XILINX OPEN HARDWARE 2018 DESIGN CONTEST

A PARALLEL FRAMEWORK FOR SIMULATING CELLULAR AUTOMATA ON FPGA LOGIC

JUNE 2018

Participant: Nikolaos Kyparissas
Supervisor: Prof. Apostolos Dollas

School of Electrical and Computer Engineering
Technical University of Crete
Chania, Greece

XOHW18-220

nkyparissas@isc.tuc.gr
dollas@ece.tuc.gr

youtu.be/HYWyuwxIZ94

Open Hardware 2018

The project was implemented as part of the
Open Hardware 2018 Design Contest run in partnership by
the Xilinx University Program, the Europractice Software Service, and CNFM.

Contents

1. Introduction 1
2. Brief Theoretical Background 2
3. Design 4
   3.1 CA Engine 8
   3.2 Grid Lines Buffer 9
   3.3 Graphics Controller 10
   3.4 Memory Access Arbitrator 11
   3.5 UART Controller 11
4 Design Reuse and Simulation Examples 12
   4.1 Artificial Physics 12
   4.2 The Hodgepodge Machine 14
5. Results 15
6. Conclusion 16
7. Bibliography in Alphabetical Order 17

1. Introduction

Cellular automata (CA) are discrete mathematical models discovered in the 1940s by John von Neumann and Stanislaw Ulam. They constitute a general paradigm for massively parallel computation. Through time, these powerful mathematical tools have been proven useful in countless ways, both as models of complexity and as models of non-linear dynamic systems in a variety of scientific fields.

In this project we propose a customizable parallel framework on FPGA which can be used to efficiently simulate weighted, large-neighborhood totalistic and outer-totalistic 2D CA.

In terms of results, our pipelined application-specific architecture successfully surpasses the computation and memory bounds found in a general purpose CPU and has a measured speedup of up to 46x against an Intel Core i7-6500 CPU running highly optimized software programmed in C.

2. Brief Theoretical Background

Cellular automata (CA) are discrete mathematical models used in numerous scientific fields. CA's main advantage is their flexible rules - just by setting a few simple states and rules, a CA can model incredibly complex systems.

In this project we will be dealing with 2D CA simulations. A 2D CA consists of a rectangular grid of cells, each in one of a finite number of 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 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.

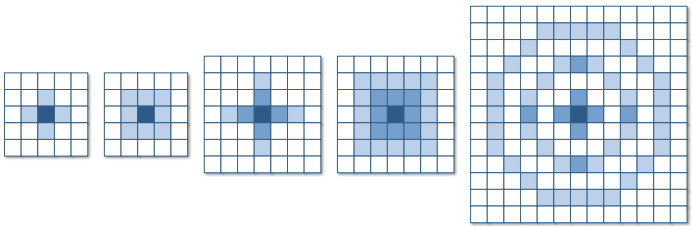

Figure 2.1: Basic types of neighborhoods in a 2D CA:
a) von Neumann, b) Moore, c) Weighted, extended von Neumann,
d) Weighted, extended Moore, e) Weighted, custom neighborhood

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/or weighted form (Figure 2.1). There are two ways to define a cell's neighborhood; either by its nxn 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 only on the

sum of the current values of the cells in its neighborhood[1]:

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

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

where r is the neighborhood's radius, $w(x,y)$ the neighborhood's 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 and the total of its neighbors ($w(0,0) \neq 0$), then the CA belongs to the class of outer totalistic CA.[1] 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.

Even broader versions of totalistic CA include transition rules with more steps involved, like the Hodgepodge Machine discussed later in this report.

Calculating CA with a small set of states and small neighborhoods has been thoroughly explored and can be accomplished efficiently by general purpose CPUs thanks to algorithms like the *Hashlife* algorithm[2]. As the number of states rises and the neighborhood size grows, the complexity of simulating such a CA rule rises dramatically and the use of efficient accelerators becomes crucial.

[1] A. Ilachinski, *Cellular Automata: a Discrete Universe*, World Scientific, 2001.
[2] W. Gosper, *Exploiting Regularities in Large Cellular Spaces*, Physica 10D, 1984.

3. 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 framework for real-time 2D CA simulations which helps users save precious time as we will demonstrate later in this report (Chapter 4, Design Reuse and Simulation Examples). The user can use the board's "up" and "down" push-buttons to control the simulation's speed.

This design is significantly improved and offers enhanced capabilities compared to our older competition entry, *Game of Complex Life - Modeling of Urban Growth Processes with Cellular Automata* (Xilinx Open Hardware 2015 Design Contest):

- Full-HD (1080p, 60 Hz) graphical projection of a 1920x1080 grid vs. HD (1280x800) graphical projection of a 640x400 grid.
- Fully pipelined – 60 FPS real-time simulation vs. $\approx$ 2 FPS simulation.
- Use of Nexys 4 DDR's external DDR2 memory vs. using only the FPGA's available Block RAM.
- More versatile customization.

Before generating the design, the user is free to choose any neighborhood size and one of the two available CA cell sizes: 4-bit (up to 16 states/cell) or 8-bit (up to 256 states/cell). The neighborhood size and the cell size in bits are the two key parameters that affect the design to be generated. Both parameters determine the number and size of the buffers used in the design, as well as the pipeline stages needed to meet the timing constraints.

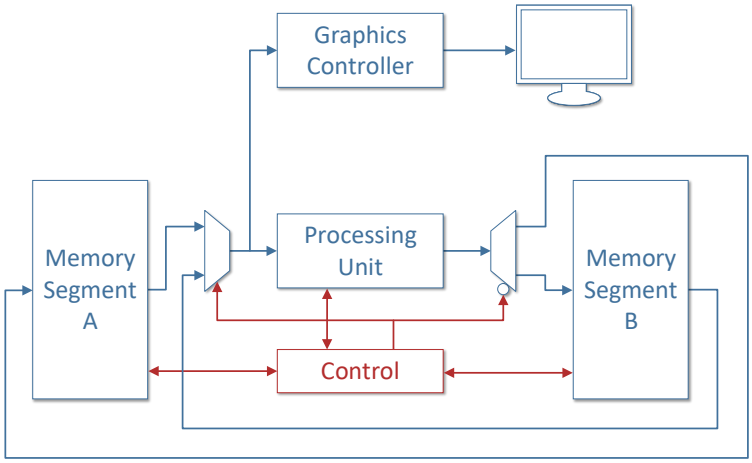

Figure 3.1: Double buffering.

In order to calculate a new CA state (aka the next timestamp) we need to have a complete timestamp of the previous CA's state. Thus, double buffering constitutes the best choice for us (Figure 3.1) - a completed timestamp of our CA's state is presented to the user while the same timestamp's data is processed by the CA Engine in order to produce the next timestamp.

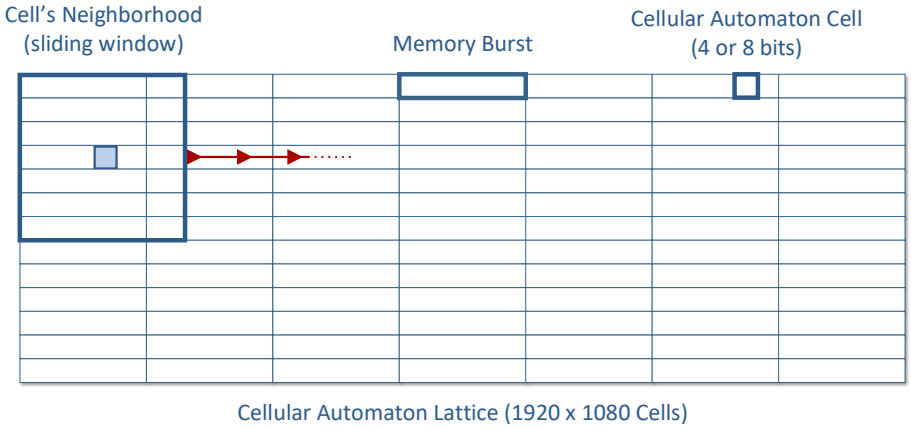

Figure 3.2: Grid Representation and sliding window swift.

A CA grid timestamp is represented in our system as shown in Figure 3.2: 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's size will vary measured in bursts/line as the burst size is fixed by the memory controller.

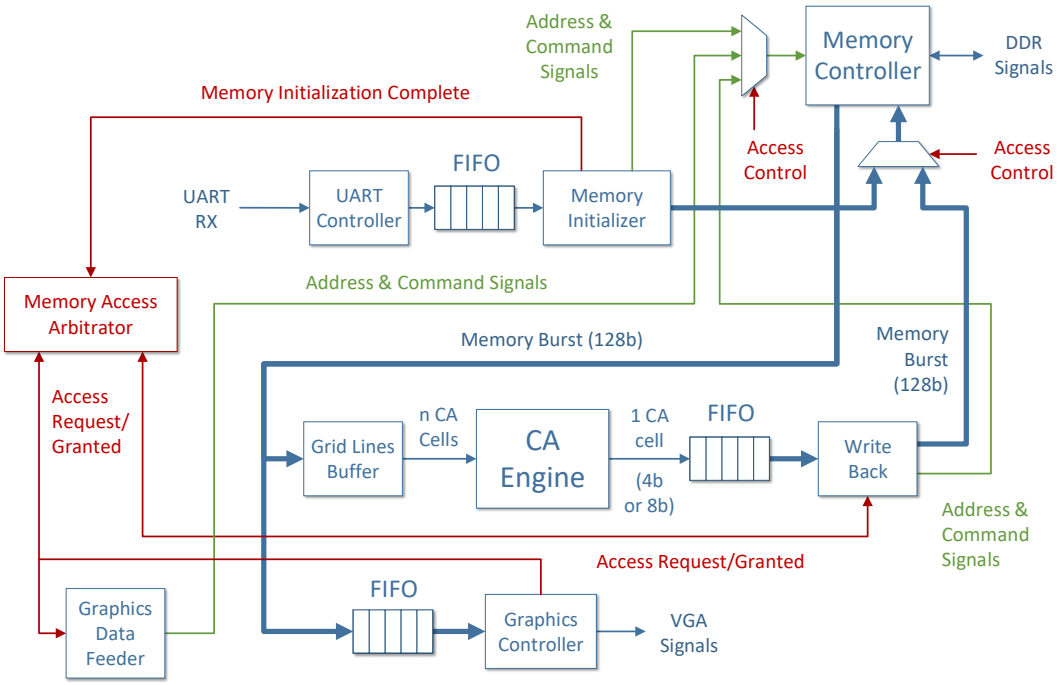

Figure 3.3: A simplified schematic of the system architecture.

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 3.3. At first 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 UART from a computer. The software needed for creating and transmitting 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's buffer following the controller's request, is also

loaded into the CA Engine's buffer. The CA Engine's 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 3.2), processing 1 cell / cycle.

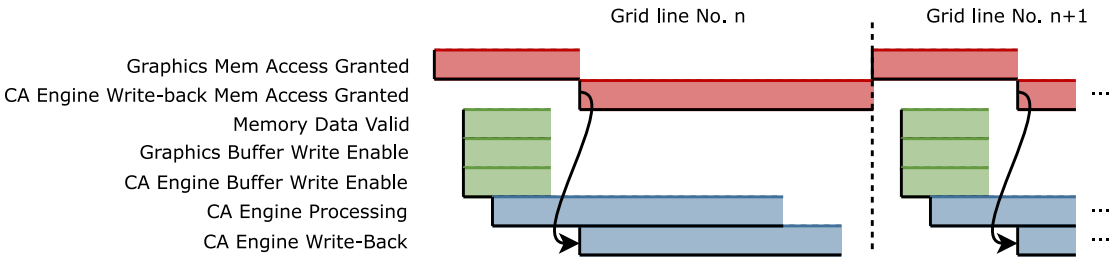

Figure 3.4: Timing diagram of the system's operation while processing 1 Line of the CA's grid.

Before the Graphics Controller requests for 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 (Figure 3.4). This is feasible because the CA Engine datapath is fully-pipelined and operates in a much higher frequency than the Graphics Controller. Combined with the fact that Xilinx's *Memory Interface* maximizes aggregate memory bandwidth, this results in a real-time CA simulation architecture with each graphics frame displaying a new CA generation.

The communication required between the system's modules is minimal, with each part of the architecture having its own independent control unit based on the number of bursts/line and the number of lines that have been processed so far. This results in a versatile system architecture which the user can customize without risking jeopardizing the system's synchronization integrity, in spite of the clock domain crossings required.

3.1 CA Engine

The CA Engine is responsible for calculating the new values of the CA grid's cells. This module was completely designed by the participant. It operates at 200 MHz, receives a new neighborhood column from the Grid Lines Buffer in every clock cycle and produces 1 cell/cycle.

This module is fully pipelined, with the pipelined neighborhood window providing all the data needed for calculating a new CA cell per clock cycle. The sum is calculated by a pipelined binary adder tree.

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.

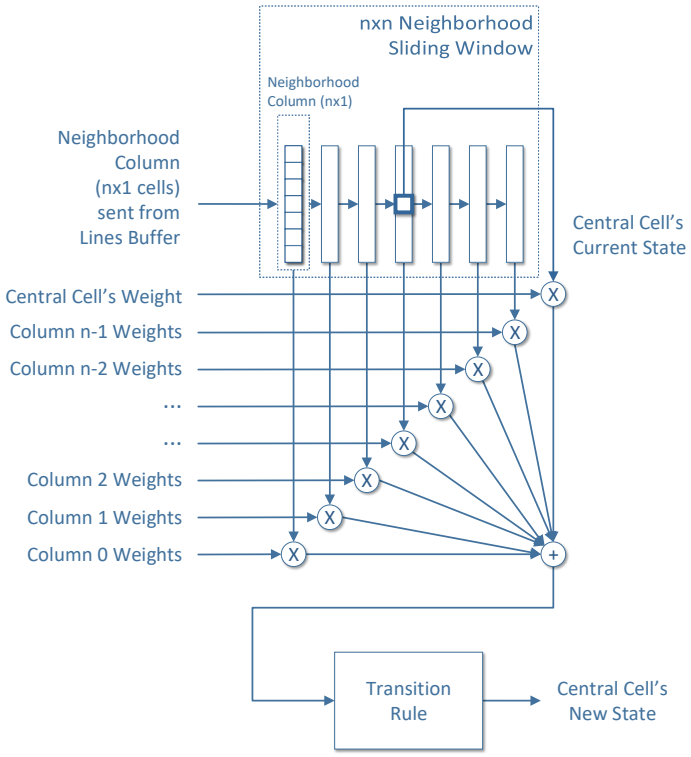

Figure 3.5: Cellular Automaton Processing Unit.

3.2 Grid Lines Buffer

The Grid Lines Buffer is responsible for providing the CA Engine with a new neighborhood column in every clock cycle. This module was completely designed by the participant, excluding the BRAM modules which were generated by *Xilinx Block Memory Generator (8.4)*.

The idea behind this module is based on N. Margolus's use of a corner-turning buffer responsible for feeding his proposed FPGA Architecture for systolic computations with data[1].

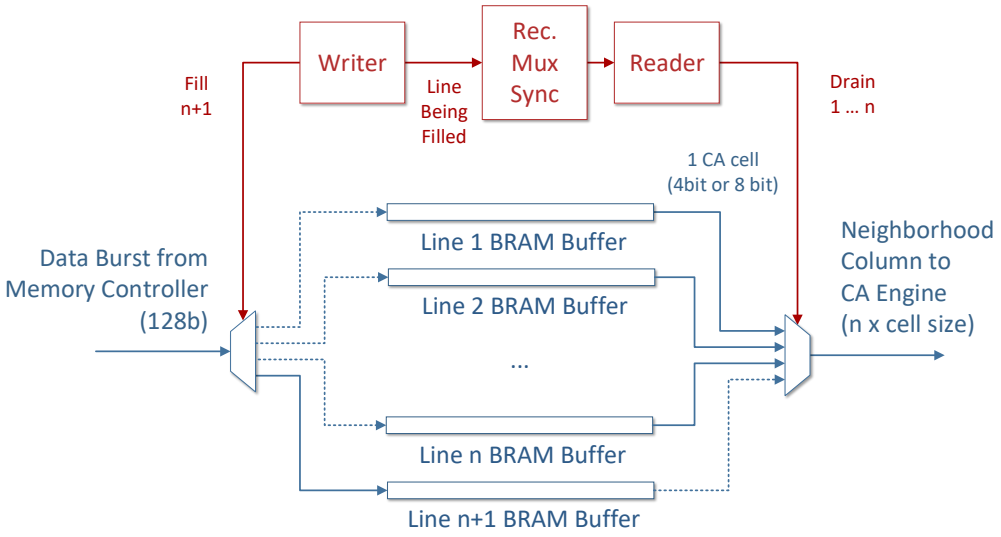

Figure 3.6: Grid Lines Buffer.

The Grid Lines Buffer (Figure 3.6) consists of n BRAM modules representing the lines in the CA rule's nxn neighborhood, plus one BRAM module used as a write buffer. While the Line Buffer's control logic drains the n BRAM modules in order to feed the CA Engine, it loads the BRAM module No. $n+1$ with the new line requested by the Graphics Controller in order to be displayed on the screen.

[1]N. Margolus, *An FPGA Architecture for DRAM-Based Systolic Computations*, Proceedings of the 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '97), 1997.

As soon as a complete grid line has been drained/filled, the Grid Line Buffer's control logic will "shift down" the buffer window and will be ready to drain the BRAM modules $2, ...n+1$ and fill the BRAM module No. 1.

This module's control logic consists of two parts, one responsible for filling the buffer and one responsible for draining it. The writer operates at 81.25 MHz, which is the Memory Controller's user interface clock's frequency, and the reader operates at 200 MHz, which is the CA Engine's clock frequency. The reader and the writer communicate with each other with the help of a recirculation multiplexer synchronizer.

The Grid Lines Buffer module does not require any intervention by the user. The number and size of the BRAM modules required and the depth of the pipeline are generated automatically based on the size of the CA rule's neighborhood and the size of the CA's cell in bits.

3.3 Graphics Controller

The Graphics Controller consists of two internal hardware components, FHD Sync Controller and FHD Color Controller.

FHD Sync Controller is based on Ulrich Zoltan's "VGA Controller" (Digilent, 2006). This module generates the video synch pulses for the monitor to enter 1920x1080@60hz resolution state (Full HD). It also provides horizontal and vertical counters for the currently displayed pixel.

FHD Color Controller was completely designed by the participant. It reads serially 4-bit or 8-bit data from a memory burst stored in its FIFO buffer, transforms it into a 12-bit color output signal, and transmits it according to FHD Sync Controller's counter signals.

The Graphics Controller collaborates with Graphics Data Feeder, a module designed by the participant. The Graphics Controller requests for a new frame line to be loaded into its FIFO buffer every time it is almost finished with displaying the current line of the frame. The Graphics Data feeder is nothing more than an address generator generating the addresses corresponding to a frame line's data stored in memory.

3.4 Memory Access Arbitrator

The Memory Access Arbitrator module is responsible for deciding whether the CA Engine's Write-back module or the Graphics Data Feeder will be allowed to control the memory bus for each bus cycle.

The Graphics Data Feeder has priority over the CA Engine, in order to ensure than the Graphics Controller will never fail to meet the screen's timing requirements. The memory access time offered to the CA Engine's write-back module is enough to preserve the real-time characteristics of the system. This would not be feasible without Xilinx's *Memory Interface* which maximizes aggregate memory bandwidth.

3.5 UART Controller

The UART controller used during the system's memory initialization phase is implemented by Jakub Cabal (github.com/jakubcabal/uart_for_fpga) and is used in this project as is.

The UART settings that are required by our system:

1. baud = 2000000
2. 8 data bits
3. 1 stop bit
4. No parity bit

The UART Controller fills a FIFO buffer which is drained by the Memory Initializer, a module designed by the participant. The Memory Initializer is nothing more than an address generator generating the memory addresses required in order to drain the UART's FIFO burst by burst.

4. 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 and the software needed to use our FPGA framework can be found in github.com/nkyparissas/XOHW18.

4.1 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 4.1). 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”!

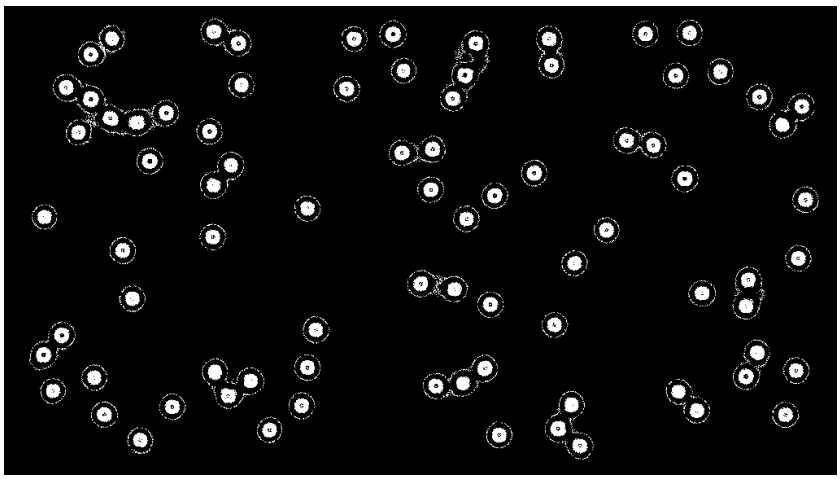

Figure 4.1: ”Artificial Physics” CA rule being simulated with our system.

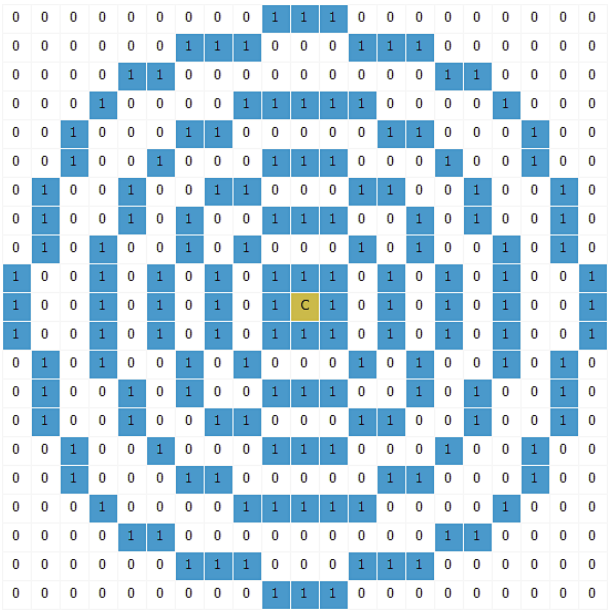

Figure 4.2: "Artificial Physics" rule neighborhood.

The rule requires a quite large $21x21$ neighborhood with binary weights as shown in Figure 4.2. The cell's state transition function is defined as:

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

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

As we mentioned earlier, the user chooses the neighborhood size and the CA cell's size in bits and the system is automatically generated based on these parameters. The only part the user needs to edit by hand is the CA Engine's sum and rule transition function. The implementation for "Artificial Physics" can be found in github and can be used as a template for weighted, large-neighborhood, totalistic and outer totalistic CA rules. The system's parameters used here were: $neighborhood\ size = 21$ and $cell\ size = 4\ bits$.

4.2 The Hodgepodge Machine

The second example is a CA rule known as the "Hodgepodge Machine": an outer totalistic CA rule with q states. Normally, the Hodgepodge Machine rules require a $3x3$ Moore neighborhood, but larger-neighborhood rules have been explored during the last decade.

For this example we used a $19x19$ neighborhood and a simplified version of the rule which calculates 1 sum for both ill ($cell\ state = q$) and infected cells ($0 < cell\ state < q$). The cell's state transition function is defined as:

$$c_t(i,j) = \begin{cases} \text{total number of infected and ill cells}/k & \text{if } c_t(i,j) = 0 \\ 0 & \text{if } c_t(i,j) = q \\ \text{total sum of neighborhood's cells}/l + g & \text{otherwise} \end{cases}$$

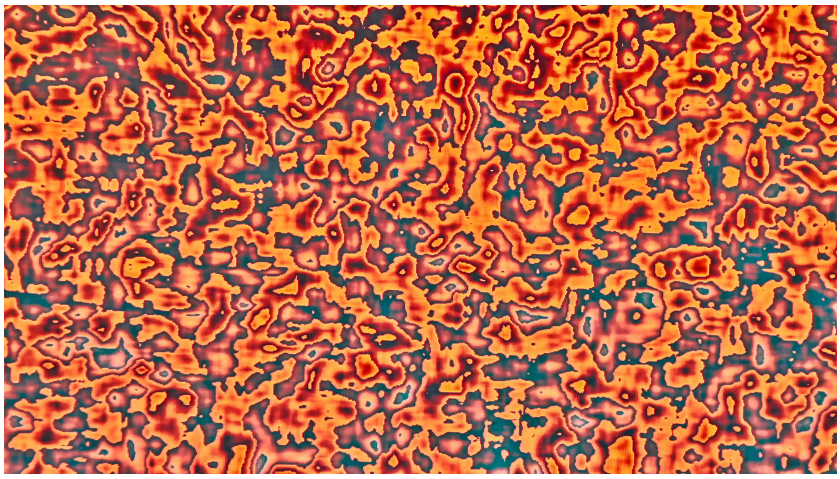

Figure 4.3: The "Hodgepodge Machine" CA rule being simulated with our system.

The implementation for the Hodgepodge Machine can be found in github and can be used as a template for CA rules that require more than 1 sums to be calculated for each cell state transition. The Hodgepodge Machine's parameters used were: $k = 64$, $l = 1024$, $g = 5$ and $q = 255$. The system's parameters used here were: $neighborhood\ size = 19$ and $cell\ size = 8\ bits$.

5. Results

In terms of results, our design calculates 60 CA generations / second regardless the rule running. The grid's size is 1920x1080 cells, which gives us 124416000 cells / second.

Our design successfully surpasses the computation and memory bounds found in a general purpose CPU and has a measured speedup of up to 46x against an Intel Core i7-6500 CPU (1 core) running highly optimized (-O3) software programmed in C as shown below.

CA Rule	**i7-6500, 1000 generations**	**Our Design, 1000 generations**	**Our Design's Speedup**
Artificial Physics, n = 21	657.69 sec	16.67 sec	39.45x
The Hodgepodge Machine, n = 19	769.83 sec	16.67 sec	46.18x

Table 5.1: Comparative results: execution time and speedup.

The amount of FPGA resources utilized for each rule differs and depends mainly on the CA rule's neighborhood size. As the size of the neighborhood grows, the number and size of the BRAM modules rise as well. Some indicative results can be seen in Table 5.2 after the implementation of the Hodgepodge Machine CA rule.

Resource	Utilization	Utilization %
LUT	10811	17.05
LUTRAM	1099	5.78
BRAM	45	33.33
DSP	1	0.42
IO	70	33.33
BUFG	7	21.88
MMCM	3	50
PLL	1	16.67

Table 5.2: Resource utilization.

6. Conclusion

Our FPGA framework for real-time CA simulations helps the user save precious time by offering a significant speedup compared to a high-end, general-purpose CPU.

The design is automatically generated according to the user's needs and is fully customizable.

The relatively low resource utilization leaves plenty of room for further exploration of the design's maximum range of capabilities.

5. Bibliography in Alphabetical Order

1. F. Berto and J. Tagliabue, *Cellular Automata*, Stanford Encyclopedia of Philosophy, 2017.

2. A. W. Burks, *Essays on Cellular Automata*, University of Illinois Press, 1971.

3. A. K. Dewdney, *Computer Recreations: The Hodgepodge Machine Makes Waves*, Scientific American, Vol. 43, 1988.

4. J. Drieseberg and C. Siemers, *C to Cellular Automata and Execution on CPU, GPU and FPGA*, Proceedings of the International Conference on High Performance Computing and Simulation (HPCS '12), 2012.

5. W. Gosper, *Exploiting Regularities in Large Cellular Spaces*, Physica 10D, 1984.

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

7. K. Ishimura, K. Komuro, A. Schmid, T. Asai and M. Motomura, *FPGA Implementation of Hardware-Oriented Reaction-Diffusion Cellular Automata Models*, Nonlinear Theory and Its Applications, Vol. 6, No. 2, IEICE, 2015.

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

9. N. Margolus, *An FPGA Architecture for DRAM-Based Systolic Computations*, Proceedings of the 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '97), 1997.

10. N. Margolus, *An Embedded DRAM Architecture for Large-Scale Spatial-Lattice Computations*, Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00), 2000.

11. C. Maxfield, *The Design Warrior's Guide to FPGAs*, Newnes – Elsevier, 2004.

12. S. Murtaza, A. G. Hoekstra, and P. M. A. Sloot, *Performance Modeling of 2D Cellular Automata on FPGA*, Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '07), 2007.

13. S. Murtaza, A. G. Hoekstra, and P. M. A. Sloot, *Compute Bound and I/O Bound Cellular Automata Simulations on FPGA Logic*, ACM Transactions on Reconfigurable Technology and Systems, Vol. 1, No. 4, ACM, 2009.

14. S. Murtaza, A. G. Hoekstra, and P. M. A. Sloot, *Cellular Automata Simulations on a FPGA cluster*, International Journal of High Performance Computing Applications, Vol. 25, Issue. 2, SAGE, 2010.

15. J. von Neumann, *The General and Logical Theory of Automata*, Cerebral Mechanisms in Behavior, The Hixon Symposium, John Wiley & Sons, New York, 1951.

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

17. T. Toffoli and N. Margolus, *Cellular Automata Machines – A New Environment for Modeling*, MIT Press, 1987.