Field Programmable Gate Array Technology as an Enabling Tool Towards Large-Neighborhood Cellular Automata on Cells with Many States

Nikolaos Kyparissas, Apostolos Dollas
School of ECE, Technical University of Crete

Conference Paper | Accepted Manuscript
Published in: 2019 International Conference on High Performance Computing & Simulation (HPCS)

DOI: 10.1109/HPCS48598.2019.9188084

© 2019 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Field Programmable Gate Array Technology as an Enabling Tool Towards Large-Neighborhood Cellular Automata on Cells with Many States

Nikolaos Kyparissas, Apostolos Dollas*
School of Electrical and Computer Engineering
Technical University of Crete
Chania, Greece
nkyparissas@isc.tuc.gr, dollas@ece.tuc.gr

***Abstract*—Cellular Automata (CA) have been used for many decades to simulate physical processes. From the 3×3 and 5×5 neighborhoods of the 1950's, and typically on binary images, as recently as the mid-2010's the neighborhoods went up to 15×15 on images with a few states. Field Programmable Gate Array (FPGA) technology, already applicable to CA simulation since the early 1990's, has reached such maturity levels that a small device can simulate large-neighborhood CA. In this work we present an architecture which we have fully implemented, that can simulate CA with up to 29×29 neighborhoods on 256-state cells for Full High Definition (FHD) image input/output with real-time 60 frames-per-second capability. Emphasis of the present work is on the game-changing opportunities that FPGA technology creates to the CA community. We present results from the Greenberg-Hastings and Hodgepodge models, as well as a large-neighborhood anisotropic model. Large neighborhoods either yield qualitatively different results vs. smaller neighborhoods, or lead to results which are merely impossible to produce with small neighborhoods. A comparison of FPGA technology for CA shows advantages vs. conventional Central Processing Units (CPUs) or Graphics Processor Units (GPUs).**

***Keywords*—cellular automata, large neighborhood, FPGA, 29×29, real time**

I. Introduction

Cellular automata (CA) were proposed by John von Neumann and Stanislaw Ulam during the 1950s [1]–[4] and constitute an abstract, massively parallel discrete model of computation. Through time, they have been thoroughly used not only as models of complexity but also as models of non-linear dynamic systems.

CA are one of the first-ever applications of FPGAs as custom hardware computation accelerators. We built on four decades of custom hardware CA machines and used today's technology in order to create a scalable architecture which leads to a powerful environment for the exploration of large-neighborhood 2D CA with the use of FPGAs.

In this paper we present the basic key points of our architecture and its operation, before presenting and discussing some interesting, unprecedented results that were produced from experimentation with large-neighborhoods on some well-known excitable CA. These promising results demonstrate that our generic FPGA-based architecture is a powerful tool for the exploration of large-neighborhood 2D CA and their modeling capabilities.

This paper consists of five sections. Following the present, introductory section, Section II describes the related work and discusses the intrinsic characteristics of FPGA technology vs. CPUs and GPUs; Section III briefly presents the basic elements of the design; Section IV has results from actual runs of three different models, two of which are standard and a third one which shows how large neighborhoods open up interesting possibilities in anisotropic CA simulation. Lastly, Section V has a discussion on the results and conclusions.

II. Related Work and FPGAs vs. Competitive Technology

This section presents an overview of related works and approaches that have been proposed to accelerate cellular automata simulations with the use of FPGAs. In addition, it provides some basic background on FPGA technology and compares it to CPUs and GPUs.

A. FPGA Technology and Related Work

The FPGA technology, originally proposed in the 1980's and steadily evolving ever since, is a technology in which algorithms are mapped directly to the hardware resources of the integrated circuit. Loosely described, small memories implement basic logic functions (logic gates, 1-bit adders, comparators, etc.) in the form of Look-Up Tables (LUT); programmable interconnects allow for the connection of these units (Figure 1). In addition, basic memory cells such as Flip-Flops hold (as needed) results in registers, and larger units such as on-chip memory (called with various names, such as Block Random Access Memory - BRAM), Digital Signal Processors (DSP), clock synchronizing circuits, and even microprocessors complement the designer's palette of elements to use. FPGAs operate at a much slower clock rate compared to CPUs, however, the immense internal bandwidth as well as the intrinsic opportunities for parallelism - if it can be exploited - allow for very fast processing at a fraction of the energy requirements vs. competitive technologies. FPGA

*Also at the Telecommunication Systems Institute, Chania, Greece technology has been used to accelerate CA simulations ever since its infancy in the early 1990s, however, to the present day, the capabilities of FPGAs have evolved in such a major way that the corresponding architectures for CA simulation have changed drastically as well. We will highlight these architectures, historically.

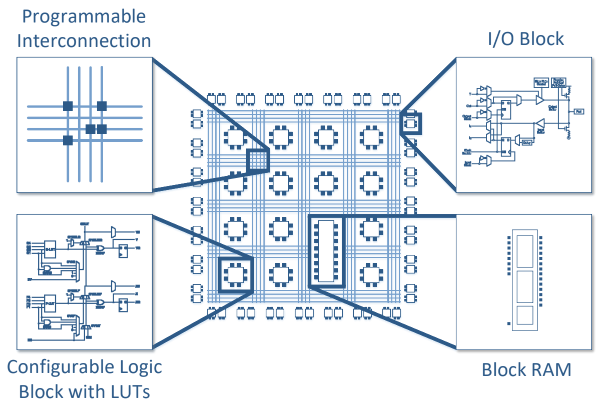

Programmable Interconnection
I/O Block
Configurable Logic Block with LUTs
Block RAM

Fig. 1: The basic components of an FPGA.

Toffoli and Margolus's *Cellular Automata Machines* (*CAM*) were the first special-purpose computers designed to accelerate CA simulations. The first results were published in 1984 [5]. The approach followed by Toffoli consists of only one hardware module implementing the transition function which 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. In 1986, the next version of *CAM*, *CAM-6*, was completed by Toffoli and Margolus and was produced commercially as a PC expansion board [6]. The latest version of *CAMs*, *CAM-8*, made its appearance in 1993 [7]. It was a multiprocessor version of its predecessors, with interconnected *CAM*-like modules processing separate sectors of the CA grid simultaneously. The machine supported arbitrarily large neighborhood sizes and cell sizes in bits, as its computation was based on shifting the data of a sector accordingly before processing them. Following the steps of its predecessors, *CAM-8*'s performance was outstanding at that time and led to Margolus's later work on custom FPGA machines, up to 2000 [8], [9].

During the 1990s the *Cellular Processing Architecture* (*CEPRA*), an FPGA-based architecture, was developed 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 to compute 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 [10]. It contained 8 FPGA-based CA processors which could access all their 3×3 neighborhood cells simultaneously thanks to a computation window buffer. *CEPRA-1X*, *CEPRA-8L*'s successor, was completed in 1997 [11]. 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.

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 [12]. They introduced their *Scalable Parallel Architecture for Concurrency Experiments* (*SPACE*) and proposed a different approach to design hardware for CA simulations. 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 10x speedup over 2 *CAM-8* modules.

In 2001, Kobori, Maruyama and Hoshino from the University of Tsukuba in Japan presented their own FPGA-based CA system [13]. Their streaming architecture consisted of an array of PEs sweeping across the CA grid. 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 $g+n$. This FPGA-based CA system comprised of an off-the-shelf PCI board with one FPGA and used the host computer to display the results. It could simulate a 2048×1024 FHP lattice gas automaton and calculate 400 generations per second, achieving nearly a 155x speedup over a high-end CPU at the time. However, the CA visualization was in pseudo-real time, as most calculated generations never reached the PE array's output.

The contribution of the aforementioned projects to the field of custom CA computers is substantial. However, they comprise only a fraction of the landscape. During the last 3 decades, many other significant projects and developments have contributed to the exploration of the field and FPGAs have been widely used to simulate CA. Most implementations have been custom to a specific CA rule without the use of large neighborhoods [14].

As we have seen so far, there are two prevailing approaches to design custom hardware accelerators for CA simulation:

- To exploit a CA's spatial parallelism by implementing it as an array of PEs. Each PE represents a CA cell, interconnected to its adjacent PEs which are 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 neighborhood sizes, a PE's demand in logic and routing resources increases, and performance drops.
- To design a streaming architecture which processes the CA as a stream of cells. This approach is more suitable

for rules with large neighborhoods on large grids. Our work falls within this approach, but with a new architecture as present-day FPGAs offer different capabilities as opposed to those of ten years ago.

B. *FPGAs vs. Competitive Technologies for CA Simulations*

The work presented in this paper has been evolving over the last four years and previous versions of our architecture have been submitted to the Xilinx Design Competition, with 11×11 neighborhoods in 2015 [15] and 21×21 neighborhoods in 2018 [16], reaching a top-12 distinction in both cases among more than one hundred designs. The main reason why this project has been continuing for close to five years remains the untapped full potential of present-day FPGAs vs. competitive technologies such as CPUs. In this subsection we will try to highlight the reasons why *both* the CA neighborhood and the states simulated on an FPGA can increase substantially before we reach the hard limits of the technology. We will try to be very specific, as the purpose of this work is to prompt the CA community to direct us to well-sized architectures which are meaningful and useful.

FPGAs generally run at one order of magnitude slower clock rate vs. CPUs. A design for non-trivial problems running at 300MHz is often considered to be very good. This means that with a parallelism level of 10 we could break even vs. a CPU if input/output (I/O) were to scale evenly - which they do not. The only characteristic that may be considered similar to a CPU is the latency and the bandwidth of the main memory (e.g. DDR3, DDR4, etc.). Other than that, a CPU has typically two levels of cache memory with 3-5 cycles access time for the first level, and around 10 cycles for the second level. These figures are important because the number of registers in almost all present-day architectures is 32. Therefore, if we increase the neighborhood of a CA even to the 29×29, which is the figure of our present architecture, the weights alone are 841 and each one needs to operate on a 4-bit or 8-bit region (if we have 16 or 256 states, respectively). Present-day CPUs have many cores, and they employ pipelining (i.e. overlapping different stages of successive instructions so that instructions with many cycles each are completed on successive clock cycles). The above description is somewhat of an oversimplification, as modern computer architects employ other techniques as well in order to gain performance. Nonetheless, as the CA neighborhoods get larger, the use of the so-called Level-1 cache is mandatory. Therefore, limitations in CA simulation on CPUs do not stem from the required processing but rather from the mandatory data movement. Simply stated, for the computation of a single CA cell there need to be multiple CPU memory accesses if the neighborhood becomes large enough. Even techniques such as storing multiple weights in a single word, can only push a bit further the hard limit of a CPU's capabilities.

By contrast, the computational resources on-chip of an FPGA are fully customizable, This means that even a small FPGA could store not only the 29×29 weights presented here, but much larger neighborhoods as well. In order for each cell to enter the FPGA only once, the internal BRAM storage needs to store $n\times(k-1)+k$ elements (size n lines for $k-1$ rows of the neighborhood - all but the last line to be processed, plus k elements of the last line) and the $k\times k$ weights of the neighborhood, for $n\times n$ grids and $k\times k$ weights of the neighborhoods. There may be additional considerations such as whether the grid is planar or a torus, or whether we have multiple sets of rules for the CA, but as a rough computation the figures are correct. The capability of FPGAs to store the internal state plus the required buffers means that for each external memory access to read an input datum we have O(k^2) operations which are performed internally to the FPGA at a multi-terabyte internal bandwidth. In addition, the word size of operands is fully customizable, yielding a user-defined trade-off between accuracy and level of exploitable parallelism (the speed is less of a concern as due to pipelining only the initial latency is affected in CA simulations).

Unlike CPUs, GPUs can muster a very substantial level of parallelism, and hence they may prove to be a very useful technology for CA simulation in the future. Nonetheless, the situation in the late 2010's is that GPUs are still highly optimized for vector operations on floating point numbers. Although support of GPUs of certain operations (designed in for deep learning applications) could prove useful towards CA simulation, GPUs require an order of magnitude more energy for this type of computation vs. FPGAs, and their computational model would leave vast resources underutilized. This said, the parallelism which is built-in GPUs could prove to be very valuable, especially if at some future point CA computations evolve into floating point representation and for neighborhoods which are beyond on-chip storage of FPGAs.

III. Design and Architecture

A detailed and thorough description of our architecture is beyond the scope of this paper as this would come at the expense of not showing the CA simulation capabilities that the new architecture allows for. However, the key elements of the design are briefly presented here for readability purposes.

A simplified schematic of our system is shown in Figure 2. In order for it to begin its operation, the system's memory needs to be loaded with the initial CA state (an initial state for each cell of the grid at time $t=0$) via a serial port (UART) from a computer. The software needed to create and transmit a compatible file to our system has been developed as part of this project and is provided to the user.

After the memory initialization process is complete, the system starts displaying the stored CA grid on screen. In order to calculate a new CA state we need to have a complete timestamp of the previous state of the automaton. Thus, for the purpose of double buffering a completed timestamp of the automaton state is presented to the user while the same timestamp data is processed by the *CA Engine* in order to produce the next generation of the automaton.

Every line that is loaded into the graphics buffer following the controller's request, is also loaded into the *CA Engine* buffer. The *CA Engine* buffer holds all the lines needed to

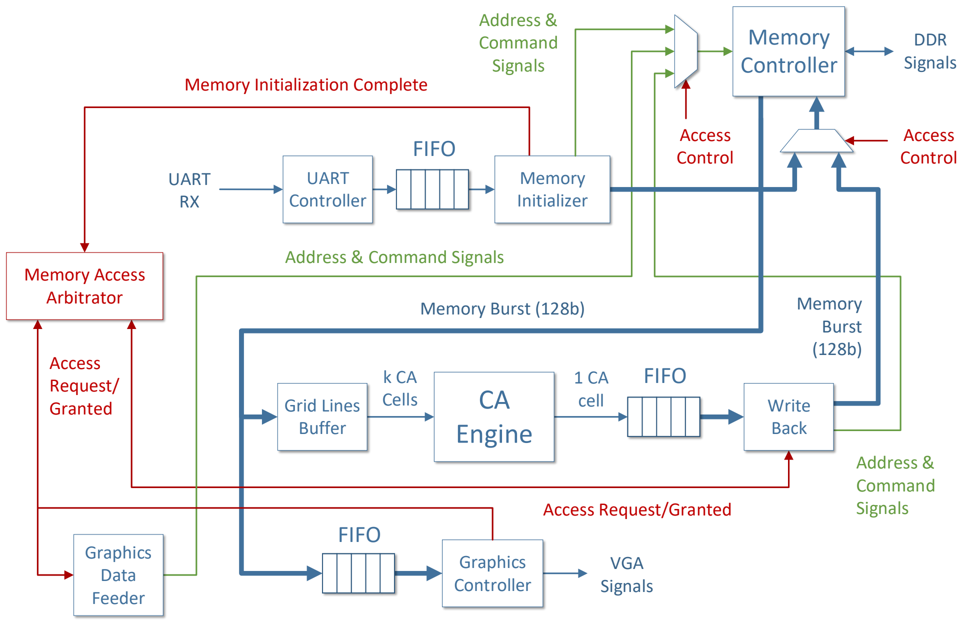


Fig. 2: A simplified schematic of the system architecture.

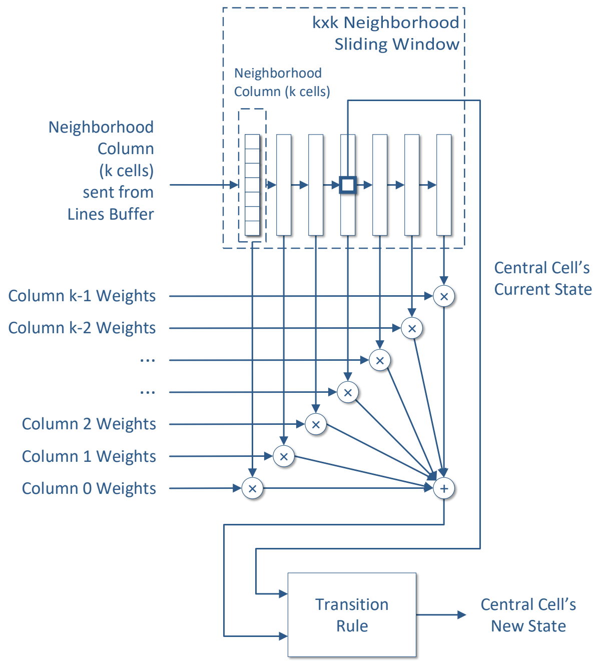

kxk Neighborhood Sliding Window
Neighborhood Column (k cells)
Neighborhood Column (k cells) sent from Lines Buffer
Central Cell's Current State
Column k-1 Weights
Column k-2 Weights
…
…
Column 2 Weights
Column 1 Weights
Column 0 Weights
Transition Rule
Central Cell's New State

Fig. 3: The *CA Engine*.

provide the engine with the neighborhood of each cell of the line being processed.

This custom buffer combined with the *CA Engine*'s neighborhood sliding window provide the arithmetic logic with the complete neighborhood of a different cell at every clock cycle (Figure 3). As a result, loading a cell's neighborhood data into the processing unit is reduced from a $O(k^2)$ problem into a $O(1)$ task in terms of time (but with $O(k^2)$ resources - i.e. level of parallelism), where k is the size of the neighborhood in each dimension. The FPGA's large internal bandwidth allows for the design of a fully parallel *CA Engine* which can produce a new cell value per clock cycle.

Our design was implemented in VHDL using the *Xilinx Vivado 2018.1* tools and mapped for testing and experimentation to the *Digilent Nexys 4 DDR Artix-7* FPGA board. Before generating the design, the user needs to set its generic variables accordingly based on the CA rule they wish to simulate. These parameters, which include the cell size in bits and the neighborhood size, determine the number and size of the buffers used in the design, as well as the number of pipeline stages required to meet the timing constraints. The aforementioned dimensioning procedure is performed automatically during the design instantiation process. The *CA Engine* runs at 200 MHz, fast enough to produce and display 60 CA generations per second at 1080p (Full HD), with each cell of the automaton being represented by one pixel on the screen.

The cellular grids that can be simulated by our architecture are virtually unlimited, since our design constraints are not due to the time complexity of the arithmetic calculations but rather due to the amount of the FPGA's internal BRAM resources. At present we limit the resolution to that of the display for practical reasons.

IV. Simulation Examples and Results

In order to demonstrate the capabilities of our design we chose two well-known CA which we expanded so that they utilize large, 29×29 neighborhoods and many cell states. In addition, we experimented with large-neighborhood anisotropic rules which show interesting self-organization behavior.

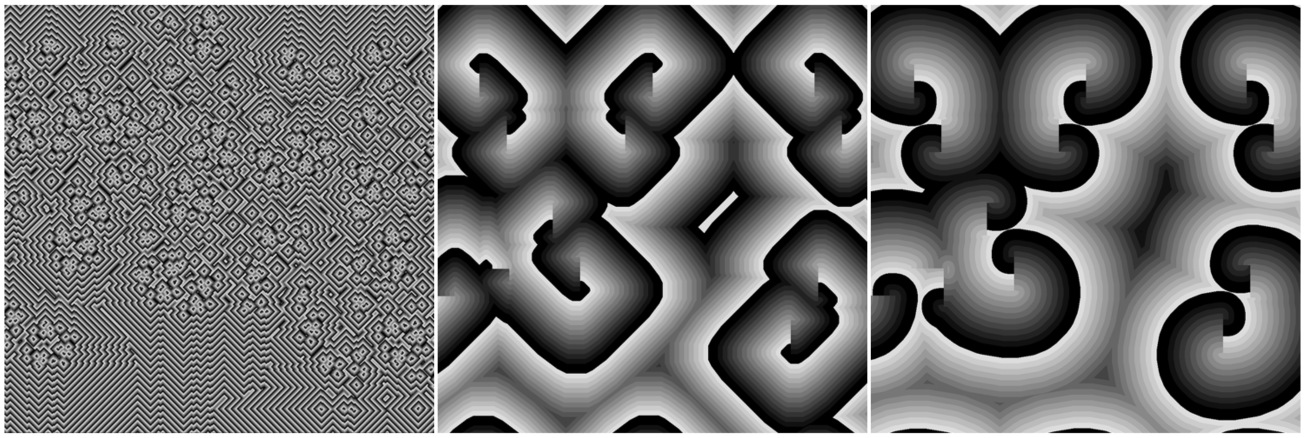


Fig. 4: The *Greenberg-Hastings* CA after 500 iterations with a 3×3 von Neumann, a 29×29 von Neumann and a 29×29 circular neighborhood respectively. The three images above are of the same resolution, with each pixel representing a cell of the automaton's grid.

The rules chosen are the *Greenberg-Hastings* CA and the *Hodgepodge Machine*. Both CA belong to the group of excitable CA, a type of CA which models excitable media. An excitable medium is a dynamical system which has the ability to propagate spatially distributed periodic waves. The neighborhood and state expansion of the CA rules, above, is not new [17], [18]. However, to our knowledge, it is the first time that such large neighborhoods are utilized, yielding interesting results.

A. The Greenberg-Hastings Model

The *Greenberg-Hastings* Model (GHM) is a 3-state CA with a von Neumann neighborhood with radius 1, designed to model excitable media [19]. In spite of their simplicity, the carefully designed rules produce interesting excitable media patterns. In the past, more sophisticated attributes have been added to GHM with the use of FPGA technology, such as stochasticity, in order to create a more realistic and isotropic excitable media model [20].

In its original form, there are numerous initial conditions which lead to periodic behavior [21]. Such initial conditions are proven to exist in the extended version of the rule as well, where the width of the waves has also been proven to be proportional to the neighborhood size [17]. For this simulation example we used a 29×29 von Neumann neighborhood and cells with 16 states. A cell can be "quiescent" (state 0), "excited" (state 1) or in a sequence of "refraction" (states 2 to 15). The cell's transition function is defined as:

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

where t is a threshold value. As shown in Figure 4, by using a circular neighborhood instead of a von Neumann neighborhood, the patterns formed are heavily affected by the change and the waves and vortices become curved, which is an interesting qualitative result in its own right.

B. The Hodgepodge Machine

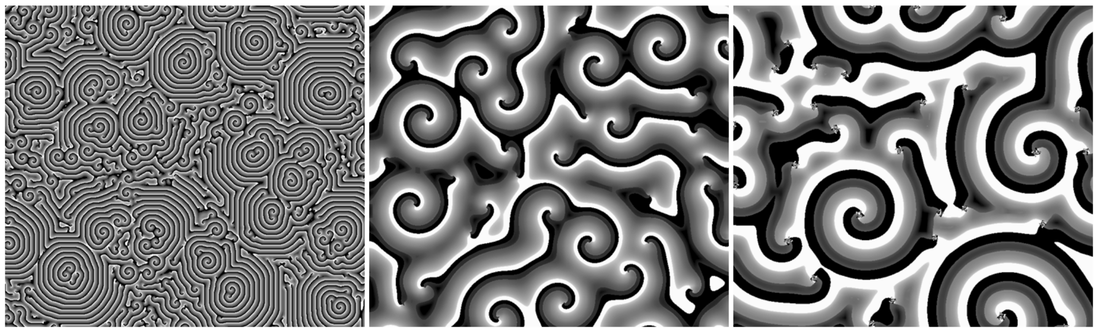
The second example is a CA rule known as the *Hodgepodge Machine*, an excitable CA with q states. It was designed in 1988 by Martin Gerhardt and Heike Schuster and was popularized by Alexander Dewdney and his column in Scientific American's Computer Recreations [22], [23]. Normally, the Hodgepodge Machine rules require a 3×3 Moore neighborhood.

For this simulation we used a 29×29 Moore neighborhood and a simplified version of the rule which calculates 1 sum for both ill ($cell\ state = q$) and infected cells ($0 < cell < q$). The cell's state transition function is defined as:

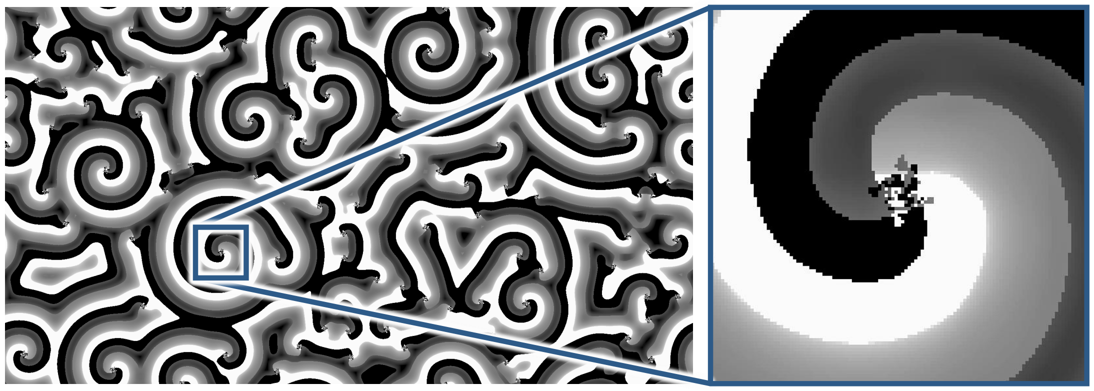
$$c_{t+1}(i,j) = \begin{cases} \dfrac{\text{number of infected and ill neighbors}}{k} & \text{if } c_t(i,j) = 0 \\ 0 & \text{if } c_t(i,j) = q \\ \dfrac{\text{sum of all neighbors}}{\text{sum of infected neighbors}} + g & \text{otherwise} \end{cases}$$

where k and g are the two parameters of the rule that determine when and how fast the "infection" will spread. The automaton's parameters used for this simulation were $q = 255$, $k = 5$ and $g = 105$.

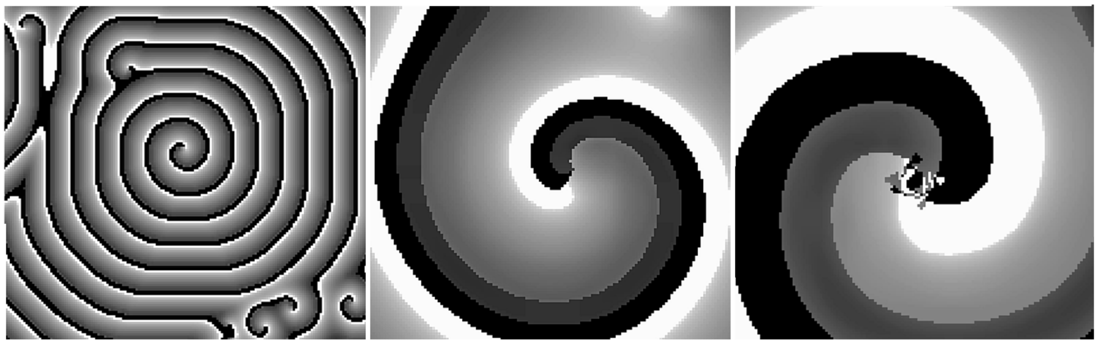
As we can see in Figure 5, we notice that the vortices produced co-exist with small, stable, vortex-like patterns located in the center of the larger vortices. This phenomenon was not present in any of our earlier experiments with smaller neighborhood sizes reaching up to 19×19 cells. An additional qualitative difference vs. simulation with smaller neighborhoods was the apparent lack of convergence of the CA when the initial data were completely random, but with convergence when 19×19 sized tiles of random numbers are placed randomly in the grid - a phenomenon which is not present in the 3×3 neighborhood.



(a) The *Hodgepodge Machine* after 500 iterations with a 3×3, a 19×19 and a 29×29 Moore neighborhood respectively. The three images above are of the same resolution, with each pixel representing a cell of the automaton's grid.



(b) The use of 29×29 neighborhoods results in large vortices with stable core patterns.



(c) The stable vortex cores are not formed with smaller neighborhoods, such as the 3×3 or the 19×19 Moore neighborhood.

Fig. 5: The *Hodgepodge Machine* with a 29×29 Moore neighborhood produces results which differ from those when smaller neighborhoods are used.

C. *Anisotropic Rules*

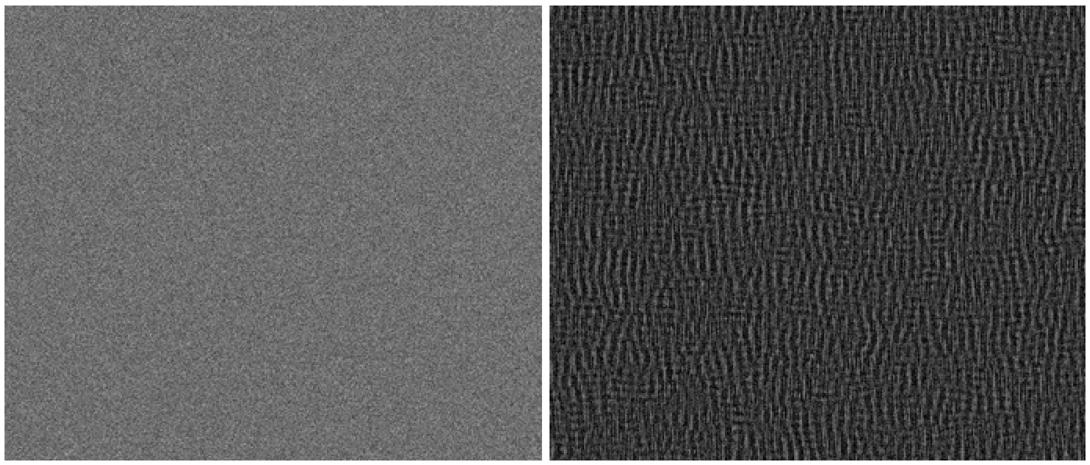
The anisotropy of CA lies either in the anisotropy of the grid, the anisotropy of the neighborhood or both. We experimented with the neighborhood's anisotropy. Our anisotropic 29×29 Moore neighborhood contains the largest weights at its far right (eastern) edge, with the weight value gradually being reduced to 1 towards the far left (western) edge of the neighborhood. The state transition function for each 256-state cell is quite simple and defined as:

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

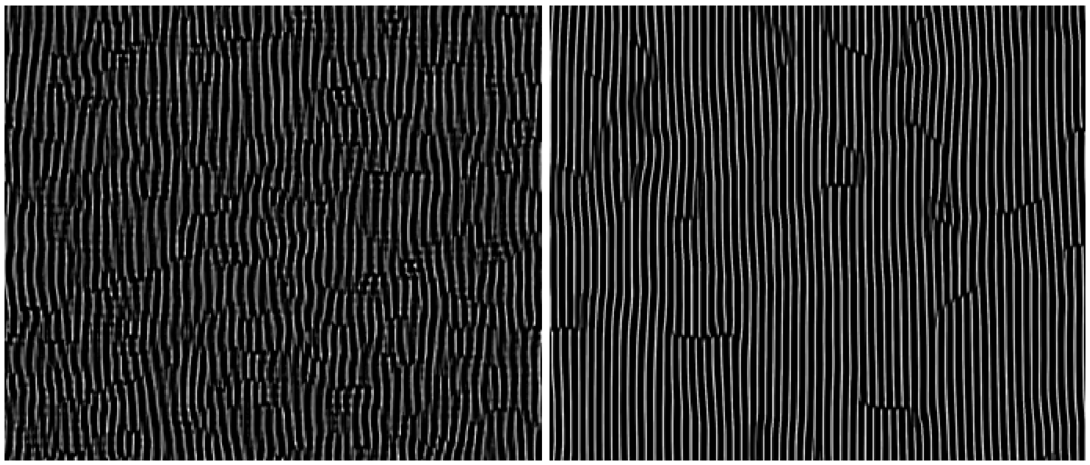
The self-organization properties of this relatively simple anisotropic CA can be seen in Figure 6. Starting from a randomly filled grid, the automaton forms long, thin stripes of cells after several thousand generations. As time goes by, the worm-like ripples shown in Figure 6b tend to propagate to the left by virtue of the anisotropic neighborhood's weights. The long, thin, rope-like structures become more coherent once a ripple is straightened and the two edges of the ropes merge into one.

D. *Performance Results*

Our architecture calculates 60 CA generations per second on 1920×1080 FHD grids with 8-bit pixel representation,



(a) Starting from a randomly filled grid, the automaton cells form horizontal and vertical structures after 120 generations.



(b) The evolution of the automaton after 500 and 10000 generations respectively.

Fig. 6: The self-organization properties of a simple anisotropic CA with a large 29×29 Moore neighborhood after 1, 120, 500 and 10,000 generations. The horizontal and vertical cell concentrations after 120 generations, and the worm-like patterns after 500 and especially after 10,000 generations are not artifacts from the image resolution in the paper - they exist on the large screen as well.

regardless of the rule complexity. The output is fed to a VGA interface, and the user can "slow down" the simulation. As shown below, our system has a measured speedup of up to $51\times$ against an *Intel Core i7-7700HQ* CPU (1 core) running highly optimized (-O3) software programmed in C. Table I shows the CPU and FPGA result averages from multiple runs of two CA models. The speedup of a contemporary GPU vs. a CPU is comparable to our speedup for similar CA rules, but for smaller, 11×11 neighborhoods [24], [25]; unlike our architecture, the GPU performance gets degraded as neighborhoods become larger, and a GPU consumes at least 10 times more energy than an FPGA for these computations.

The amount of required FPGA resources for each rule depends on the CA rule's neighborhood and the size of each cell in bits. Results from our implementation of the *Hodgepodge Machine* on the modest-sized FPGA of the Xilinx Nexys 2 board are shown in Table II.

TABLE I. COMPARATIVE RESULTS: EXECUTION TIME AND SPEEDUP

Cellular Automaton	**i7-7700HQ, 1000 gen.**	**Our Design, 1000 gen.**	**Our Design's Speedup**
Greenberg-Hastings, n = 29	469.58 sec	16.67 sec	28.16×
The Hodgepodge Machine, n = 29	851.29 sec	16.67 sec	51.06×

V. Discussion and Conclusions

In the present paper we focused on the enabling characteristics of present-day FPGAs for large neighborhood CA simulation. Although exceptionally elegant and powerful at the time, the FPGA-based CA architectures of the early 1990s have a different philosophy than the one we present here. All aspects of computation, internal storage, and even development tools are different now with respect to that time, and hence

TABLE II. RESOURCE UTILIZATION

Resource	Utilization	Utilization %
LUT	20375	32.14
LUTRAM	1555	8.18
FF	27224	21.47
BRAM	65	48.15
DSP	1	0.42
IO	73	34.76
BUFG	7	21.88
MMCM	3	50
PLL	1	16.67

the solution is different as well. Table III, below, shows the current work, recently published work on GPUs, as well as the older, classic FPGA-based CAM architectures.

A small present-day FPGA can easily hold in terms of the main parameters $n = 2000$ and $k = 40$, (being limited by the BRAM resources and the on-chip logic, respectively). Subject to the above constraints, the internal bandwidth is such that with pipelining we can have one cell processed per cycle at the rate of the cells, as they enter the FPGA from external memory. More complex rules and state sensitive rules (i.e. rules that change, depending on the state of a cell) can be implemented as well.

TABLE III. LARGE-NEIGHBORHOOD CA IMPLEMENTATIONS ON HARDWARE

Architecture	Neighborhood Size	Performance
Margolus, 1993-2001, CAMs	experimented with up to 11×11	10 gen./sec for a 512×512 grid with 3-bit cells
Gibson et al., 2015, Workstation with Nvidia GTX 560 Ti	experimented with up to 11×11	≈ 65× over serial for Game of Life on a 2048×2048 grid
Millan et al., 2017, Nvidia TitanX GPU	experimented with up to 11×11	21.1× over serial for Game of Life on a 4096×4096 grid
Kyparissas & Dollas, 2019, Artix-7 FPGA	experimented with up to 29×29	51× over serial for the Hodgepodge Machine on a 1920×1080 grid

We have demonstrated experimentally that present-day FPGAs are a "game changer" in terms of capabilities to simulate CA with large, 29×29 neighborhoods. A trade-off between performance and flexibility comes from the designs being "compiled in" but applicable to varying datasets. At present, we have a framework in which we enter in an easy form the desired rules, and the framework generates the hardware project which compiles the design with *Xilinx Vivado 2018.1* for *Digilent's Nexys 4 DDR Artix-7* FPGA board - we could have a generic design at the trade-off of somewhat smaller neighborhoods or smaller number of frames per second.

We hope that CA engines with very large k, very large n, and even "strange" rules (e.g. state-based rules, location-based rules, probabilistic rules, anisotropic rules, etc.), will result to interesting, new CA experiments.

Acknowledgements

This work was partially funded by the European commission in the context of the FP7 ICT project QualiMaster (under grant no. 6195525). The authors wish to acknowledge Xilinx Corp. for donating the Nexys 4 DDR board that was used in this work.

References

[1] J. von Neumann, "The General and Logical Theory of Automata," *Cerebral Mechanisms in Behavior: The Hixon Symposium, John Wiley & Sons*, 1951.

[2] J. von Neumann and A. W. Burks, *Theory of Self-Reproducing Automata*. University of Illinois Press, 1966.

[3] S. Ulam, "Random Processes and Transformations," in *International Congress of Mathematicians*, Cambridge, 1950.

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

[5] T. Toffoli, "CAM: A High-Performance Cellular-Automaton Machine," *Physica D: Nonlinear Phenomena*, vol. 10, no. 1-2, 1984.

[6] T. Toffoli and N. H. Margolus, *Cellular Automata Machines - A New Environment for Modeling*. MIT Press, 1987.

[7] N. H. Margolus, "CAM-8: A Computer Architecture Based on Cellular Automata," *Pattern Formation and Lattice-Gas Automata, AMS*, 1993.

[8] ——, "An FPGA Architecture for DRAM-Based Systolic Computations," in *5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '97)*, 1997.

[9] ——, "An Embedded DRAM Architecture for Large-Scale Spatial-Lattice Computations," in *27th Annual International Symposium on Computer Architecture (ISCA '00)*, 2000.

[10] R. Hoffmann, K.-P. Völkmann, and M. Sobolewski, "The Cellular Processing Machine CEPRA-8L," *Mathematical Research*, no. 81, 1994.

[11] C. Hochberger, R. Hoffmann, K.-P. Völkmann, and J. Steuerwald, "The CEPRA-1X Cellular Processor," *Reconfigurable Architectures: High Performance by Configware, IT Press, Bruchsal*, 1997.

[12] P. Shaw, P. Cockshott, and P. Barrie, "Implementation of Lattice Gases Using FPGAs," *Physica D: Nonlinear Phenomena*, vol. 12, no. 1, 1996.

[13] T. Kobori, T. Maruyama, and T. Hoshino, "A Cellular Automata System with FPGA," in *9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '01)*, 2001.

[14] G. C. Sirakoulis, "Cellular Automata Hardware Implementation," in *Cellular Automata: A Volume in the Encyclopedia of Complexity and Systems Science, Second Edition*, A. Adamatzky, Ed. New York, NY: Springer US, 2018.

[15] N. Kyparissas and A. Dollas, "Game of Complex Life - Modeling of Urban Growth Processes with Cellular Automata," in *Xilinx Open Hardware European Design Contest*, 2015, http://www.openhw.eu/2015-finalists.html.

[16] ——, "A Parallel Framework for Simulating Cellular Automata on FPGA Logic," in *Xilinx Open Hardware European Design Contest*, 2018, http://www.openhw.eu/2018-finalists.html.

[17] R. Fisch, J. Gravner, and D. Griffeath, "Threshold-Range Scaling of Excitable Cellular Automata," *Statistics and Computing*, vol. 1, 1991.

[18] S. Robles, "Cellular Automata," *Fractal Design [Online], http://www.fractaldesign.net*, Accessed: April 2019.

[19] J. M. Greenberg and S. P. Hastings, "Spatial Patterns for Discrete Models of Diffusion in Excitable Media," *SIAM Journal on Applied Mathematics*, no. 54, 1978.

[20] N. Vlassopoulos, N. Fatès, H. Berry, and B. Girau, "An FPGA Design for the Stochastic Greenberg-Hastings Cellular Automata," in *2010 International Conference on High Performance Computing & Simulation (HPCS '10)*, 2010.

[21] J. M. Greenberg, C. Greene, and S. Hastings, "A Combinatorial Problem Arising in the Study of Reaction-Diffusion Equations," *SIAM J. Matrix Analysis Applications*, vol. 1, 1980.

[22] A. K. Dewdney, "Computer Recreations: The Hodgepodge Machine Makes Waves," *Scientific American*, vol. 259, no. 2, 1988.

[23] M. Gerhardt and H. Schuster, "A Cellular Automaton Describing the Formation of Spatially Ordered Structures in Chemical Systems," *Physica D: Nonlinear Phenomena*, vol. 36, no. 3, 1989.

[24] M. J. Gibson, E. C. Keedwell, and D. A. Savić, "An Investigation of the Efficient Implementation of Cellular Automata on Multi-Core CPU and GPU Hardware," *Journal of Parallel and Distributed Computing*, vol. 77, 2015.

[25] E. N. Millán, N. Wolovick, M. F. Piccoli, C. G. Garino, and E. M. Bringa, "Performance Analysis and Comparison of Cellular Automata GPU Implementations," *Cluster Computing*, vol. 20, no. 3, 2017.