A Multiprocessor Architecture for the Massively Parallel Model GCA

Wolfgang Heenes, Rolf Hoffmann, and Johannes Jendrsczok

TU Darmstadt, FB Informatik, FG Rechnerarchitektur
Hochschulstraße 10, D-64289 Darmstadt, Germany
Phone +49 6151 16 {5312, 3606}, Fax +49 6151 16 5410
{heenes, hoffmann, jendrsczok}@ra.informatik.tu-darmstadt.de

Abstract

The GCA (Global Cellular Automata) model consists of a collection of cells which change their states synchronously depending on the states of their neighbors like in the classical CA model. In differentiation to the CA model the neighbors are not fixed and local, they are variable and global. The GCA model is applicable to a wide range of parallel algorithms. In this paper a multiprocessor architecture for the massively parallel GCA model is presented. In contrast to a special purpose implementation of a GCA algorithm the multiprocessor system allows the implementation in a flexible way through programming. The architecture mainly consists of a number of cell processors and a network. The cell processors are dedicated RISC processors, the network is a crossbar implemented with multiplexers. Only read-accesses through the network are necessary in the GCA model leading to a simplified structure. A system with 32 processors was implemented as a prototype on a FPGA. The analysis and implementation results have shown that the performance of the system scales very well with the number of processors.

1. Introduction

The GCA (Global Cellular Automata) model [8] is an extension of the classical CA (Cellular Automata) model [9]. In the CA model the cells are arranged in a fixed grid with fixed connections to their local neighbors. Each cell computes its next state by the application of a local rule depending on its own state and the states of its neighbors. The data accesses to the neighbors states are read-only and therefore no write conflicts can occur. The rule can be applied to all cells in parallel and therefore the model is inherently massively parallel. The CA model is suited to all kind of applications with local communication, like physical fields, lattice-gas models, models of growth, moving particles, fluid flow, routing problems, picture processing, genetic algorithms, and cellular neural networks.

The GCA model is a generalisation of the CA model which is also massively parallel. It is not restricted to the local communication because any cell can be a neighbor. Furthermore the links to the neighbors are not fixed; they can be changed by the local rule from generation to generation. Thereby the range of parallel applications is much wider for the GCA model. Typical applications besides the CA applications are graph algorithms, hypercube algorithms, logic simulation [10], numerical algorithms, communication networks, neuronal networks, games, and graphics.

The state of a GCA cell consists of a data part and one or more pointers (Fig. 1). The pointers are used to dynamically establish links to global neighbors. We call the GCA model one handed if only one neighbor can be addressed, two handed if two neighbors can be addressed and so on. In our investigations about GCA algorithms we found out that most of them can be described with only one link.

The aim of our research is the hardware and software support of this model. There are mainly three possibilities for an implementation.

1. **Fully Parallel Architecture.** A specific GCA algorithm is directly mapped into the hardware using registers, operators and hardwired links which may also be switched if necessary. The advantage of such an implementation is a very high performance [3], but the problem size is limited by the hardware resources and the flexibility to apply different rules is low.

2. **Partially Parallel Architecture with Memory Banks.** This architecture [5, 7] offers also a

1-4244-0054-6/06/$20.00

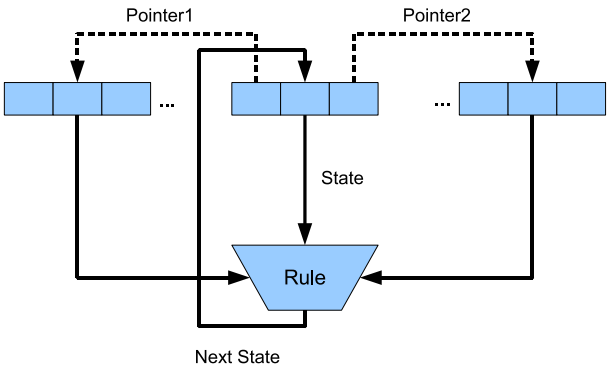

Figure 1. The operation principle of the GCA

high performance, is scalable and can cope with a large number of cells. The flexibility to cope with different rules is restricted.

3. **Multiprocessor Architecture.** This architecture [4] is not as powerful as the above two ones, but it has the advantage that it can be tailored to any GCA problem by programming. It allows also integrating standard or other computational models.

In this contribution we are presenting a multiprocessor architecture for the GCA model which was also implemented in FPGA logic.

2. A Multiprocessor Architecture

2.1. Design Goals

- The system shall consist of a master processor, p cell processors with local memories and an interconnection network (Fig. 2).
- Each cell processor can hold a part of the GCA cell field of the application.
- A cell processor can modify only its own cells in its local memory.
- Each cell processor has only read access to the other (external) cell processors, write accesses need not to be implemented due to the GCA model.
- The local GCA rule shall be programmable by processor instructions.
- The processor instructions shall support the accesses to the cells in the local memory the read accesses to external cells stored in the other processors.

The tasks of the master are

- initializing the cell processors with program and data.
- central control and synchronization.
- optionally supplying the cell processors with general parameters, counters or identical instructions.

The network interconnects the master and the cell processors. Depending on the type of the GCA algorithm the communication pattern between the cells can be simple (regular and symmetric) or complex (irregular and not symmetric). Therefore the network complexity depends on the complexity of the communication patterns which is needed for the class of GCA algorithms to be implemented. For many GCA algorithms [2] the communication pattern is rather simple which simplifies the design of the network. Simple or specialized networks can be implemented with multiplexers or fixed connections, complex networks have to be able to manage concurrent read accesses to arbitrary external memory locations. As in the GCA model a cell is not allowed to modify the contents of another cell, the network design is simplified because write accesses need not to be implemented.

2.2. Evaluation of the General Architecture

In order to get a feeling about the performance of such a GCA multiprocessor architecture a mathematical model was developed. The model takes into account the probabilities for internal (local) and external memory accesses and allows predictions upon the time for the computation of a cell rule depending of the number of processors.

- The number of cell values to be computed in one generation is N. Each cell has L (global) neighbors.

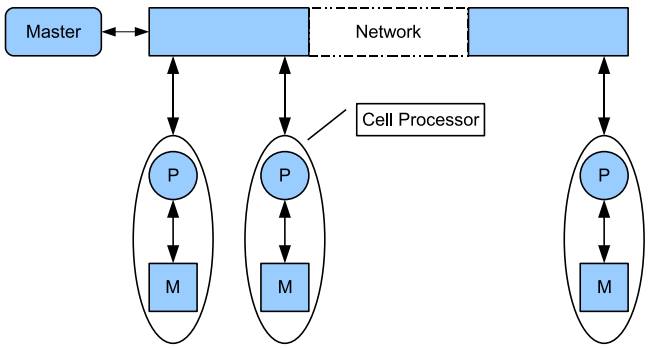

Figure 2. General System Architecture

- The number of cell processors is p.
- Each cell processor processes $n = N/p$ cells, where N can be divided by p without remainder.

The time to compute one generation (n cells in parallel) is $T = n \cdot t_{Rule}$, where t_{Rule} is the time to compute the local rule on a cell processor. The time t_{Rule} consists of the following parts:

- $t_{ReadSelf(internal)} = t_0$
  The time to read the cell data from the local/internal memory.
- $t_{ReadNeighbor}$
  The time to read the cell data of a neighbor. In the case the neighbor cell is in the internal memory the time shall be $t_{ReadNeighbor(internal)} = t_0$. In the other case if the neighbor is in an external memory (memory of another cell processor) the time shall be defined as $t_{ReadNeighbor(external)} = e(p) \cdot t_0$ meaning that it takes e(p) times longer than an internal access. Also the external access may increase with the number of processors p in a certain way.
- $t_{Compute} = c \cdot t_0$
  The time to compute the local rule.
- $t_{WriteSelf(internal)} = t_0$
  The time to store the resulting cell value in the internal memory.

The probability to hit a neighbor cell in the internal memory is

$$P(ReadNeighbor(internal)) = \frac{n}{N},$$

if the probability to access an arbitrary cell is equally distributed. Then the probability to access a neighbor which is located on another cell processor (external access) is

$$(1 - P(ReadNeighbor(internal))).$$

The average time to access a neighbor cell is then

$$\begin{aligned} t_{ReadNeighbor} &= t_{ReadNeighbor(internal)} \\ &\quad \cdot P(ReadNeighbor(internal)) \\ &\quad + t_{ReadNeighbor(external)} \\ &\quad \cdot (1 - P(ReadNeighbor(internal))). \end{aligned}$$

The time T to compute one generation with access to L neighbors will be

$$\begin{aligned} T &= (t_{ReadSelf(internal)} \\ &\quad + (t_{ReadNeighbor(internal)} \\ &\quad \cdot P(ReadNeighbor(internal)) \\ &\quad + t_{ReadNeighbor(external)} \\ &\quad \cdot (1 - P(ReadNeighbor(internal)))) \cdot L \\ &\quad + t_{Compute} + t_{WriteSelf(internal)}) \cdot \frac{N}{p}. \end{aligned}$$

With $P(ReadNeighbor(internal)) = \frac{1}{p}$ we get the following result

$$T = (1 + (\frac{1}{p} + e(p) \cdot (1 - \frac{1}{p})) \cdot L + c + 1) \cdot t_0 \cdot \frac{N}{p}.$$

Now the relative speed-up shall be evaluated. If only one processor is available, no external references are necessary and the formula for T can be simplified to

$$T = (2 + L + c) \cdot t_0 \cdot N.$$

In the other extreme case the number of cells is equal to the number of processors, the time to compute one generation is

$$T = (2 + e(p) \cdot L + c) \cdot t_0.$$

In the normal case, the number of cells is greater than the number of processors and the cells can be equally distributed to the processors, the relative speed-up will be

$$S(p) = \frac{(2 + L + c) \cdot p}{(2 + (\frac{1}{p} + e(p) \cdot (1 - \frac{1}{p})) \cdot L + c)}.$$

Fig. 3 shows the speed-up for the common case L=1 (one neighbor). The parameter $e(p)$ describes the cost of an external access relative to an internal access. In the case the external access cost is not a constant, the function $e(p) = 1 + h \cdot p$ (fixed cost plus incremental cost) was assumed as an example. The real cost of external accesses will depend on how efficient the interconnection network can handle the communication pattern desired by the GCA algorithm.

3. FPGA Prototype Implementation

A prototype of the multiprocessor architecture was designed and implemented for a FPGA (Altera Cyclone II) [1]. The system consists of p cell processors without a dedicated master. One of the cell processors takes over the tasks of the master. The cell processors are RISC processors. The processor design is simple compared to standard microprocessor because the resources or the FPGA were limited. The goal was to

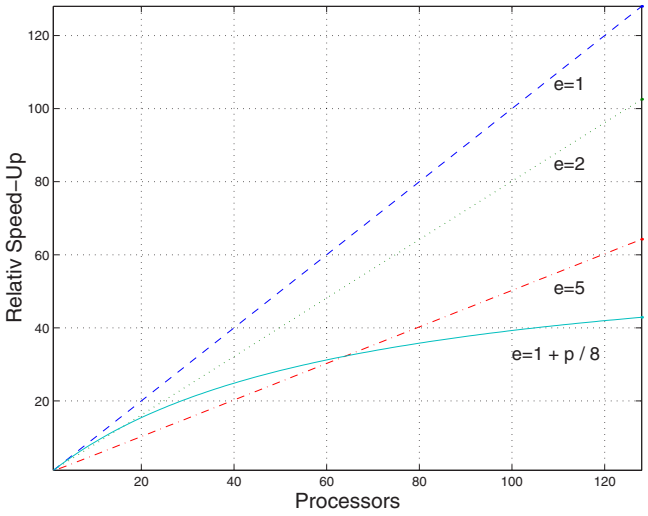

Figure 3. Speed-Up

build a prototype system in order to study the principal behaviour of such an implementation. Therefore the processors are not optimized, e. g. the execution of the instructions is not pipelined.

The components of a cell processor are:

- **Program Memory.** The size is 256 words of 24 bits. The program is loaded into it during the initialization phase.
- **Data Memory.** The size is 256 words of 16 bits. It consists of two parts, each with 128 words. The two parts are necessary, because the old generation of cells has to be available whilst the new generation is computed. Beside the cells data also arbitrary local variables are stored in this memory.
- **Register File.** The size is 16 words of 16 bits. The registers R8..RE are general purpose, the registers R0..R7 and RF are special purpose.
- **Dedicated Registers:**
  - R0 = S: status register
  - R1 = PC: program counter
  - R2 = CD: actual cell data
  - R3 = ECA: pointer to an external cell in another processor
  - R4 = ECD: cell data of the addressed external cell
  - R5 = MD: data to/from the data memory
  - R6 = MA: address to data memory

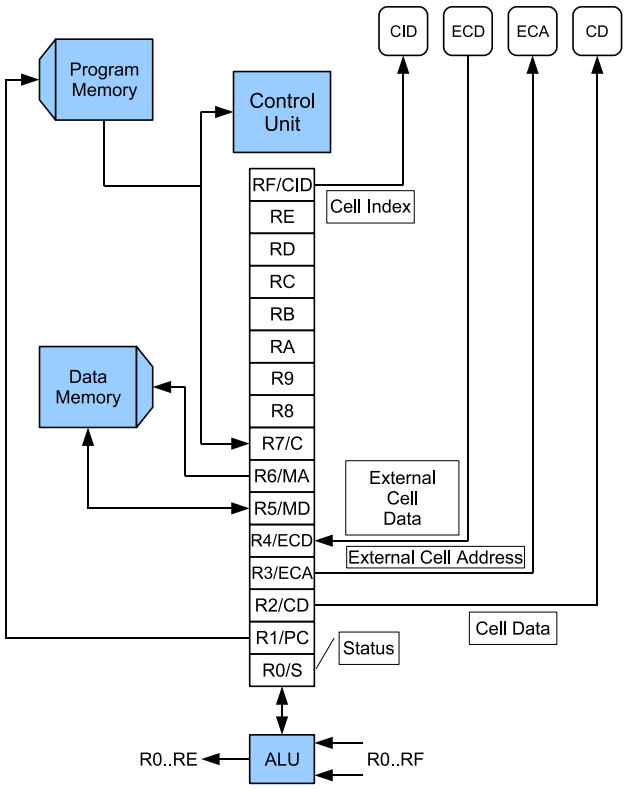

Figure 4. Cell Processor

  - R7 = C: register which is loaded with constants from program memory
  - RF = CID: actual address/position of the cell
- **ALU**. The ALU is connected to arbitrary registers and to the status register.
- **Program Address Logic**. The program address logic computes the next program address.
- **Control Unit**. The Control Unit interprets the master control information and the local instruction.

3.1. Instruction Set of the Cell Processor

The instructions are 24 bits wide (Tab. 1). The instructions operate only on registers, therefore we call this type of architecture VRISC (very much reduced instruction set computer). The instruction fields are:

- T: Instruction type
- OPC: ALU operation
- Rd: Register address destination

- Rs1, Rs2: Register address source
- L: Target address (label)
- K: Constant
- C: Selection of a condition

All instructions are only executed if the selected condition is true. A condition (C) is a selected bit from the status register S. Available conditions among others are Carry=ST(8), Zero=ST(9), Negative=ST(10), Overflow=ST(11), False=ST(0), True=ST(1).

T	OPC	Rd	Rs1	Rs2	C	
0	dop	Rz	Rx	Ry	i	Rz := Rx dop Ry
1	mop	Rz	Rx		i	Rz := mop Rx
9	K				i	R7 := constant
11			Rx	Ry	i	compare
2	L				i	jmp L
3		R5	R6		i	R5 := MEM(R6)
4		R6	R5		i	MEM(R6) := R5
5		R4	R3		i	R4 := ext.(R3)
10					i	nop
12					i	end
23..16		15..8		7..0		

Table 1. Instructions

Instruction typ 0 is a dyadic operation on registers. Operators are AND, OR, ADD and MUL. Instruction 1 is a monadic operation on registers. Monadic operations are SHIFT, NOT, MOV and logical reduction. Instruction 9 loads a 16-bit constant K into the register R7. Instruction 11 compares two registers and sets the condition bits of the status register. Instruction 2 conditionally sets the program counter to the target address L. Instruction 3 reads data from the address R6 of the local memory MEM into the register R5. Instruction 4 writes data into the local memory. Instruction 5 reads data from an external memory location R3 to the register R4. There are also some special instructions for synchronization of the cell processors (WAIT, READY, GO).

3.2. Some Implementation Details

The prototyping platform was a Cyclone FPGA with the Quartus synthesis software from Altera. The Cyclone II FPGA contains 68,416 logic elements (LE) and 1,152,000 RAM bits. The implementation language was Verilog HDL. The cell processor system with $p = 32$ processors was implemented with 56% of the available logic elements and 28% of the RAM bits (Tab. 2). The maximum clock frequency was around 85 MHz.

In the current prototype implementation every instruction needs six cycles for execution. The optimizing of the cell processor is under work, e. g. minimizing the number execution cycles, pipelining and the extension of the instruction set.

p	total LE	LEs for network	memory bits	register bits	max. clock (MHz)
1	843	0	10,240	355	96.45
4	3,573	192	40,960	1,424	91.27
8	7,553	768	81,920	2,844	90.63
16	16,625	2,816	163,840	5,692	89.69
32	38,367	11,264	327,680	11,392	84.65

Table 2. Ressources and clock rate

The communication network was implemented as a read-only crossbar consisting of multiplexers. Each processor has direct access to any other processors. The cost of the network in terms of logic elements and the time delay was not of significant relevance for $p = 32$ processors.

In order to test cost and delay of the network was investigated. The network consists of multiplexers. A one bit multiplexer was synthesized separately. The number of logic elements shown in the table 3 has to multiplied with the number of processors and the width of the external data bus.

The multiprocessor system had been simulated in JAVA before the synthesis process was started. A cross assembler is available to facilitate the machine programming.

multiplexer	logic elements	stages	delay
4 : 1	3	2	5.2 ns
8 : 1	6	3	6.4 ns
16 : 1	11	4	7.5 ns
32 : 1	22	5	10.2 ns
64 : 1	43	6	11.9 ns
128 : 1	86	7	12.7 ns
256 : 1	171	8	16.5 ns
512 : 1	342	9	17.5 ns

Table 3. Cost and delay of multiplexers

4. An Application: Merging of Bitonic Sequences

The principle of operation of the cell processor system will be demonstrated by the parallel merging of bitonic sequences. The result of merging two bitonic sequences is a sorted sequence of values. A sequence is bitonic if the values are increasing to a maximum and then decreasing. If such a sequence is cyclically shifted it remains bitonic. Bitonic sequences can be constructed from an unsorted sequence by applying nearly the same principle as merging.

In the fully parallel GCA model each cell holds its own value and compares it with the neighbor's cell which is in changing distance which is powers of two. If the own value has not the desired relation (e. g. ascending order) it will change its own value to the value of the neighbor. If a pointer points to higher indexed cell the minimum is computed, otherwise the maximum. The number of generations is $log_2(N)$, where N is the number of cells.

If the GCA model is sequentially executed on one processor, n steps are necessary in each generation leading to a total number of $n \cdot log_2(N)$ steps.

The algorithm can be described in the language CDL [6] which was developed in order facilitate the description of cellular algorithms. A cell state consists of *cell.data* and the pointer *cell.other*. The pointer to the global neighbor is denoted with *other*.

Listing 1. CDL Program for Bitonic Merge

```
var other : celladdress;
 w, a     : integer;
rule begin
 if ((cell.own_pos
   and cell.other_pos)= 0) then begin
   other := [cell.other_pos];
   w := *other.data;
   a := *cell.data;
   if (w < a) then cell.data := w;
 end
 else begin
   other := [cell.other_pos];
   w := *other.data;
   a := cell.data;
   if (a < w) then cell.data := w;
 end;
 cell.other_pos := cell.other_pos / 2;
 /* Address 64,32,16,8,4,2,1 */
end;
```

The CDL program (Listing 1) accesses the neighbors via relative addresses $(\pm m/2)$ in the first generation, $(\pm m/4)$ in the second and so on. The neighbor may also be accessed via absolute addresses. In this case the neighbor's address can be derived from the own address (or space index) of the cell, inverting a bit of the own address. The bits to be inverted are counted from the MSB to the LSB, according to the generation increment. In the prototype implementation absolute addressing of the neighbors was used.

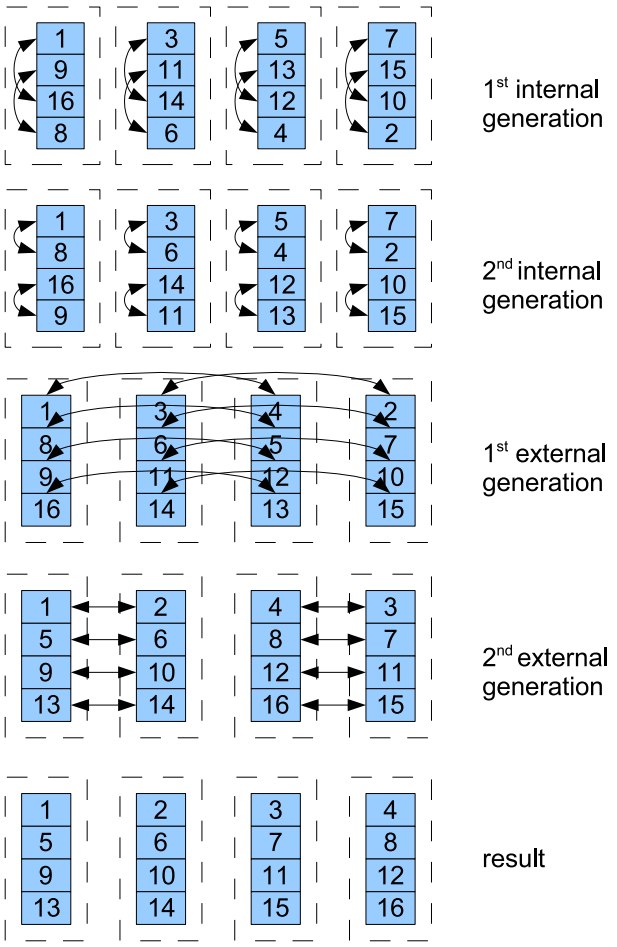

Figure 5. Bitonic Merge with four processors and 16 cells

The system was implemented for $p = 1, 2, 4, 8, 16, 32$ processors and $N = 128$ cells. In each processors are hold $n = N/p$ cells. In the first processing part only internal cells are compared (Fig. 5). We call these generations internal generations (G_I). In the second processing part only external cells are compared. We call these generations external generations (G_E).

The total number of generations is

$$G = log_2(N) = G_E + G_I = log_2(p) + log_2(n).$$

In order to compute the time for the external processing, G_E has to multiplied with the number n of internal cells (because they sequential exchanged via the communication net) and multiplied with a factor

t_e, representing the time for one external operation (operation with external access) $T_E = G_E \cdot n \cdot t_e$.

In order to compute the time for the internal processing, G_I has to be multiplied with the number n of internal cells (because they sequentially activated) and multiplied with a factor t_i, representing the time for one internal operation $T_I = G_I \cdot n \cdot t_i$.

Thus the total estimated time T is

$$T = T_E + T_I = log_2(p) \cdot \frac{N}{p} \cdot t_e + log_2(\frac{N}{p}) \cdot \frac{N}{p} \cdot t_i.$$

For our implementation (Listing 2) the time was exactly counted in number of instructions

$$T_I = 7 + G_I \cdot (17 \cdot n + 7)$$
$$T_E = (4 + 10 \cdot n) \cdot G_E$$

These formulas include additional constant parts which correspond to initialization code.

The number of needed instructions and the relative speed-up for p and $N = 128$ is shown in the table 4.

processors	instructions	rel. speed-up
p=1	15288	1.00
p=2	7221	2.12
p=4	3410	4.48
p=8	1615	9.47
p=16	772	19.80
p=32	377	40.55

Table 4. Number of instructions

The table shows a slight super linear speed-up. This is due to the fact that the code for the external generations is faster than the code for the internal generations. Two internal cells which are linked with each other can simultaneously change their values whilst two linked internal cells have to be processed one after the other.

The listing 2 shows the assembler program running on each cell processor. It consists of one part which handles the external operations and one part which handles the internal operations.

Listing 2. Program for the first processor, case p = 4

```
0  load_c              – begin of part 1
1  r8 = r7             – constant 1 for incrementing
2  load_c 0            – constant 0 for comparing
3  r9 = r7
4  load_c 16           – relative distance for the
5  ra = r7             – first step
6  rf = r9             – address offset, address space 0
7  load_c 32           – address offset, address space 1
8  rc = r7
9  load_c 31           – Label 5: inner counter variable
10 rb = r7
11 r6 = rb add rf      – Label 3: load value A to
12 load                – register Rd
13 rd = r5
14 r0 = rb and ra      – relative distance & address
15 if st(9) jump 22    – jump to label 1
16 r6 = r6 sub ra      – address minus the current
17 load                – relative distance
18 re = r5             – load value B to register Re
19 cmp rd,re           – compare the values of A and B
20 if st(2) rd = re    – if greater then exchange
21 jump 28             – jump to label 2
22 r6 = r6 add ra      – Label 1: address plus relative
23 load                – distance
24 re = r5             – load value B to register Re
25 cmp re,rd           – compare the values of A and B
26 if st(2) rd = re    – if greater then exchange
27 jump 28             – jump to label 2
28 r6 = rb add rc      – Label 2: calculate new address
29 r5 = rd
30 store               – store the cell
31 rb = rb sub r8      – inner counter variable minus one
32 if st(a) jump 34    – if zero jump to label 4
33 jump 11             – else jump to label 3
34 r0 = rf             – Label 4: exchange offsets
35 rf = rc
36 rc = r0
37 ra = shr ra         – calculate the relative distance
38 if st(9) jump 40    – if zero jump to label 6
39 jump 9              – else jump to label 5
40 load_c 63           – Label 6: begin of part 2
41 r6 = r7             – address = address of the last
42 load_c 2            – value in memory
43 r3 = r7             – cell address = 2
44 load_c 31           – 1st address of the memory space
45 load                – Label 7: load value
46 r2 = r5             – cell content = data value
47 load_g              – global load
48 cmp r4,r5           – compare cellcontent & local data
49 if st(2) r5 = r4    – if greater then exchange
50 store               – store value
51 r6 = r6 sub r8      – address minus one
52 cmp r7, r6          – compare values
53 if st(6) jump 55    – break condition jump to label 8
54 jump 45             – else next step
55 load_c 63           – Label 8: begin of part 3
56 r6 = r7             – address = address of the last
57 load_c 1            – value in memory
58 r3 = r7             – cell address = 1
59 load_c 31           – 1st address of the memory space
60 load                – Label 9: load value
61 r2 = r5             – cell content = data value
62 load_g              – global load
63 cmp r4,r5           – compare cellcontent & local data
64 if st(2) r5 = r4    – if greater then exchange
65 store               – store value
66 r6 = r6 sub r8      – address minus one
67 cmp r7, r6          – compare values
68 if st(6) jump 70    – break condition jump to label 10
69 jump 60             – else next step
70 end                 – Label 10: end of part 3
```

5. Conclusion

A programmable multiprocessor architecture for the massively parallel GCA model was designed and implemented as a prototype in FPGA technology. The architecture consists of p cell processors with internal memories and a read-only interconnection network.

Compared to a dedicated implementation the proposed architecture is very flexible because it can be easily adapted to different GCA algorithms by programming. The speed-up of the prototype increases linear with the number of processor for the investigated algorithm. Also for other implemented algorithms (vector reduction, transitive hull) the speed-up was linear. The implementation of the network is relatively simple because it consists only of cascaded multiplexers. If the number of processors gets very high, the cost and time delay of the network have to be taken into account.

If the external processor offers the external cell data at the right moment to the demanding processor, no synchronization overhead downgrades the performance. Therefore the program should reflect the desired communication pattern of the GCA algorithm in order to minimize the synchronization overhead.

References

[1] Altera. `http://www.altera.com/literature/hb/cyc2/cyc2_cii5v1.pdf`, 2005.

[2] W. Heenes. Globaler Zellularer Automat: Algorithmen und Architekturen. Masterthesis, Technische Universität Darmstadt, 2001.

[3] W. Heenes, R. Hoffmann, and S. Kanthak. FPGA Implementations of the Massively Parallel GCA Model. In *International Parallel & Distributed Processing Symposium (IPDPS), Workshop on Massively Parallel Processing (WMPP)*, 2005.

[4] W. Heenes, J. Jendrsczok, and R. Hoffmann. Eine massivparallele Rechnerarchitektur für das GCA Modell. In *PARS Workshop, Gesellschaft für Informatik (GI)*, 2005.

[5] W. Heenes, K.-P. Völkmann, and R. Hoffmann. Architekturen für den globalen Zellularautomat. In *19th PARS Workshop*, 2003.

[6] C. Hochberger. *CDL - Eine Sprache für die Zellularverarbeitung auf verschiedenen Zielplattformen.* PhD thesis, Technische Universität Darmstadt, 1998.

[7] R. Hoffmann, K.-P. Völkmann, and W. Heenes. GCA: A massively parallel Model. In *International Parallel & Distributed Processing Symposium (IPDPS), Workshop on Massively Parallel Processing (WMPP)*, 2003.

[8] R. Hoffmann, K.-P. Völkmann, S. Waldschmidt, and W. Heenes. GCA: Global Cellular Automata, A Flexible Parallel Model. In *6th International Conference on Parallel Computing Technologies (PaCT)*, 2001.

[9] J. von Neumann. *Theory of Self-Reproducing Automata.* University of Illinois Press, Urbana and London, 1966.

[10] C. Wiegand, C. Siemers, and H. Richter. Definition of a configurable architecture for implementation of global cellular automaton. In *17th International Conference on Architecture of Computing Systems (ARCS 2004)*, 23-26 March 2004.