Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200101

A Processor Architecture for Executing Global Cellular Automata as Software

Christian RISTIG[a,1] and Christian SIEMERS[a]
[a] *Department of Informatics, Clausthal University of Technology, Germany*

Abstract. Cellular automata are a massively parallel programming model that are capable to solve many algorithmic problems efficiently. The complexity of defining a suitable cell rule for a concrete problem can be overcome by the use of the extended model of global cellular automata in conjunction with specialized compilers, to translate a high-level imperative programming language to cellular automata. Obviously, the execution on universal multicore processors does not make use of the full parallel potential of cellular automata and the workflow for direct hardware implementations is slow and hard to debug. In this paper, we propose a novel processor architecture that can execute a global cellular automaton as software and can still compete with other software or hardware implementations.

Keywords. FPGA, Cellular Automata, Processor Architecture, Parallel Processing, Dataflow

1. Introduction

Cellular automata (CA) are a massively parallel model that can easily be implemented in hardware [1]. There exist several application fields for cellular automata, e.g. image processing, machine learning, fluid dynamic or traffic simulation [2]. On the other side, writing a program for a cellular automaton that can solve a given problem is extremely challenging, as a cell has a strict and homogenous neighborhood and there is only one rule for all cells. Mortensen [3] presented a method to compile a high-level imperative programming language to the cellular automata model, so developers can write their algorithms in the usual way they do, and although benefit from the parallel execution. Unfortunately, the resulting automaton is somehow inefficient as information has to be passed from one cell to another over long distances, which is done by a message passing protocol.

To overcome the restrictions of a cellular automaton, the so-called global cellular automata (GCA) have been introduced [4] [5].

Definition. For a given natural number k, a global cellular automaton (GCA) is a 4-tuple (Z, Q, ν, δ) consisting of

- the set of cells Z,
- the neighborhood function ν: $Z \times Q \rightarrow Z^k$,
- the set of cell states Q,
- the state transition function δ: $Z \times Q \times Q^k \rightarrow Q$.

[1] Corresponding Author: Christian Ristig, Department of Informatics, Clausthal University of Technology, Julius-Albert-Str. 4, 38678 Clausthal-Zellerfeld, Germany. E-mail:christian.ristig@tu-clausthal.de

In contrast to a cellular automaton, the neighborhood of a cell does not only depend on the cell itself, but also of its current state. Thus, a cell might have a totally different neighborhood from one generation to another. The number of neighbors is denoted with k and might be constant or variable as well. Furthermore, the definition of the state transition function δ allows the use of individual rules for each cell. While it is easier to implement algorithms in software, an implementation in hardware is more complex because of the varying neighborhood. Several approaches have been presented in [6] [7] [8]. Common to all, there is always a compromise between generality and runtime. Specialized approaches are fast, but running a different algorithm is a complex process. In contrast, more general approaches have a great overhead in space and time.

Drieseberg et al. [9] use global cellular automata and combine them with the approach of Mortensen. They presented a compiler for a subset of the C programming language. It generates a GCA that can be run on a multicore CPU, a GPU or an FPGA. Experimental results show, that a speedup could be achieved compared to a program that was compiled with a standard C compiler and executed on an ordinary CPU. This was especially true for the implementation on an FPGA, which could achieve the highest speedup factor compared to the runtime of the same algorithm on a CPU. However, the development cycle includes the synthesis and place & route process of the hardware description and is very time-consuming. Additionally, the presented compiler creates a huge number of cells, already for small algorithms with a few lines of code. This results in a very slow or even impossible routing [10].

In this paper, we present a hardware architecture that executes global cellular automata written as software. The architecture is generalized and can execute a high number of cells. Nevertheless, it aims to reach high speedup factors compared to the execution of cellular automata on universal hardware. We also propose a mapping process of standard syntax elements of imperative programming languages (such as operators, if and while) to our cellular processor architecture, which results in much fewer cells than the approach of Drieseberg.

2. Hardware Architecture for Execution of Global Cellular Automata

In this section, we present a hardware architecture for a processor that is able to calculate a global cellular automaton $GCA = (Z, Q, v, \delta)$ where

- $Z \subset \mathbb{N}$ is a finite subset of the natural numbers (called the *cell IDs*),
- v is the neighborhood function with $k = 0$ or $k = 1$ depending on the local cell state,
- $Q = \{(pc, a, v, s) | pc \in \mathbb{N}; a, v \in \mathbb{Z}; s \in \mathcal{S}\}$ is the set of all possible cell states,
- δ is the state transition function (called the *rules*).

An element of Q is a 4-tuple where pc is the program counter, a is the accumulator, v the cell value and s the local cell state. The set $\mathcal{S} := \mathcal{I} \cup \mathcal{V}$ is composed of the subset of invalid states $\mathcal{I}$, containing the states *Busy*, *LikelyTrue*, *LikelyFalse* and *Reset*, and of the subset of valid states $\mathcal{V}$, containing the states *Ready*, *True*, *False* and *LoopReset*.

In a global cellular automaton, the neighborhood of a cell depends not only of the cell ID, but also of its state. More precisely, it depends on the value of the program counter, which might change in every new generation. The definition of the GCA above

implies that a cell has either one neighbor cell at a given time or no neighbor at all. We agree that if a cell has a neighboring cell, it might use its cell value v or its local cell state s, but not its program counter and accumulator to calculate the cell's new cell state in Q.

The number of cells in Z and the state transition function δ highly depends on the function the cellular automaton computes. For a hardware architecture, we have to limit the number of cells and to define a set of general rules the designer of the function can choose from.

2.1. Ruleset

The ruleset can be divided into four categories: initialization, arithmetic/logic, comparison and control. Every rule may be annotated with a valid flag that has two effects when the rule actually changes the cell's program counter. First, the new value of the accumulator is copied to the cell's value v. Second, if the cells local state s is in $\mathcal{I}$, it is changed to a corresponding state in $\mathcal{V}$ according to Table 1. The local states *Reset* and *LoopReset* have a special meaning and are not affected by the valid flag.

Table 1. Local cell state changes when valid flag is set in a rule

Old State	New State
Busy, Ready	Ready
LikelyTrue, True	True
LikelyFalse, False	False

Initialization rules are used to set a definite value in the cell's accumulator. This value might be a constant (*set* rule) or the cell value $\bar{v}$ of a neighboring cell if its local state $\bar{s}$ is valid (*read* rule). There are also variants of the set and read rules which only set the accumulator when the condition $s \notin \mathcal{V}$ is met (*init* rule), or increase the cells program counter by two instead of one (*skip* rule). A complete list of the initialization rules and their impacts is shown in Table 2. They do not change the cell value or the local cell state if the rule is not annotated with a valid flag.

Table 2. List of initialization rules

Rule	Program Counter	Accumulator
Set	$pc = pc + 1$	$a = c$
Set and Skip	$pc = pc + 2$	$a = c$
Set Init	$pc = pc + 1$	$a = \begin{cases} c, & \mathrm{s} \notin \mathcal{V} \\ a, & \text{else} \end{cases}$
Read	$pc = \begin{cases} pc+1, & \bar{s} \in \mathcal{V} \\ pc, & else \end{cases}$	$a = \begin{cases} \bar{v}, & \bar{s} \in \mathcal{V} \\ a, & else \end{cases}$
Read and Skip	$pc = \begin{cases} pc+2, & \bar{s} \in \mathcal{V} \\ pc, & \text{else} \end{cases}$	$a = \begin{cases} \bar{v}, & \bar{s} \in \mathcal{V} \\ a, & \text{else} \end{cases}$
Read Init	$pc = \begin{cases} pc+1, & \mathrm{s} \in \mathcal{V} \vee \bar{s} \in \mathcal{V} \\ pc, & \text{else} \end{cases}$	$a = \begin{cases} \bar{v}, & \mathrm{s} \notin \mathcal{V} \\ a, & \text{else} \end{cases}$

Arithmetic/logic rules implement unary operators (denoted with $\ominus$) such as negation or binary operators (denoted with $\oplus$) such as addition or subtraction. The first operand is always the accumulator of the cell, the second is either a constant or the value of a neighboring cell. Like initialization rules, they have no impact on the cell value or the local cell state if a valid flag is not present, but all the other effects are presented in Table 3.

Table 3. List of arithmetic/logic rules

Rule	Program Counter	Accumulator
Unary operator	$pc = pc + 1$	$a = \ominus a$
Binary operator (constant)	$pc = pc + 1$	$a = a \oplus c$
Binary operator (neighbor cell)	$pc = \begin{cases} pc+1, & \bar{s} \in \mathcal{V} \\ pc, & else \end{cases}$	$a = \begin{cases} a \oplus \bar{v}, & \bar{s} \in \mathcal{V} \\ a, & else \end{cases}$

Comparison rules compare the accumulator with a constant or the value of another cell, and write the result into the accumulator as stated in Table 4. Afterwards, they also change the local cell state as defined in Eq. 1.

$$s = \begin{cases} LikelyFalse, & a = 0 \wedge s \in \mathcal{I} \\ False, & a = 0 \wedge s \in \mathcal{V} \\ LikelyTrue, & a \neq 0 \wedge s \in \mathcal{I} \\ True, & a \neq 0 \wedge s \in \mathcal{V} \end{cases} \tag{1}$$

Table 4. List of Comparison rules

Rule	Program Counter	Accumulator
Comparison with constant	$pc = pc + 1$	$a = \begin{cases} 0, & a \not\preccurlyeq c \\ 1, & a \preccurlyeq c \end{cases}$
Comparison with neighbor	$pc = \begin{cases} pc+1, & \bar{s} \in \mathcal{V} \\ pc, & else \end{cases}$	$a = \begin{cases} 0, & a \not\preccurlyeq \bar{v} \wedge \bar{s} \in \mathcal{V} \\ 1, & a \preccurlyeq \bar{v} \wedge \bar{s} \in \mathcal{V} \} \\ a, & \text{else} \end{cases}$

Control rules do not change the cells accumulator, but the program counter as well as the local cell state. The available rules are listed in Table 5 and can be divided into three groups:

- Rules that wait for a neighboring cell to take over a specific local state $\hat{s} \in \mathcal{V}$ (wait rule)
- Rules that increase the program counter by two instead of one when the own local state or the local cell state of a neighbor is *True* (skip rule)
- Rules for halting or resetting a cell

Table 5. List of control rules

Rule	Program Counter	Local Cell State
Wait rule	$pc = \begin{cases} pc+1, & \bar{s} = \hat{s}, \hat{s} \in \mathcal{V} \\ pc, & else \end{cases}$	No change
Skip rule (local)	$pc = \begin{cases} pc+2, & s \in \{LikelyTrue, True\} \\ pc+1, & sonst \end{cases}$	No change
Skip rule (neighbor)	$pc = \begin{cases} pc+2, & \bar{s} = True \\ pc+1, & \bar{s} \in \mathcal{V} \setminus \{True\} \\ pc, & else \end{cases}$	No change
Reset rule (local)	$pc = \begin{cases} 0, & s = Reset \\ pc, & else \end{cases}$	$s = \begin{cases} Busy, & s = Reset \\ Reset, & else \end{cases}$
Reset rule (neighbor)	$pc = \begin{cases} 0, & s = Reset \\ pc, & else \end{cases}$	$s = \begin{cases} Reset, & \bar{s} = Reset \\ Busy, & s = Reset \\ s, & else \end{cases}$
Loop (reset) rule	$pc = \begin{cases} 0, & \bar{s} = Reset \\ pc, & else \end{cases}$	$s = \begin{cases} Ready, & \bar{s} = Reset \\ LoopReset, & else \end{cases}$
Halt rule	No change	No change

2.2. Processor Architecture

The processor is built of several so-called *cell compute units* (CCUs) and an on-chip-network for interconnection, as presented in Figure 1. There is also a *control unit* (CU) connected to the network that has access to an external main memory. The CU is responsible for loading the individual cell rules from main memory into the CCUs before the automaton can be run. It also monitors the state of the automaton and signals the completion of work when all cells become inactive, thus there are no more state changes from one generation to another.

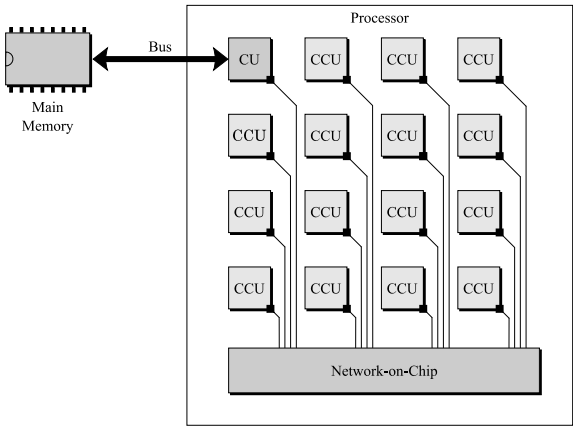

Figure 1. Overall processor architecture

A cell compute unit is designed for minimal hardware consumption. It uses a 1-operand-machine architecture, as presented in Figure 2, which is sufficient to execute a cell rule, because a rule has at most one operand that might be a constant (also: *immediate*) value or the value of a neighboring cell. Each CCU has four registers to store

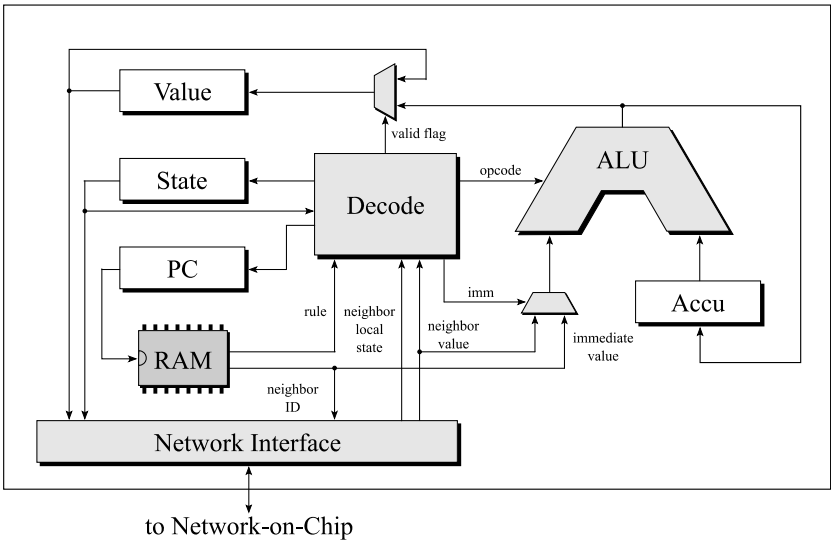

Figure 2. CCU architecture in details

the value of the cell's state tuple (pc, a, v, s). The program counter addresses a small memory belonging to the cell which is capable to hold a couple of rules and their operands. If a cell has a neighbor in the current generation, the ID of the neighbor is routed to the network-on-chip to request its local state and value. The cell's own local state and value are always provided to the network as well. A decoder unit decodes the rule and generates following control signals:

- an opcode for an arithmetic/logical unit (ALU)
- a signal to select the correct operand (imm)
- a signal to take over the ALU result as new cell value (valid flag)
- the next program counter
- the new local cell state

2.3. Mapping imperative programming languages to GCA ruleset

As already mentioned, writing a program for a cellular automaton is different than writing it in a programming language like C. Therefore, it is desirable to have a mapping algorithm from an imperative programming language to the ruleset of the presented hardware architecture. Based on the ideas of Drieseberg and Mortensen we propose the following procedure:

1. Transform the imperative program into static single assignment (SSA) form (as described in [11])
2. Build a dataflow graph of the transformed program (see example in Figure 3)
3. Replace each node in the dataflow graph with a cell and create the rules for each created cell
4. Optimize the resulting *cell graph* to reduce the number of cells

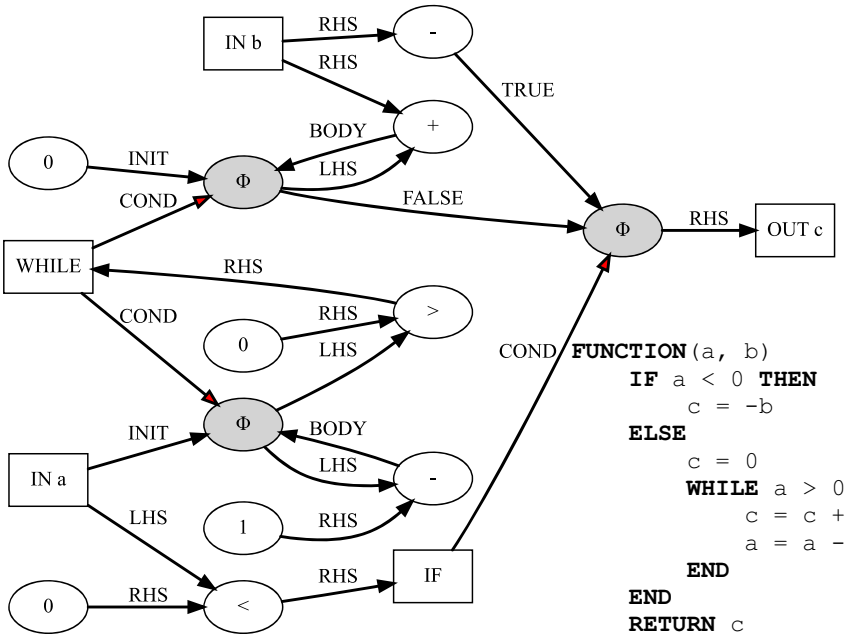

Figure 3. Example of a dataflow graph and the corresponding program in pseudocode

Step 1 includes several sub-procedures, like parsing the source code or building the control flow graph and (abstract) syntax trees, before the SSA form can be generated. Step 2 is very simple once the program is in SSA form. Additional optimizations (for instance removing unused variables, balancing expressions, etc.) could take place before. An example dataflow graph is presented in Figure 3. It consists of nine types of nodes that are generated from the keywords and literals in the original program code. The edges of the graph show the actual flow of data, beginning from the input nodes (IN) to the output nodes (OUT). They are annotated with additional information, so that the target node can handle the incoming data correctly. For example, a binary operator node needs to know the order of its operand, whereat LHS (left-hand side) denotes the left operand and RHS (right-hand side) the right one. Yet another example is the phi-nodes that belong to the while-node: they require the loop condition (COND) to decide if they have to choose the initial value, that is valid *before* the loop body is executed (INIT), or the value computed *in* the loop body (BODY). In step 3, the nodes of the dataflow graph are replaced with cells according to Figure 4. Every generated cell is assigned an ID that is used by other cells in their cell rules to access the cell's state. The cell rules are abbreviated with a self-explanatory mnemonic. Mnemonics written in bold font have the

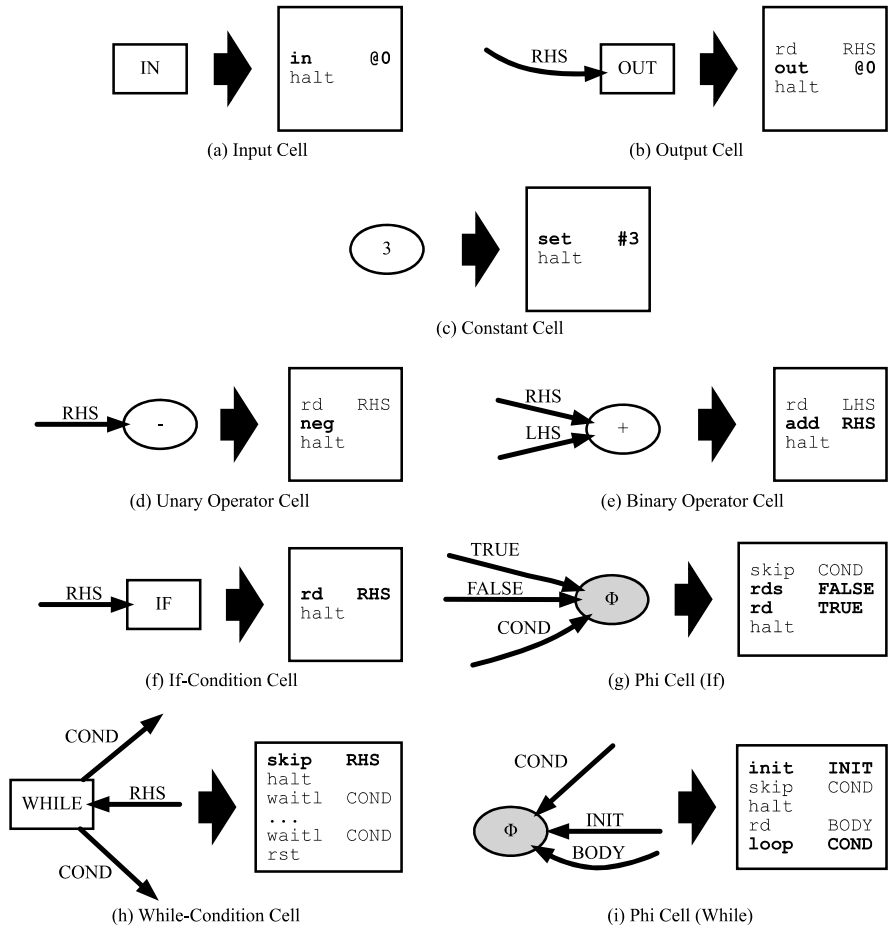

Figure 4. Translation of dataflow graph nodes to cells

valid flag set. Input and output cells have a special @i-operand that denotes to the i-th argument or return value, respectively. Numbers with a leading # symbol denote a literal value; without it they represent the ID of another cell. In some situations, an additional wait-rule must be prepended to a cell. This is when it wants to read from a phi-cell of a while loop outside of the loop body. Because the value of such a phi-cell is always valid during the execution of the loop, the cell outside must wait for the loop condition to be false (see Figure 5). Step 4 tries to eliminate cells by inserting their rules into another one. For example, literal nodes are easy to integrate into another cell, just by replacing a read access with a literal value. In Figure 5, the optimization process has been executed for the illustrated algorithm.

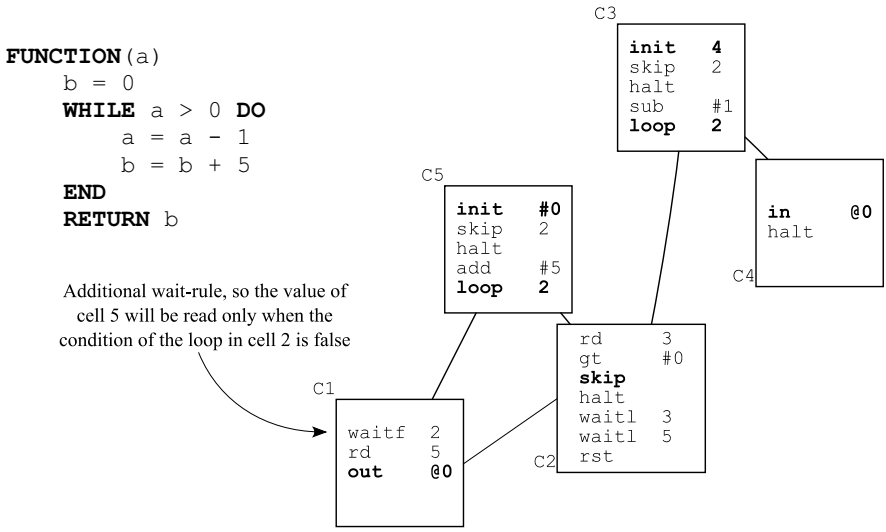

Figure 5. Optimized cell graph for an algorithm

3. Results

The presented processor architecture has been implemented in VHDL and synthesized with the Xilinx Vivado toolchain. A single cell compute unit with a 32-bit word size, including a multiplier unit using DSP slices, occupies around 480 lookup tables and 100 flip-flops. In a high-end 7-series Virtex FPGA with 2 million logic cells (type XC7V2000T) this corresponds to only 0.04% of logic consumed, so hundreds of CCUs can be integrated into a single FPGA. The local cell memory is implemented as LUT-RAM and uses 40 of the total 480 lookup tables (32 for the operand and 8 for the encoded rule). Depending on the FPGA technology, the possible number of rules per CCU varies. For a 6-input LUT this means 64 rules per cell.

For our experiments, we used a crossbar as network-on-chip. Although this network architecture uses a lot of FPGA logic (large multiplexer structures are needed for each single data bit), it has a low latency and is easy to implement. There exist lots more network architectures that might be suitable for the processor architecture (a good survey can be found in [12]), especially the so-called Benes-network is an appropriate candidate [13], as it is collision-free and resource-efficient. The complex routing algorithm can also be implemented in hardware [14].

The first algorithm we analyzed consist of a matrix multiplication of two 3x3 matrices. This operation has a high degree of parallelism as all 9 elements of the resulting matrix can be computed concurrently. Even the three multiplications needed to compute an element can run in parallel. The corresponding global cellular automaton has a total of 38 cells and was executed on a system with a maximum of 63 cells. Simulation shows that it took only five generations to compute the result, as illustrated in Figure 6. Cells highlighted with a red border did change their state in the current generation, and yellow cells are in a valid local state.

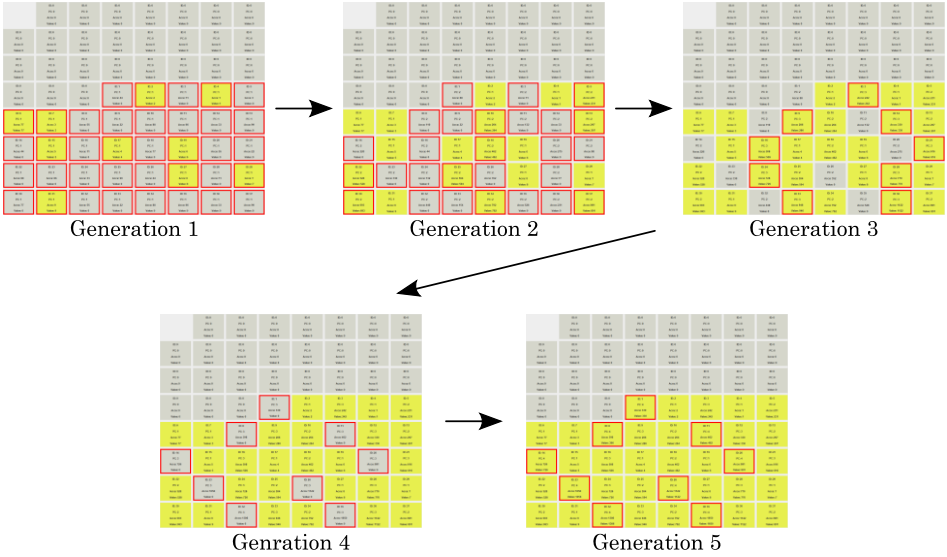

Figure 6. Execution of a matrix multiplication

Another algorithm we implemented was Stein's algorithm [15], the binary version of the *greatest common divisor (GCD)*, and compared the results with the FPGA implementation of Drieseberg et al. Their GCD global cellular automaton consists of 258 cells, whereas our optimized automaton consists of only 24 cells, which is less than 10%. We simulated the execution of the algorithm with the two input values 3528 and 3780. It took 217 generations to calculate the result of 252. This is also less, and only a third, of the number of generations compared to Drieseberg et al. It must be pointed out, that the number of generations is only an indicator to the actual execution time, as our simulation does not consider any hardware properties. Thus, the achievable clock frequency of the proposed processor architecture might be much slower due to propagation delays. Further research has to be done here. If a generation needs one clock cycle to compute and we assume a clock frequency of only 50 MHz, our automaton would need around 4 ms to calculate the GCD. Indeed, this is much slower than the FPGA implementation of Drieseberg et al. but still faster than the CPU and GPU variants.

4. Conclusion

In this paper, we presented a new processor architecture that is able to execute global cellular automata with a specialized ruleset as a software program. The ruleset is

designed in such a way that algorithms written in an imperative programming language can easily be mapped to a global cellular automaton with a low number of cells needed. The automation of this process is one of our next steps. In first experiments, we confirmed that the presented architecture can achieve faster execution times than a software implementation on a universal processor. This advantage outweighs even more, if the executed algorithm has a high degree of parallelism. Nevertheless, further research has to be done with real-world applications to fully prove the advantages of the architecture. Furthermore, the processor has to be implemented and evaluated on an FPGA to analyze the scalability of the architecture and to obtain actual execution times.

References

[1] M. Dascalu, "Cellular Automata Hardware Implementations - an Overview," *Romanian Journal of Information, Science and Technology,* 2016.

[2] A. C. Lima and J. C. Ferreira, "Automatic Generation of Cellular Automata on FPGA," *IX Jornadas sobre Sistemas Reconfiguráveis,* Februar 2013.

[3] M. Mortensen, High Level Parallel Programming Language Compiling to a Cellular Automata Processing Model, Aarhus: Master's thesis, Aarhus Universitet, 2007.

[4] R. Hoffmann, K. P. Völkmann and S. Waldschmidt, "Global Cellular Automata GCA: An Universal Extension of the CA Model," *ACRI 2000 "work in progress" session, Karlsruhe, Germany, Oct. 4th - 6th,* 2000.

[5] R. Hoffmann, K. P. Völkmann, S. Waldschmidt and W. Heenes, "GCA: Global Cellular Automata. A Flexible Parallel Model," *Parallel Computing Technologies, 6th International Conference, PaCT 2001, Novosibirsk, Russia, September 3-7, 2001, Proceedings,* 2001.

[6] R. Hoffmann, W. Heenes and M. Halbach, "Implementation of the Massively Parallel Model GCA," *Parallel Computing in Electrical Engineering,* 2004.

[7] W. Heenes, R. Hoffmann and J. Jendrsczok, "A Multiprocessor Architecture for the Massively Parallel Model GCA," *Parallel and Distributed Processing Symposium,* 2006.

[8] C. Schäck, W. Heenes and R. Hoffmann, "A Multiprocessor Architecture with an Omega Network for the Massively Parallel Model GCA," in *Lecture Notes in Computer Science (LNCS, volume 5657)*, Berlin, Heidelberg, Springer, 2009.

[9] J. Drieseberg and C. Siemers, "C to Cellular Automata and execution on CPU, GPU and FPGA," *International Conference on High Performance Computing & Simulation (HPCS),* pp. 216-222, 2012.

[10] J. Drieseberg, Abbildung sequentieller C-Programme auf parallel arbeitende Zellularautomaten, Clausthal-Zellerfeld: Dissertation, Technische Universität Clausthal, 2016.

[11] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," in *ACM Transactions on Programming Languages and Systems, Vol 13, No 4*, New York, 1991.

[12] T. Schwederski and M. Jurczyk, Verbindungsnetze: Strukturen und Eigenschaften, Stuttgart: Teubner, 1996.

[13] S. C. Stilkerich, C. Siemers and C. Ristig, "Appropriate Multi-core Architecture for Safety-critical Aerospace Applications - Certifiable Real-time Switching Network," *Proceedings of the 4th International Conference on Pervasive and Embedded Computing and Communication Systems - Volume 1: PECCS,* pp. 180-185, 2014.

[14] S. Aust, ConPar - Ein Echtzeitparallelrechner zur Rezentralisierung von Steuergeräten im Automobil, Clausthal-Zellefeld: Institut für Informatik TU Clausthal, 2013.

[15] J. Stein, "Computational problems associated with Racah algebra," *Journal of Computational Physics, Volume 1, Issue 3,* pp. 397-405, 1967.