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

Conference Paper | Accepted Manuscript
Published in: 2019 29th International Conference on Field Programmable Logic and Applications (FPL)

DOI: 10.1109/FPL.2019.00024

An FPGA-based Architecture to Simulate Cellular Automata with Large Neighborhoods in Real Time

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*—In this paper we present a reconfigurable logic-based parallel architecture for the computation of 29×29 large-neighborhood cellular automata at 60 frames-per-second (FPS) real time update rate, using a small FPGA. The computation for each one of the n^2 elements of a two-dimensional input is O(k^2), where k is the size of the neighborhood in each dimension. All buffering and computation is performed internally in the FPGA. In terms of performance results, our architecture outperforms a general-purpose CPU running highly optimized software programmed in C by up to 51$\times$; in neighborhoods up to 11×11 in which there are published results from GPUs our architecture has similar performance to GPUs at one-tenth the energy requirements, however, our architecture has the same performance for 29×29 neighborhoods whereas GPU performance drops as neighborhood grows. We expect this work to provide enabling new tools for the use of cellular automata models in the physical sciences.**

I. Introduction

Cellular automata (CA) were proposed by von Neumann in the 1950s [1], [2]. Their "simulation" (a term used by the CA community) is an abstract mathematical paradigm for massively parallel computation. Countless non-linear dynamic systems and physical processes, from molecular dynamics [3], [4] to large ecosystems [5] are modeled with CA. Hardware technology has been widely used as CA computation accelerators [6], including FPGA technology since its earliest days.

In the early 1980s, Toffoli and Margolus from the Massachusetts Institute of Technology developed their first Cellular Automaton Machine (CAM) [7]. CAM was a streaming architecture which used DRAM to store the CA grid and SRAM look-up tables to calculate the next state of the cell. Further development led to CAM-6 and in the 1990's CAM-8 (a multiprocessor version of CAM-6) [8]–[10].

During the 1990s an FPGA-based architecture for CA simulations was developed at the TU Darmstadt [11], [12]. The Cellular Processing Architecture (CEPRA) was a streaming architecture like CAM and used a similar neighborhood buffer. The difference between the two machines was that CEPRA used arithmetic logic to calculate the transition rule, in contrast to CAM which used SRAM look-up tables.

In 2001 Kobori *et al.* used a different approach to simulate CA with the use of FPGAs [13]. Their streaming architecture consisted of an interconnected array of processing elements sweeping across the grid. In this computation method, each cell of the grid is processed consecutive times within the FPGA. As a result, the output cells belong to a generation which is several generations ahead of the input cells. This approach is very fast for small neighborhoods as it reduces memory transfers, however, connecting all cells to all possible neighbors becomes inefficient as neighborhoods become larger due to routing demands.

None of the above significant contributions supported the efficient simulation of large-neighborhood CA, largely due to limitations of the underlying technology - each approach was excellent for the technologies at hand. Present-day FPGAs have on-chip available resources which, combined with the TBytes/sec internal bandwidth and customizable datapaths, mandate the re-visitation of CA computations. This paper presents an efficient architecture to take full advantage of modern FPGA capabilities and design tools, in order to yield an efficient parallel architecture for the simulation of large-neighborhood cellular automata on large grids.

Section II has a brief theoretical background, Section III has the design of our proposed architecture, followed by Section IV with simulation examples. Section V has results from the proposed architecture, and Section VI has conclusions from this work.

II. Brief Theoretical Background

A 2D CA (the scope of the present work) consists of a rectangular grid of cells, each in one of a finite number of states. For each cell, a set of cells is defined relative to the specified cell, called its neighborhood. 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 that determine the new state of each cell on the next time interval in terms of the current state of the cell and the states of the cells in its 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/or weighted form (figure 1). 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 number of cells from the central cell to the neighborhood's edge.

A distinct, widely-used class of CA are totalistic CA. 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

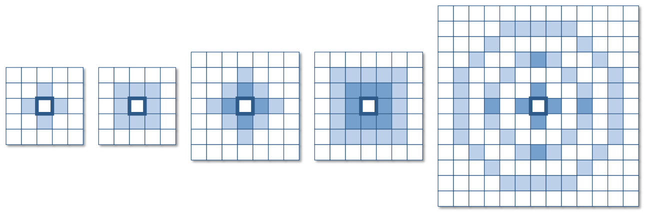

Fig. 1. Basic types of neighborhoods in 2D cellular automata (left to right, shaded areas are within the neighborhood): von Neumann; Moore; weighted, extended von Neumann; weighted, extended Moore; weighted, custom neighborhood.

only on the sum of the current values of the cells in its neighborhood [14]:

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

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

where r is the neighborhood's radius, $w(x,y)$ the neighborhood's weights with $w(0,0) = 0$ and $a, b, \ldots \in \mathbb{Z}$.

If the next state of a cell depends on both its own current state ($w(0,0) \neq 0$) and the total of its neighbors, then the CA belongs to the class of outer totalistic CA [14]. Conway's Game of Life, one of the best-known CA, is an example of an outer totalistic CA with 2 states and a simple Moore neighborhood [15].

Calculating CA with a small set of states and small neighborhoods can be accomplished efficiently by general purpose CPUs with algorithms like the Hashlife algorithm [16] but as the state complexity and neighborhood grow the use of accelerators is crucial. For real-time simulation, we arbitrarily chose 60 FPS to match the screen update rate at Full-HD definition, although even with the small FPGA we use, we can go up to close to 100 FPS.

III. Design

Our design was implemented in VHDL, using Xilinx Vivado 2018.1. Digilent's Nexys 4 DDR Artix-7 FPGA Board, which was connected to a monitor via a VGA cable, was used for testing. It constitutes a fully-pipelined parallel architecture for real-time 2D CA simulations which helps users save precious time as we will demonstrate later in this paper (Section IV, Design Reuse and Simulation Examples). The user can use the board's "up" and "down" push-buttons to control the simulation speed. This design has evolved with substantial improvements from our older design *Game of Complex Life - Modeling of Urban Growth Processes with Cellular Automata* (one of top-12 designs in the Xilinx Open Hardware 2015 Design Contest, unpublished outside the scope of the competition). Our fully implemented design, supports:

- Full-HD (1080p, 60 Hz) graphical projection of a 1920×1080 grid
- Fully pipelined – 60 FPS real-time simulation (the present design can reach 100 FPS)
- Use of Nexys 4 DDR's external DDR2 memory for virtually unlimited input grids - even if they exceed the graphics resolution ability of the engine to display them (at present we limit the resolution to that of the display, however, this is not a real constraint)
- More versatile customization.

Initially, the user has to choose any neighborhood size and a 4-bit or 8-bit cell size (for up to 16 or 256 states/cell). These parameters affect the design to be generated, such as the size of the buffers used in the design, as well as the pipeline stages needed to meet the timing constraints. The aforementioned dimensioning procedure is performed automatically during the design instantiation process.

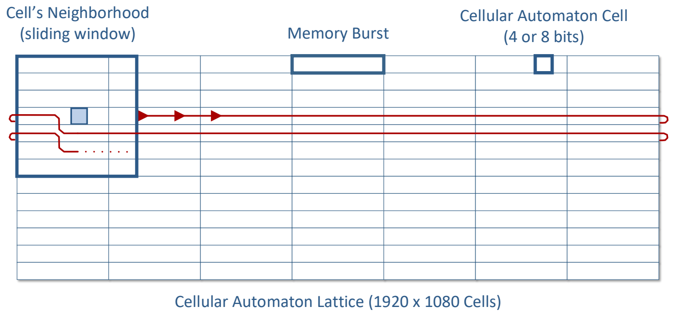
Cell's Neighborhood (sliding window)
Memory Burst
Cellular Automaton Cell (4 or 8 bits)
Cellular Automaton Lattice (1920 x 1080 Cells)

Fig. 2. Grid Representation in memory and sliding window swift.

In order to calculate a new CA state (a.k.a. the next timestamp), we need to have a complete timestamp of the previous CA state. Thus, double buffering is mandatory. A CA grid timestamp is represented in our system as shown in figure 2: The grid consists of 1080 lines, and each line consists of 1920 CA cells. According to the CA cell size chosen by the user, a grid line size will vary measured in bursts/line as the burst size is fixed by the memory controller.

Our system consists of four basic sections:

1) Memory Initialization 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 running at 200 MHz.
4) Graphics running at 148.5 MHz.

A simplified schematic of our system is shown in figure 3. At first the system memory needs to be loaded with the initial CA state (an initial state for each cell of the grid at time $t = 0$) via 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 via VGA at 1080p. Every line that is loaded into the Graphics Controller

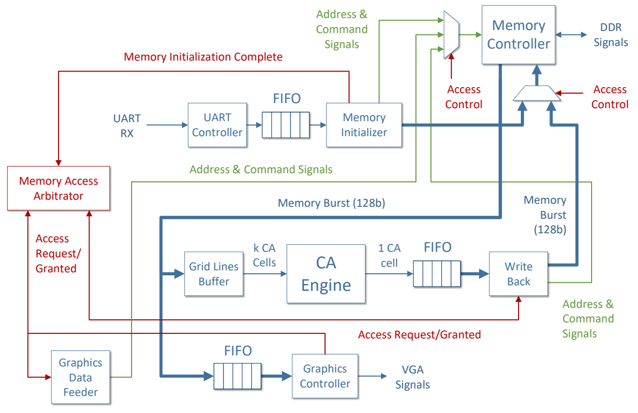

Fig. 3. A simplified schematic of the system architecture.

buffer following the controller's request, is also loaded into the CA Engine buffer. The CA Engine buffer holds all the 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 (figure 2), processing 1 cell/cycle.

Before the Graphics Controller requests a new line to be loaded from the memory, the CA Engine has completed processing the previous line requested by the Graphics Controller and has written the new cell values of that line back to the memory segment that currently represents the CA's next timestamp. This is feasible because the CA Engine datapath is fully-pipelined and operates at a much higher frequency than the Graphics Controller. The communication required between the system modules is minimal, with each part of the architecture having its own independent control unit based on the number of bursts/line and counting the number of lines that have been processed so far.

A. *CA Engine*

The CA Engine calculates the new values of the CA grid cells. This module operates at 200 MHz, receives a new neighborhood column from the Grid Lines Buffer in every clock cycle and produces 1 cell/cycle. The module is fully pipelined and implemented as an adder tree (DSPs could be used, however, more complex routing would impact negatively the pipeline speed). The CA Engine is the only module that needs customization by the user, since the transition rule and the sums required are different for every CA rule.

B. *Grid Lines Buffer*

The Grid Lines Buffer provides the CA Engine a new neighborhood column in every clock cycle. This module uses BRAM modules which were generated by Xilinx Block Memory Generator (v. 8.4). The idea behind this module is based on N. Margolus's use of a corner-turning buffer to feed his proposed FPGA Architecture for systolic computations with data [9]. The Grid Lines Buffer consists of k BRAM modules representing the lines in the CA rule's $k \times k$ neighborhood, plus one BRAM module used as a write buffer. Our design's performance stems from the Grid Lines Buffer and the fact that each cell enters our system only once per frame. In order for each cell to enter our system only once, for a $k \times k$ neighborhood and $n \times n$ datasets (e.g. the HD screen), we need to store internally in the FPGA $(k-1) \times n + k$ elements. This allows for the $k \times k$ neighborhood to move along.

In addition to the above, a Graphics Controller manages the output to the screen, and a Memory Access Arbitrator determines whether the CA computation engine or the Graphics Controller will take the memory bus.

IV. Design Reuse: Simulation Examples

In this section, we demonstrate the reusability of our design by using it to explore 2 large-neighborhood totalistic CA rules. The following examples can be used as a template by any user who wants to simulate their own CA rule.

The source code for user-generated hardware and constraints, the software needed to use our FPGA architecture design and some example designs can be found in https://github.com/nkyparissas/Cellular_Automata_FPGA. The CA Engine needs to be designed by the user with the use of a HDL, since the transition rule and the sums required are different for every CA rule, however, the source code of the following simulation examples can be used as a template.

A. *Artificial Physics*

The first example is a CA rule known as "Artificial Physics": an outer totalistic CA rule with 2 states and a weighted, large neighborhood. The rule's name originates from its fascinating behavior (figure 4). As the CA simulation moves on "atoms" appear in the CA's universe. As time goes by, these "atoms" attract and bind together forming "molecules".

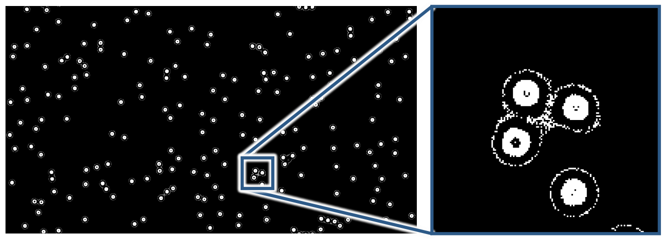

Fig. 4. Artificial Physics

The rule requires a quite large 21×21 neighborhood with binary weights as shown in figure 5. The cell state transition function is defined as:

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

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

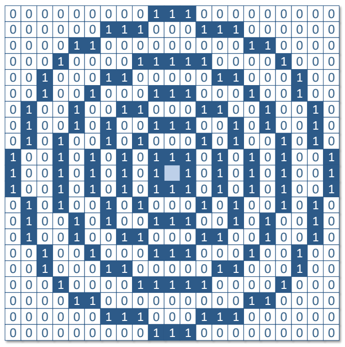

Fig. 5. Artificial Physics 21x21 neighborhood.

B. The Hodgepodge Machine

The second example is a rule known as the "Hodgepodge Machine", an outer totalistic CA rule with q states [17], [18].

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

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

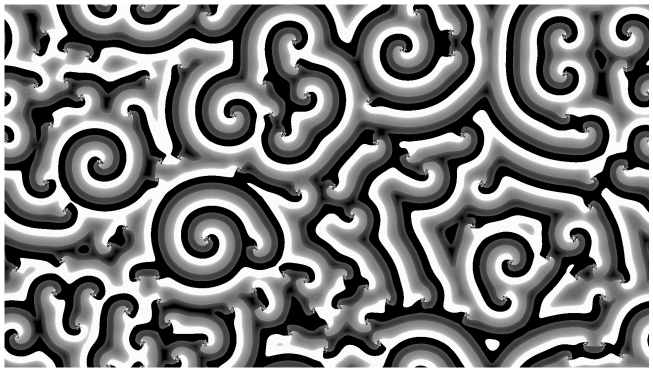

Fig. 6. The Hodgepodge Machine.

The implementation for the Hodgepodge Machine can be found in github and can be used as a template for complex rules with multiple sums per cell state transition. The Hodgepodge Machine's parameters used were: $k = 5$, $g = 105$ and $q = 255$. The system parameters used here were: $neighborhood\ size = 29$ and $cell\ size = 8\ bits$.

V. Results

Our design calculates 60 CA generations/second regardless of the rule running. The grid size is 1920×1080 cells, which gives us 124,416,000 cell updates/second.

Compared to a general-purpose CPU it has a measured speedup of up to $51\times$ against an Intel Core i7-7700HQ CPU (1 core) running highly optimized (-O3) C code, see table I. The speedup offered by contemporary GPUs is comparable to our design's speedup for up to 11×11 neighborhoods (no larger neighborhoods have been reported on GPUs), however, a GPU consumes at least 10 times more power than an FPGA [19], [20]. At the system level a contemporary GPU requires 170 W, whereas our Nexys 4 is powered by USB, i.e. maximum of 7.5 W; in addition, GPU performance drops as the neighborhood gets larger, whereas in our architecture the performance remains the same.

TABLE I
Comparative Performance Results

CA Rule	**i7-7700HQ, 1000 gen.**	**Our Design, 1000 gen.**	**Speedup**
Artificial Physics, n = 21	538.77 sec	16.67 sec	32.32x
The Hodgepodge Machine, n = 29	851.29 sec	16.67 sec	51.06x

The amount of FPGA resources depends mainly on the CA rule's neighborhood size. Some indicative results can be seen in table II after the implementation of the Hodgepodge Machine CA on XC7A100T-1CSG324C, a modest-sized FPGA.

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

TABLE II
Resource utilization.

VI. Conclusions

Our architecture for CA simulations, based on a modest FPGA, has a major speedup vs. a high-end CPU, and very competitive performance vs. a GPU at an order-of-magnitude lower required energy for the computation. The design is automatically generated according to the user's needs, with minimal user code, and is fully customizable. We believe that by exploiting the FPGAs TB/sec internal bandwidth and increasing the computational complexity vs. I/O, we have a new architecture which is enabling technology for large neighborhood CA models. A framework is under development to further aid the process.

Acknowledgements

This work was partially funded by the European commission in the context of the FP7 ICT project QualiMaster (under grant no. 6195525) and by a travel grant from the School of ECE of the Technical University of Crete. 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] J. B. Salem and S. Wolfram, “Thermodynamics and Hydrodynamics with Cellular Automata,” in *Theory and Applications of Cellular Automata*. World Scientific, 1986.

[4] D. H. Rothman and S. Zaleski, *Lattice-Gas Cellular Automata: Simple Models of Complex Hydrodynamics*. Cambridge University Press, 2004.

[5] P. Hogeweg, “Cellular Automata as a Paradigm for Ecological Modeling,” *Applied Mathematics and Computation*, vol. 27, no. 1, 1988.

[6] 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.

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

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

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

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

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

[12] 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.

[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] A. Ilachinski, *Cellular Automata: a Discrete Universe*. World Scientific, 2001.

[15] M. Gardner, “Mathematical Games - The Fantastic Combinations of John Conway’s New Solitaire Game “Life”,” *Scientific American*, vol. 223, no. 4, 1970.

[16] W. R. Gosper, “Exploiting Regularities in Large Cellular Spaces,” *Physica D: Nonlinear Phenomena*, vol. 10, no. 1-2, 1984.

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

[18] 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.

[19] 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.

[20] 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.