their neighborhoods sequentially in order to, eventually, update the whole grid and produce a new frame of the simulation. The issue occurring with this approach is that, as soon as a cell is updated, the neighboring cells will immediately see the new state rather than the old one which is needed. To address this and other problems (e.g., grid boundary conditions), *CAM* uses *double buffering*, i.e. two copies of each frame ("previous" and "next"), a technique used in virtually all CA architectures.

The *CAM* architecture was fully pipelined and its memory consisted of 8 bit-planes, with each plane contributing a single bit to each 8-bit cell (fig. 1). The memory planes' size was 256×256 sites and they provided the transition function with all 3×3 neighborhood cells simultaneously.

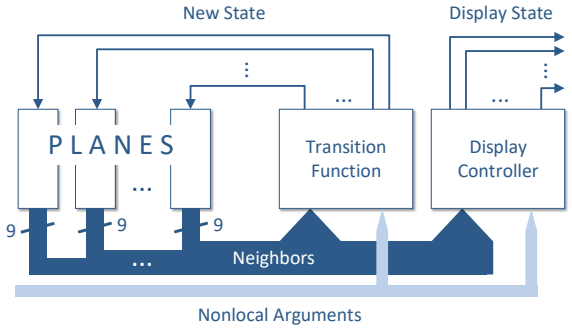

Fig. 1. CAM's basic computational loop. The transition function hardware module is an SRAM LUT.

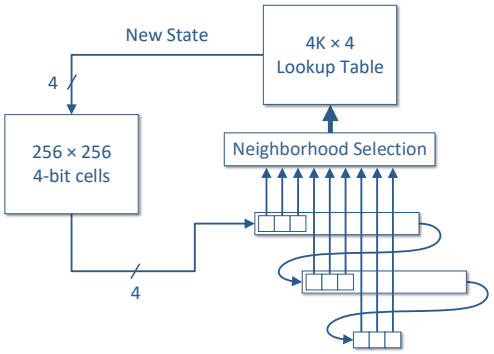

Fig. 2. CAM-6's pipeline buffer provides the transition function with all 3×3 neighborhood cells simultaneously.

These features made the system significantly faster than any general-purpose computer at the time, as *CAM* could display the evolution of 256×256 8-bit cells in real time.

Norman Margolus, a PhD student at the time, worked with Toffoli on further developing the prototype. During the next two years the machine's capabilities started growing, many versions followed and in 1986 *CAM-6* was completed. It spawned a book [55], which showcases the machine's numerous applications, and was produced commercially as a PC expansion board.

CAM-6's architecture was fully pipelined and its memory consisted of 4 bit-planes, with each plane contributing a single bit to each 4-bit cell. The size of each plane was 256×256 sites. A pipeline buffer (fig. 2) provided the transition function with all 3×3 neighborhood cells simultaneously. Similar to its predecessor, *CAM-6* could display the evolution of 256×256 cells in real time. A most interesting feature was the ability to rearrange the interconnection of the planes. The user could, at the expense of having 16 states per cell, either obtain multidimensional CA simulations by "stacking" planes on top of one another, or simulate a larger grid by "gluing" planes edge-to-edge. Larger grids could also be simulated by using a technique called *scooping*: the large grid is stored in the host computer's memory and *CAM-6*'s internal 256×256 grid is used as a cache memory.

Margolus continued developing *CAMs* within the next few years, with *CAM-8* published in 1995 [34]. It was a multiprocessor version of its predecessors, with interconnected *CAM*-like modules processing separate sectors of the CA grid simultaneously (fig. 3). The sectors could be rearranged

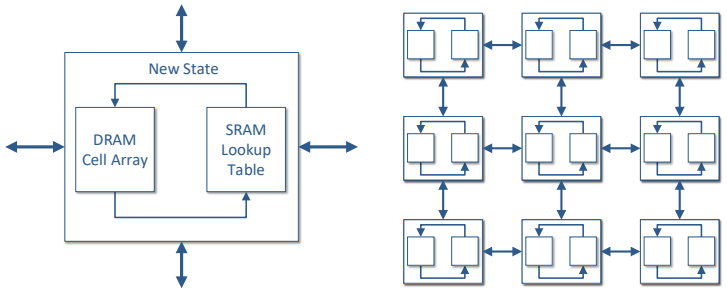

Fig. 3. CAM-8 architecture. Each processing node was connected only to its nearest-neighboring module.

in any way forming n-dimensional spaces of any size, provided the user had enough *sector modules* in their *CAM-8*. The machine supported arbitrarily large neighborhood sizes and cell sizes in bits. Following the steps of its predecessors, *CAM-8*'s performance was outstanding at that time. Its shift-based data manipulation was ideal for simulation of lattice gas automata and it could process a 1000×2000 FHP gas model at a rate of 190 frame updates per second. In general *CAM-8* could display the evolution of 512×512 cells in real time. However, for large neighborhoods the performance dropped significantly. For example, when simulating a CA rule with 3-bit cells and an 11×11 neighborhood size, *CAM-8* could display 10 generations per second.

Even though *CAMs* originally were not FPGA-based, Margolus' later work on custom FPGA machines [35, 36] was based on these architectures, and hence we have included these architectures here. Through time, his ideas of data permutation and the effective use of fast memories as custom buffers proved to be useful not only for CA simulation, but also for DRAM-based systolic computation in general, and many subsequent works (including our own) employ similar structures for different aspects of FPGA-based custom machines.

2.2 CEPRA: Cellular Processing Architecture (1994–2000)

The *Cellular Processing Architecture* (*CEPRA*) was an FPGA-based architecture developed during the 1990s at the Technical University of Darmstadt. It was a streaming architecture with an internal dataflow similar to that of *CAM*. The key difference between the two systems was that *CEPRA* used pipelined arithmetic logic instead of LUTs for computing the CA's transition function. As a result, the advantage of *CEPRA* compared to *CAM* was that complex rules could be computed in one step, whereas *CAM* had to convey their computation through cascaded LUTs.

CEPRA-8L, the first member of the *CEPRA* family, was completed in 1994 [22]. It contained 8 FPGA-based CA processors which could access all their 3×3 neighborhood cells simultaneously thanks to a computation window buffer. *CEPRA-8L* could display 22 generations of 512×512 8-bit cells per second. *CEPRA-1X*, *CEPRA-8L*'s successor, was completed in 1997 [20]. It was an FPGA co-processor mounted on a PC expansion board and used the memory of the host computer to store the CA grid. *CEPRA-1X* supported 2D and 3D CA with neighborhoods of radius $r = 1$ and could display the evolution of 1024×1024 16-bit cells in real time.

Within the next 3 years the designers of *CEPRA-1X* also created a high-level *Cellular Description Language* (*CDL*) which translates complex CA rules into Verilog HDL [21]. *CDL* programming does not require any special knowledge of the system architecture and can be used as a general CA description language.

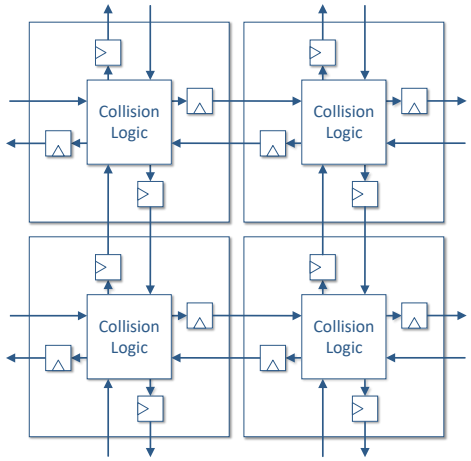

Fig. 4. Top level view of the lattice gas automaton as it was implemented in SPACE.

2.3 SPACE: Scalable Parallel Architecture for Concurrency Experiments (1996)

In 1996, Shaw, Cockshott and Barrie from the University of Strathclyde in the UK argued that, as far as lattice gas automata are concerned, parallel machines can outperform LUT-based computers such as *CAM* and yield more useful results [51]. They introduced their *Scalable Parallel Architecture for Concurrency Experiments* (*SPACE*) and proposed a different approach towards hardware for CA simulations. As shown in fig. 4, their FPGA-based architecture consisted of an array of interconnected processing elements (PEs), each one of which represented a cell of the HPP model, a fundamental lattice gas automaton. A *SPACE* board, which contained 16 FPGA chips, could simulate a 9×30 lattice gas automaton, achieving nearly a 10× speedup over 2 *CAM-8* modules. The architecture was scalable and larger lattice gas automata could be simulated by obtaining more *SPACE* boards.

As we will see later in this section, the size of a lattice gas automaton that can fit in only one of today's FPGA chips is an order of magnitude larger than that of two *SPACE* boards, and these quantitative differences greatly affect scalability issues of the architecture.

2.4 Kobori, Maruyama and Hoshino (2001)

In 2001, Kobori, Maruyama and Hoshino from the University of Tsukuba in Japan presented their own FPGA-based CA system at the 9th IEEE International Symposium on Field-Programmable Custom Computing Machines [28]. Their design combined the two approaches discussed in the previous subsections. Their streaming architecture consisted of an array of PEs sweeping across the CA grid (fig. 5). In this computation method, if the depth of the PE array is n, each cell of the grid is processed n consecutive times within the FPGA. As a result, if the input cells belong to generation g, the output cells will belong to generation $g + n$.

The problem arising from this method is that when the width of the CA grid is larger than that of the FPGA's I/O, as the computation moves on within the FPGA, the cells located at the edge of the PE array cannot access the new state of the cells located in their neighborhood, therefore, their new values are invalid. As a result, after each sweep of the grid, the cells that the system has read outnumber the ones that have been produced in its output. The issue is tackled by overlapping consecutive scans of the grid, as shown in fig. 6. This FPGA-based CA system consisted of an off-the-shelf PCI board with one FPGA and used the host computer to display the results. It could simulate

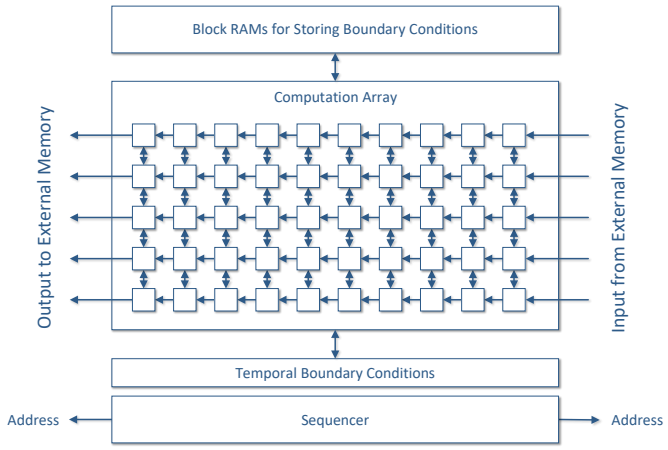

Fig. 5. Overview of Kobori, Maruyama and Hoshino's system architecture.

a 2048 × 1024 FHP lattice gas automaton and calculate 400 generations per second, achieving nearly a 155× speedup over a high-end CPU at the time. However, the CA's visualization was in pseudo-real time, as most calculated generations never reached the PE array's output. Their system was complemented by a custom high-level language which could be translated into Verilog HDL. By using that language, the user could specify the size of the PE array and the CA rule.

2.5 Other Significant Work

The above architectures comprise only a fraction of the landscape. During the last 3 decades, many other significant projects and developments have lead to advances in the field.

In 1991, Bouazza *et al.* used the *ArMen Machine* to implement CA in a way similar to that of *CEPRA-8L* [3]. The *ArMen Machine* consisted of interconnected FPGAs arranged in a ring and was controlled by a host computer interface board. While the system's performance results were comparable to those of *CAM*, *ArMen*'s routing resources were not sufficient for the simulation of CA rules with large neighborhoods.

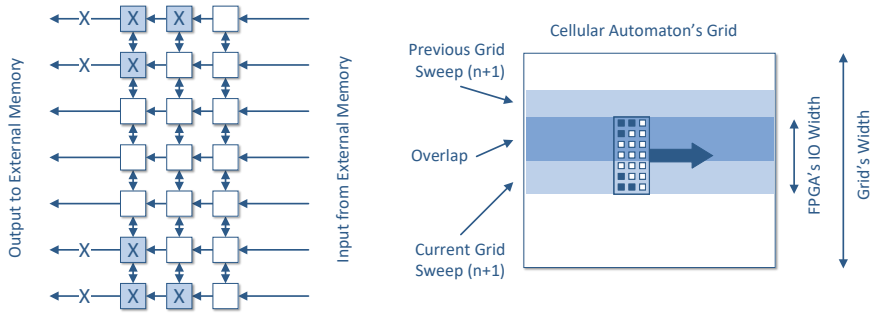

Fig. 6. When the width of the CA grid is larger than that of the FPGA's I/O, data integrity is preserved by overlapping consecutive scans of the grid.

Ten years later, Cappuccino and Cocorullo introduced *CAREM*, a configurable CA co-processor [6]. The processor's architecture consisted of a variable number of PEs which depended on each particular CA that the processor would be executing. For example, simulating a CA with only 2 states per cell (1-bit cell) resulted in generating 32 PEs within *CAREM*, since 32 is the maximum number of 1-bit cells that can fit in the system's 32-bit memory word. Although it might seem restricting, this method actually gave the designers plenty of freedom by keeping the memory management unit simple.

From 2007 to 2010, Murtaza, Hoekstra and Sloot from the University of Amsterdam performed a series of studies on the performance modeling of FPGA-based CA systems [39–42]. With architectures similar to those of Kobori, Maruyama and Hoshino, they experimented with different topologies, sizes and types of PEs, depending on whether a particular CA simulation is compute-bound or memory-bound. Their experiments concluded with the floating point execution of lattice Boltzmann fluids on FPGA clusters.

In 2013, Lima and Ferreira from the University of Porto presented their own reconfigurable CA architecture [33]. The approach followed was similar to that of *SPACE*. The processing unit consisted of a PE array which implemented the whole CA within the FPGA. The system could be configured with the use of a Graphical User Interface (GUI) running at the host computer and it could simulate any CA with small neighborhoods. The size of the automaton grid varied and could reach up to 72×72 cells depending on the rule's complexity.

Since then, FPGAs have been widely used to simulate CA, however, most implementations have been custom to a specific CA rule without the use of large neighborhoods [52].

2.6 Our Approach

As we have seen in this section, there are two prevailing approaches for the design of custom hardware accelerators to simulate CA. A commonplace approach is to exploit a CA's spatial parallelism by implementing it as an array of PEs. Each PE represents a CA cell and is interconnected to its adjacent PEs which are, in turn, the neighboring cells. This method results in outstanding performance when simple CA rules are concerned. However, when it comes to complex rules with many states per cell and large neighbordhood sizes, a PE's demand in logic and routing resources increases and performance drops. The other approach, which is the one employed in the current work, is to design a streaming architecture which processes the CA as a stream of cells. This approach is effectively more scalable for complex rules with large neighborhoods on large grids.

3 THEORETICAL BACKGROUND

This section provides the theoretical background necessary to understand the subject of this paper. The first part of this section gives an overview of the basic CA theory. The second part has a brief description of major versions of our work through time and the features of our latest design.

3.1 Cellular Automata

The execution of CA in computing systems is termed by the community "CA simulations" and we will keep this terminology, with the understanding that it refers to actual runs on actual hardware, and in the case of our work with results from downloaded designs and not simulations of the hardware, for 2D CA. A 2D CA consists of an infinite rectangular grid of homogeneous cells, each in one of a finite number of discrete states. For each cell, a set of cells called its neighborhood is defined relative to the specified cell. An initial state of the CA at time $t = 0$ is selected by assigning a state for each cell. A new generation is created according to some fixed mathematical rules described with a transition function which determines the new state of each cell in the next time interval in terms of the current state of the cell and the states of the cells in its neighborhood.

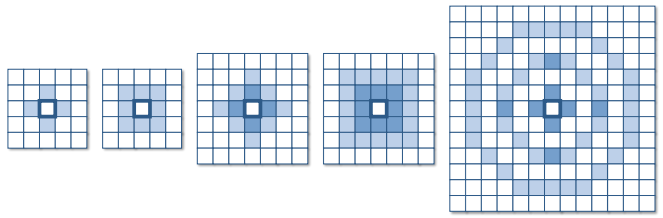

Fig. 7. Basic types of neighborhoods in 2D CA (left to right): a) von Neumann, b) Moore, c) Weighted, extended von Neumann, d) Weighted, extended Moore, e) Weighted, custom neighborhood.

There are three basic types of neighborhoods in 2D CA: von Neumann, Moore and custom. All three types can be used in an extended and weighted form (fig. 7). There are two ways to define a cell's neighborhood: either by its $n \times n$ dimensions in cells, or by its radius r measured as the distance from the central cell to the neighborhood edge. Note that extending the Moore neighborhood for radius $r \geq 1$ gives a set of cells situated at Chebyshev distance r from the central cell, while extending the von Neumann neighborhood for radius $r \geq 1$ gives a set of cells situated at Manhattan distance r from the central cell.

3.2 Totalistic Cellular Automata

Totalistic CA form a distinct, widely-used class. The state of each cell in a totalistic CA is represented by an integer value drawn from a finite set of possible states $S = \{s_0,\ s_1,\ ...,\ s_n\}$, and the next state of a cell depends only on the sum of the current values of the cells in its neighborhood [24]:

$$c'(i,j) = \sum_{x=i-r}^{i+r} \sum_{y=j-r}^{j+r} w(x-i, y-j) \cdot c_t(x,y)$$

$$c_{t+1}(i,j) = \begin{cases} s_0 & \text{if } c'(i,j) \leq a \\ s_1 & \text{if } a < c'(i,j) \leq b \\ \dots & \\ s_n & \text{otherwise} \end{cases}$$

where r is the neighborhood radius, $w(x,y)$ the neighborhood weights with $w(0,0) = 0$ and $a, b, ... \in \mathbb{Z}$. If the next state of a cell depends on both its own current state ($w(0,0) \neq 0$) and the total sum of its neighbors, then the CA belongs to the class of *outer totalistic CA* [24]. *Conway's Game of Life*, one of the best-known CA, is an example of an outer totalistic CA with 2 states per cell and a simple Moore neighborhood [14].

3.3 Totalistic Cellular Automata Broadened

Even broader versions of totalistic CA include transition rules with more steps involved, like the *Hodgepodge Machine* discussed later in this paper. Calculating CA with a small set of states per cell and small neighborhoods has been thoroughly explored and can be accomplished efficiently by general-purpose CPUs thanks to algorithms like the *Hashlife* algorithm [18], a memoized *(sic)* algorithm which accelerates computation by several orders of magnitude. As the number of states rises, the neighborhood size grows and weights are introduced, the complexity of simulating such CA rules rises dramatically and the use of efficient accelerators becomes crucial.

3.4 Design Timeline

The advanced modeling capabilities and capacity of large-neighborhood CA came to our attention in 2015 while developing the *Game of Complex Life*, an FPGA design which models basic urban grown processes with CA (fig. 8) [29]. CA have been successfully used to simulate urban growth both in basic and high-end models[1, 11, 48]. In the *Game of Complex Life* a small city located on the map, which is the CA grid, is developing into a large metropolis while taking into account the geographical morphology of the area. This means, for example, that the city will develop slower on hills than it does on plains and it won't cross a river unless a bridge is first built across it. The city's form is also based on a few basic socioeconomic rules, as there has to be an equilibrium between the number of apartment buildings, businesses and recreation facilities.

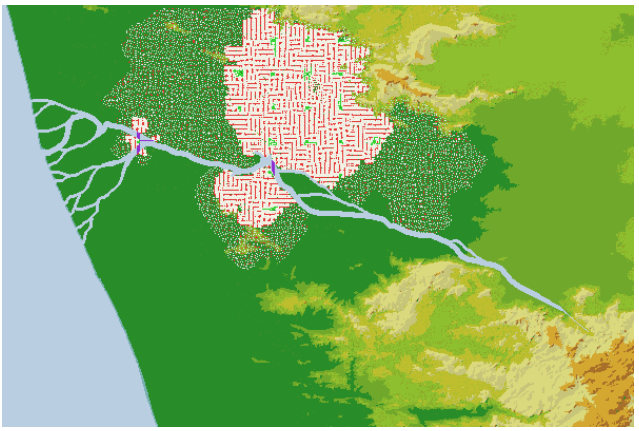

Fig. 8. Our 2015 project, *Game of Complex Life: Modeling of Urban Growth Processes with CA.*

The rule is on a 7×7 Moore neighborhood and it is context-sensitive: it changes depending on the type of area in which the currently-processed cell is located. Within the next three years we built on those ideas and created the first version of a scalable architecture to simulate CA with large neighborhood on reconfigurable logic [30, 32]. The user could use this architecture to simulate CA with neighborhood sizes up to 21×21 cells and 256 states per cell. Since then, as shown in table 1, the architecture has been transformed into a versatile general-purpose CA framework. The resulting designs have already produced original results [31], such as corner vortices which do not appear when neighborhoods are small. The framework also provides the user with the ability to simulate rules with even larger 29×29 neighborhood sizes and a large number of states per cell in different types of large grids. The present paper is the only complete publication of this project - the Xilinx design competition entries are only the corresponding videos in the competition home page (not papers), and the two 2019 conference publications include examples, performance and architectural highlights but not the complete architecture and framework.

3.5 Real-time Performance

In computer graphics, "real-time" means that the processing engine can produce new frames at the rate of the medium projecting the images, e.g. a computer monitor. We arbitrarily chose 60 frames per second (fps) to match the screen update rate at Full-HD definition. This is not a real constraint though; even with the modest FPGA that we use, we can go up to close to 100 fps as described in section 6.7. However, this would mean that there are many iterations between outputs, whereas at present a fast CPU can deliver for this type of neighborhood slightly more than 1 fps.

Year	Neighborhood Size	Number of Cell States	Comments
2015	7×7	8 (v1), 11 (v2)	Xilinx Open Hardware 2015 Contest Finalist [29]
2018	up to 21×21	up to 256	Xilinx Open Hardware 2018 Contest Finalist [30]
2019	up to 29×29	up to 256	Latest Work, partially published in [31, 32]

Table 1. The timeline of our hardware designs.

4 AN EFFICIENT FPGA-BASED LARGE NEIGHBORHOOD ARCHITECTURE

In this section, the architecture designed for this project is presented in full detail, with emphasis on how performance tradeoffs affected architectural decisions and choices.

4.1 Design Features: Number of States, Neighborhood Size and Types of Grid

A CA is ultimately described by the number of states per cell, the size and shape of the cell's neighborhood and the transition rule. The number of cell states and the cell's neighborhood are the two key parameters which determine the rule's capability to model various phenomena accurately. Some models require rules with only a few cell states (e.g., in the *Game of Life* a cell is in one of two states: "alive" or "dead"). Other models require a higher number of states in order to describe either the evolution of gradual phenomena, (e.g., a temperature gradient), or dynamic properties (e.g., spatial orientation or heading direction). The size of a cell's neighborhood defines the level of detail a model can reproduce both macroscopically and microscopically. Adjusting the neighborhood shape and weights can result in complex behavior, such as movement, interlacing and anisotropy. Our system supports either 4-bit or 8-bit cells (up to 16 or 256 states per cell, respectively) and neighborhood sizes of up to 29×29 cells. In section 5, we explain how these two parameters can be set by the user so that the resulting architecture will be generated by our framework.

In theory, CA evolve on an infinite grid. In practice, computers have finite memory and can only store and simulate CA on a finite grid, such as a Cartesian grid, which is commonly used. However, its use requires the need of explicitly defined boundary conditions, as the cells located at the edge of the grid have no access to a complete neighborhood. One solution is to create a zone of fixed-state cells around the rectangular grid (for example, zero padding) and special rules for those cells in order to enclose the CA's evolution within the said grid. This technique is useful when it comes to spatially bounded phenomena, such as a lattice gas automaton propagating within a container. However, it is quite restrictive as far as spatially large phenomena are concerned, as it would be better to completely avoid the need for "special cases". Another type of grid is a finite grid without edges, in the shape of a torus. The toroidal grid, which is often described as "periodically infinite", is formed by wrapping-around the edges of the rectangular grid as shown in fig. 9. As we saw earlier in section 2, all previous CA hardware simulators supported toroidal grids. The types of grids supported by our system are the rectangular grid, the toroidal grid and an intermediate type of grid which is the cylindrical grid (fig. 9). Large CA neighborhoods are of no use if the automaton is executed in small grids, as they produce large patterns which are visible only within a large grid. The grid size of our system is 1920×1080 cells, large enough for large patterns to form, propagate and interact with each other across the CA universe. The grid size chosen is also the resolution of modern Full-HD screens, with each pixel of the screen representing a CA cell. The grid size for the computation engine can be easily extended (to tens of thousands of cells in the X direction and as much as the DDR memory will allow in the Y direction), however, this would affect synchronization of the computation engine with the graphics engine, and thus the graphical output of the system.

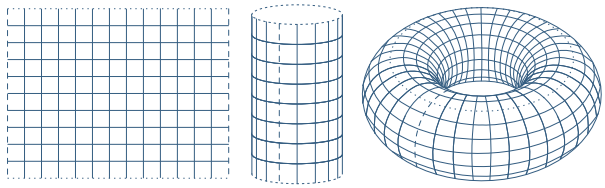

Fig. 9. Our design supports three different types of grids, Cartesian, intermediate and toroidal (left to right). The "periodically infinite" toroidal grid eliminates the need for special boundary conditions.

4.2 System Architecture and Implementation

A simplified schematic of the design is shown in fig. 10. As will be shown below, a number of parameters can be chosen by the user. These parameters lead into a compiled-in design, however, thanks to the framework, which will be presented in section 5, the user only needs to define the parameter values, and all connectivity, timing, synchronization, etc. are known to work for all cases (including the corner cases). The only actual code that the user needs to write has to do with the rule computations, which generally means that the user will modify one of the supplied templates in order to determine weights, neighborhoods, and state transitions. Our system was designed in VHDL and consists of four basic subsystems:

(1) *Memory Initialization* and *Frame Extraction* running at 100 MHz.
(2) *Memory Controller* generated by *Xilinx's Memory Interface Generator* running at 325 MHz, providing a user interface clock at 81.25 MHz (4:1).
(3) *CA Engine* datapath and buffers running at 200 MHz.
(4) Full-HD Graphics running at 148.5 MHz.

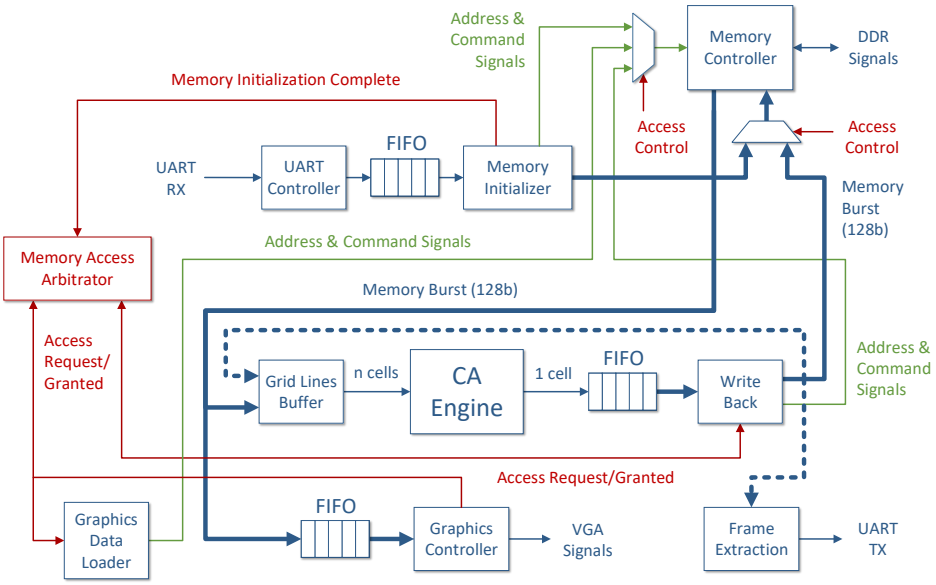

Fig. 10. A simplified schematic of the system architecture.

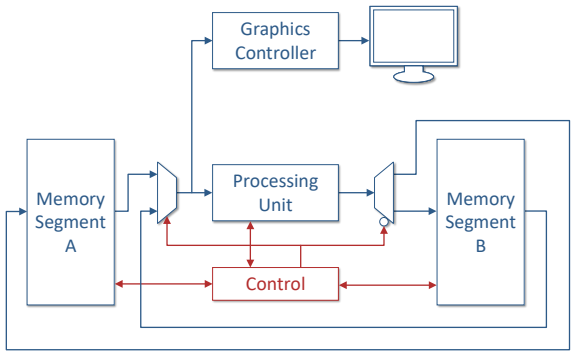

Fig. 11. Double buffering.

Initially, the system's external DDR memory needs to be loaded with the value for each cell of the grid at time $t = 0$. In standalone FPGA boards this can be done via UART from a computer. After memory initialization, the system starts displaying the stored CA grid on screen via VGA at 1080p. Every line that is loaded into the *Graphics Controller*'s buffer is also loaded into the *CA Engine*'s buffer. The *CA Engine*'s buffer holds all of the grid lines needed to provide the engine with the neighborhood of each cell of the line being processed. This results in a sliding window in the size of the cell's neighborhood moving across the grid, processing 1 cell per clock cycle.

This project constitutes a framework on which the user can build their own CA hardware simulations. The automatic generation process will be described in section 5; in this section we will present the key variables which affect the design's inner structure, dimensioning and resources without yet describing how the framework will implement the design. These variables are:

- n, $n \times n$ neighborhood size, $n \in [3..29]$. For example: $n = 29$ for a 29×29 neighborhood.
- c, cell size in bits, $c \in \{4, 8\}$. For example: $c = 4$ for a 4-bit cell (16 states).
- b, memory burst size in bits, $b \in \mathbb{N}$. For example: $b = 128$ for a 128-bit burst.
- x, y, the grid dimensions in cells, $x, y \in \mathbb{N}$. For example: $x = 1920$ and $y = 1080$ for a 1920×1080 grid.

A CA is a discrete world with discrete time, operating in distinct time steps. In each CA time step (iteration) the next state of all cells has to be completed. Like most previous architectures we use double buffering - one memory copy for each cell's current state, one copy for the next, with the memories for current and next state alternating after each complete iteration, as shown in fig. 11.

4.3 Memory Controller, Grid Memory Mapping and Graphics Synchronization

The two memory segments shown in fig. 11 are two distinct parts of the external DDR memory. Each segment has to hold a frame of $x \times y \times c$ bits each. Thus, for the same grid size, the segments may require a different amount of memory space depending on c, the cell size in bits. The external memory in our system was a 128 MB DDR2 memory; it receives and transmits data in the form of word bursts. Our system uses *Xilinx's Memory Interface Solution* as its *Memory Controller* which supports DDR, DDR2 and DDR3 memory interfaces. In our case, the controller runs at 325 MHz and provides a user interface clock at 81.25 MHz (4:1). It sends and receives 128-bit bursts ($b = 128$) which contain eight 16-bit words each. In our application, the memory word size is irrelevant - in order to dimension our system we need to define the number of CA cells per memory burst:

$$c_b, \text{ number of cells per memory burst, } c_b = \frac{b}{c},\ c_b \in \mathbb{N}.$$

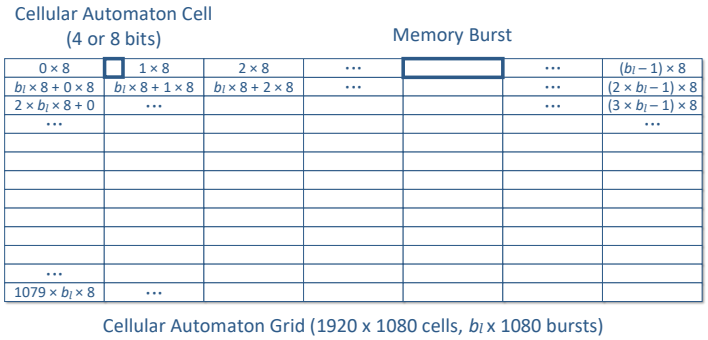

Fig. 12. Grid representation in memory and burst addressing.

The 1920×1080 grid is represented in memory as shown in fig. 12. Each line consists of 1920 cells parcelled up in bursts, with each burst containing c_b cells. The number of cells per burst is a user-defined variable, however, the number of memory bursts per grid line which is affected by the cell size, is automatically determined by the framework in section 5:

$$b_l, \text{ number of memory bursts per grid line, } b_l = \frac{x \times c}{b} = \frac{x}{c_b},\ b_l \in \mathbb{N}.$$

The memory holds two consecutive frames of the grid for the purpose of double buffering. Both memory segments can be accessed with the same addressing pattern as shown in fig. 12, and the only thing distinguishing them is the address's most significant bit. This addressing pattern is known as *simple horizontal scan*. However, it must be clarified that this method has nothing to do with the way the system sweeps the CA grid in order to process it. The horizontal scan concerns only the way the system reads data from the external memory and loads it into its buffers.

An initial configuration of the automaton grid, also commonly known as the CA's initial state, is set by assigning a state for each cell of the grid. The initialization of the grid's configuration can be performed by the host computer and downloaded to our system before it begins its operation. After initialization is complete, the system starts displaying the contents of the memory on screen, alternating between the two memory segments due to double buffering. Our design is deliberately slowed down so that it will display 60 CA generations per second for real-time operation.

The graphics subsystem has a VGA output, and comprises of three modules:

(1) The *Graphics Data Loader* which loads from memory the data to be displayed.
(2) The *Full-HD controller*, the graphics controller which generates the video synchronization pulses.
(3) The *Color Palette* which associates each pixel datum with a color.

The graphics controller is the one defining the system's timing and synchronization requirements as a whole, as we have to make sure that the processing engine will have generated a new frame before the graphics subsystem requests for it.

The graphics subsystem must have immediate access to the external memory whenever it needs to load part of the frame. Since memory access is shared between the graphics loading part of the frame and the processing engine storing part of the new frame, the memory access time had to be kept as short as possible for the graphics. For that purpose, the graphics subsystem loads one frame line at a time into its buffer in order to render it on the screen. *Horizontal scanning*, which we described earlier in this section, performs the load operation by matching closely the computation speed to the graphics requirements, and as a result we do not need a complete graphics frame memory or the associated memory transfers.

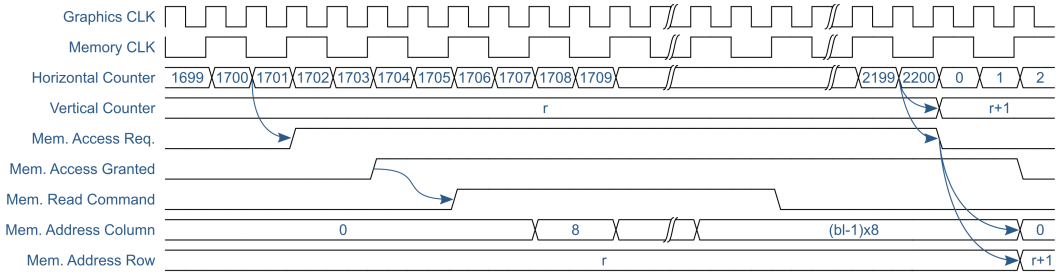

Fig. 13. The timing diagram of a frame line being loaded into the system's buffers. The system loads line r while the graphics subsystem is almost done rendering line $r-1$ on the screen.

Loading a new frame line into the graphics' buffer is carried out by the *Graphics Data Loader* once the graphics controller has completed drawing 75% of the line currently being displayed on screen (fig. 13). The loader is in direct communication with the graphics controller and acts as a mediator between the graphics controller and the memory controller. Every time the loader completes loading a frame, it alternates reading between the two memory segments of the double buffering. The *Graphics Data Loader* is also the entire system's data loader, as it is the only module requesting data to be loaded from memory. The *CA Engine* accesses the same data without requesting any additional loads from memory.

4.4 CA Engine

The *CA Engine* computes the new values of the CA grid cells. It operates at 200 MHz and it produces a new cell value per clock cycle once its pipeline is full. For each cell the *CA Engine* has to:

(1) Load the neighborhood cells.
(2) Multiply the cells with the neighborhood weights and then compute their sum (i.e. compute a dot product with"Multiply and Accumulate" operations).
(3) Calculate the cell's new state depending on the sum's value (Transition Function).

The *CA Engine* accepts a neighborhood column consisting of $n \times c$ bits as its input and shifts the entire window at every clock cycle. This results in a sliding window in the size of the cell's neighborhood moving across the grid, as shown in fig. 14. This way, the engine has the complete neighborhood of the cell being processed available at clock rate. All three steps, above, have to be optimized for the engine to be able to process 1 cell per clock cycle, and so our *CA Engine* uses a pipelined neighborhood window implemented as an array of shift registers, as shown in fig. 15.

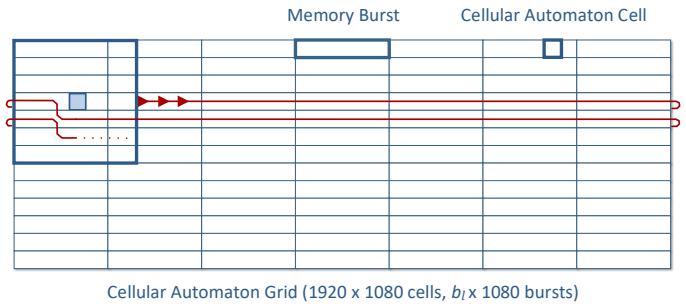

Fig. 14. The pipelined neighborhood sliding window sweeping across the grid.

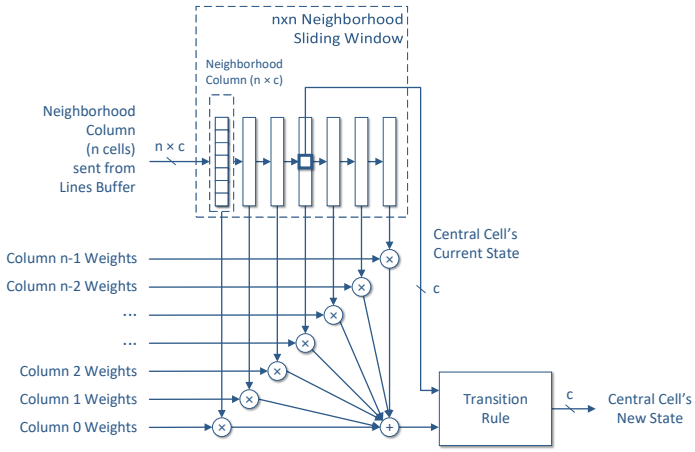

Fig. 15. The CA Engine.

After loading the neighborhood, all the cell values located in the cell's neighborhood have to be multiplied and accumulated (MAC). Each of the $n \times n$ cells is multiplied with its weight value, before entering an adder tree which calculates the sum of all the weighted cells. The adder tree performs the addition of $n \times n$ elements, each of which is $c + w$ bits wide, where w is the size of each neighborhood weight in bits. With the neighborhood reaching sizes up to 29×29 cells, the amount of the required arithmetic logic resources limits the width of the weights to 4 bits (with any 4-bit value), the limiting factor being the required multipliers (841 in the case of 29×29 neighborhood).

The arithmetic logic of the *CA Engine* is completely pipelined, providing the *Transition Function* with a new result at every clock cycle. The engine's *Transition Function* is a simple look-up table and does not require more than 1 clock cycle to produce the new value of the cell being processed. As a result, the *CA Engine* as a whole can produce a new cell per clock cycle.

Another important feature, which needs to be mentioned for future reference, is that there is no input signal enabling the *CA Engine*'s data input. The *CA Engine* reads a neighborhood column at clock rate, without taking into account what lies on the bus. Instead, it also accepts a 1-bit *Valid* signal which characterizes the central cell of the column currently being read. If the central cell of the column is valid, then the cell's new value will be written back to the external memory, otherwise it will be discarded by the system's *Write-Back* module.

The *CA Engine*'s high performance is apparent when the window slides horizontally across the grid. However, the window can not slide vertically. The system has plenty of time between frame lines to unload the last cells of the line currently being processed and load the neighborhood window with the first valid cells of the next line.

4.5 Grid Lines Buffer: Rectangular and Cylindrical Grid

The *CA Engine* produces a new cell per clock cycle, provided that it is supplied with a neighborhood column of cells at clock rate. This is possible due to the *Grid Lines Buffer*, which was designed based on the ideas of Margolus' *CAM 6* pipeline buffer [55]. Advanced variations of such dataflow buffers are used today in streaming architectures and stencil computing [8, 61]. We should note that stencil computations have similarities to our work, however, there is a major difference as well: as stencil computations usually apply to computationally more complex problems vs. CA (e.g. solving Navier-Stokes equations on a large grid for weather prediction) the computations of derivatives,

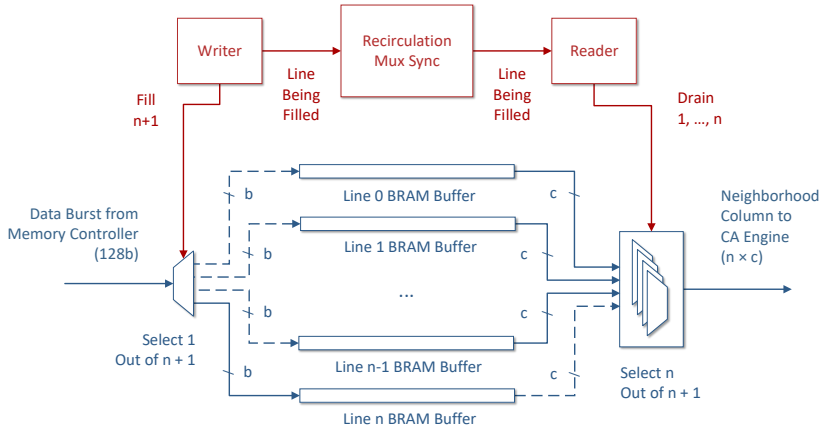

Fig. 16. The Grid Lines Buffer's internal structure.

etc. means that conventional CPUs and GPUs are better suited for the computation because the computation to memory access ratio is different, and the type of computation (typically, floating point) is better suited to CPUs and GPUs - parallelism is achieved by different means.

The *Grid Lines Buffer* (fig. 16) consists of n BRAM modules which store the grid lines needed to supply the *CA Engine* with the $n \times n$ neighborhood of the all the cells located in a grid line, plus one BRAM module used as a write buffer. As a result, the amount of BRAM resources required equals to $(n+1) \times x \times c$ bits $= (n+1) \times b_l \times b$ bits. In our case, the maximum values for our architecture (corner case) are $x = 1920, n = 29, c = 8$.

The *Grid Lines Buffer*'s control is implemented as two communicating FSMs; a reader and a writer. The writer operates at 81.25 MHz, which is the frequency at which the memory bursts arrive, and writes the buffer every time the system receives a grid line requested by the *Graphics Feeder*. In the meantime, the reader drains n BRAM modules at 200 MHz in order to feed the *CA Engine*. As soon as a complete grid line has been drained/filled, the buffer's control logic will "shift down" the buffer window. The line that was loaded last is ready to be drained along with the $n-1$ most recently loaded lines, and the earliest loaded line is ready to take upon the role of the write buffer. The reader and the writer communicate with each other via a recirculation multiplexer synchronizer, so that the system will not start filling a line that has not yet been drained.

In order to maintain correct system timing without user intervention in the *Grid Lines Buffer*, the number and size of the required BRAM modules and the depth of the pipeline are generated automatically based on n and c. Hence, the latency of the buffer is also variable and equal to n cycles. In effect, we make sure that the timing between the *CA Engine* and the *Graphics Engine* remains valid no matter what the user's neighborhood size and weight patterns, at the cost of the pipeline latency, which does not affect system performance because we have slowed down the *CA Engine* in order to match the speed of the *Graphics Engine*. In section 4.4 we have shown how the *CA Engine* always reads a neighborhood column at clock rate without having an input signal enabling its data input. This feature allows us to pre-load the neighborhood of the *CA Engine* before providing it with valid cells as shown in fig. 17. As a result, we can define the neighborhood of the cells located at the edge of the rectangular grid, which would normally have incomplete neighborhoods. By pre-loading the *CA Engine*'s pipelined neighborhood window with zero values, we create a zero-padded rectangular grid for our CA simulation.

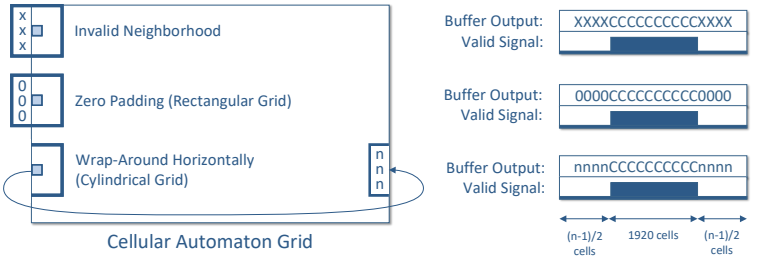

Fig. 17. Pre-loading the CA Engine's neighborhood window.

The cylindrical grid is implemented by pre-loading the *CA Engine*'s neighborhood window with the $(n-1)/2$ last cells of the buffer's lines before sending for processing the valid cells. For the rightmost cells of the grid, the buffer sends the first $(n-1)/2$ cells of the buffer's lines even after it has finished sending valid cells. The user choice of rectangular, cylidrical, or toroidal grid does not lead into any manual design. The toroidal grid's implementation is described in section 4.8.

4.6 Write-Back

Every time the *CA Engine* processes a grid line, it produces $x = 1920$ valid cells, 1 cell per clock cycle. The engine's output is connected to a FIFO buffer which parcels up the cells in bursts. The FIFO's data output is connected directly to the *Memory Controller*'s data bus.

Write-Back, which operates at the *Memory Controller*'s interface clock rate, acts as the mediator between the FIFO buffer and the *Memory Controller*, making sure that a burst has been successfully written to the memory before requesting for the next burst from the FIFO buffer. It handles the *Memory Controller*'s and the FIFO buffer's control signals, including the memory address bus.

The *Write-Back* module keeps track of how many bursts and lines it has written to memory. Once it has written $y \times b_l$ bursts, it alternates writing between the two memory segments as dictated by double buffering. As soon as a burst is available within the FIFO buffer, *Write-Back* attempts to write it in memory, provided it has access to do so, i.e., that the *Graphics Data Feeder* does not read anything from the memory at the time. The *Write-Back* module is fully pipelined and can write 1 datum of the burst per clock cycle if both the FIFO buffer and the *Memory Controller* are available.

4.7 Memory Access Arbitrator

The *Memory Access Arbitrator* is controlling which subsystem has access to the memory bus at any given clock cycle. The *Graphics Data Loader* has priority over *Write-Back* in order to not disrupt the graphics' 60 frames per second refresh rate. The loader occupies the memory bus only for 25% of the time needed to render a frame on the screen. The remaining 75% of the time is sufficient for the *CA Engine* and *Write-Back* to process and write an entire CA generation in memory.

4.8 Grid Lines Buffer: Toroidal Grid

In a toroidal grid, the neighborhood can wrap-around both horizontally and vertically. The wrap-around of the vertical edges of the grid in order to form a cylinder, known as a *toroidal wrap-around*, is implemented as in the cylindrical grid.

Folding the horizontal edges of the cylinder is called a *poloidal wrap-around*, and is implemented with extra BRAM line buffers and logic since the wrap-around involves grid lines that the system has accessed before and will not be loaded again by the *Graphics Data Loader*. For a cell located in

the first line of the grid, all of its above-located neighborhood cells are found in the last $(n-1)/2$ lines of the frame. Yet the last lines of the frame will not be loaded by the *Graphics Data Loader* until the rendering process reaches the end of the frame. These lines are kept in the *Grid Lines Buffer* when *Write-Back* transfers them to memory while processing the *previous* frame. This mechanism is implemented with "data forwarding" - a well-known method from computer design [43].

In a similar way, the neighborhood cells located under cells in the last line are found in the first $(n-1)/2$ lines of the frame, and they have been long loaded by the *Graphics Data Loader* at the start of the frame rendering process. In order to overcome this problem, the toroidal *Grid Lines Buffer* stores those lines in separate BRAM line buffers when the *Graphics Data Loader* loads them at the start of the frame rendering process and keeps them until needed at the end of the frame. The amount of BRAM resources required to implement the toroidal *Grid Lines Buffer* equals to $[(n+1)+(n-1)/2+(n-1)/2] \times x \times c$ bits $= 2 \times n \times x \times c$ bits $= 2 \times n \times b_l \times b$ bits. The extra BRAM line buffers needed for the toroidal buffers are called *Poloidal Line Buffers*, as they involve the poloidal movement of the neighborhood across the grid. Figure 18 shows the grid lines affected, the new structure of the *Grid Lines Buffer* and the new data flow. The rest of the system remains as is, except for *Write-Back* which now passes to the toroidal *Grid Lines Buffer* which line is written in the memory at any given time, in case the buffer needs it for its *Poloidal Line Buffers*. This approach leads to distributed control of this aspect of the architecture as well.

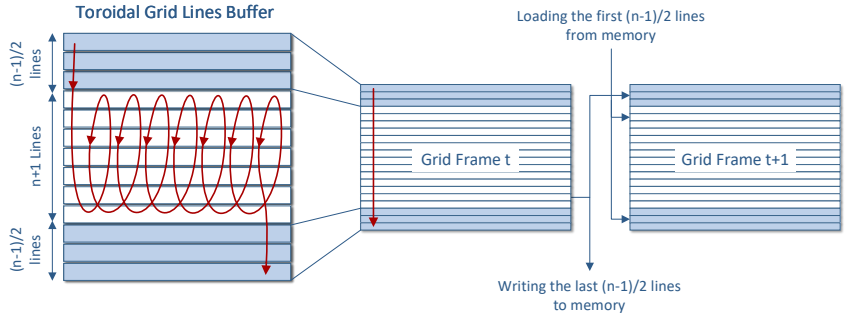

Fig. 18. Toroidal Grid Lines Buffer.

4.9 Putting It All Together: Real-Time Execution

The key ideas in this work are in the transformation of external, serial DDR memory accesses into parallel, internal FPGA resources: loading a cell's neighborhood data is reduced from $O(n^2)$ to $O(1)$ memory accesses, but with $O(n^2)$ internal-to-FPGA resources. Combined with the well-timed *Graphics Controller*, the *Memory Controller* maximizes aggregate memory bandwidth, and the system has no memory bottlenecks despite the built-in graphics controller. As shown in fig. 19, the system loads, processes and stores a grid line during the time it takes the *Graphics Controller* to render a line on the screen. The synchronization of subsystems is minimal, with distributed, independent control for each subsystem, based on b_l, the number of bursts per line and the number of lines that have been processed so far. This results in a generic system architecture which the user can customize in an automated fashion despite multiple clock domain crossings taking place.

4.10 Design Tradeoffs: Graphical Output vs. Off-line Results

One major tradeoff in the project has to do with graphics. Some, but not all CA accelerators have built-in graphics (e.g. Lima and Ferreira [33] have no graphics output), as graphics complicate and potentially slow down a design. The present architecture solves the built-in graphics architectural

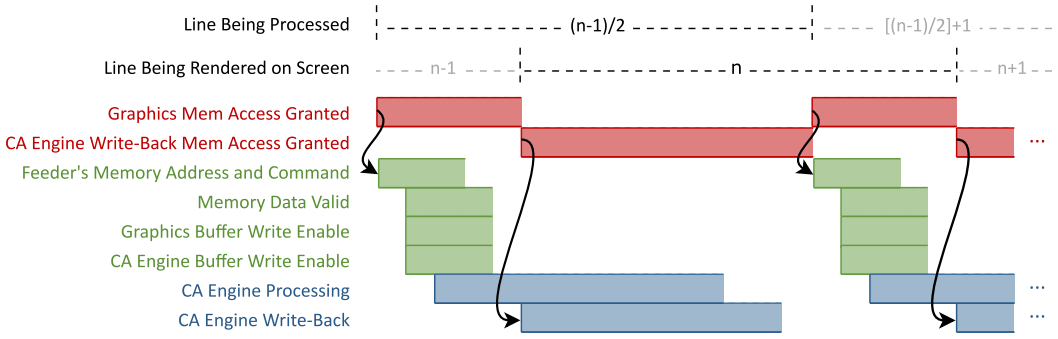

Fig. 19. The timing diagram of the system to load, process and write 1 line of the CA's grid.

problems, but can be easily adapted to a faster, no-graphics execution (at roughly 100 fps in our case). If we want to store all results on DDR memory (per iteration or after some iterations), regardless of graphical output, a mere replacement of the registers containing the DDR frame memory baseline addresses with counters would suffice. Successive frames would then be stored in different parts of the DDR memory. Lastly, the number of states could be easily increased, as such a change would only affect the initial pipeline latency and rules of any complexity can be implemented with BRAM (e.g. for 16-bit states, the associated "next state" would require 64K $\times$ 16 bit BRAM). Similarly, the grid size could be increased substantially: in the Y axis there is no limit as long as there exists DDR memory, whereas in the X axis the BRAM has to hold $n-1$ grid lines (plus one more for the output), which with our modest FPGA would result in many tens of thousands of cells (design complications which arise in toroidal grids do not change the main architecture). To conclude, in order to offer a versatile architecture, we solved the synchronization problems with a graphics controller, and provided the tools and the design files to re-purpose the architecture as needed.

5 A FRAMEWORK TO GENERATE CUSTOM CELLULAR AUTOMATA ARCHITECTURES FOR FPGAS

The architecture and system timing were presented in detail in the previous section not only for completeness, but also in order for the reader to gain insight on how the architecture is supported by a framework to automate the design process for CA with variable-sized neighborhoods up to 29 $\times$ 29 and display the results at 60 fps. This framework is presented below.

5.1 Automatic Code Adjustment

The variables which were used in previous sections to describe the system's inner structure dimensions and resources are not abstract theoretical entities but rather VHDL code in the form of *Generics*, used to dimension the design each time it is to be synthesized. Through simple changes in the design's top-level VHDL file *Generics*, the designer can parametrize the entity's structure and behavior:

```
Entity TOP_LEVEL is
    Generic(
        Grid_x : integer := 1920; -- Number of cells in a line
        Grid_y : integer := 1080; -- Number of lines
        Cell_size : integer := 8; -- Cell size in bits
        -- Cell_size = 4 => 2^4 = 16 states
        -- Cell_size = 8 => 2^8 = 256 states
```

```
        Neighborhood_size : integer := 29;
        -- Neighborhood size must be an odd number >= 3
        Grid_type : string := "TOROIDAL";
        -- Valid values: "RECTANGULAR", "CYLINDRICAL" and "TOROIDAL"
        Burst_size : integer := 128; -- Number of bits, depends on the memory controller
        Number_of_bursts_per_line : integer := Grid_x / (Burst_size / Cell_size);
        -- Cell_size = 4 => Number_of_bursts_per_line = Grid_x * Cell_size / Burst_size = 60
        -- Cell_size = 8 => Number_of_bursts_per_line = Grid_x * Cell_size / Burst_size = 120
        Palette : string := "WINDOWS"; -- Valid values: "WINDOWS" and "GRADIENT"
        Speed : integer := 60; -- Speed: every n frames => new generation
        -- For example, our graphics here run at 60 fps
        -- If Speed = 120 then we have a new frame @ 60/120 = 0.5 Hz
        Memory_addr_width : integer := 27
        -- Number of address bits, depends on the memory controller
    );
Port( ...
```

The user only needs to set the values of the *generics* found in the top-level file. The subsystems' VHDL files inherit their generic values from the top-level file without any intervention by the user and adjust their internal structure and behavior accordingly. One of the most useful features of this process is the dimensioning and allocation of the *Grid Lines Buffer*'s BRAM resources, which depend on n, the automaton's neighborhood size and c, the cell size in bits. This is code which the user does *not* need to worry about, as the framework generates it automatically, as shown below in a small example of the framework's output:

```
Generate_line_buffers: for i in 0 to Neighborhood_size Generate
-- 0 to Neighborhood_size - 1 lines + one for buffering
    Line_buffer_4: if (Cell_size = 4 ) Generate
        Line_buffer: Line_buffer_4b
        Port Map(
            clka => UI_CLK,
            wea => Write_enable(i),
            addra => Write_address(5 downto 0),
            dina => Data_in,
            clkb => CLK_200,
            addrb => Read_address,
            doutb => Line_data(0, i)
        );
    End Generate Line_buffer_4;

    Line_buffer_8: if (Cell_size = 8 ) Generate
        Line_buffer: Line_buffer_8b
        Port Map(
            clka => UI_CLK,
            wea => Write_enable(i),
            addra => Write_address,
            dina => Data_in,
            clkb => CLK_200,
            addrb => Read_address,
            doutb => Line_data(0, i)
        );
    End Generate Line_buffer_8;
End Generate Generate_line_buffers;
```

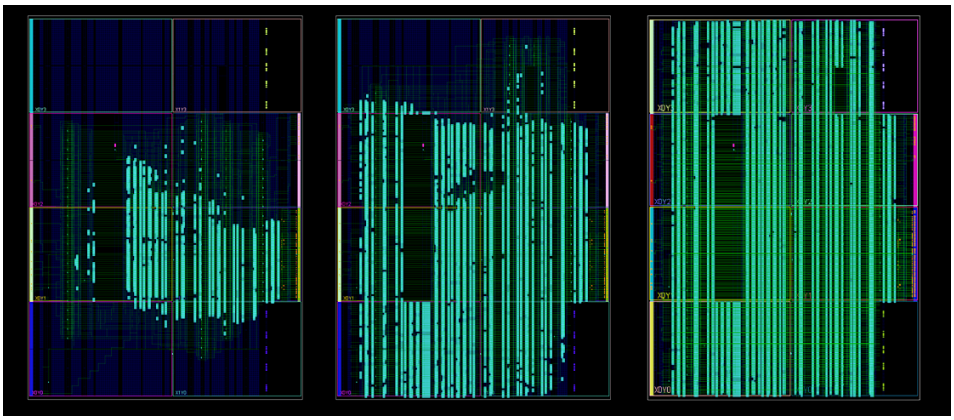

Fig. 20. Placed and routed design implementations of Artificial Physics ($n = 21$), the Hodgepodge Machine ($n = 29$), and the Hodgepodge Machine weighted ($n = 29$). All designs are fully functional.

Memory addressing is also affected by *Generics*. Both *Write-Back* and *Graphics Data Loader* set and reset their address counters based on the values of y, the number of grid lines and b_l, the number of memory bursts per grid line.

Automatic code adjustment is a fundamental feature which results from our framework. While in general our design is automatically generated based on the user's preferences, the *CA Engine*'s function depends specifically on the CA rule. For that reason, the engine must be redesigned every time according to the application, with the aforementioned general guidelines applying to the internal structure in all cases. Therefore, the engine's latency varies and depends directly on the neighborhood size n and the rule's complexity (without the user to have to worry about timing). In the future, we plan to design a user-friendly tool to generate the *CA Engine* based on a high-level CA rule description. Once the user has prepared their version of the *CA Engine*, our design generates a system which can execute any CA simulation in real time without any additional intervention by the user, provided that the automaton's characteristics lie within the system's specifications. In fig. 20 you can see implementation results of different CA rules with the use of our framework.

6 EXPERIMENTAL RESULTS AND PERFORMANCE EVALUATION

As Toffoli stated in 1984, applying CA modeling to real world problems is a challenging task, as one would have to experiment with various CA configurations, as well as develop a solid theoretical background [54]. Large-neighborhood CA have not been thoroughly explored, although, as we will describe below, their advanced modeling capabilities over automata with simple $r = 1$ neighborhoods have been proven in various fields like physics and chemistry [12, 13, 26, 46].

This section will demonstrate our design's capabilities and reusability by simulating four different large-neighborhood CA and their implementation results (the term "simulate" is used by the CA community - all examples are from actual runs on actual hardware). The FPGA platform is *Digilent's Nexys 4 DDR* board featuring *Xilinx's Artix 7* FPGA and a 128 MB DDR2 memory module. The *Artix 7* part mounted on board is the medium-sized *XC7A100T-1CSG324C* which contains 15850 logic slices (4 x 6-input LUTs and 8 flip flops each), 4860 Kbits of BRAM and 240 DSP slices. The VHDL design was implemented using *Xilinx's Vivado 2018.1 Design Suite* (currently migrated to *2019.2*). The FPGA board was connected via USB to a computer running *Microsoft Windows 10* and via a VGA cable to a Full-HD computer monitor.

6.1 Artificial Physics

The *Artificial Physics* rule is an outer totalistic CA with 2 cell states and a weighted, large neighborhood. The rule's name originates from its fascinating behavior (fig. 21). As the simulation moves on, "atoms" appear in the automaton's universe. These "atoms" attract each other and bind together to form "molecules". Each cell of the grid can be either "alive" or "dead". The CA's initial state of the grid must have 1/7 of its cells alive and randomly distributed across the grid in order for atoms to appear, otherwise the sea of alive cells disappears within a few generations. A bitmap image of the initial state with these properties is easy to create with the use of tools like *Matlab* by filling a 1920×1080 array randomly with ones and zeroes. We run it with a large 21×21 neighborhood with binary weights as shown in fig. 21. The cell's state transition function is defined as:

$$c'(i,j) = \sum_{x=i-r}^{i+r} \sum_{y=j-r}^{j+r} w(x-i, y-j) \cdot c_t(x,y)$$

$$c_{t+1}(i,j) = \begin{cases} 0 & \text{if } c'(i,j) \leq 19 \\ 1 & \text{if } 20 < c'(i,j) \leq 23 \\ 0 & \text{if } 24 < c'(i,j) \leq 58 \\ 1 & \text{if } 59 < c'(i,j) \leq 100 \\ 0 & \text{otherwise} \end{cases}$$

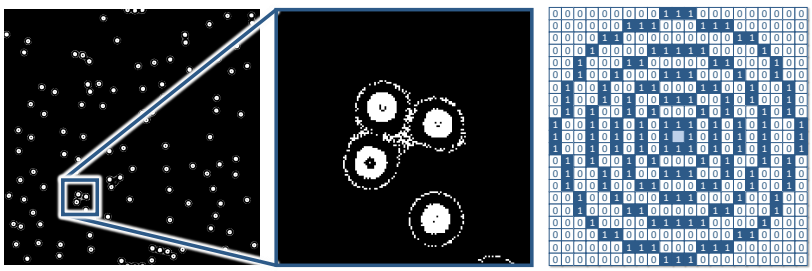

Fig. 21. Artificial Physics, an outer totalistic CA with a large neighborhood.

As we mentioned in section 5, the user chooses the neighborhood size and the automaton's cell size in bits, and the system is automatically generated based on these parameters. The only part of the design that the user needs to edit by hand is the *CA Engine* whose operation is determined by the rule's transition function. To help in this process, sample files are included in our open-source repository. In this simulation the system parameters were $n = 21$ and $c = 4$ bits, with toroidal grid.

6.2 The Greenberg-Hastings Model

The second CA simulation example is an excitable media model called the *Greenberg-Hastings* model. Excitable media are non-linear dynamical systems which are able to support the propagation of excitation waves, which usually appear periodically after a certain period of time called refractory time. Forest fires and chemical reaction-diffusion systems are two well- known examples of excitable media and have been succesfully implemented with the use of reconfigurable technology [25, 44]. Excitable media are often described by CA as a simpler replacement for complex differential equations [27, 60]. The *Greenberg-Hastings* model is one of the simplest CA to model excitable

media [19]. It originally had a 3×3 Moore neighborhood and 3 cell states: "quiescent", "excited" and "refractory", but it was later expanded to support more states and larger neighborhoods [12]. Excitable CA usually generate spiral or horns patterns and converge into an excitation equilibrium. As convergence is not guaranteed when starting from a randomly-filled grid, the initial grid condition is often set to patterns which are known to lead to an excitation equilibrium [12].

For this example we used a 29×29 Moore neighborhood and 16 cell states on a cylindrical grid, thus $n = 29$ and $c = 4$. A cell can be "quiescent" (state 0), "excited" (state 1) or in a sequence of "refraction" (states 2 to 15). The cell's state transition function is defined as:

$$c_{t+1}(i,j) = \begin{cases} 1 & \text{if } c_t(i,j) = 0 \text{ AND the number of excited neighbors} > t, \\ & \text{where } t \text{ is a threshold value} \\ (c_t(i,j) + 1) \ mod \ 16 & \text{if } c_t(i,j) > 0 \\ c_t(i,j) & \text{otherwise} \end{cases}$$

As shown in fig. 22 the *Greenberg-Hastings* CA has reached an excitation equilibrium with eight oscillating horns producing waves. The large neighborhood size plays an important role in the convergence of the automaton, as it has been proven that the width of the waves produced is proportional to the neighborhood's radius [13]. The ability to have circular radii is facilitated by the large neighborhood, which is a qualitative difference vs. previous results, clearly shown in fig. 22.

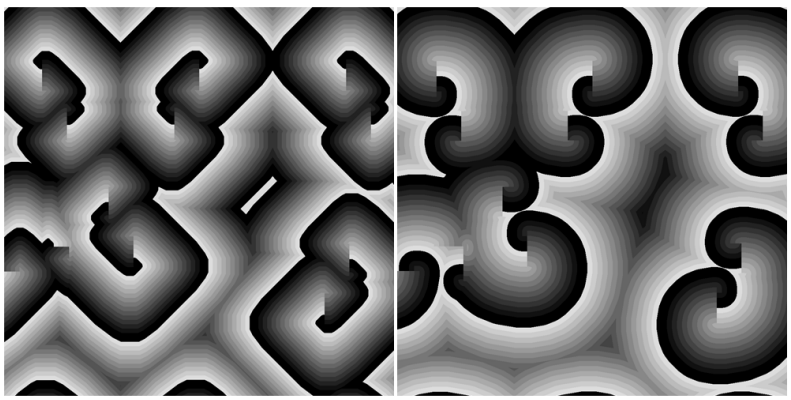

Fig. 22. The Greenberg-Hastings CA with a 29×29 von Neumann neighborhood (left) and a 29×29 circular neighborhood (right). Both images are at the same resolution.

6.3 The Hodgepodge Machine

The third example of our system's applications is the simulation of the *Hodgepodge Machine*. The *Hodgepodge Machine* models excitable media in a more complex fashion than that of the *Greenberg-Hastings* CA and is often used to model the Belousov-Zhabotinsky chemical reaction [10, 15]. Originally the *Hodgepodge Machine* had a 3×3 Moore neighborhood, but large-neighborhood versions of the rule have been explored during the last decade. The excitation equilibrium, also known in CA as auto-organization, is found empirically as in the case of the *Greenberg-Hastings* rule and depends on the rule's parameters as well as the initial state of the grid. For this example we used a 29×29 Moore neighborhood and 256 cell states, thus $n = 29$ and $c = 8$. A cell can be "healthy" (state 0), "ill" (state 255) or in a sequence of "infection" (states 1 to 254). The cell's state transition function is defined as:

$$c_{t+1}(i,j) = \begin{cases} \text{number of infected and ill cells } /k & \text{if } c_t(i,j) = 0 \\ 0 & \text{if } c_t(i,j) = 255 \\ (\text{sum of cells } / \text{ sum of infected cells}) + g & \text{otherwise} \end{cases}$$

All the cells mentioned in the transition function concern the cells located in $c_t(i,j)$'s neighborhood, and k and g are the two parameters of the rule that determine when and how fast the "infection" will spread.

As shown in fig. 23 our simulation of the *Hodgepodge Machine* has reached an excitation equilibrium with numerous horns and spirals producing waves. Note that the waves seem to consist of only four kinds of cells forming four wide cell bands. However, that is not the case. The noticeable gradient "glow" appearing between those four bands of cells is not a result of image processing, but is actually created by the CA itself and its many different cell states. The automaton's parameters used for this simulation were $k = 5$ and $g = 105$. The grid was toroidal and initially filled with random spots whose diameter was equal to n cells.

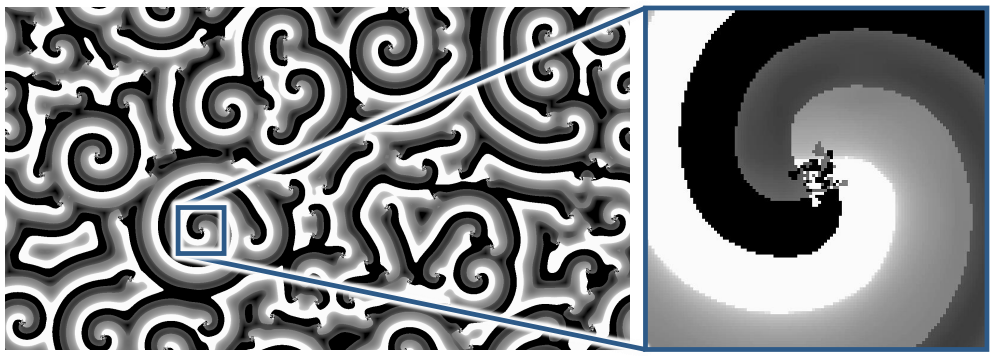

Fig. 23. The Hodgepodge Machine makes waves.

The neighborhood size affects directly the automaton's behavior and convergence. As we can see in fig. 24 the neighborhood size affects the width of the waves produced in a much similar way as in the Greenberg-Hastings model. The results produced by large neighborhoods have not only quantitative but also qualitative importance. As far as the *Hodgepodge Machine* is concerned, as the neighborhood radius grows we notice that the vortices produced are the result of small, stable, vortex-like patterns located in the center of the larger vortices (detail shown in magnification in fig. 23). This result is not observed with smaller neighborhood sizes and as best we know this work is the first one which produced it, however, if we notice closely, the vortices of the 19×19 neighborhood *Hodgepodge Machine* contain a "spike" in their center which could be a precursor of the stable vortex-like core patterns.

6.4 Anisotropic Rules

The fourth and final application example is an anisotropic CA with a large, 29×29 neighborhood. The anisotropy of CA lies either in the anisotropy of the grid, the anisotropy of the neighborhood or both. In this example, we experimented with the neighborhood's anisotropy: the neighborhood contains the largest weights at its far right (eastern) edge, with the weight value gradually being reduced to 1 towards the far left (western) edge of the neighborhood. The cell's state transition function is defined as:

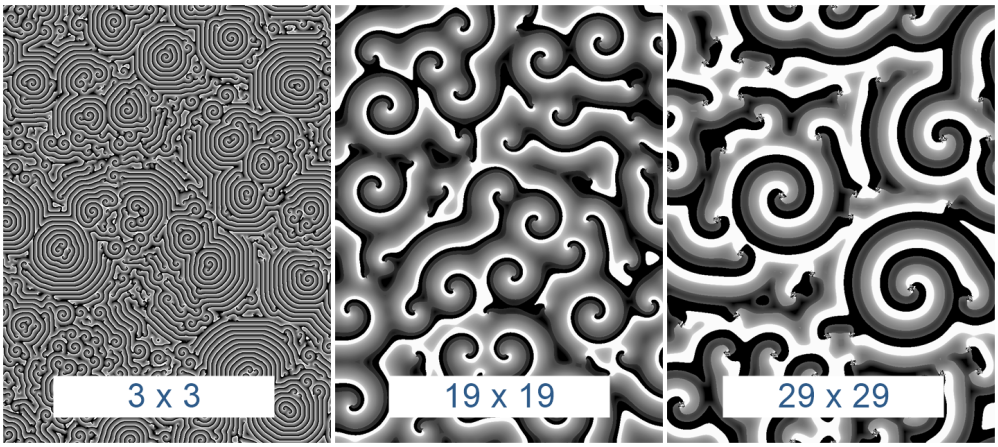

Fig. 24. The Hodgepodge Machine with three different neighborhood sizes. The images are of the same resolution, with each pixel representing an automaton cell.

$$\text{weighted sum} = \sum_{x=i-r}^{i+r} \sum_{y=j-r}^{j+r} w(x-i, y-j) \cdot c_t(x, y)$$

$$c_{t+1}(i,j) = \begin{cases} (c_t(i,j) - 1) \ mod \ 256 & \text{if weighted sum} > \text{threshold} \\ (c_t(i,j) + 1) \ mod \ 256 & \text{if weighted sum} < \text{threshold} \\ c_t(i,j) & \text{otherwise} \end{cases}$$

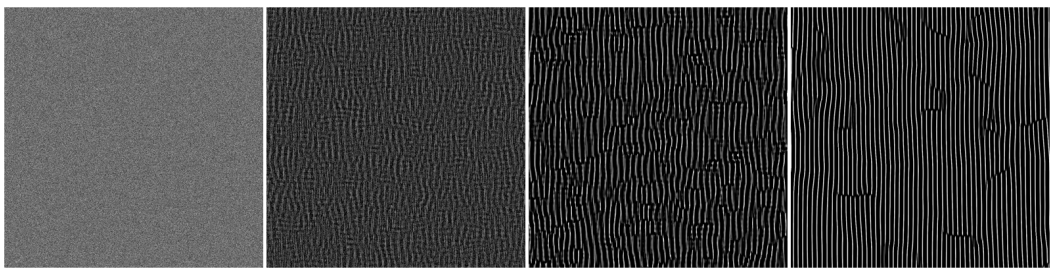

Fig. 25. The self-organization properties of a simple anisotropic CA with a large 29×29 Moore neighborhood after 1, 120, 500 and 10000 generations.

This quite simple anisotropic rule exhibits interesting self-organization properties as shown in fig. 25. Starting from a randomly filled grid, the automaton cells form long, thin stripes after a few thousand generations. In the course of time, ripples appear and tend to propagate to the left by virtue of the anisotropic neighborhood's weights. The long, rope-like structures are in the output and they are not an aliasing artifact of the image. They become more coherent once the rules allow for a ripple to disappear. As a result, the two edges of the rope merge into one, creating a ring around the toroidal grid. For this anisotropic CA example we used a 29×29 Moore neighborhood and 256 cell states, thus $n = 29$ and $c = 8$. The grid was toroidal and randomly filled using a uniform distribution $U\{0, 255\}$.

Cellular Automaton	i7-7700HQ, 1000 generations	Our Design, 1000 generations	Our Design's Speedup
Artificial Physics (n = 21)	538.77 sec	16.67 sec	32.32×
Greenberg-Hastings (n = 29)	469.58 sec	16.67 sec	28.16×
Hodgepodge Machine (n = 29)	851.29 sec	16.67 sec	51.06×

Table 2. Comparative results for CPUs: execution time and speedup.

6.5 Weighted Neighborhoods

The previous rules do have large neighborhoods, but they do not have any computationally demanding weights. Large-neighborhood CA often use binary weights that enable or disable a cell, as in *Artificial Physics*, or normalized weights that raise the significance of the cells located closer to the center of the neighborhood. In order to test and ensure that our design can support weighted neighborhoods, we applied random 4-bit weights to the *Hodgepodge Machine*'s 29×29 neighborhood, which translates into 841 multipliers operating in parallel. Our medium-sized FPGA could fit all of them with no problems in placing and routing the design, and it had no timing issues. If a larger FPGA is used or if numerous weighted cell values can be calculated only with the use of shift operations, one can use larger weights and multipliers.

6.6 Performance Results

In terms of results, our design calculates 60 CA generations per second regardless of the rule. The grid size is 1920×1080 cells which gives us $1920 \times 1080 \times 60 = 124,416,000$ cell updates per second, each cell requiring up to 841 multiplications, 840 additions, and a threshold comparison, i.e. roughly 195 GOPS (not counting the graphics) with a modest-size FPGA. As shown in Table 2 our system has a measured speedup of up to 51× against an *Intel Core i7-7700HQ* CPU (1 core) running highly optimized (-O3) software programmed in C. The CPU-based model was done both with `int` and `char` datatypes, both of which had the same performance. Although this is equivalent to 32 bits per datum, this cannot be circumvented in a conventional CPU because it stems from the ISA. We should note, however, that although 32 bits could lead to an incredibly large state space, in practice this is of no value to a conventional CPU because the rule would inevitably lead to an inordinately large decision tree, directly impacting the performance of the CPU. Regarding the cores, even with an 8-core machine we would get a 4-7× speedup (not addressing cost and energy requirements), even if we assume zero parallelization or memory contention overhead.

The speed offered by contemporary GPUs is comparable to that of our design for similar CA rules [5, 17, 37], to the extent of reported results, i.e. for neighborhoods up to 11×11, as shown in Table 3. Current GPU studies focus mainly on increasing the neighborhood radius of simple CA rules, which results in memory-bound algorithms. However, Gibson *et al.* demonstrate that by increasing the neighborhood radius there is a drop in the speedup yielded by GPUs [17], which means that with very large neighborhoods FPGA technology is not only more power-efficient by a factor of more than 10×, but also faster than GPUs. Power-wise, at the system level a contemporary reference GPU requires 170 W, whereas our *Nexys 4 DDR* board is powered by USB, i.e. maximum of 7.5 W. We should briefly discuss the types of applications on Table 3. In order to compare against the latest published data on GPUs, from the associated published optimized designs, we have a somewhat apples-to-oranges situation. However, by demonstrating that our architecture has roughly par performance vs. a GPU on a 11×11 neighborhood (FPGA: smaller grid but more complex rules), and by virtue of our architecture having exactly the same performance for the 29×29 grid which corresponds to some 7 times more calculations per cell, combined with the published results by

Neighborhood	Gibson et al., 2015, Workstation with Nvidia GTX 560 Ti	Millan et al., 2017, Nvidia TitanX GPU	Kyparissas and Dollas, 2019, Xilinx Artix 7
11×11	✓	✓	✓
19×19	–	–	✓
29×29	–	–	✓
Performance	≈ 65× over serial for Game of Life on a 2048×2048 grid	21.1× over serial for Game of Life on a 4096×4096 grid	51× over serial for the Hodgepodge Machine (29×29) on a 1920×1080 grid
Power Consumption	≈ 170 Watt at maximum load	≈ 250 Watt at maximum load	≈ 4.5 Watt running the Hodgepodge Machine

Table 3. Comparative results for GPUs for large-neighborhood CA.

Gibson *et al.* indicating that GPU performance drops with radius increase, we believe that the trend is very clear - FPGAs are better performance-wise in addition to being more energy efficient vs. GPUs when the neighborhood is large enough.

In order to offer a more complete perspective of system dimensioning and speed, table 4 has a comparison with major milestones in FPGA-based CA architectures. Expectedly, our system has larger neighborhoods and faster performance, however, the trend of a major architecture every decade or so is clear - technology and CAD tool evolution leads to new architectures.

Some indicative results of the resource utilization for the examples in the previous section can be seen in table 5. The amount of FPGA resources utilized for each rule differs and depends mainly on the CA's neighborhood and cell size. As the size of the neighborhood grows, the number and size of the BRAM modules grows as well. Another architectural aspect which also affects the required resources is the type of the grid. The toroidal grid requires more BRAM resources in

Architecture	Neighborhood size	Grid size	Number of states	Simulation speed
CAM-8, 1993	experimented with up to $r = 5$ (11×11)	arbitrarily large, typ. 512×512	arbitrarily large	real-time for 512×512 grids and $r = 1$, 10 fps for $r = 5$
SPACE, 1996	$r = 1$ (3×3)	9×30	16 (4-bit cells, HPP model)	10× speedup over two *CAM-8* modules
CEPRA-1X, 1997	$r = 1$ (3×3)	1024×1024	65536 (16-bit cells)	real-time
Kobori, Maruyama and Hoshino, 1997	$r = 1$ (3×3)	2048×1024	256 (8-bit cells, FHP model)	400 generations per second
Murtaza, Hoekstra and Sloot, 2007	$r = 1$ (3×3)	arbitrarily large, typ. 1024×1024	up to 32-bit cells	≈ 1000 generations per sec. (HPP model)
Kyparissas and Dollas, 2019	experimented with up to $r = 14$ (29×29)	arbitrarily large, typ. 1920×1080	16 or 256 (4 or 8-bit cells)	real-time regardless of the rule size

Table 4. Comparative results for FPGA architectures over time.

Cellular Automaton, Neighborhood Size, Type of Grid	LUT	LUT RAM	FF	BRAM	DSP	IO	BUFG	MMCM/PLL
Artificial Physics, n = 21, Rect.	11%	7%	6%	36%	1%	35%	22%	50%/17%
Artificial Physics, n = 21, Tor.	13%	7%	6%	66%	1%	35%	22%	50%/17%
Greenberg-Hastings, n = 29, Cyl.	16%	6%	9%	48%	1%	35%	22%	50%/17%
The Hodgepodge Machine, n = 29, Tor.	37%	9%	27%	90%	1%	35%	22%	50%/17%
The Hodgepodge Machine, n = 29 Weighted, Tor.	47%	9%	51%	90%	1%	35%	22%	50%/17%

Table 5. Resource utilization on a medium-sized FPGA for different CA implementations.

order to implement the poloidal wrap-around. A large part of the LUT and FF required resources is determined by the rule's complexity itself and has nothing to do with the framework's design, as we can see for example in the case of the *Hodgepodge Machine*. The two designs of the *Hodgepodge Machine* are identical, with the only difference being the addition of the neighborhood weight multipliers (table 5).

6.7 Performance Analysis: Omitting the Graphical Output Restrictions

With the *Graphics Controller* currently acting as the synchronizer of the system, it takes 2200×6.73 *ns* for the system to load, compute and write-back a frame line, and then wait for the completion of a frame line to be rendered on screen. As described in section 4.3, there are 2200 horizontal synchronization pulses for a screen line (including the 1920 visible pixels, the front-porch pixels, etc.) and the graphics clock period is 6.73 *ns*. In case the *Graphics Controller* is omitted, loading a frame line on the buffer can take place in parallel with the computation of a previous line (the *CA Engine* can keep writing in the write-back FIFO buffer in the meantime). The time it takes to process a line and a frame without the graphics restrictions can be calculated as follows:

$$t_{line} = (n + l + x - 1) \times t_{CA\_Engine\_CLK\_period} + t_{mem\_UI\_CLK\_period} \Rightarrow$$
$$t_{frame} = t_{line} \times y + [(n-1) \times b_l/2] \times t_{MEM\_UI\_CLK\_period}$$

where n in t_{line} represents the latency between the *Grid Lines Buffer* and the *CA Engine*, l the latency of the *CA Engine* which depends on the CA rule, and $t_{mem\_UI\_CLK\_period}$ the time for the last burst to be written in memory for every line. In t_{frame}, $[(n-1) \times b_l/2] \times t_{MEM\_UI\_CLK\_period}$ describes the time it takes for the buffer to load the necessary lines between frames. For instance, for the *Hodgepodge Machine* we have $c = 8$, $n = 29$, $l = 60$, $x = 1920$ and $y = 1080$:

$$t_{line} = (29 + 60 + 1919) \times 5ns + 12.31\ ns = 10052.31\ ns \Rightarrow$$
$$t_{frame} = 10052.31\ ns \times 1080 + 14 \times 120 \times 12.31\ ns = \ 0.010877\ sec \Rightarrow 1.088\ sec \text{ for 100 frames}$$

7 CONCLUSIONS AND FUTURE WORK

In this paper we presented in detail a highly parallelized and pipelined FPGA-based architecture, as well as a framework on reconfigurable logic to produce designs which efficiently run and display CA with up to 29 × 29-sized neigborhoods at 60 fps. In addition, the framework allows the user to automatically perform the process of resource dimensioning, allocation, interconnection and synchronization. The new design yields significant speedup vs. a high-end general-purpose CPU

and comparable speedup to contemporary GPUs up to the reported 11×11 neighborhoods but at a small fraction of the required energy, whereas for large neighborhoods it is expected to be faster than GPUs in addition to more power-efficient. Both the architecture and the framework are open-source in order to facilitate extensive use and further development of our system.

In the future, it would be useful to have a graphical or command-line interface for the user to enter all desired dimensions and rules, and a CAD tool to automatically generate the design. By CAD tool we do not mean the user interface but the ability of the tool to take input from user information, such as type of grid, number of bits per state, CA rules, etc. and produce the *Xilinx Vivado* files, run the tools to produce a design, and have the design at the user's disposal without the user having to know or understand any hardware-related aspects of the design. Such a tool could be Python based, and lead to execution on the *Amazon EC2 F1* cloud. The CA structure and function would be described with a high-level language and be translated into an HDL, as in [21].

In order to simulate high-order CA, in which a cell's state depends also on its state in previous generations [57], our system can be expanded by storing in every cell information from previous states. Thus, our existing *CA Engine* will process a timeline of previous generations without changing the architecture, but rather the rules. Other space tessellations are easy to simulate as well. The triangular grid and the hexagonal grid have many applications in CA. However, since the rectangular grid remains most efficient memory-wise, the best way to represent a triangular or a hexagonal grid would be to map it on a rectangular grid, a procedure that does not require any changes to the architecture. Furthermore, our design's current performance provides an indication for its potential performance while simulating 3D CA. Our system can simulate neighborhood sizes of up to 29×29 cells in real time, which means it can process 841 cells per clock cycle. It also loads, processes and writes back $1920 \times 1080 = 2,073,600$ cells per frame. The above values can also fit a 3D CA with a $9 \times 9 \times 9$ neighborhood evolving on a 3D grid $100 \times 100 \times 207$ cells large.

Last, since the maximum frequencies that an FPGA design can reach today are much higher than our implementation's 200 MHz, our framework's general outline can be extended to implement continuous CA [7, 45] in hardware using floating point arithmetic in large present-day FPGAs.

ACKNOWLEDGMENTS

The authors wish to acknowledge Xilinx Corp. for their generous support of TUC ECE educational programs as well as multiple project-specific donations which made this work possible.

REFERENCES

[1] José I. Barredo, Marjo Kasanko, Niall McCormick, and Carlo Lavalle. 2003. Modelling Dynamic Spatial Processes: Simulation of Urban Future Scenarios through Cellular Automata. *Landscape and Urban Planning* 64, 3 (2003), 145 – 160.

[2] Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. 2001. *Winning Ways for Your Mathematical Plays* (2 ed.). A K Peters Ltd.

[3] K. Bouazza, Joël Champeau, Pius Ng, Bernard Pottier, and Stéphane Rubini. 1992. Implementing Cellular Automata on the ArMen Machine. In *2nd International Workshop on Algorithms and Parallel VLSI Architectures*. Gers, France, 317–322.

[4] Arthur W. Burks. 1971. *Essays on Cellular Automata*. University of Illinois Press.

[5] Daniel Cagigas-Muniz, Fernando Diaz del Rio, Manuel Ramon Lopez-Torres, Francisco Jimenez-Morales, and Jose Luis Guisado. 2020. Developing Efficient Discrete Simulations on Multicore and GPU Architectures. *Electronics* 9, 189 (2020).

[6] Gregorio Cappuccino and Giuseppe Cocorullo. 2001. Custom Reconfigurable Computing Machine for High Performance Cellular Automata Processing. *Electronic Engineering Times (www.eetimes.com, TechOnLine Publication)* (2001).

[7] Bert W. C. Chan. 2019. Lenia: Biology of Artificial Life. *Complex Systems* 28, 3 (2019), 251–286.

[8] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. 2018. SODA: Stencil with Optimized Dataflow Architecture. In *2018 IEEE/ACM International Conference on Computer-Aided Design*. San Diego, CA, USA, 1–8.

[9] Edgar F. Codd. 1968. *Cellular Automata*. Academic Press, NYC.

[10] Alexander K. Dewdney. 1988. Computer Recreations: The Hodgepodge Machine Makes Waves. *Scientific American* 259, 2 (1988), 104–107.
[11] Iro Dimitriou. 2013. *Cellular Automata in Design: An Approach in Recurring Design Processes.* School of Architecture, Technical University of Crete. Dipl.Eng. thesis.
[12] Richard Durrett and David Griffeath. 1993. Asymptotic Behavior of Excitable Cellular Automata. *Experimental Mathematics* 2, 3 (1993), 183–208.
[13] Robert Fisch, Janko Gravner, and David Griffeath. 1991. Threshold-Range Scaling of Excitable Cellular Automata. *Statistics and Computing* 1, 1 (1991), 23–39.
[14] Martin Gardner. 1970. Mathematical Games - The Fantastic Combinations of John Conway's New Solitaire Game "Life". *Scientific American* 223, 4 (1970), 120–123.
[15] Martin Gerhardt and Heike Schuster. 1989. A Cellular Automaton Describing the Formation of Spatially Ordered Structures in Chemical Systems. *Physica D: Nonlinear Phenomena* 36, 3 (1989), 209–221.
[16] Felix A. Gers, Hugo de Garis, and Michael Korkin. 1998. CoDi-1Bit : A Simplified Cellular Automata Based Neuron Model. *Lecture Notes in Computer Science* 1363 (1998), 315–333.
[17] Michael J. Gibson, Edward C. Keedwell, and Dragan A. Savić. 2015. An Investigation of the Efficient Implementation of Cellular Automata on Multi-Core CPU and GPU Hardware. *J. Parallel and Distrib. Comput.* 77 (2015), 11–25.
[18] William R. Gosper. 1984. Exploiting Regularities in Large Cellular Spaces. *Physica D: Nonlinear Phenomena* 10, 1-2 (1984), 75–80.
[19] James M. Greenberg and Stuart P. Hastings. 1978. Spatial Patterns for Discrete Models of Diffusion in Excitable Media. *SIAM J. Appl. Math.* 54 (1978), 515–523.
[20] Christian Hochberger, Rolf Hoffmann, Klaus-Peter Völkmann, and Jens Steuerwald. 1997. The CEPRA-1X Cellular Processor. *Reconfigurable Architectures: High Performance by Configware, IT Press, Bruchsal* (1997).
[21] Christian Hochberger, Rolf Hoffmann, Klaus-Peter Völkmann, and Stefan Waldschmidt. 2000. The Cellular Processor Architecture CEPRA-1X and its Configuration by CDL. In *IPDPS 2000. Lecture Notes in Computer Science, vol 1800.* 898–905.
[22] Rolf Hoffmann, Klaus-Peter Völkmann, and Mark Sobolewski. 1994. The Cellular Processing Machine CEPRA-8L. *Mathematical Research* 81 (1994), 179–188.
[23] Paulien Hogeweg. 1988. Cellular Automata as a Paradigm for Ecological Modeling. *Appl. Math. Comput.* 27, 1 (1988), 81–100.
[24] Andrew Ilachinski. 2001. *Cellular Automata: a Discrete Universe.* World Scientific.
[25] Kazuyoshi Ishimura, Katsuro Komuro, Alexandre Schmid, Tetsuya Asai, and Masato Motomura. 2015. FPGA Implementation of Hardware-Oriented Reaction-Diffusion Cellular Automata Models. *Nonlinear Theory and Its Applications, IEICE* 6, 2 (2015), 252–262.
[26] Katrin Jahns, Kamil Balinski, Martin Landwehr, Jürgen Wübbelmann, and Ulrich Krupp. 2017. Prediction of High Temperature Corrosion Phenomena by the Cellular Automata Approach. *Materials and Corrosion* 68, 2 (2017), 125–132.
[27] Anastasiya Kireeva, Karl K. Sabelfeld, and Sergey Kireev. 2019. Synchronous Multi-particle Cellular Automaton Model of Diffusion with Self-annihilation. In *Parallel Computing Technologies*, Victor Malyshkin (Ed.). Springer International Publishing, 345–359.
[28] Tomoyoshi Kobori, Tsutomu Maruyama, and Tsutomu Hoshino. 2001. A Cellular Automata System with FPGA. In *9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines.* Rohnert Park, CA, USA, 120–129.
[29] Nikolaos Kyparissas and Apostolos Dollas. 2015. Game of Complex Life - Modeling of Urban Growth Processes with Cellular Automata. In *Xilinx Open Hardware European Design Contest.* http://www.openhw.eu/2015-finalists.html.
[30] Nikolaos Kyparissas and Apostolos Dollas. 2018. A Parallel Framework for Simulating Cellular Automata on FPGA Logic. In *Xilinx Open Hardware European Design Contest.* http://www.openhw.eu/2018-finalists.html.
[31] Nikolaos Kyparissas and Apostolos Dollas. 2019. Field Programmable Gate Array Technology as an Enabling Tool Towards Large-Neighborhood Cellular Automata on Cells with Many States. In *2019 International Conference on High Performance Computing and Simulation.* Dublin, Ireland, 940–947.
[32] Nikolaos Kyparissas and Apostolos Dollas. 2019. An FPGA-based Architecture to Simulate Cellular Automata with Large Neighborhoods in Real Time. In *29th International Conference on Field Programmable Logic and Applications.* Barcelona, Spain, 95–99.
[33] André C. Lima and João Canas Ferreira. 2013. Automatic Generation of Cellular Automata on FPGA. In *9th Portuguese Meeting on Reconfigurable Systems.* Coimbra, Portugal, 51–58.
[34] Norman H. Margolus. 1996. CAM-8: A Computer Architecture Based on Cellular Automata. In *Pattern Formation and Lattice-Gas Automata*, Anna T. Lawniczak and Raymond Kapral (Eds.). AMS, Providence, RI, 167–187.
[35] Norman H. Margolus. 1997. An FPGA Architecture for DRAM-Based Systolic Computations. In *5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines.* Napa Valley, CA, USA, 2–11.

[36] Norman H. Margolus. 2000. An Embedded DRAM Architecture for Large-Scale Spatial-Lattice Computations. In *27th Annual International Symposium on Computer Architecture*. Vancouver, Canada, 149–160.

[37] Emmanuel N. Millán, Nicolás Wolovick, Maria Fabiana Piccoli, Carlos Garcia Garino, and Eduardo M. Bringa. 2017. Performance Analysis and Comparison of Cellular Automata GPU Implementations. *Cluster Computing* 20, 3 (2017).

[38] Melanie Mitchell. 1998. Computation in Cellular Automata: a Selected Review. *Non-Standard Computation, Wiley-VCH Verlag in Weinheim* (1998), 95–140.

[39] Shakeeb Murtaza, Alfons G. Hoekstra, and Peter M. A. Sloot. 2007. Performance Modeling of 2D Cellular Automata on FPGA. In *2007 International Conference on Field Programmable Logic and Applications*. Amsterdam, The Netherlands, 74–78.

[40] Shakeeb Murtaza, Alfons G. Hoekstra, and Peter M. A. Sloot. 2008. Floating Point Based Cellular Automata Simulations Using a Dual FPGA-Enabled System. *2nd International Workshop on High-Performance Reconfigurable Computing Technology and Applications* (2008), 1–8.

[41] Shakeeb Murtaza, Alfons G. Hoekstra, and Peter M. A. Sloot. 2009. Compute Bound and I/O Bound Cellular Automata Simulations on FPGA Logic. *ACM Transactions on Reconfigurable Technology and Systems* 1, 4 (2009), 23.

[42] Shakeeb Murtaza, Alfons G. Hoekstra, and Peter M. A. Sloot. 2010. Cellular Automata Simulations on a FPGA Cluster. *International Journal of High Performance Computing Applications, SAGE* 25, 2 (2010), 193–204.

[43] David A. Patterson and John L. Hennessy. 2012. *Computer Organization and Design - The Hardware / Software Interface (Revised 4th Edition)*. Academic Press.

[44] Pavlos Progias and Georgios Ch. Sirakoulis. 2013. An FPGA Processor for Modelling Wildfire Spreading. *Mathematical and Computer Modelling* 57, 5-6 (2013), 1436–1452.

[45] Stephan Rafler. 2011. *Generalization of Conway's Game of Life to a Continuous Domain - SmoothLife*. arXiv:1111.1567.

[46] Ehsan Rahimi and Shahram Mohammad Nejad. 2013. Radius of Effect in Molecular Quantum-dot Cellular Automata. *Molecular Physics* 111, 5 (2013), 697–705.

[47] Paul W. Rendell. 2011. A Universal Turing Machine in Conway's Game of Life. In *2011 International Conference on High Performance Computing and Simulation*. Istanbul, Turkey, 764–772.

[48] Andreas Rienow. 2018. The Future of Central European Cities – Optimization of a Cellular Automaton for the Spatially Explicit Prediction of Urban Sprawl. In *Cellular Automata: A Volume in the Encyclopedia of Complexity and Systems Science, Second Edition*, Andrew Adamatzky (Ed.). Springer US, New York, NY, USA.

[49] Daniel H. Rothman and Stephane Zaleski. 2004. *Lattice-Gas Cellular Automata: Simple Models of Complex Hydrodynamics*. Cambridge University Press.

[50] James B. Salem and Stephen Wolfram. 1986. Thermodynamics and Hydrodynamics with Cellular Automata. *Theory and Applications of Cellular Automata, World Scientific* (1986), 5.

[51] Paul Shaw, Paul Cockshott, and Peter Barrie. 1996. Implementation of Lattice Gases Using FPGAs. *Physica D: Nonlinear Phenomena* 12, 1 (1996), 51–66.

[52] Georgios Ch. Sirakoulis. 2018. Cellular Automata Hardware Implementation. In *Cellular Automata: A Volume in the Encyclopedia of Complexity and Systems Science, Second Edition*, Andrew Adamatzky (Ed.). Springer US, New York, NY, USA.

[53] Tommaso Toffoli. 1977. Computation and Construction Universality of Reversible Cellular Automata. *J. Comput. System Sci.* 15, 2 (1977), 213–231.

[54] Tommaso Toffoli. 1984. CAM: A High-Performance Cellular-Automaton Machine. *Physica D: Nonlinear Phenomena* 10, 1-2 (1984), 195–204.

[55] Tommaso Toffoli and Norman H. Margolus. 1987. *Cellular Automata Machines - A New Environment for Modeling*. MIT Press.

[56] Stanislaw Ulam. 1950. Random Processes and Transformations. In *International Congress of Mathematicians*. Cambridge, 264–275.

[57] Gérard Y. Vichniac. 1984. Simulating Physics with Cellular Automata. *Physica D: Nonlinear Phenomena* 10, 1-2 (1984), 96–116.

[58] John von Neumann. 1951. The General and Logical Theory of Automata. *Cerebral Mechanisms in Behavior: The Hixon Symposium, John Wiley & Sons* (1951), 1–41.

[59] John von Neumann and Arthur W. Burks. 1966. *Theory of Self-Reproducing Automata*. University of Illinois Press.

[60] Yifan Zhao, Stephen A. Billings, and Alexander F. Routh. 2007. Identification of Excitable Media Using Cellular Automata Models. *International Journal of Bifurcation and Chaos* 17, 01 (2007), 153–168.

[61] Hamid Reza Zohouri, Artur Podobas, and Satoshi Matsuoka. 2018. Combined Spatial and Temporal Blocking for High-Performance Stencil Computation on FPGAs Using OpenCL. In *2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Association for Computing Machinery, New York, NY, USA, 153–162.

[62] Konrad Zuse. 1970. *Calculating Space*. MIT Technical Translation AZT-70-164-GEMIT, Massachusetts Institute of Technology (Project MAC).