

# FPGA Implementation of Cellular Automata Compared to Software Implementation

Mathias Halbach, Rolf Hoffmann, Patrick Röder

TU Darmstadt, FB Informatik, FG Rechnerarchitektur

Alexanderstraße 10

D-64283 Darmstadt

Phone +49 6151 16 3713, 3606

Fax +49 6151 16 5410

halbach@informatik.tu-darmstadt.de

hoffmann@informatik.tu-darmstadt.de

patrick.roeder@web.de

**Abstract:** In order to optimize applications in the Cellular Automata model we have searched for a performant platform to run billions of simulations. The question was how much speed-up could be gained by using the FPGA technology compared to optimized software. As an example we implemented two cellular automata rules in software on a PC and also in FPGA logic. On our low end experimental platform we reached a speed-up of 19 for a medium complex rule and 14 for a complex rule. If we would use the latest high end FPGA technology, speed-ups up to many thousand are realistic. A cluster of thousands of workstations would be necessary to reach the same performance which is much more costly than a FPGA solution.

## 1 Motivation

The Cellular Automata (CA) dates back to John von Neumann [vN66] and Konrad Zuse [Zu69]. It is a very elegant computing model which can be applied to many real problems in physics, chemistry, or biology and also to computational or artificial problems. Such problems are described by a field of cells and a local rule. The new cell state is defined by the rule which takes all the states of its neighbours into account. All cells can work in parallel because each cell can independently update its own state. Therefore the model is massively parallel and is an ideal candidate to be implemented in hardware.

We have investigated the question how the CA model can efficiently be implemented in hardware and how much performance can be gained out of a hardware solution compared to a software solution.

In the past we have developed a series of CEPRA-machines (Cellular Processing Architectures) which are all based on programmable FPGA technology. The CEPRA-8L [HVS94] used 8 FPGAs or 8 DSPs for the 8-fold parallel computation of the rule. The CEPRA-1X [HHVS97] [HHVW00] is a PCI plug-in card, which works in a stream like mode. The lines of the cell array are held in the host and are streamed line by line to a line buffer, consisting of 3 lines. The line buffer feeds a window of 3 by 3 cells which is taken as parallel input to the FPGA computation. The CEPRA-S [HUVW00] [HHV01] consists of two FPGAs. The first FPGA is dedicated for parallel computation, 9 data memories are connected to it. The second FPGA interprets instructions which are stored in a control memory and controls the data path by generating addresses for the data RAMs. The programming of the FPGAs is supported by the language CDL [Ho98], which allows describing cellular rules in a concise

notation. CDL can be compiled into FPGA logic design files.

At the moment we work also with the Altera FLEX-10k evaluation board. It allows rapid prototyping of new ideas in parallel architectures to reduce the cost and time for the development of special printed circuits. Our results presented in this paper are based on prototypes implemented on this board.

In the following we will compare the efficiency of software implementations to prototyped FPGA implementations.

The first rule is called the WireWorld [De90] (in the following RuleWW), which allows to describe digital circuits.

The description of this rule in a pseudo-code notation is

```
function RuleWW (NE, N, NW, E, C, W, SE, S, SW) ;
const wire=3;
    head=2;
    tail=1;
    empty=0;
var sum;
begin
    if C=empty then RuleWW:=empty
    else if C=wire then
        begin
            sum:= (N=head) + (NE=head) + (E=head) + (SE=head) +
                (S=head) + (SW=head) + (W=head) + (NW=head) ;
            if (sum=1) or (sum=2) then RuleWW:= head
                else RuleWW:=wire;
        end
    else if C=tail then RuleWW:=wire
    else if C=head then RuleWW:=tail;
end;
```

The default data type is unsigned integer with range 0..3. The background without wires is the empty state. Electrons can move along wires. A moving electron consists of a head and a tail. An example circuit (full adder) is shown in fig. 1.

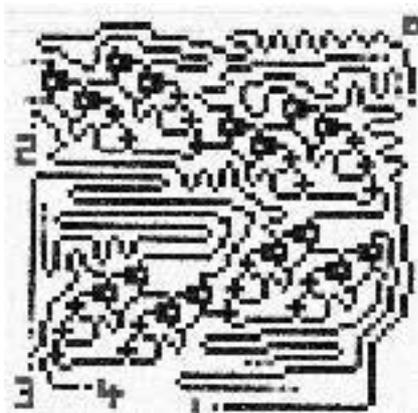


Figure 1: Wire World 4 bit full adder

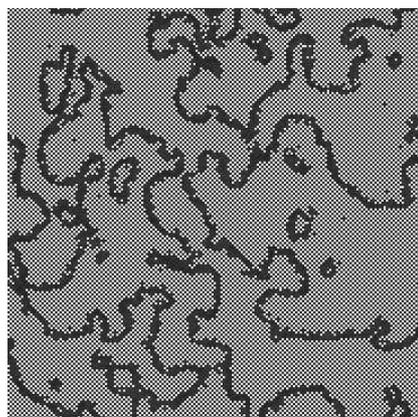


Figure 2: Rule Coast Line

The WireWorld rule is neither very simple nor very complex. In order to find out how a more complex rule behaves, we have defined a second rule, called RuleCoast, which generates coast line structures (fig. 2):

```
function RuleCoast (Cell, North, East, South, West);
// Rule Coast Line
const s1=40, s2=215;
A:= max (Cell, North, East, South, West);
B:= min (Cell, North, East, South, West);
small:= (Cell<s1) + (North<s1) + (East<s1)
        + (South<s1) + (West<s1);
big := (Cell>s2) + (North>s2) + (East>s2)
        + (South>s2) + (West>s2);
RuleCoast:= Cell/2 + (A-B)/2 + 4 * small - 4 * big;
```

By default the data type is unsigned integer with range 0..255.

## 2 Implementing Cellular Automata in Software (C)

The straight forward naive implementation uses a two dimensional array of cells A[0..width+1, 0..height+1] and two loops. A second field B of the same dimensions is recommended to hold the new calculated cells, i. e. to have access to the "old" values earlier in the calculation sequence.

### Compute-Phase

```
for(x=1; x<=width; x++)
  for(y=1; y<=height; y++) {
    Cell = A[x][y];
    North = A[x][y-1];
    NorthEast = A[x+1][y-1];
    East = A[x+1][y];
    SouthEast = A[x+1][y+1];
    South = A[x][y+1];
    SouthWest = A[x-1][y+1];
    West = A[x-1][y];
    NorthWest = A[x-1][y-1];
    B[x][y] = RuleWW(Cell, North, NorthEast, East,
                    SouthEast, South, SouthWest,
                    West, NorthWest, x, y);
  }
```

### Update-Phase

```
for(x=1; x<=width; x++)
  for(y=1; y<=height; y++)
    A[x][y] = B[x][y];
```

The description of the rule RuleWW in C is

```
celltype RuleWW(celltype Cell, celltype North,
                celltype NorthEast, celltype East, celltype SouthEast,
```

```

celltype South, celltype SouthWest, celltype West,
celltype NorthWest, int x, int y)
{
switch (cell) {
case 0: // Background
return 0;
case 1: // Electron Tail
return 3;
case 2: // Electron Head
return 1;
case 3:{ // Wire
int sum =
(y>1 && North == 2? 1 : 0) +
(x<width && y>1 && NorthEast == 2? 1 : 0) +
(x<width && East == 2? 1 : 0) +
(x<width && y<height && SouthEast == 2? 1 : 0) +
(y<height && South == 2? 1 : 0) +
(x>1 && y<height && SouthWest == 2? 1 : 0) +
(x>1 && West == 2? 1 : 0) +
(x>1 && y>1 && NorthWest == 2? 1 : 0);
return (sum == 1 || sum == 2)? 2 : 3; }
}
}

```

To reduce the computation time in software, the code can be optimized for the underlying machine. The main methods of optimizing the code are

1. Use a one-dimensional field to reduce address calculations.
2. Use pointers instead of indices.
3. Write the calculation function inline to the compute phase.
4. Implement border with additional memory instead of if-statements.
5. Delete the update-phase by using two pointers, which point to A and B in one computation and interchange the pointers for the next computation.
6. Change the loop iteration index from 1..n to n-1..0 (faster loop terminate check).
7. Reuse variables and intermediate results.
8. Reduce conditional statements, e.g. by using tables.
9. Use "case"(switch) instead of multiple if-statements, also nest if necessary.
10. Keep the cache filled.
11. Copy whole lines by a fast copy procedure (memcpy) into a three line buffer and operate on the lines instead of accessing the whole field directly.

The rules were implemented in C with different degrees of code optimization using the mentioned optimization methods. The used compiler was GNU gcc 3.3.1 with optimization parameter `-O3`. The computation time on the platform Pentium 4 2.4 GHz (Fujitsu/Siemens) with Windows XP and Cygwin 1.25.5 for the best optimized version is shown in table 1, using the abbreviations displayed in table 2.

Size n x n	#cells	gen./s	t/co	c/co
128 x 128	16k	4296.9	14.2 ns	34
256 x 256	65k	1024.0	14.9 ns	36
512 x 512	256k	179.6	21.2 ns	51
1024 x 1024	1M	41.4	23.1 ns	55
2048 x 2048	4M	10.3	23.1 ns	56
4096 x 4096	16M	2.6	22.9 ns	55
8192 x 8192	64M	0.6	22.9 ns	55

Table 1: Wire World, optimized C-code

<b>#cells</b>	number of cells
<b>gen./s</b>	generations per second
<b>t/co</b>	time per cell operation
<b>c/co</b>	clock cycles per cell operation
<b>ps</b>	pipeline stages
<b>AS</b>	Design automatically optimized for Area or Speed (values: area=0..speed=10)
<b>%uca</b>	percentage of used chip area
<b>#ulc</b>	used logic cells
<b>max. MHz</b>	max. clock frequency in MHz

Table 2: Abbreviations for tables

The rules can be implemented relatively fast in software and that the time per cell operation only slightly increases with the number of cells. Possibly the increase is due to the fact that the caches have limited size, but in general the used PC system is well balanced with respect to memory accesses. An astonishing result was also that the computation time for the unoptimized rule was only about two times longer, meaning that the compiler is able to optimize the code efficiently.

The results for the most optimized C-code for the RuleCoast are shown in the table 3.

Size n x n	gen./s	t/co	c/co
128 x 128	1719.0	35.5 ns	85.2
256 x 256	455.1	33.5 ns	80.5
512 x 512	110.1	34.6 ns	83.1
1024 x 1024	26.0	36.7 ns	88.2
2048 x 2048	5.5	43.3 ns	103.9
4096 x 4096	1.1	53.0 ns	127.3
8192 x 8192	0.3	54.5 ns	130.9

Table 3: Rule Coast Line, optimized C-Code

Compared to the RuleWW the computation time is about 2.5 times longer due to the fact that RuleCoast is more complex and contains more if-statements, which even in the optimized form can not be removed. The time per cell operation increases moderately with the size of the field which is possibly due to increasing cache faults.

In the next section we investigate the question how much faster these two rules can be implemented in FPGA hardware.

### 3 Hardware Prototype Implementation

We have implemented both rules in FPGA logic, using the Altera-Flex10k evaluation board (FPGA chip Flex EPF10K70RC240-4) and the MaxPlus II tools. The rules were described in the Hardware Description Language Verilog.

For example the description of the RuleCoast in Verilog is

```
module rule(clk, cnew, n, e, c, w, s);
  input  clk;
  input  [7:0] c,n,e,s,w;
  output [7:0] cnew;
  parameter s1 = 42, s2 = 213;

  function [7:0] min;
  input [7:0] a, b;
  begin min = a<b? a : b; end
  endfunction

  function [7:0] max;
  input [7:0] a, b;
  begin max = a>b? a : b; end
  endfunction

  wire [2:0] as1 = (n<s1)+(e<s1)+(c<s1)+(w<s1)+(s<s1);
  wire [2:0] as2 = (n>s2)+(e>s2)+(c>s2)+(w>s2)+(s>s2);
  wire [7:0] as1_8bit = as1;
  wire [7:0] as2_8bit = as2;
  wire [7:0] as1_as2 = (as1_8bit<<2)-(as1_8bit<<2);
  wire [7:0] maxne = max(n,e); //pipe#1
  wire [7:0] maxcw = max(c,w); //pipe#1
  wire [7:0] maxnecw = max(maxne,maxcw); //pipe#2
  wire [7:0] max3 = max(maxnecw,s); //pipe#3
  wire [7:0] minne = min(n,e); //pipe#1
  wire [7:0] mincw = min(c,w); //pipe#1
  wire [7:0] minnecw = min(minne,mincw); //pipe#2
  wire [7:0] min3 = min(minnecw,s); //pipe#3
  wire [7:0] max3_min3 = max3-min3;
  wire [7:0] cnew = (max3_min3>>1)+as1_as2+(c<<1); //pipe#4
endmodule
```

The RuleWW was implemented in a register array of sizes 10 x 10, 64 cell operations in the kernel field (8 x 8) work in parallel. The RuleCoast was implemented in a register array of sizes 6 x 6, 16 cell operations in the kernel field (4 x 4) work in parallel. The designs were synthesized with different parameters

- (1) A parameter AS tells the synthesizer whether the area or speed should be optimized. AS runs from 0 to 10.
- (2) the number of pipeline stage was varied and optimized by hand. If pipeline stages are used, only the optimal number and placement of stages are taken into consideration.

ps	AS	%uca	#ulc	max. MHz	t/co	speed-up
0	0	68	2554	19.5	0.80 ns	18
0	5	68	2554	21.2	0.74 ns	19
0	10	68	2554	19.7	0.79 ns	18

Table 4: Rule WireWorld synthesized, 64 rules in parallel

For the RuleWW (table 4) the maximal clock frequency varies between 19.5 and 21.2 MHz. The best time reached per cell operation was 0.74 ns. Compared to the best software solution (14.2 ns) a speed-up of 19 is reached, using only 2554 logic cells (68 % of the total amount). For the implementation of one rule in hardware approximately  $2554/64 = 40$  logic cells are needed.

The number of needed logic cells does not significantly depend on the number of pipeline stages. The optimal choice of the parameter AS can lead to a design which is up to 9 % faster.

For the RuleCoast (table 5) the maximal clock frequency varies between 6.6 and 25.8 MHz (38.76 ns). The best time reached per cell operation was 2.7 ns. Compared to the best software solution (33.48 ns  $\approx$  33 ns) a speed-up of 13.8 ( $\approx$  14) is reached, using only 2259 logic cells (60 % of the total amount). For the implementation of one rule in hardware about  $2259/16 = 141$  logic cells are needed. Even this complex rule can operate with a clock frequency of 25.8 MHz using pipelining. This means that even very complex rules can produce a new result in every clock cycle. It is recommended to use pipelining if the straight forward approach without pipelining is too slow (6.6 MHz). It is interesting to observe that the chosen value of the AS parameter does not always yield the expected optimization.

ps	AS	%uca	#ulc	max. MHz	t/co	speed-up
0	0	71	2673	6.6	9.47 ns	3.5
0	10	80	2878	7.2	8.59 ns	3.9
3	0	60	2259	22.8	2.74 ns	12.2
3	5	60	2295	23.2	2.69 ns	12.4
3	10	69	2604	25.8	2.42 ns	13.8

Table 5: Rule Coast Line synthesized, 16 rules in parallel

We call the 16 x 16, respective 4 x 4 cell field computation window. If larger cell fields are to be computed, the computation window has to be shifted over the whole field. To solve this problem in hardware, different approaches are feasible. One approach is to allocate enough registers to hold the whole field. Another approach is the use of the internal memory banks on the chip and/or external memories. In the second approach, which was demonstrated in the CEPRA-1X, shift register or FIFOs can be used to reuse cell values in order to cut down the memory-read bandwidth.

## 4 Comparison Hardware vs. Software

The computation time for one new cell state is in software

$$t_S = k_S * T_S \tag{1}$$

with  $k_s$  = number of clock cycles per rule,  $T_S$  = time for one computer clock.

The computation time for one new cell state is in FPGA hardware

$$t_H = k_H * T_H / p \quad (2)$$

with  $k_H$  = number of clock cycles per rule,  $T_H$  = time for one FPGA clock.  $p$  = degree of parallelism in hardware.

The speed-up using the FPGA is

$$S = \frac{t_S}{t_H} * \frac{k_S}{k_H} * \frac{T_S}{T_H} * p \quad (3)$$

In our test cases the parameters are

Rule 1: RuleWW

$$S = \frac{34}{1} * \frac{0.416 \text{ ns}}{47.17 \text{ ns}} * 64 = 19.2 \quad (4)$$

Rule 2: RuleCoast

$$S = \frac{80.5}{1} * \frac{0.416 \text{ ns}}{38.76 \text{ ns}} * 16 = 13.8 \quad (5)$$

Using more advanced FPGA technology (Xilinx Virtex-E, Altera Stratix) we could operate with higher clock rates and with a higher degree of parallelism. The FPGA Altera EPS120 supplies 114'140 logic cells and the Xilinx 2VP125 supplies 125'136 logic cells.

For the implementation of the rule RuleWW about 40 logic cells are needed, and for the rule RuleCoast 141 logic cells. We assume that 50 % of the chip is available to implement the compute window and the other 50 % for logic to feed the window and to communicate to the outside. For the EPS120 we have 57070 cells available to implement the rules.

The degree of parallelism would then be

$$p(\text{Rule 1}) = 57070/40 = 1426 \quad (6)$$

$$p(\text{Rule 2}) = 57070/141 = 404 \quad (7)$$

If we assume a FPGA technology which operates at 1/15 of the actual PCs of clock rate we reach the following speed-ups

$$S(\text{Rule1}) = 24 * 1/15 * 1426 = 2282 \quad (8)$$

$$S(\text{Rule2}) = 142 * 1/15 * 404 = 3824 \quad (9)$$

The speed-up is higher for the more complex rule 2. This is mainly due to the fact, that even complex rules can be computed in one clock cycle by the use of pipelining.

## 5 Results

The implementation of the Cellular Automata computing model can be implemented significantly faster in FPGA hardware than in optimized software. Whereas the execution time in software depends on the complexity of the rule, the execution time (throughput) in hardware is almost independent of the rule complexity because of pipelining. With pipelining one new cell state computation can be completed in each clock cycle even for very complex rules. Although the effort to implement a rule in hardware is much more time consuming and not standard, the hardware solution is much cheaper than hundreds or thousands of PCs working parallel together in software to gain the same performance.

## References

- [De90] Dewdney, A. K.: Draht- und eppichwelten. *Spektrum der Wissenschaft*. 1990.
- [HHV01] Heenes, W., Hoffmann, R., und Völkman, K.-P.: Hardware-Unterstützung für Eingebettete Hochleistungsanwendungen. *thema Forschung* 1/2001. TU Darmstadt. 2001. ISSN 1434-7768.
- [HHVS97] Hochberger, C., Hoffmann, R., Völkman, K.-P., und Steuerwald, J.: The CEPRA-1X Cellular Processor. In: *IPPS97*. Genf, Schweiz. 1997.
- [HHVW00] Hochberger, C., Hoffmann, R., Völkman, K.-P., und Waldschmidt, S.: The Cellular Processor Architecture CEPRA-1X and its Configuration by CDL. In: *RAW 2000 (IPDPS 2000), Parallel Computing Technologies (LNCS 1800)*. Springer Verlag. 2000.
- [Ho98] Hochberger, C.: *CDL - Eine Sprache für die Zellularverarbeitung auf verschiedenen Zielplattformen*. PhD thesis. Darmstädter Dissertation D17. 1998.
- [HUVW00] Hoffmann, R., Ulmann, B., Völkman, K.-P., und Waldschmidt, S.: A Stream Processor Architecture Based on the Configurable CEPRA-S. In: *FPL 2000, in LNCS 1896*. Springer. 2000.
- [HVS94] Hoffmann, R., Völkman, K.-P., und Sobolewski, M.: The Cellular Processing Machine CEPRA-8L. In: Jesshope, Jossifov, und Wilhelmi (Hrsg.), *Parcella 94*. volume 81 of *Mathematical Research*. S. 179–188. Akademie Verlag. 1994.
- [Rö03] Röder, P.: *Effiziente Implementierung zellularer Automaten in Software*. Studienarbeit. TU Darmstadt. 2003.
- [TM87] Toffoli, T. und Margolus, N.: *Cellular Automata Machines*. MIT Press. 1987.
- [vN66] von Neumann, J.: *Theory of Self-Reproducing Automata*. Univ. of Illinois Press. 1966.
- [Zu69] Zuse, K.: *Rechnender Raum*. Vieweg Braunschweig. 1969.