Minimising the Hardware Resources for a Cellular Automaton with Moving Creatures

Mathias Halbach Rolf Hoffmann

TU Darmstadt, FB Informatik, FG Rechnerarchitektur
Hochschulstraße 10, D-64289 Darmstadt, Germany
E-Mail: {halbach, hoffmann}@ra.informatik.tu-darmstadt.de

Abstract: Given is the following "creature's exploration problem": n creatures are moving around in an unknown environment in order to visit all cells in shortest time. This problem is modelled as a cellular automaton (CA) because the CA model is massively parallel. Therefore, it is perfectly suited to be supported by hardware. We are trying to simulate the CA as fast as possible by the use of the FPGA technology. We need a very fast simulation because we want to observe and evaluate much different behaviours of the creatures. Our main goal in the background is the optimisation of the behaviours of the creatures. In this contribution we have investigated the question how the creature's exploration problem can be implemented in hardware with a minimum of hardware resources in order to maximise the number of cells which can be computed in parallel (or to speed up the simulation). We have designed and evaluated four different implementations that vary in the combination or separation of the logic for the environment, for the creatures and for the collision detection.

1 Introduction

In order to speed up significantly the execution of a Cellular Automaton hardware support is necessary. Previous investigations [HH04] have shown that a speed up of hundreds to thousands is possible by the use of dedicated FPGA logic compared to software simulation on a personal computer. The resources needed to implement a Cellular Automaton in hardware depend on the number of cells in the field, the kind of neighbourhood and the complexity of the rule.

In this contribution, we are researching a special problem, in which an automaton cell consists of two parts, a simple one and a complex one. The goal is to minimise the hardware resources of a FPGA implementation in order to enlarge the size of the field that can be computed fully in parallel.

The problem used as example is the following: A number of creatures with local intelligence shall move around autonomously in an environment in order to visit all empty cells in shortest time (with a minimum number of time steps, i. e. generations). The *environment* is given by a field of cells, called the *environment cells*. The environment cells are either of type *empty* or of type *obstacle*. A creature can move to an empty cell if no other creature tries to move to the same position (otherwise this is a conflict situation). The

creature's behaviour is given by a state machine, which reacts on an input signal m (*move*, creature can move). If the creature cannot move, it will turn to the right or to the left. The state machine is implemented by a state table.

The main goal of our so called "creature's exploration problem" – which is not addressed here in detail – is to find out the absolute optimal state machines for a given number of states to explore an unknown environment. It is very time consuming to find out the optimal behaviour if the number of states are larger than 5 or 6. We found an optimal behaviour (fig. 1) for one creature for a given set of initial configurations (environment plus the initial position and state of the creature) by the use of FPGA logic. Even with FPGA technology, it is not easy to find the optimum because the set of solutions is growing exponentially with the number of states. Like in software, the hardware implementation must try to simplify the complexity of the search procedure, e.g. by avoiding to test equivalent state machines and by not generating state machines with less states than required.

The problem of finding optimal solution of moving agents using a state machine has also been addressed in [MSPPU02], and the problem has practical applications like mowing a lawn [Koz92] or exploring an unknown environment by robots. In contrast, this contribution is focused on the minimisation of the hardware resources. We use the cellular automata model, because all the cells can update their state independently in a synchronous operation mode. Such models are called *massively parallel* and they are perfect models to be efficiently mapped into hardware.

Results of our preceding investigations were presented in [HHHT04, HH05, HHH04, HH04]. The goal of these investigations was to find out the best behaviour for several moving creatures by the aid of FPGA acceleration. The current work is focussed on minimising the hardware resources in order to be able to simulate as many automatons as possible in parallel.

2 Formal description of the problem

The problem consists of two types of cells: (a) environment cells and (b) creatures. The environment cells are simple, static in their state and have four fixed links to their neighbours. The creatures are variable in their location and they have a variable state (direction, control state). Moreover, a creature has only one dynamic link to the neighbour in front of its moving direction. In the classical uniform CA model the union of these types forms a complex cell, which can be switched dynamically to the actual needed type. Environment cells carry either the value *free* or *obstacle*. Free cells can be visited whereas obstacles cannot. The border of an environment must be defined by cells of type obstacle. A rectangular environment can be described by

- the size n_x and n_y with $n_x, n_y \in \mathbb{N}$,
- the positions of the obstacles including the border positions $\mathbb{H} \subset \{c \mid c = (x, y) \in \mathbb{N}_0 \times \mathbb{N}_0 \ \wedge \ 0 \leq x < n_x \ \wedge \ 0 \leq y < n_y\} =: \mathbb{P}$,
- the border $\{c \mid c = (x, y) \in \mathbb{P} \ \wedge \ (x \in \{0, n_x - 1\} \vee y \in \{0, n_y - 1\})\} \subseteq \mathbb{H}$

where $\mathbb{P}$ is the set of all possible positions. The free cells are given by $\mathbb{F} := \mathbb{P} \backslash \mathbb{H}$.

Each creature (with index $i \in \mathbb{I}$, $|\mathbb{I}| =$ number of creatures) is defined by its actual position, direction and control state at the time step $t \in \mathbb{N}_0$:

- position: $p_{i,t} \in \mathbb{F}$,
- direction: $r_{i,t} \in \{0, 1, 2, 3\} =: \mathbb{D}$, where 0 represents north, 1 represents east etc.,
- control state: $s_{i,t} \in \mathbb{S} = \{v \mid v \in \mathbb{N}_0 \ \wedge \ 0 \leq v < S\}$ with $s_{i,0} := 0$

The number of possible control states is S which is a measure for the "brain power" of the creature. The creature looks one cell ahead in its actual moving direction $r_{i,t}$ and is able to read information from that position. We call the corresponding cell *front cell* and the position *front position*, which is defined as $\vec{p}_{i,t} \in \mathbb{F}$ by

$$\vec{p}_{i,t} := \begin{cases} (x_{i,t},\ y_{i,t} + 1) & \text{if } r_{i,t} = 0 \text{ (north)} \\ (x_{i,t} + 1,\ y_{i,t}) & \text{if } r_{i,t} = 1 \text{ (east)} \\ (x_{i,t},\ y_{i,t} - 1) & \text{if } r_{i,t} = 2 \text{ (south)} \\ (x_{i,t} - 1,\ y_{i,t}) & \text{if } r_{i,t} = 3 \text{ (west)} \end{cases} \qquad \text{with } (x_{i,t}, y_{i,t}) = p_{i,t}.$$

Other front cell's features are tagged in the same way, e. g. $\vec{h}$ is the obstacle information of the front cell.

A creature must move to its front position if the front cell is reachable. The front cell is reachable (1) if the environment cell is free (not an obstacle) and (2) not occupied by another creature and (3) there is no conflict: If more than one creature wants to move to the same front cell, a conflict exists that must be resolved. There are several solutions to resolve the conflicts, such as

1. Only one creature is allowed to move, selected by an arbitration process that needs to be defined.
2. No creature will move, because the creatures can look one position farther (increased neighbourhood distance = 2)
3. The front cell overtakes the additional task to observe conflicts and in such a case, it sends a stop signal to the creatures. Thereby the creatures in conflict are prevented from moving.

Hochberger [Hoc98] has solved the problem of conflict resolution in another way using a two phase algorithm. In the first phase, one of the creatures is selected by the front cell and the front cell changes some hidden control bits accordingly. In the second phase, each creature knows how to behave by interpreting the hidden control bits.

For the cases 2 and 3 the *moving condition* $m_{i,t}$ describes whether the creature i can move or not:

$$m_{i,t} := \begin{cases} \text{true} & \text{when } p'_{i,t} \in \mathbb{F} \ \wedge \ \forall_{j \in \mathbb{I}} \left((i = j) \vee (p'_{i,t} \neq p_{j,t} \ \wedge \ p'_{i,t} \neq p'_{j,t}) \right) \\ \text{false} & \text{otherwise} \end{cases}$$

and the next position $p_{i,t+1}$

$$p_{i,t+1} := \begin{cases} \vec{p}_{i,t} & \text{if } m_{i,t} = \text{true} \\ p_{i,t} & \text{if } m_{i,t} = \text{false} \end{cases} .$$

Depending on the moving condition the next position is

$$p_{i,t+1} := \begin{cases} \vec{p}_{i,t} & \text{if } m_{i,t} = \text{true} \\ p_{i,t} & \text{if } m_{i,t} = \text{false} \end{cases} .$$

Simultaneously with a possible move, the creature may change its state $s_{i,t}$ and its direction $r_{i,t}$ according to the next state function f and the output function g (turn right or left) which both are stored in the "brain" of the creature, i. e. in a memory table.

$$s_{i,t+1} \leftarrow f(s_{i,t}, m_{i,t})$$
$$r_{i,t+1} \leftarrow (r_{i,t} + g(s_{i,t}, m_{i,t})) \bmod 4$$

A state machine is formed by connecting the memory with a state register s and a direction register r as shown in figure 1a. The output actions of the creature are

R = turn right if the creature can't move (moving condition = false), coded by "0"
L = turn left if the creature can't move (moving condition = false), coded by "1"
Rm = move and turn right, if the creature can move, coded by "0"
Lm = move and turn right, if the creature can move, coded by "1"

Note that the state machine belongs to the creature and also has to move if the creature moves. A move of a creature in the CA model can be accomplished by copying the creature's state to the front cell and deleting the creature on its current position. The best 6-state behaviour for one creature exploring 24 different environments was determined by enumeration using FPGA technology (fig. 1b) in preceding investigations.

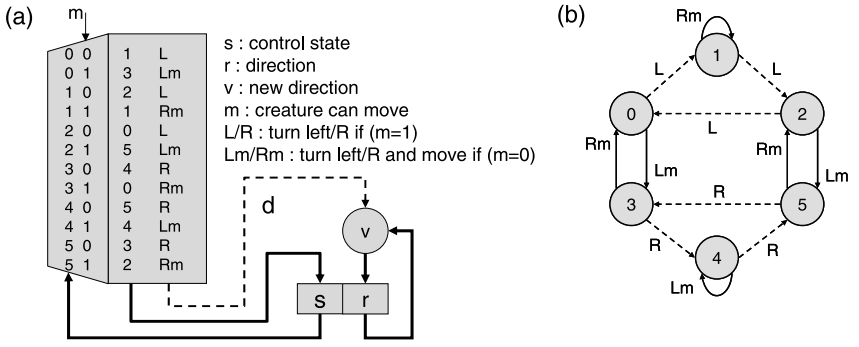

Figure 1: A state machine (a) models the behaviour of a creature. The best 6-state algorithm (b).

3 Implementation Variants

The current goal is to reduce the amount of hardware resources needed for an FPGA implementation in order to increase the amount of cells, which can be computed in parallel.

The following variants have been considered (1) **Uniform:** classical uniform cellular automata, (2) **Separated:** separate creatures (complex rule) attached to the environment, (3) **Augmented A:** The environment is augmented with parts of the creature's state and additional logic, (4) **Augmented B:** The environment is augmented with an index and additional logic.

In the uniform variant, the state of a cell is the union of the type *environment* and the type *creature*. In the separated variant, the creatures are stored as individuals which can read the environment. In the augmented variants the environment is augmented with additional information such as creature's index (identification number), creature's direction or index (own position) of the environment cell.

In the **Uniform Variant**, all cells are of the same type which is the union of all types which have to be modelled. Each cell has the same capabilities (attributes and rules) which are not necessarily used in every generation. It's also possible to say that the cells are polymorphic. In our case, the capabilities of the environment are joined with the capabilities of the creature.

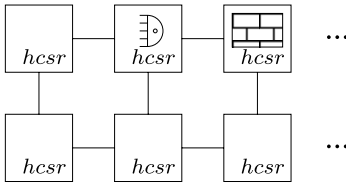

Figure 2: Uniform Implementation

The capabilities that must be combined in such an uniform cell (fig. 2) is

- h: environment attribute – obstacle (true/false)
- c: type selection – creature is on the cell (true/false)
- s: control state of the creature
- r: direction of the creature
- ST: state table defining the creature's behaviour
- CD: logic which generates a move signal m

The attributes c, s and r have to be variables. The environment attribute h shall be implemented as a variable (register) in order to be able to change the environment during the simulation.

A state table ST defines the behaviour of the creatures. In this investigation, we assume that the state table is common to all creatures. It shall be variable, meaning that it should consist of registers which contents may be changed dynamically (e. g. in an optimisation procedure). Optionally additional information for statistics (e. g. if the cell was already visited) may be stored in each cell. The next state (s') and next direction (r') of the uniform cell are given by the following formulas

$$
\begin{aligned}
\dot{m}^w &:= c^w \wedge (r^w \equiv w + 2 \bmod 4) \text{ for } w \in \mathbb{D} \\
\dot{m} &:= \exists_{w_1 \in \mathbb{D}} m^{w_1} \wedge \neg \exists_{w_2 \in \mathbb{D}, w_2 \neq w_1} m^{w_2} \\
\dot{s} &:= \begin{cases}
s^{\text{north}} & \text{if } c^{\text{north}} \wedge r^{\text{north}} \equiv \text{south} \\
s^{\text{east}} & \text{if } c^{\text{east}} \wedge r^{\text{east}} \equiv \text{west} \\
s^{\text{south}} & \text{if } c^{\text{south}} \wedge r^{\text{south}} \equiv \text{north} \\
s^{\text{west}} & \text{if } c^{\text{west}} \wedge r^{\text{west}} \equiv \text{east}
\end{cases} \\
s' &= \begin{cases}
f(s, \text{false}) & \text{if } c \wedge \vec{h} \\
f(\dot{s}, \text{true}) & \text{if } \neg c \wedge \dot{m} \\
\text{any} & \text{otherwise}
\end{cases} \\
r' &= \begin{cases}
r + g(s, \text{false}) \bmod 4 & \text{if } c \wedge \vec{h} \\
\dot{r} + g(\dot{s}, \text{true}) \bmod 4 & \text{if } \neg c \wedge \dot{m} \\
\text{any} & \text{otherwise}
\end{cases}
\end{aligned}
$$

with the same definition for $\dot{r}$ as for $\dot{s}$.

The CD logic for collision detection generates the signal m (move) which decides whether the creature can move or not. This signal move becomes false if (1) the front cell is an obstacle, (2) a creature is placed on the front cell, (3) two or more creatures want to move to the same front cell if the front cell is free.

Each creature can check the first and second condition by testing the front cell. The testing of the third condition is more complicated: First, the front cell checks if there are more than one creature in the neighbourhood, which want to move to it (conflict). Second, the conflict situation is send back to all the creatures, which have caused the conflict.

Separated Variant: Another way is to separate the environment and the creature, such that the changing state of the creatures is separated and therefore minimised. The advantage is that complex rules for the creatures need not to be replicated in the cells, which would results in a poor utilisation of the hardware.

Each creature has to be able to read the status of the environment cell (obstacle or not) from the current front position. This can be achieved in hardware by the use of a multiplexer (see fig. 3). A technical problem arises when the number of cells exceeds a certain limit, e. g. $256 = 16 \times 16$ with our current FPGA technology. When exceeding the limit, the hardware needs too many resources (wires and cascaded multiplexers) or the time delay will be too large. Therefore, this approach is limited to a small field of 16×16 if the limit is 256.

Another problem arises with the detection of several creatures that are in conflict. There must be a central logic that detects and resolves the possible conflicts. This logic becomes very complex when the number of creatures is increasing. For a creature i this conflict exists if $p_i \equiv p_j \vee p_i \equiv \vec{p}_j$ gets true for any j with $i \neq j \in \mathbb{I}$. The generation of this logic is described in the hardware description language AHDL (Altera HDL) by the following statements.

```
DEFAULTS collisionfree[] = VCC; END DEFAULTS;
FOR i IN 1 TO objects - 1 GENERATE
  FOR j IN i + 1 TO objects GENERATE
    IF object[j].pos[] == front[i].pos[] THEN
      collisionfree[i] = GND;
    END IF;
    IF object[i].pos[] == front[j].pos[] THEN
      collisionfree[j] = GND;
    END IF;
    IF front[i].pos[] == front[j].pos[] THEN
      collisionfree[i] = GND; collisionfree[j] = GND;
    END IF;
  END GENERATE;
END GENERATE;
```

It can be seen from these statements, that complexity of the logic increases quadratically with the number of creatures.

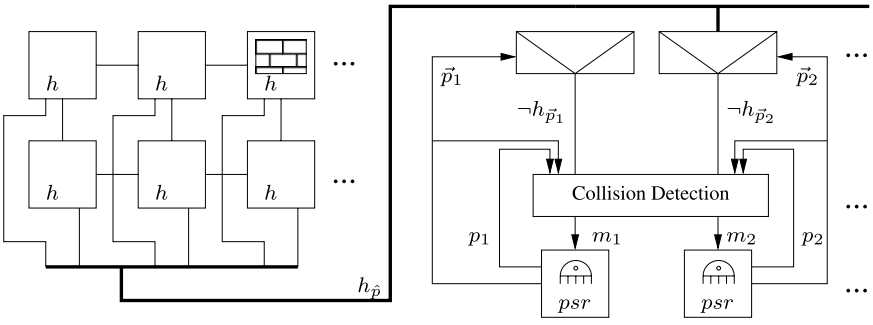

Figure 3: Each creature is connected the environment, the collision detection logic is placed between the creatures.

In the implementation of **Augmented A Variant**, the cells of the field consist of so called *augmented cells* which are the union of the environment h, the direction r of a creature and the index c_i of the creature which is active on that cell. The state of the augmented cells is (h, ci, r). The creature index c_i is stored decoded, which means that for each creature an active bit is reserved. If such an active bit is set, automatically a connection is routed to the appropriate creature via a bus system. Each creature drives an output bus d_i (one bit

wide: turn left/right as output from the state machine) and listens to an input bus (one bit wide: move yes/no) as can be seen in fig. 4.

Each active augmented cell which is occupied by a creature i sends a request to its front cell defined by r_i. The front cell computes the move signal m_i by checking all the conditions that were mentioned above for the uniform variant. The move condition is then sent to the appropriate creature. After inspecting the state table, the creature returns d_i (turn left/right). If the creature cannot move, the augmented cell will change its direction accordingly. In the case, the creature can move to the front cell and copies the active bit from the augmented cell and the new direction from the bus.

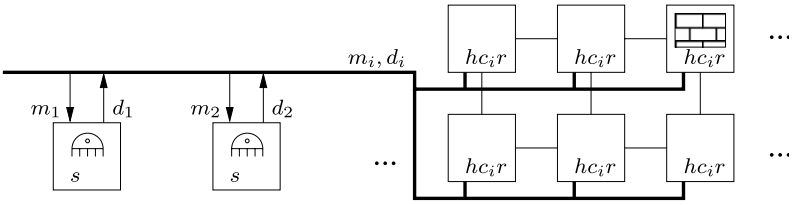

Figure 4: Each augmented cell automatically contacts the appropriate creature via a bus system, computes the move condition and changes its direction according to the answer of the creature.

In **Augmented B Variant**, the implementation of the environment cells are augmented only with an index (own position of the cell). The index is used to connect automatically the appropriate creature to the environment cell. A creature sends its current position p_i via a bus to all cells. If a dedicated cell detects its own position on that bus, it connects to it. In conjunction the direction r_i of that creature is also connected to that cell, and a backward connection m_i is established. This is illustrated in figure 5. The environment cell is augmented with special logic in order to compute the move signal, which is send back via a bus to the appropriate creature.

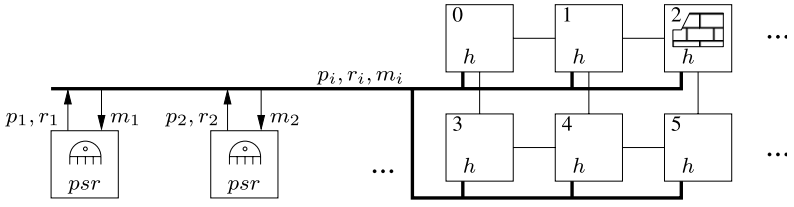

Figure 5: Augmented, Variant B – mixture of Variant A and Multiplexer

An interesting feature is the conflict detection. An active cell (environment cell connected to a creature) asks the front cell (the neighbour in the current direction r_i), if there might be a conflict. The front cell checks if its neighbours may cause conflicts and returns the required information. The active cell then sends the required move signal back to the creature. By this technique, the creatures can indirectly perceive that conflicting creatures are two cells ahead or right/left in front. Using the logic of a front cell during the com-

putation of the next generation (asynchronously) the neighbourhood of a creature can be indirectly increased. The hardware implementation causes no problem and asynchronous oscillations does not occur, because the logic of the front cell uses only local inputs.

4 FPGA Implementation

The problem was described in the hardware description language AHDL and synthesised and configured for the Altera FPGA Cyclone EP1C20F324C7 using the Quartus II tools. AHDL was used instead of Verilog or VHDL, because it offers much better features to generate parameterised logic in an array. The most relevant parameters are the implementation variant, the size of the field, the number of control states of the state machine and the number of creatures. The resources are counted in the number of needed logic elements for that FPGA. The results of the synthesise are shown in figure 6. The maximum clock rate depends on the variant, the field size and the number of creatures. The highest reached clock rate was 81 MHz (8×8, variant 1, 2 creatures); the lowest was 40 MHz (field size 8×8, variant 3, 8 creatures). Note that the clock rate can be influenced by changing AHDL description or by changing the synthesis parameters of the Quartus tool, e. g. setting the clock rate influences the number of fitter passes. In order to compare the performance of the different solutions, also the clock rate has to be taken into account.

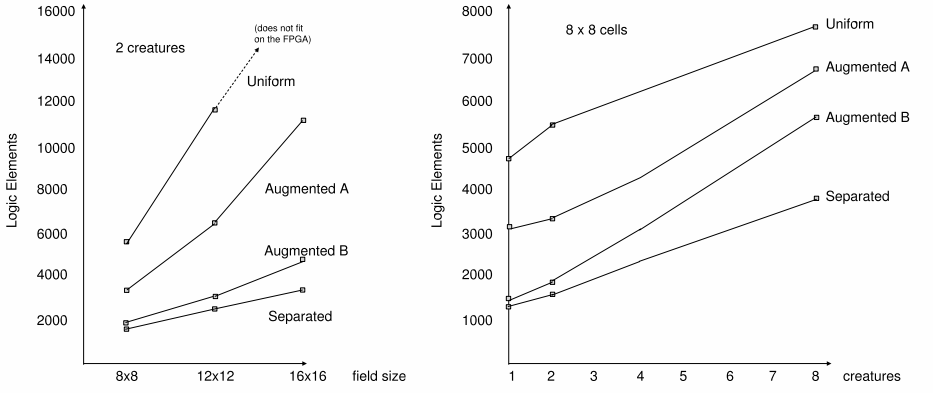

Figure 6: Resources needed for a FPGA implementation

For our implementation we didn't use internal or external memory, although this would be a possible approach. In order to process large CA fields which reside in memory the techniques of parallel pipelined streaming have already been used in previous implementations [WH95] and [HUVW00]. By these techniques p new cell states can be computed in each memory cycle by the use of p pipelines. In the presented approach we can compute 64 cell states in parallel in one clock cycle without using memories. The shortcoming of our approach is a limited number of cells in the field. In order to optimise the creatures behaviours, the limited field size of a 8×8 was sufficient.

5 Conclusion

The creature's exploration problem was modelled as a cellular automaton (CA) because CAs are massively parallel and can be perfectly supported by hardware. Time-consuming simulations are necessary when the creatures' behaviour has to be optimised. In order to reduce the simulation time, as many cells as possible should be computed in parallel in the hardware (FPGA chip). Therefore, the hardware resources have to be minimised.

Four different implementation variants have been designed synthesised for a field of different size and up to 8 creatures. Comparing the different variants, the "Separated" variant was best in counts of logic elements, followed by the "Augmented B" variant. For fields of bigger size and more creatures, the "Augmented B" variant may outperform the other solutions because of fewer interconnections.

References

[HH04] Mathias Halbach and Rolf Hoffmann. Implementing Cellular Automata in FPGA Logic. In *International Parallel & Distributed Processing Symposium (IPDPS), Workshop on Massively Parallel Processing (WMPP)*. IEEE Computer Society, April 2004.

[HH05] Mathias Halbach and Rolf Hoffmann. Optimal Behavior of a Moving Creature in the Cellular Automata Model. In Victor Malyshkin, editor, *Parallel Computing Technologies*, number 3606 in LNCS. Springer, September 2005.

[HHH04] Wolfgang Heenes, Rolf Hoffmann, and Mathias Halbach. Implementation of the Massively Parallel Model GCA. In *Parallel Computing in Electrical Engineering (PARELEC), Parallel System Architectures*, September 2004.

[HHHT04] Mathias Halbach, Wolfgang Heenes, Rolf Hoffmann, and Jan Tisje. Optimizing the Behavior of a Moving Creature in Software and in Hardware. In *Sixth International conference on Cellular Automata for Research and Industry (ACRI 2004)*, number 3305 in LNCS, October 2004.

[Hoc98] Christian Hochberger. *CDL – Eine Sprache für die Zellularverarbeitung auf verschiedenen Zielplattformen*. PhD thesis, TU Darmstadt, 1998.

[HUVW00] Rolf Hoffmann, Bernd Ulmann, Klaus-Peter Völkmann, and Stefan Waldschmidt. A Stream Processor Architecture Based on the Configurable CEPRA-S. In *Field-programmable Logic: The Roadmap to Reconfigurable Systems (FPL 2000)*, number 1896 in LNCS, Villach, Austria, August 2000. Springer Verlag.

[Koz92] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[MSPPU02] B. Mesot, E. Sanchez, C. A. Pena, and A. Perez-Uribe. SOS++: Finding Smart Behaviors Using Learning and Evolution. In Standish, Abbass, and Bedau, editors, *Artificial Life VIII*, page 264ff., 2002.

[WH95] Stefan Waldschmidt and Christian Hochberger. FPGA synthesis for cellular processing. In *IEEE/ACM International Workshop on Logic Synthesis*, May 1995.