International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 1, Number 2, pages 191–210, July 2011

Specialized Multicore Architectures Supporting Efficient Multi-Agent Simulations

Christian Schäck, Rolf Hoffmann, Wolfgang Heenes
Technische Universität Darmstadt
FB Informatik, FG Rechnerarchitektur
Hochschulstraße 10, 64289 Darmstadt, Germany
{schaeck,hoffmann,heenes}@ra.informatik.tu-darmstadt.de

Received: January 30, 2011
Revised: May 20, 2011
Accepted: June 20, 2011
Communicated by Yasuaki Ito

Abstract

Two new multiprocessor architectures to accelerate the simulation of multi-agent systems based on the massively parallel GCA (Global Cellular Automata) model are presented. The GCA model is suited to describe and simulate different multi-agent systems. The designed and implemented architectures mainly consist of a set of processors (NIOS II) and a network. The multiprocessor systems allow the implementation in a flexible way through programming, thus simulating different behaviors on the same architecture. Two architectures, one with up to 16 processors, were implemented on an FPGA. The first architecture uses hardware hash functions in order to reduce the overall simulation time, but lacks scalability. The second architecture uses an agent memory and a cell field memory. This improves the scalability and further increases the performance.

Keywords: simulation of multi-agent systems, global cellular automata, hardware hash function, FPGA, NIOS II, multiprocessor, multicore

1 Introduction and Related Work

Multi-agent modeling can be applied to a variety of disciplines, e.g. traffic simulation [13], simulation of biologic systems [21, 18] or ecosystems [5]. In previous work [15, 17] we have shown how multi-agent systems (MAS) can be described and simulated using the GCA (Global Cellular Automata) model [9, 10]. Other investigations have incorporated the characteristics of CAs (Cellular Automata) to simulate HIV-Immune Interaction Dynamics [24]. For this application the model of cellular automaton has been a good way to study the adaptation and self-regulation properties. Multi-agent simulations have also been realized on GPUs (Graphics Processing Unit) using the CUDA (Compute Unified Device Architecture) framework [19, 20]. The authors of [6] present a new technique to simulate agent-based models on GPUs. They used sugarscape as a test application. With this test application a high simulation performance was reached. But the authors criticized the bad programmability and denote it as counterintuitive.

The realization of cellular automata on GPUs can be efficient if a small neighborhood is needed. This way the local properties of the GPUs architecture can be used efficiently. Multiple cells can be processed at a time allowing fast accesses to neighbor cells as these are loaded in a bunch. Larger neighborhoods can not be realized as efficient as local neighborhoods [23, 22].

In this paper we will present new architectures for fast simulation of multi-agent systems. Using the GCA model the programmability can be kept intuitive. New applications can then be developed with low effort. The new architectures are FPGA (Field Programmable Gate Array) based. Therefore the absolute performance will not be able to keep up with the highly advanced GPUs. Developing new architectures allows us to investigate specially designed commonly used agent functions. Simulating a MAS of size 45×45 on a former multiprocessor architecture (MPA) [15] lasted $0.44\,ms$ per generation for four processors ($p = 4$). Simulation of the same MAS, but without any agents (empty world), lasted $0.37\,ms$ per generation on the same architecture. So the simulation of the empty MAS consumes about 83.58 % of the overall execution time. Thus this overhead has to be minimized. This is done by the design of novel architectures. When superfluous computations are reduced then the complexity of the architectures increases in general. We are presenting two new architectures. The first architecture (HA) shows good simulation results even for a low number of processors at a low clock rate, but lacks scalability. The second architecture (DAMA) scales better and contains an additional hardware function to further increase the performance.

2 Global Cellular Automata Model

The GCA model is a generalization of the CA model using dynamic global links to select neighbor cells. It consists of a set of cells that update their state synchronously in parallel according to a local rule stored in each cell. Each cell can hold multiple data and link fields. We will not distinguish between data and link fields and such fields will be called *blocks*. The term block describes a memory location holding any kind of data (data or link information). The interpretation of a block is determined by the application. For each cell a local cell rule is applied calculating the next data and link states. All cell states at a certain time step t constitute a so called *generation*. Each cell has read access to any other cell using the dynamic links. Write conflicts cannot occur, therefore the model can easily be supported by hardware for a large number of cells. Fig. 1 shows the operation principle for n blocks.

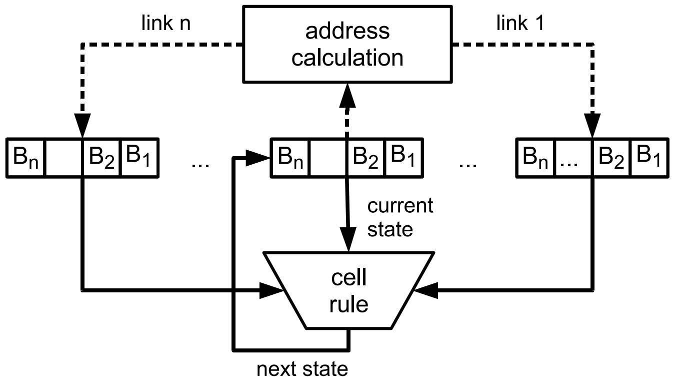

Figure 1: The GCA operation principle

3 Agent System Layer

Several different architectures and enhancements for the GCA model have been developed [14, 15, 16, 17, 11, 12]. While the GCA model can be used for many different applications our focus is the simulation of MAS. At first general architectures have been developed. These architectures are capable of running all GCA algorithms according to the GCA model. Focusing on the task of multi-agent simulation the general architectures can be enhanced and/or restricted. The resulting architectures are no longer necessarily capable of running all GCA applications but achieve a higher simulation performance regarding multi-agent simulations.

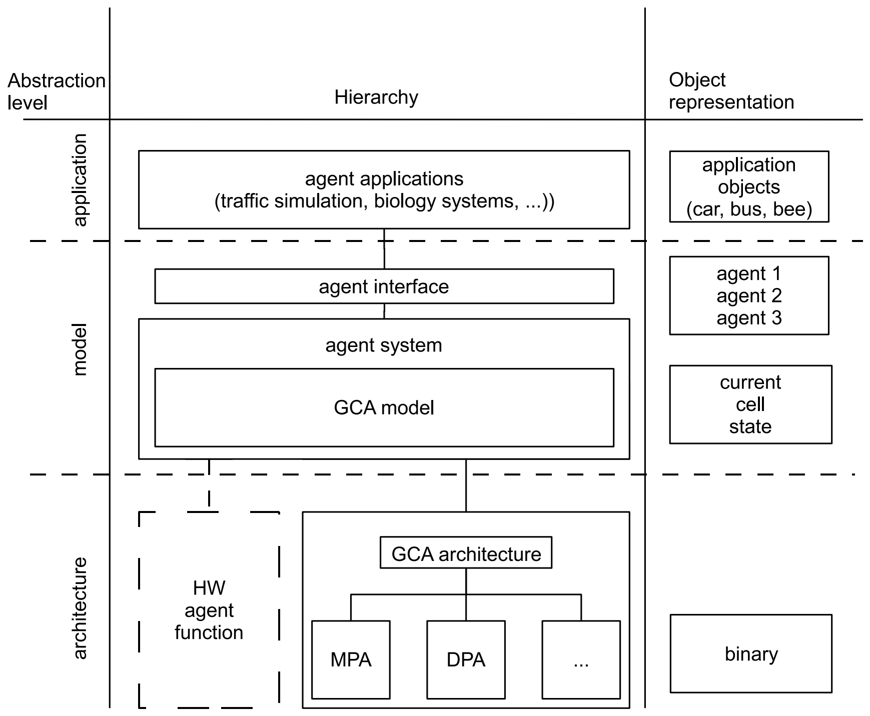

Figure 2: Model Stack

With many different agent behavior rules [21, 18, 13, 5], MAS and many different simulation architectures there is the need of a common interface. Therefore the agent system layer has been introduced to the overall system (Fig. 2). The agent system layer defines how MAS are simulated using the GCA model itself or modified versions of it. With the agent system layer the GCA model can be enhanced or restricted and a mapping of the agent behavior rule into the cell rule is defined. Furthermore, the agent system layer defines commonly used agent functions (such as cell checks for moving agents) and abstracts from different architectures. So far, the agent system layer defines the interface of the commonly used functions. The functions can either be implemented in software or hardware, depending on the applications need. In a complete agent simulation system a complete well defined function library can be considered as given. Fig. 2 shows the complete system model stack. The actual application uses the interface (and the defined functions) for simulation. The agent system layer, which is based on the GCA model defines the actual architecture.

4 Test Application - Agent Behavior Rule

In order to compare the architectures, an artificial MAS was defined. Agents are moving around on a 2D grid according to a simple rule. The 2D grid including the agents constitutes the MAS. The agent's behavior is implemented as C code running on multiple processors and applied to all agents of the MAS. The two different object types are: agent (A) and obstacle (O) (Fig. 3).

A: An agent checks four neighbor positions (A1, A2, A3, A5) located in the moving direction of the agent. For the front position (A3) an easy check is executed as this position only needs to be empty. The remaining three positions (A1, A2, A5) are checked for an agent with a moving direction to the front position (A3). If this is the case none of the agents can move to this position to avoid a collision. If only the agent from the current position (A4) wants to move

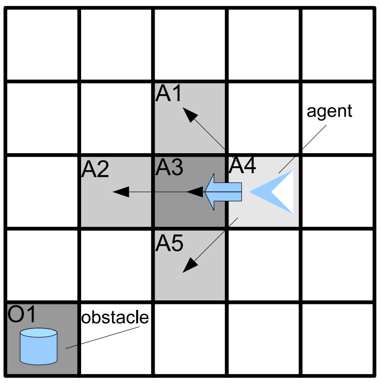

Figure 3: Multi-agent system. Agent A4 checks four cells in its moving direction in order to detect an obstacle, another agent or a conflict.

to the front position (A3) it is deleted from that position (A4). The agent in Fig. 3 shows this situation.

O: An obstacle (O1) does not need to check any neighbor, it is not altered and copied to the next generation.

Each agent (A) behaves according to the following rule:

- if agent can move then move forward and simultaneously turn (toggle between S and W or between N and E)
- if agent can not move then turn right

The implemented direction of rotation for the two cases is shown in Fig. 4.

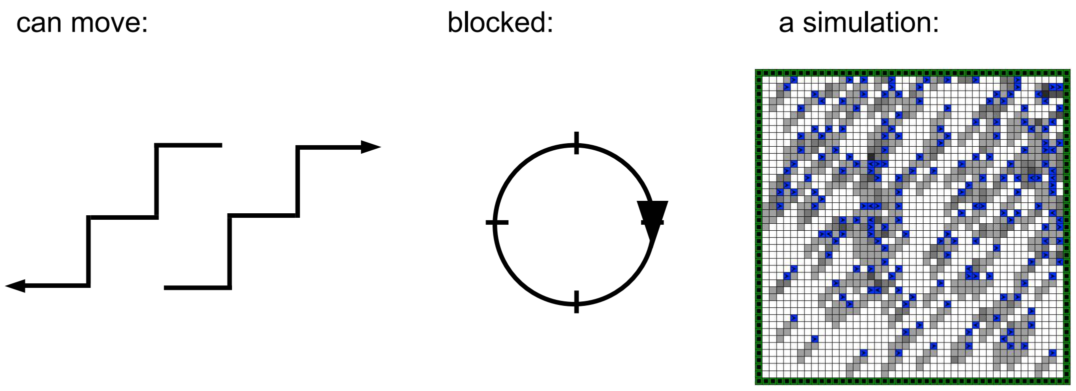

Figure 4: Agents behavior. An Agent is moving the SW-NE diagonal up or down if it can move, otherwise it turns right

5 Multiprocessor Architecture with Hash Function (HA)

5.1 System Overview

The new architecture supporting the GCA model consists of p NIOS II softcore processors [4] each supplied with a program memory and a data memory. The program memory holds the cell rule (agent behavior). The data memory holds the active cell states. In contrast to the former architecture

(MPA) [15], the one presented here stores only part of the cells in order to save resources and to process large cell fields that are sparsely occupied by agents. As not all cells are stored in the data memory we first have to distinguish between the data that is stored and the data that is not stored. Therefore we defined two cell types:

- **active cell:** A cell that requires calculation and has relevant data that has to be stored in the data memory. In the context of agent simulation, this cell is some kind of agent cell. (here: agent or obstacle)
- **inactive cell:** A cell that holds no data and does not require to be processed. Therefore inactive cells are not stored in the data memory. In the context of agent simulation, this cell is an empty cell.

Each data memory holds a subset of all active cells. Each NIOS II processor applies the rule to the cells data in its associated data memory. To be able to read the data of a neighbor, all processors are connected to a network (via various hardware-software functions implemented through custom instructions). The data memories are implemented as dual port memories (Fig. 5). The first port is used for read accesses by the associated processor and by a write arbiter, the second port is used for read accesses by other processors via the network. The data memories are capable of holding two generations at a time. The current generation is used for all read accesses while the next generation is used for all write accesses. Thereby data consistency is given at any time (synchronous updating between generations).

A NIOS II processor, its associated data and program memory, a read/write arbiter, a flag register and a local buffer is called *Processing Unit* (PU) (Fig. 5 highlights processing unit 3. The program memory is not shown). The design of the network is essential for the overall performance. Previous investigations [15] have shown that a bus network with dynamic arbitration performs very well for multi-agent simulation in comparison to other networks (ring, omega). Different networks were also investigated for the hash architecture. It turned out that the bus network with dynamic arbitration performed best for our application.

All active cells are distributed among the available data memories of all PUs based on their cell index. As each memory location can only store one cell at a time a hash function ($h(i)$ in Fig. 5) is used to determine a free memory location. The hash function is used to determine the PU and the memory location. The address calculated by the hash function consists of a lower part determining the memory location within the data memory and an upper part selecting the PU.

A cell in the agent world described by its coordinates (x, y) is transformed to a cell index i. The top left cell is the cell with the coordinates $(0, 0)$. The cell index can be calculated by the following function where (x, y) are the cells or agents coordinates and MAXX is the dimension of the agent world in the X-direction:

$$i = f(x, y) = y \cdot MAXX + x$$

Within the cell rule (or agent's behavior) only the cell index i is used. The neighbor cells of the cell i can be determined by: $i + 1$ (right neighbor), $i - 1$ (left neighbor), $i - MAXX$ (upper neighbor), $i + MAXX$ (bottom neighbor).

Note that the amount of agents per data memory varies and changes during the execution of the cell rule due to the usage of a hash function. If the amount of agents is smaller than the capacity of one data memory all agents could concentrate in one data memory. In that case one processor has to execute all agents while the other processors are idle. This situation depends on the hash function, the size of the data memories and the amount of agents. It also varies during the execution, so it is probably unlikely to happen during multiple generations. This circumstance needs further investigations. Using a cell rule that does not generate new agents it is guaranteed that all agents can be stored at any time. If a cell rule is used that generates new agents during execution, the programmer has to check that the total amount of agents does not exceed the total memory capacity (size of all data memories) at any time.

The general processing steps are:

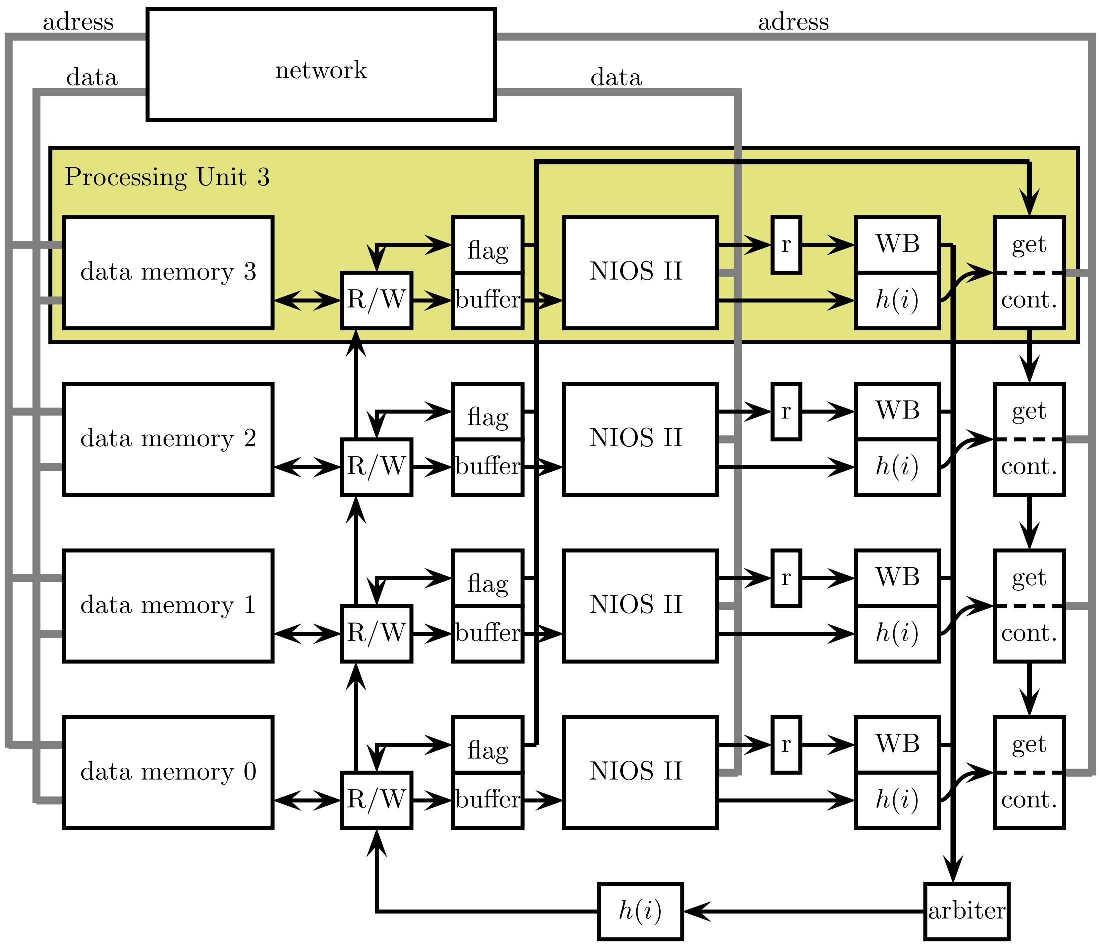

Figure 5: Architecture with hash functions (HA) for $p = 4$. Active cells are stored in the data memories. NIOS II/f processor with 512 Byte instruction cache, hardware multiplication and hardware division.

1. Search for the data in the data memory (R/W-Arbiter) and store it in a local buffer.
2. Load an agent from the local buffer into the NIOS II processor.
3. Check neighbor cells by using the hash function in conjunction with the get and contains functions.
4. Write the new agent into a local write buffer.
5. The data in the buffers is written back by a write arbiter.

The read unit of the R/W arbiter searches in the memory for active cells and transfers them into a buffer. A flag field indicates which memory location is actually holding valid data, avoiding unnecessary memory accesses and speeding-up the search process. The processor reads the data out of the local buffer and executes the cell rule. Thus reading data out of the memory is done in parallel to the computation, and the processor does not have to wait for the search process. After a cell has been processed the new cell data is written into a register (r in Fig. 5). All registers are transferred into a write buffer by a dedicated custom instruction. A write arbiter fetches the data of all processors out of the write buffer. The cell index is transformed into a valid write address by the hash function. The cell state is then stored at this address. External read accesses are more complex. As not all cells are stored in the data memories a neighbor access first has to check if the cell exists. The processor initiates the cell check. The read address of the cell is transformed by the hash function. A CONTAINS hardware function (cont. in Fig. 5) checks the flag field for data at the corresponding address. If the flag is not set the cell is not stored in any of the data memories. If the flag is set the data has to be read via the network using the GET hardware function (*get* in Fig. 5). After the cell has been read, the index of the cell has to be compared with the index of the requested cell. Multiple read operations might be necessary if rehashes had occurred during the write operation. As the cell index is not equal to the memory address the cell index has to be stored in the data memory. During external read operations the stored cell index always has to be compared with the requested index.

5.2 Step-by-Step Sequence of the Process

Given an agent A at cell (a, b) where (a, b) equals cell index c. Let's assume that A is stored in data memory 3 at memory location 5 (initial configuration).

The R/W arbiter of PU 3 scans the flag register of PU 3 for all memory positions and finds a flag for memory location 5. The agent stored at that position is copied out of the memory into the local buffer. Stored alongside with the agent is the cell index c. The NIOS II processor reads the data out of the buffer. With the information about the agent type and the cell index c the agents behavior is determined. As an example the agent might want to move from the current cell c to the neighbor cell $c + 1$. In this case cell $c + 1$ and the neighbor cells of cell $c + 1$ need to be checked (to avoid collisions and to maintain data consistency). Let's just consider how cell $c+1$ is checked. The NIOS II processor sends the cells index $(c + 1)$ through the hash function. The result of the hash function is a global memory address (gma) consisting of data memories address (lower bits) and the index of the PU (upper bits). The amount of bits used for the upper part is $log_2\,(\#PUs)$ where # PUs is the amount of PUs. The amount of lower bits depends on the size of the data memory. Using the CONTAINS function first the flag register of the gma is checked. If it is set the gma is used to determine the right data memory via the network and the right location within the data memory. It is then verified if the agent stored at that location belongs there or if another agent has been stored there due to rehashes. Further accesses might be necessary. The GET function is then used to load the actual agents data (e.g. direction, type, ...). With that information the Agent A can determine its new position. The new data of Agent A are stored at the new cell index d. The data is transfered through the register r into the WB buffer. An arbiter loads the data out of the WB buffers and transformes the cell index d into a gma. Again, the upper bits of the gma are used to determine the PU and the lower bits are used to determine the memory location.

5.3 Generation Transition, Flag Handling

After each processor has processed all cells in its data memory the processor needs to be synchronized with all other processors. This synchronization is done with a barrier-synchronization technique. During the synchronization phase the memories are switched from the current generation to the next generation. The flag field of the current generation has to be reset to indicate that all memory locations are empty. The flag field has been implemented as registers in order to be able to reset the complete flag field within one clock cycle.

5.4 Hash Function

The HA uses an agent based approach. The processors do not calculate one cell after another and therefore can not access each cell by its address. The processors only calculate the agents. In order to be able to check neighbor cells, the cells address first has to be transformed by a hash function to get the real memory address. A hash function is used to resolve the addresses, as the total amount of data memory is reduced and does not store empty cells. The hash function converts the cell index into a memory address. The cell is stored at that address if the memory location is free. If the memory location is occupied, a rehash has to be executed determining a new memory address. Good results for the agent behavior rule (section 4) have been achieved with the following hash function: $h(i) = (i + 2 \cdot n) \bmod size$, where n is incremented for every rehash for a cell with cell index i. $size$ denotes the size of all data memories together (number of all data memory locations). The mod operation is realized by using the least significant bits only. These least significant bits are sufficient to address any of the data memories. Thus, the upper bits determine the PU while the lower bits determine the right memory position of the data memory. The amount of bits depend on the size of the data memories.

5.5 FPGA Prototype Implementation

Table 1: Multiprocessor architecture with hash function. Resources and clock frequency for the Cyclone II FPGA

PUs	LEs	network LEs	memory bits	reg. bits	clock (MHz)
1	13,487	-	153,664	3,802	75.00
2	14,993	62	186,464	5,059	76.19
4	22,579	85	252,064	7,576	70.84
8	103,333	143	383,264	12,614	can't fit

The prototyping platform was a Cyclone II FPGA [3] with the Quartus II 8.1 synthesis software from Altera. The Cyclone II FPGA contains 68,416 logic elements (LE) and 1,152,000 RAM bits [3]. The implementation language was Verilog HDL. The NIOS II processor was built with the SOPC-Builder. The synthesis settings were optimized for speed. Tab. 1 shows the numbers of logic elements (LE), logic elements for network, the memory usage and the maximum clock frequency. Scalability is limited by the use of the flag registers which results in a high usage of combinatorial logic.

Using the Cyclone II FPGA platfom only up to four processor could be configured. In order to decrease the implementation resources and to increase the number of softcore processors a second architecture (DAMA) was designed.

5.6 Simulation Results

The simulation results of the MAS on the HA are shown in Tab. 2. The cycle speed-up is slightly higher compared to the real speed-up as the clock rate degrades. With the four processors that could be realized on the Cyclone II FPGA a real speed-up of 2.27 could be reached.

Table 2: Simulation results of the MAS on the HA

processing units (PUs)	cycles per generation	cycle speed-up	execution time per generation (ms)	real speed-up
1	30,221	-	0.403	-
2	18,107	1.67	0.238	1.70
4	12,600	2.40	0.178	2.27

6 Dedicated Agent Memory Architecture (DAMA)

6.1 System Overview

The architecture consists of multiple NIOS II processors each with its own program memory and two data memories (Fig. 6). One data memory (*agent memory*) holds the agent data, the other memory (*cell field memory*) holds pointers to the agent memory. The agents are distributed *equally* among all agent memories. Each cell field memory holds a part of the overall cell field. Depending on the application additional hardware functions can be implemented to enhance the architecture. For our test application the function (*check four neighbors*) has been added in order to accelerate the simulation. All these components form a processing unit PU (Fig. 6). The complete architecture consists of p PUs connected through a ring network.

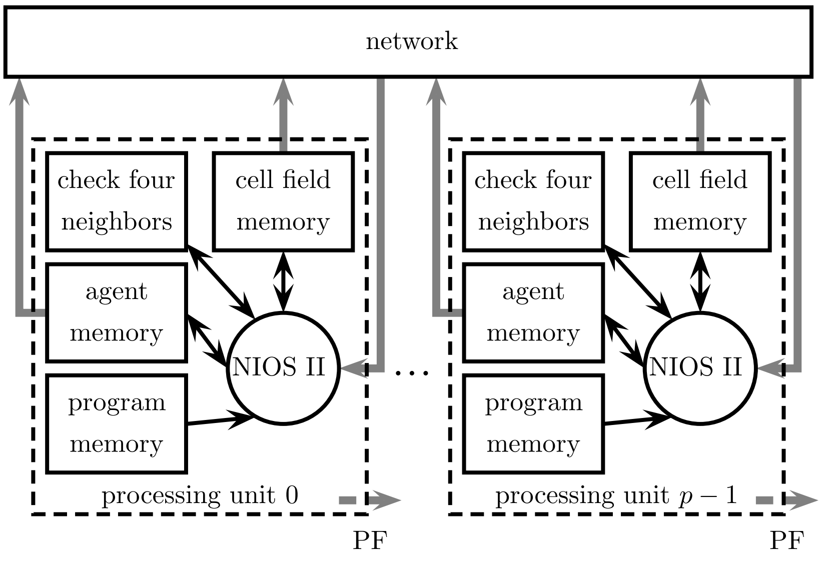

Figure 6: Dedicated agent memory architecture (DAMA). Agents are stored in the agent memory. Pointers to the agents are stored in the cell field memory. A special hardware function checks the movement condition. PU_i can forward a pointer to PU_{i+1} using Pointer Forwarding (PF). NIOS II/f processor with 512 Byte instruction cache, hardware multiplication and hardware division.

The components of a PU are in detail:

- **agent memory:** The agent memory holds the actual data of each agent. This includes the position (x,y) on the cell field. Other data may be stored for each agent if needed (e.g. agent type, agent direction).
- **cell field memory:** The cell field indicates which cell is actually occupied by an agent, and if so, it stores a pointer to the agent in one of the p agent memories. The cell field does not contain any agent data, it only holds pointers to the agent memory locations.
- **program memory:** The program memory holds the cell rule that is applied to each agent. Depending on the agents' state and the states of the selected neighbors, the next state of the agent is calculated.
- **check four neighbors:** A dedicated hardware function that checks the four neighbor cells in moving direction of an agent (Fig. 3). This hardware avoids multiple hardware software transitions and therefore speeds up the computation of the cell rule.

The hardware function *check four neighbors* is an example that is useful for agent simulation. Further commonly used functions have to be detected and added to the architecture. As described in section 3 these functions can be implemented in software or hardware and can be provided to the programmer as a system library. Missing functions can be implemented within the cell rule.

The simulation of a MAS is executed by the following steps:

(1) An agent is loaded out of the agent memory into the NIOS II processor.

(2) The destination cell for the agent is determined by checking the neighbor cells (function: *check four neighbors*).

(3) The position and direction of the agent has to be updated in the agent memory.

(4) The pointer to the agent in the cell field has to be updated.

The cell field memories are used to determine if a cell is occupied by an agent, and if so, where to find that agent. For occupied cells, accesses to the agent memory are possible (e.g. to check the agents direction). The new destination cell of an agent requires two memory updates. First the position of the agent has to be updated in the agent memory. Secondly the cell field has to be updated. If the agent has moved to a cell that is handled in the cell field of another PU the write access has to be forwarded to the next PU (Pointer Forwarding (PF) as shown in figure 6). Write accesses are forwarded until they reach the PU that is responsible for that write access. The processor does not have to wait for a write access to finish. The write accesses are buffered in each PU an forwarded whenever possible. No further processor actions are necessary to fully execute the write process. Note that only the pointer information for the cell field memory is forwarded during a write operation. The actual agent data is kept in the agent memory and does not move across the PUs.

6.2 Custom Instructions

The NIOS II CPU has been enhanced by several custom instructions (CI). All custom instructions define a function call and a swap from software to hardware and back. The custom instructions enrich the NIOS II instruction set with special functions implemented in hardware. This way the additional instrictions such as reading a neighbor cell or generation synchronization are realized.

The following custom instructions have been added to the architecture:

- **NEXTGEN**: generation synchronization of all PU's. All PU's are synchronized by AND-gating.
- **WR_B{X}**: writes the data of block X into a hardware register.
- **WR_SEND**: transfers all hardware registers written by WR_B{X} into the agent memory.
- **RD_NXT_AGT_B{X}**: Read the data of block X of the current agent (internal access).
- **GET_NXT_AGT**: Set the address of the agent to be calculated. The agent's data can be read using the RD_NXT_AGT_B{X} functions. The function is used to access all agents stored in associated PU.
- **RD_B{X}**: Reads the data block of an agent. Automatic decision between internal (memory and processor are in the same PU) and external accesses using the network (memory and processor are in different PU).
- **RD_AGT_ADR**: Reads the link information from the cell field memory.
- **CHECK_CELL**: Calls the hardware function to check the four neighbors of a cell. Returns the amount of agents {0,1,2,3,4} whose destination is that cell.

6.3 NIOS II Cell Rule / C-Code

The NIOS II cell rule implements the agent behavior and is shown in Listing 1. For the used application two blocks are used. The first block is used to store agents position. The second block is used to store the agent type. This architecture does not store empty cells and only calculates the agents movements. Therefore the agents are loaded one after another out of the agent memory[1]. The address of the agent memory is first checked for validity (using the *GET_NXT_AGT* custom instruction). If an agent is stored at that address, the agents data is preloaded for easy and fast access through the custom instructions *RD_NXT_AGT_B0* and *RD_NXT_AGT_B1*. The software implementation of the *CHECK_CELL* custom instruction is shown in Listing 2. It substitutes lines 24 to 27 in Listing 1. The *AGENT_NR* only holds the address of the local agent memory. It does not include an processor index. But the addresses stored in the cell field memory consist of an processor index and the address for the agent memory. The pointer or address *AGENT_NR* is used to go through the local memory and to process the agents one by one. The pointer or address *NEW_AGENT_ADR* consists of an index and address part. With *NEW_AGENT_ADR* a memory location in one of the cell field memories can be addressed. Inside the cell field memories pointers to the agent memories are stored. The pointers stored in the cell field memory consist of an index and address part.
RD_AGT_ADR reads the pointer stored at *NEW_AGENT_ADR* out of one of the cell field memories. There are no further accesses encapsulated inside *RD_AGT_ADR*. *CHECK_CELL* hides multiple *RD_AGT_ADR* and *RD_B*$\{X\}$ accesses. A software implementation of these accesses is shown in Listing 2.

Listing 1: Agent behavior rule implementing the agent behaviour (DAMA)

```
1  int main(){
2   int g, cond;
3   int AGENT_NR, NEW_AGENT_ADR;
4   int AGENT_BLOCK0, AGENT_BLOCK1;
5
6   CI(NEXTGEN,0,NXTG_KEEP);                        //startup generation synchronization
7   CI(NEXTGEN,0,NXTG_KEEP);                                       //keep data in memory
8
9   for(g=0;g<GENS;g++){
10   AGENT_NR=0;                                          //begin with first local agent
11   while(CI(GET_NXT_AGT,AGENT_NR,0)!=0)                    //check if address is valid
12   {
13    AGENT_BLOCK0 = CI(RD_NXT_AGT_B0,0,0);               //read block 0 = agent address
14    AGENT_BLOCK1 = CI(RD_NXT_AGT_B1,0,0);                  //read block 1 = agent type
15
16    if(AGENT_BLOCK1>CELL_BLOCK){                                     //if cell is agent
17     switch(AGENT_BLOCK1){                        //calculate destination cell address
18      case CELL_AGENT_N: NEW_AGENT_ADR = AGENT_BLOCK0-MAXY; break;
19      case CELL_AGENT_S: NEW_AGENT_ADR = AGENT_BLOCK0+MAXY; break;
20      case CELL_AGENT_E: NEW_AGENT_ADR = AGENT_BLOCK0+1;    break;
21      case CELL_AGENT_W: NEW_AGENT_ADR = AGENT_BLOCK0-1;    break;
22     }
23
24     if(CI(RD_AGT_ADR,NEW_AGENT_ADR,0)==-1)           //check destination neighbors
25      cond=CI(CHECK_CELL,NEW_AGENT_ADR,MAXY);                  //check four neighbors
26     else
27      cond=999;
28
29     if(cond>1){                                                     //do not move agent
30      if(AGENT_BLOCK1==CELL_AGENT_W)
31       AGENT_BLOCK1=CELL_AGENT_N;
```

[1]Other processing techniques are also supported

```
     else
      AGENT_BLOCK1+=2;

     CI(WR_B0,0,AGENT_BLOCK0);                              //write agent address
     CI(WR_B1,0,AGENT_BLOCK1);                                 //write agent type
     CI(WR_SEND,AGENT_BLOCK0,0);             //write address to cell field memory
    }
    else{                                       //move agent to destination cell
     switch(AGENT_BLOCK1){
      case CELL_AGENT_N: AGENT_BLOCK1=CELL_AGENT_E; break;
      case CELL_AGENT_S: AGENT_BLOCK1=CELL_AGENT_W; break;
      case CELL_AGENT_E: AGENT_BLOCK1=CELL_AGENT_N; break;
      case CELL_AGENT_W: AGENT_BLOCK1=CELL_AGENT_S; break;
     }

     CI(WR_B0,0,NEW_AGENT_ADR);                             //write agent address
     CI(WR_B1,0,AGENT_BLOCK1);                                 //write agent type
     CI(WR_SEND,NEW_AGENT_ADR,0);            //write address to cell field memory
    }
   }
   else{                               //treat all other cell types as obstacles
    CI(WR_B0,0,AGENT_BLOCK0);                               //write agent address
    CI(WR_B1,0,AGENT_BLOCK1);                                  //write agent type
    CI(WR_SEND,AGENT_BLOCK0,0);              //write address to cell field memory
   }

   AGENT_NR++;                                          //increment agent counter
  }

  CI(NEXTGEN,0,NXTG_DELETE);      //generation synchronization, delete old memory
 }

 return 0;}
```

Listing 2: Software implementation of the custom instruction CHECK_CELL

```
if(CI(RD_AGT_ADR,NEW_AGENT_ADR,0)==-1)           //check destination neighbors
{
 cond=0;                                 //check neighbors of destination cells
 TEMP_AGENT=CI(RD_AGT_ADR,NEW_AGENT_ADR-MAXY,0);
 if(TEMP_AGENT!=-1 && CELL_AGENT_S==CI(RD_B1,TEMP_AGENT,0))              //north
 {cond++;}

 TEMP_AGENT=CI(RD_AGT_ADR,NEW_AGENT_ADR+MAXY,0);
 if(TEMP_AGENT!=-1 && CELL_AGENT_N==CI(RD_B1,TEMP_AGENT,0))              //south
 {cond++;}

 TEMP_AGENT=CI(RD_AGT_ADR,NEW_AGENT_ADR+1,0);
 if(TEMP_AGENT!=-1 && CELL_AGENT_W==CI(RD_B1,TEMP_AGENT,0))               //east
 {cond++;}

 TEMP_AGENT=CI(RD_AGT_ADR,NEW_AGENT_ADR-1,0);
 if(TEMP_AGENT!=-1 && CELL_AGENT_E==CI(RD_B1,TEMP_AGENT,0))               //west
 {cond++;}
}
else
{cond=999;}
```

6.4 FPGA Prototype Implementation

For the prototype implementation the same FPGA, synthesis software and settings as in section 5.5 have been used. Tab. 3 shows the numbers of logic elements (LE), logic elements for network, the memory usage and the maximum clock frequency for up to 16 processors.

Table 3: Dedicated Agent Memory Architecture. Resources and clock frequency for the Cyclone II FPGA

PUs	LEs	network LEs	memory bits	reg. bits	clock (MHz)
1	3,748	-	137,952	2,075	138.89
2	7,121	355	169,376	3,807	131.25
4	13,611	592	232,224	7,025	105.00
8	26,395	1,101	357,620	13,441	90.91
16	52,330	2,230	609,312	26,284	71.88

In contrast to the implementation results of the HA (section 5.5) the resource usage is noticeably lower. Comparing the synthesis results for $p = 8$ shows that the HA needed 103,333 LEs whereas the DAMA needs 26,395 LEs.

6.5 Simulation Results

The simulation results of the MAS on the DAMA are shown in Tab. 4. Compared to the HA the real speed-up for four processors increased. The better scalability of the DAMA allows to fit 16 processors onto the Cyclone II FPGA reaching a real speed-up of 4.17 using 16 processors. The cycle speed-up is rather low as the architecture only handles the agents and therefore is already quite efficient for one processor. The distribution of the MAS (distributed cell field memories) as well as the distribution of the agent memories among the PUs introduces a high load onto the interconnecting network. With the increasing number of PUs the amount of agents per PU decreases, the amount of network accesses increases and the possible wait cycles for a generation synchronization increase.

Table 4: Simulation results of the MAS on the DAMA

processing units (PUs)	cycles per generation	cycle speed-up	execution time per generation (ms)	real speed-up
1	21,586	-	0.155	-
2	11,466	1.88	0.087	1.78
4	6,494	3.32	0.062	2.52
8	3,922	5.50	0.043	3.60
16	2,679	8.06	0.037	4.17

6.6 Performance of Check Four Neighbors

The performance of the DAMA has been evaluated with and without the *check four neighbors* hardware function. The results are shown in table 5. Adding the hardware function also influences the maximum reachable clock rate. Comparing the execution times a gain between 20% and 7% can be reached. The difference for one processing unit is high because a higher clock rate could be reached using the hardware function which additionally accelerates the execution. But for two processing units the clock rate for the architecture using the hardware function is lower. Still, a gain of 12% could be reached.

For a higher number of processing units the gain decreases. This is due to the fact, that the gain does not compare the acceleration of only the function (hardware vs. software). It shows the acceleration of the whole cell rule. Therefore the additional times needed for synchronizations or more network read accesses are included. The synchronization also always has to ensure to wait for the slowest processing unit, which is also included in these times.

Another fact is, that the hardware function uses network read accesses. These accesses generated different execution times for any processing unit, whether the function is provided in software or hardware. So the execution of the function is not constant.

Table 5: Evaluation of the performance of the check four neighbors hardware function. The gain shows the gain of using the hardware function *check fours neighbors* comparing the execution times per generation

	without check four neighbors			with check four neighbors			
PUs	clock (MHz)	cycles per generation	execution time (ms) per generation	clock (MHz)	cycles per generation	execution time (ms) per generation	gain
1	130.01	2,538,572	0.195	138.89	2,158,566	0.155	20 %
2	135.01	1,340,019	0.099	131.25	1,146,565	0.087	12 %
4	105.01	748,349	0.071	105.00	649,371	0.062	13 %
8	90.91	439,615	0.048	90.91	392,219	0.043	11 %
16	71.88	289,294	0.040	71.88	267,863	0.037	7 %

6.7 Performance with a Varying Amount of Agents

The MAS has been simulated with a varying number of agents. So far the MAS contained 185 agents in a closed agent world. The border has been realized using 176 obstacles. Additionally the MAS has been simulated with 20, 50 and 100 agents on the DAMA using 16 processing units. In this simulation the execution time increases sublinear with the number of agents (table 6). The amount of obstacles bordering the agent world are fix for all simulations.

Table 6: Performance of the DAMA with a varying amount of agents.

agents	20	50	100	185
time per generation [ms]	0.012	0.015	0.023	0.037

7 Performance and Comparison

In order to compare the performance of the two architectures (HA, DAMA) the *cell-update rate* (*CUR*) was calculated (Fig. 7). *CUR(d)* is the number of cells updated per millisecond for a certain density d (0...100%) of agents and obstacles in the cell field. The results were evaluated for the application described in section 4 for a cell field of size 45×45, 185 agents and 176 obstacles ($d = 17,8\%$).

The results are shown in Fig. 7. The DAMA performed best and reached a gain $\left(\frac{CUR(d)_{DAMA}}{CUR(d)_{HA}}\right)$ between 2.6 and 2.9 compared to the HA (for up to four processors). Furthermore the DAMA can be used with a higher number of processors. The HA was restricted to four processors on the used FPGA platform.

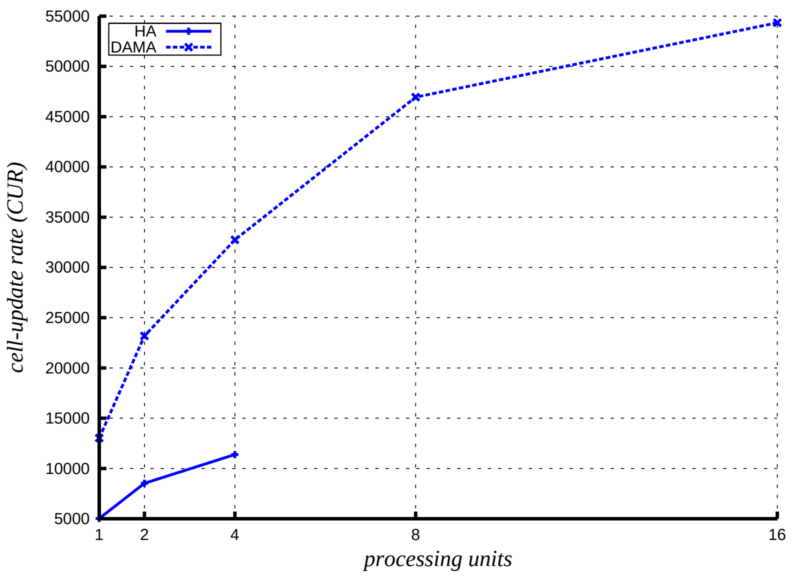

Figure 7: Cell-update rate for the two architectures: hash architecture (HA) and dedicated agent memory architecture (DAMA).

8 Investigating Clock Rate Degradation

One of the main concerns about the architectures is the decreasing clock rate. Adding more processors to the architecture results in a lower clock rate so that at some point the performance will no longer increase or might even start to decrease. To be able to use multiple NIOS II processors the instructions about creating multiprocessor NIOS II architectures have been followed [2, 8]. This way each of the processors can be programmed individually through the NIOS II IDE [2]. Different alternative approaches have been examined to create the architectures trying to reach a more stable clock rate even with a higher number of processors. To reduce any negative effect that might be introduced by creating all NIOS II processors within one SOPC project only one processor will be created in the SOPC project. Then multiple instances of that project are generated. With this method all processors are identical and only one processor can be implemented using the NIOS II IDE. All processors instruction memories are intialized with the same code. This is no problem for architectures based on the GCA model. But to enable the execution of the same programm onto different memory addresses, an additional custom instruction can be added to load an individual processor *id*. For the DAMA this is not necessary, as an agent based approach was chosen. The results of the architecture are now as shown in Tab. 7. The clock rate is now more stable and only decreases in the range of a few MHz. For the NIOS II IDE all processors are identical and not distinguishable from one another. Therefore it is no longer possible to load the cell rule into the programm memory using JTAG which used a certain amount of memory bits. The total amount of memory bits is lower compared to the previous implementation.

The simulation results using the architecture with the higher clock rates (Tab. 7) are shown in Tab. 8. With 16 processors a real speed-up of 7.35 can be reached compared to the real speed-up of 4.17 that was reached before.

8.1 Simulating a Higher Number of PUs

With the used FPGA, systems with up to 16 processors can be evaluated. As it is also interesting to determine the performance and scalability of the architecture, systems with more processors have been simulated using ModelSim Altera V6.4 [1, 7]. It has to be mentioned that not all parts of the

[2]Now replaced by the NIOS II Software Build Tools for Eclipse

Table 7: Dedicated Agent Memory Architecture. Resources and clock frequency for the Cyclone II FPGA with a more stable clock rate.

PUs	LEs	network LEs	memory bits	reg. bits	clock (MHz)
1	2,869	-	96,960	1,628	140.02
2	6,187	331	120,192	3,391	135.01
4	12,207	614	166,656	6,662	135.01
8	24,184	1,154	259,584	13,181	131.25
16	48,064	2,332	445,440	26,220	127.78

Table 8: Simulation results on the DAMA with higher clock rates

processing units (PUs)	cycles per generation	cycle speed-up	execution time per generation (ms)	real speed-up
1	21,586	-	0.154	-
2	11,466	1.88	0.085	1.82
4	6,494	3.32	0.048	3.21
8	3,922	5.50	0.030	5.16
16	2,679	8.06	0.021	7.35

NIOS II processor are initialized the same way as if it is realized on an FPGA. Therefore the amount of clock cycles needed in the simulation is slightly higher. The resource usage has been determined for a Stratix V FPGA (5SGSMB8) using Quartus II V10.1 are shown in Tab. 9. Therefore the overall clock rate for the same amount of processors is higher. The amount of logic needed for the network is shown in the network ALUT column, already included in the total amount of resources shown in the ALM and ALUT column.

Table 9: Dedicated Agent Memory Architecture. Resources and clock frequency for the Stratix V FPGA

PUs	ALMs	ALUTs	network ALUTs	memory bits	reg. bits	clock (MHz)
1	1,875	1,712	-	96,960	1,674	216.73
8	15,327	13,997	670	259,584	13,475	166.67
16	30,468	27,822	1,275	445,440	26,785	150.02
32	60,206	55,281	2,560	817,152	53,375	120.00
64	119,854	110,446	5,158	1,560,576	106,493	107.15
128	239,699	221,675	10,404	3,047,424	212,603	95.47

Tab. 10 shows the clock cycles for up to 128 processors. The best performance can be achieved with 64 processors. The amount of clock cycles starts to increases for 128 processors. With 128 processors the amount of agents per processor is low and therefore external read accesses are more often. External read accesses also need more clock cycles for each read access to accomplish. Therefore architectures with a large amount of processors are possibly only efficient if the amount of agents per processor fulfills a certain minimum.

Table 10: ModelSim simulation results using the DAMA

processing units (PUs)	cycles per generation	cycle speed-up	execution time per generation (ms)	real speed-up
1	21,644	-	0.100	-
8	4,199	5.16	0.025	3.96
16	3,098	6.99	0.021	4.84
32	2,558	8.46	0.021	4.68
64	2,358	9.18	0.022	4.54
128	2,698	8.02	0.028	3.53

9 Conclusion

The presented hash architecture HA shows a good performance for multi-agent simulation even for a low number of processors. It overcomes the need of simulating all cells and therefore minimizes the overhead. Another advantage of the HA is that it is independent of the world size and only depends on the amount of agents. The resource usage is high due to the flag control logic. This also reduces the scalability of the architecture. Therefore the maximum amount of processors on the Cyclone II is four.

The second architecture DAMA uses an agent memory (holding the agents status) and a cell field memory holding the pointers to the agents. The architecture does not only scale better compared to the hash architecture, it is also significantly faster. Due to the use of the cell field memory the architecture is not independent of the world size. The performance of the DAMA was significantly higher compared to the hash architecture. The DAMA reached a gain between 2.6 and 2.9 compared to the HA.

The two architectures allow to program agents intuitively. They also minimize the overhead of simulation which increases the performance. Additionally specialized hardware functions that are commonly used for multi-agent simulations were added to the architectures.

The clock rate of the DAMA can be kept high using an alternative approach in generating the NIOS II processors. The DAMA with 16 processors is then 1.76 times faster. A simulation for up to 128 processors shows that the amount of clock cycles starts to increase for more than 64 processors. This effect is due to the increasing network, increasing amount of network accesses (external read) and the decreasing amount of agents per processor. Simulation of different MAS using a varying amount of agents will be done in future investigations.

References

[1] Altera Corporation. *Simulating Nios II Embedded Processor Designs*, November 2008. AN 351.

[2] Altera Corporation. *Creating Multiprocessor Nios II Systems Tutorial*, February 2010. Document Version: 1.4.

[3] Altera, Datasheet Cyclone II. `http://www.altera.com/literature/hb/cyc2/cyc2_cii5v1.pdf`, 2006.

[4] Altera, NIOS II Website. `http://www.altera.com/products/ip/processors/nios2/ni2-index.html`, 2009.

[5] Alexander Keewatin Dewdney. Sharks and fish wage an ecological war on the toroidal planet Wa-Tor. *Scientific American*, 251(6):14–22, Dezember 1984.

[6] R.M. D'Souza, M. Lysenko, and K. Rahmani. SugarScape on Steroids: Simulating over a Million Agents at Interactive Rates. In *Proceedings of Agent 2007 Conference*, Chicago, IL, 2007.

[7] Ray Duran. Practical Hardware Debugging: Quick Notes On How to Simulate Altera's Nios II Multiprocessor Systems Using Mentor Graphics' ModelSim, March 2007. Staff Design Specialist FAE, Altera Corporation.

[8] Omar Elkeelany. Implementing a Multiprocessor System on an FPGA using Altera Nios II Processors & SOPC, 2008. http://iweb.tntech.edu/oelkeelany/6170/lectures/multi.pdf.

[9] Rolf Hoffmann, Klaus-Peter Völkmann, and Stefan Waldschmidt. Global cellular automata GCA: an universal extension of the CA model. In *ACRI 2000 "work in progress" session, Karlsruhe, Germany*, 2000.

[10] Rolf Hoffmann, Klaus-Peter Völkmann, Stefan Waldschmidt, and Wolfgang Heenes. GCA: Global Cellular Automata, A Flexible Parallel Model. In *Proceedings of: 6th International Conference on Parallel Computing Technologies PaCT2001, Novosibirsk, Russia, 3. bis 7. Sept., 2001*, Lecture Notes in Computer Science (LNCS 2127), Springer Verlag, 2001.

[11] J. Jendrsczok, P. Ediger, and R. Hoffmann. A scalable configurable architecture for the massively parallel GCA model. *Int. J. Parallel Emerg. Distrib. Syst.*, 24(4):275–291, 2009.

[12] Johannes Jendrsczok, Rolf Hoffmann, and Thomas Lenck. Generated horizontal and vertical data parallel gca machines for the n-body force calculation. In *ARCS '09: Proceedings of the 22nd International Conference on Architecture of Computing Systems*, pages 96–107, Berlin, Heidelberg, 2009. Springer-Verlag.

[13] Kai Nagel and Michael Schreckenberg. A cellular automaton model for freeway traffic. *Journal de Physique I*, 2(115):2221–2229, 1992.

[14] Christian Schäck, Wolfgang Heenes, and Rolf Hoffmann. A Multiprocessor Architecture with an Omega Network for the Massively Parallel Model GCA. In Koen Bertels, Nikitas J. Dimopoulos, Cristina Silvano, and Stephan Wong, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation, 9th International Workshop, SAMOS 2009, Samos, Greece, July 20-23*, volume 5657 of *Lecture Notes in Computer Science*, pages 98–107. Springer Berlin / Heidelberg, 2009. ISSN 0302-9743 (Print) 1611-3349 (Online).

[15] Christian Schäck, Wolfgang Heenes, and Rolf Hoffmann. GCA Multi-Softcore Architecture for Agent Systems Simulation. In Erik Maehle Stefan Fischer, editor, *Informatik 2009 Im Focus das Leben*, volume P-154 of *Lecture Notes in Informatics*, pages 278; 2268–82, 2009. ISSN 1617-5468.

[16] Christian Schäck, Wolfgang Heenes, and Rolf Hoffmann. Network Optimization of a Multiprocessor Architecture for the Massively Parallel Model GCA. In *22. PARS Workshop, Gesellschaft für Informatik (GI)*, volume 26, pages 48–57, 2009. ISSN 0177-0454.

[17] Christian Schäck, Rolf Hoffmann, and Wolfgang Heenes. Efficient Traffic Simulation using the GCA Model. In *24th IEEE International Parallel and Distributed Processing Symposium, Workshops and Phd Forum IPDPSW*, April 2010. 12th Workshop on Advances in Parallel and Distributed Computational Models (APDCM), IEEE Catalog Number: CFP1051J-CDR.

[18] Thomas Schmickl, Ronald Thenius, and Karl Crailsheim. Kollektive Sammel-Entscheidungen: Eine Multi-Agenten-Simulation einer Honigbienenkolonie. *Entomologica Austriaca*, 13:15–24, März 2006. ISSN 1681-0406.

[19] David Strippgen and Kai Nagel. Multi-Agent Traffic Simulation with CUDA. In *High Performance Computing & Simulation (HPCS)*, pages 106–114, Leipzig, Germany, 2009.

[20] David Strippgen and Kai Nagel. Using common graphics hardware for multi-agent traffic simulation with CUDA. In Olivier Dalle, Gabriel A. Wainer, L. Felipe Perrone, and Giovanni Stea, editors, *Simutools '09: Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, pages 1–8, ICST, Brussels, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[21] Ronal Thenius, Thomas Schmickl, and Karl Crailsheim. Einfluss der Individualität bei Sammelbienen (Apis mellifera L) auf den Sammelerfolg. *Entomologica Austriaca*, 13:25–29, März 2006.

[22] John Tran, Don Jordan, and David Luebke. New challenges for cellular automata simulation on the gpu, August 2004. ACM Workshop on General Purpose Computing on Graphics Processors.

[23] Luděk Žaloudek, Lukš Sekanina, and Vclav Šimek. Accelerating cellular automata evolution on graphics processing units. *International Journal on Advances in Software*, 3(1):294–303, 2010.

[24] Shiwu Zhang and Jiming Liu. A Massively Multi-agent System for Discovering HIV-Immune Interaction Dynamics. In *Massively Multi-Agent Systems I*, volume 3446, pages 161–173. Springer Verlag, 2005. ISSN 0302-9743 (Print) 1611-3349 (Online).