The Massively Parallel Computing Model GCA

Rolf Hoffmann

Technische Universität Darmstadt, FB Informatik, FG Rechnerarchitektur,
Hochschulstraße 10, D-64289 Darmstadt, Germany
{hoffmann}@ra.informatik.tu-darmstadt.de

Abstract. The Global Cellular Automata Model (GCA) is an extension of the Cellular Automata Model (CA). Whereas in the CA model each cell is connected via fixed links to its local neighbors, in the GCA model each cell is connected via data dependent dynamic links to any (global) cell of the whole array. The GCA cell state does not only contain data information but also link information. The cell state is synchronously updated according to a local rule, modifying the data and the link information. Similar to the CA model, only the own cell state is modified. Thereby write conflicts cannot occur. The GCA model is related to the CROW (concurrent read owners write) model and it can be used to describe a large range of applications. GCA algorithms can be described in the language GCA-L which can be compiled into different target platforms: a generated data parallel multi-pipeline architecture, and a NIOS II multi-softcore architecture.

Keywords: Global Cellular Automata, Parallel Programming Model.

1 Introduction

Since the beginning of parallel processing a lot of theoretical and practical work has been done in order to find a parallel programming model (PPM) which fulfills at least the following properties

- *User-friendly:* easy to model and to program
- *System-designer-friendly:* parallel processing target architectures supporting the model are easy to design and to implement, and programs can easily be translated into these architectures
- *Efficient:* The applications can efficiently be executed on the target architecture
- *Platform independent:* The PPM can also be mapped (interpreted, simulated) without much effort onto other standard platforms and can there be executed with a satisfying performance.

In the following sections such a model (Global Cellular Automata) will be described, and how it can be implemented and used. This model was introduced in [1], then further investigated, implemented, and applied to different problems. This paper is based on the results of former publications, mainly [1] [2] [3] [4] [5] [6] [7] [8] [9] [10].

M.R. Guarracino et al. (Eds.): Euro-Par 2010 Workshops, LNCS 6586, pp. 77–84, 2011.

2 The GCA Model (Global Cellular Automata)

The definition of the GCA model was inspired by the CA (Cellular Automata) model. The CA model consists of an array of cells arranged in an n-dimensional grid. Each cell (also called the "**C**enter Cell" is connected to its local neighbors belonging to the neighborhood, e.g. to **N**orth, **E**ast, **S**outh, **W**est. The next state of the center cell is defined by a local rule f residing in each cell: $C \leftarrow f(C, N, E, S, W)$. All cells are applying the same rule synchronously and thereby a new generation of cell states is defined. As a cell changes only its own state, no write conflicts can occur which makes the model simple and elegant. Many applications with a local neighborhood can nicely be described as a CA, and CAs can easily be simulated or implemented in hardware.

The idea for the GCA model was (1) to retain the property that a cell can only modify its own state, and (2) to introduce more flexibility. Flexibility was obtained by using (2a) computed dynamic links to the neighbors and (2b) by allowing any cell in the array to be a neighbor (global neighbors). Thus a GCA can informally be described as follows:

A GCA consists of an indexed set of cells (e.g. an n-dimensional array). The cells' states are updated synchronously according to a local rule. Each cell has k global neighbors which can dynamically be changed by the local rule (Fig. 1). Write conflicts cannot occur, therefore the model can easily be supported by hardware for a large number of cells. A GCA is initialized by an initial state for each cell (initial configuration *CFG(t=0)*). The result of the computation is the state of the finial configuration (all cell states) at time-step t_{final}. We can also speak of a "GCA algorithm", meaning the transformation of the initial generation to the final generation.

Three model variants are distinguished, the *basic model*, the *general model* and the *condensed model*. They are closely related to each other and can be transformed into each other. It depends on the application or the implementation which one will be preferred.

Basic model. The cell state is a composition $(data, pointer) = (d, p)$ (Fig. 2a). The pointer p is used to access the global neighbor $p = k^*$. The remote state (d^*, p^*) is read from the global cell via the dynamic link. Then the new state components are computed: $d' = e(d, p, d^*, p^*)$ and $p' = g(d, p, d^*, p^*)$. Then

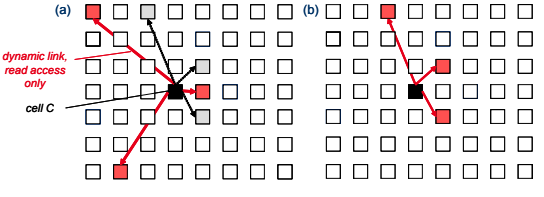

Fig. 1. (a) In generation t each cell is connected to k neighbors (e.g. $k = 3$), and it selects k new neighbors. (b) In generation $t + 1$ each cell is connected to its new neighbors.

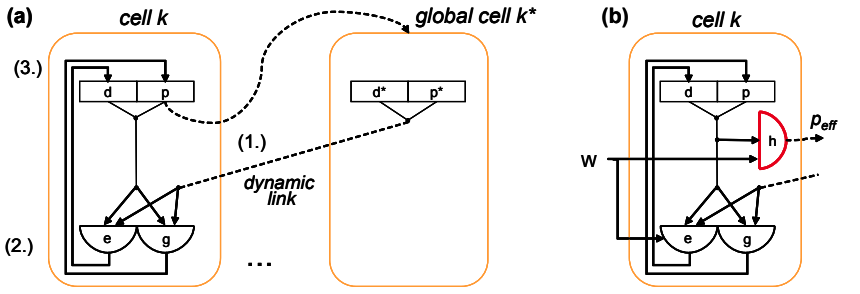

Fig. 2. (a) Basic model. The cell state is a composition of one or more data fields d and one or more pointer (link) fields p. A global cell k^* is accessed via p and a data link is dynamically established from the global cell k^* to the cell k (1.). Then the function e computes the next data and the function g computes the next pointer (2.). Then all cells are updated synchronously (3.). – (b) General model. An additional function $h(d, p)$ is used which computes an effective address p_{eff} to select the global neighbor. Central control parameters W from the simulation environment can be taken into account.

all cells are updated synchronously: $d \leftarrow d', p \leftarrow p'$. In general several data and pointer fields can be used $(d_1, d_2, ..., p_1, p_2, ...)$ and several neighbors can be accessed.

General model. In addition to the basic model the general model (Fig. 2b) uses an addressing function $h(d, p)$ which computes the effective address $p_{eff} = h(d, p)$ of the global neighbor. For many applications this addressing technique is more convenient than the direct addressing with p of the basic model. In addition, central parameters W, supplied by the simulation environment (e.g. generation counter t, cell index k, etc.) are useful to be available in the functions: $h(d, p, d^*, p^*, W), e(d, p, d^*, p^*, W)$, and $g(d, p, d^*, p^*, W)$.

Condensed model. In the condensed model the cell state is not separated into a data part and pointer part. Instead only a data part q is used. The meaning of q is a matter of interpretation, it can be interpreted as a data or a pointer field. Either parts of q are interpreted as data or pointer, or q is interpreted alternatively as data or pointer depending on a data type subfield of q (like an operation code of an instruction) or on additional information like the generation counter t.

Relation to the CROW model. The GCA model is related to the CROW (concurrent read owner write) model [11], a variant of the PRAM (parallel random access machine) models. The CROW model consists of a common global memory and P processors, and each memory location may only be written by its assigned owner processor. In contrast, the GCA model consists of P cells, each with its local state (data and pointer fields) and its local rule (together acting as a small processing unit updating the data and pointer field). Thus the GCA model is (1) "cell" based, meaning that the state and processing unit are

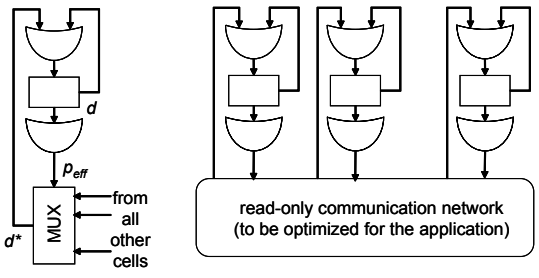

Fig. 3. Fully parallel implementation. Communication implemented by a multiplexer in each cell (a). Communication implemented by a common network (b).

encapsulated, similar as objects in the object oriented languages, and (2) the cells are structured according to the application. The processing units of the GCA can be seen as virtual processors, having just the processing features which are needed for the application. On the other hand the CROW model uses "universal" processors independent of the application. Furthermore the GCA updates in each generation the data and pointer fields simultaneously, whereas in the PRAM model only one memory location per processor is updated simultaneously.

3 GCA Architectures

A variety of architectures can be designed or used to support the GCA model. In the following three architectures are proposed. The *fully parallel architecture* can be very powerful for small dedicated applications. The *data parallel architecture* can be used for medium size dedicated applications to be configured on an FPGA. The *multisoftcore* system is programmable and can be applied to medium or large applications. Other interesting architectures could be designed, e.g. cells with programmable rules, or pipelines with programmable rules, but these are out of the scope of this paper. Furthermore standard platforms like standard multicores or GPUs can be used to execute the GCA model.

In a student's work Benjamin Milde and Niklas Büscher showed an acceleration of 13 for bitonic merging and 150 for a diffusion algorithm on an NVIDIA GFX 470 compared to an Intel Q9550@3GHz with 4 threads. Although these results cannot be generalized, GPUs seem to be a promising platform to execute the GCA model.

Fully Parallel Architecture. "Fully parallel" means that the whole GCA for a specific application is completely implemented in hardware (Fig. 3). The question is how many hardware resources are needed. The number of cells is n. Therefore the logic (computing the effective address and the next state) and the number of registers holding the cells' states are proportional to n. The local interconnections (wiring) are proportional to n, too. As the GCA generally allows to access from each cell any other global cell, the wiring effort is $(n-1) \times n$ *global* wires. The length of a global wire is not a constant, it depends on the

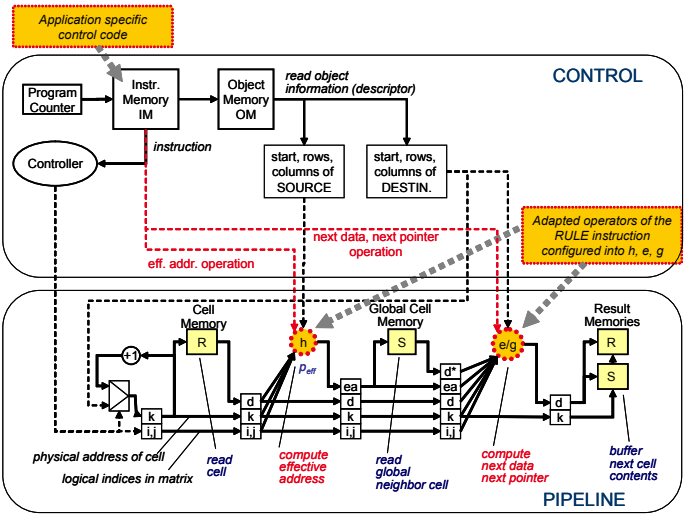

Fig. 4. Data parallel architecture (DPA) with one pipeline

physical distance. If the cells are arranged in a *2d* square grid the shortest distance is one and the longest (Manhattan) distance is $2\sqrt{n}$. Thus the total wiring (with respect to the mean distance) is of the order $O(n^2)$. Note that the longest distance also determines the maximal clock rate. Many applications / GCA algorithms do not require a total interconnection fabric because only a subset of all communications (read accesses) are required for a specific application. Therefore the amount of wires and switches can be reduced significantly for one or a limited set of applications. In addition for each global wire a switch is required. The switches can be implemented by a multiplexer in each cell, or by a common switching network (e.g. crossbar). Note that the number of switches of the network can also be reduced to the number of communication links used by the specific application. Another aspect is the multiple read (concurrent read) feature. In the worst case one cell is accessed from all the other cells which may cause a fanout problem in the hardware implementation.

Data Parallel Architecture. The data parallel architecture (DPA) uses p pipelines in order to process p cell rules in parallel [6][8]. The whole address space is partitioned into (sub) arrays, also called "cell objects". In our implementation a cell object represents either a cell vector or a cell matrix. A cell object is identified by its start address, and the cells within it are addressed relatively to the start address. The *destination object* D stores the cells to be updated, and the *source object* S stores the global cells to be read. Although for most applications D and S are disjunct, the may overlap or be the same.

The DPA consists of a control unit and p pipelines, in Fig. 4 only one pipeline is shown. In the case of one pipeline only, the cells of S are processed sequentially using a counter k. In the first pipeline stage the cell *D[k]* is read from memory

(a)

```
program
parameter logN = 3;
cellstructure = d; celltype floatcell = float; neighborhood = neighbor;
floatcell X[5]; X.d = 1,2,3,4,5;
floatcell A[5][5];
A.d = 15,2,3,4,5, 2,19,4,5,6, 5,4,15,2,1, 1,3,5,18,4, 4,2,3,1,12;
floatcell Atemp[5][5]; floatcell B[5]; B.d = 77,132,-60,53,412;
central subgen;
for gen=0 to 1000000 do
    foreach Atemp with neighbor = &A[i,j] do d <= neighbor.d; endforeach;
    foreach Atemp with neighbor = &X[0,j] do
     if (i!=j) then d <= d *neighbor.d else d <= d endif
    endforeach;
    for subgen = 0 to logN do
     foreach Atemp with neighbor = &Atemp[i+(1<<subgen)%columns,j] do
     if (i+(1<<subgen)<columns) then d <= d+neighbor.d else d <= d endif
     endforeach
    endfor;
    foreach X with neighbor = &B[i] do d <= neighbor.d; endforeach;
    foreach X with neighbor=&Atemp[i,0] do d <= d-neighbor.d; endforeach;
    foreach X with neighbor = &A[i,i] do d <= d / neighbor.d; endforeach
endfor
endprogram
```

(b)

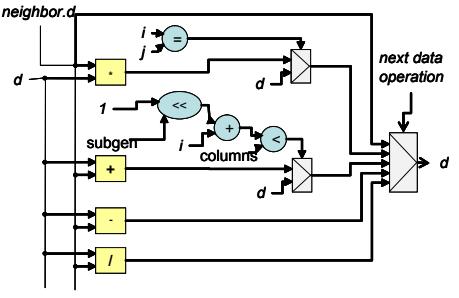

Fig. 5. (a) GCA-L program for the Jacobi iteration. – (b) Next data operator e automatically generated out of the program. It contains 4 floating point units and several integer units.

R. In the second stage the effective address ea is computed by h. In the third stage the global cell $S[ea]$ is read. In the fourth stage the next cell state d is computed. Then the next cell state is stored in the buffer memories R' and S' at location k. When all cells of the destination object are processed, the memories (R, S) and (R', S') are interchanged.

An application specific DPA with p pipelines can automatically be generated out of a high level description in the experimental language GCA-L [6]. The program (Fig. 5a) describes the Jacobi iteration [7] solving a set of linear equations.

The most important feature of GCA-L is the *foreach D with neighbor = &S[..] do .. endforeach* construct. It describes the (parallel) iteration over all cells $D[i, j]$ using the global neighbors *&S[h(i,j)]*. Our tool generates Verilog code for the functions h, e, g to be embedded in the pipeline(s). These functions are also pipelined. In addition control code for the control unit is generated. The most important control codes are the *rule* instructions. A rule instruction triggers the processing of all cells in a destination object and applies the so called *adapted operators* h, e, g coded in the rule. All necessary application specific rule instructions are extracted from the source program [7]. Fig. 5b shows a generated next data operation used by a rule instruction. It contains 4 floating point units and several integer units. The floating point operations are internally also pipelined (- :14, * :11, + :14, / :33 stages). Our tool generates Verilog code which further is used for synthesis with Quartus II for Altera FPGAs. For $p = 8$ pipelines, normalized to the amount needed for one pipeline, the relative increments for the FPGA Altera Stratix II EP2S180 were: 8.3 for the ALUTs (logic elements), 7.5 for the registers, 4.5 for the memory bits (note that the required memory bits are theoretically proportional to $(p+1)/2$ for the pipeline architecture). The speedup was 6.8 for 8 pipelines compared to one. Thus the scaling behavior was very good and almost linear for up to 8 pipelines.

Multisoftcore. The basic idea is to use many standard softcores together with specific GCA support. Each core is responsible to handle a subset of all cells

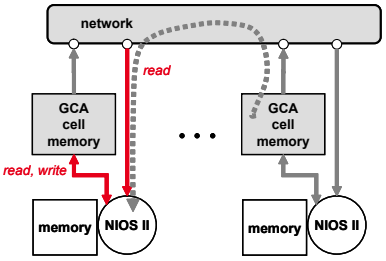

Fig. 6. Multisoftcore system implemented on an FPGA. A local GCA cell memory is attached to each NIOS II softcore. Each core can read and write its own GCA cell memory and read from any other GCA cell memory via the network.

being processed in one generation. In our implementation, p NIOS II softcores were used [9][10]. To each processor a GCA cell memory is attached (Fig. 6). A processor can read via the network the state of a global cell residing in another cell memory. Only the cells residing in the own cell memory need to be updated according to the GCA model. No write access via the network is needed, thereby the network can be simplified. In case that only a specific application has to be implemented, the network can be minimized according to the communication links used by the application. The machine instruction set of the NIOS processors was extended (custom instructions), e.g. read a cell via the network, read/write local cell memory, floating point operations, synchronize and copy new cell states into the current cell states.

A tool was developed that can automatically generate C code (extended by custom instructions) out of a GCA-L program for such a multisoftcore system. Then this C code is compiled and loaded into the cores of the system configured on an FPGA.

4 Conclusion

In the GCA parallel programming model, applications are modeled as a set of cells which are dynamically connected to other cells. Applications can be described in the experimental language GCA-L. Different GCA target architectures can easily be designed and implemented, e.g. a fully parallel architecture, a data parallel architecture, and a multisoftcore architecture. Tools allow to translate GCA-L programs into such architectures or generate them for FPGAs. These architectures can be optimized for specific applications and adjusted to the performance requirements. First investigations have shown that the GCA model can also be efficiently executed on standard multicores and especially on GPUs.

Acknowledgment. I would like to thank Benjamin Milde and Niklas Büscher who implemented the model on Quadcores and GPUs.

References

1. Hoffmann, R., Völkmann, K.P., Waldschmidt, S.: Global cellular automata GCA: an universal extension of the CA model. In: ACRI 2000 "work in progress" session, Karlsruhe, Germany, October 4-6 (2000)
2. Hoffmann, R., Völkmann, K.P., Waldschmidt, S., Heenes, W.: GCA: Global cellular automata. A flexible parallel model. In: Malyshkin, V.E. (ed.) PaCT 2001. LNCS, vol. 2127, pp. 66–73. Springer, Heidelberg (2001)
3. Hoffmann, R., Völkmann, K.P., Heenes, W.: GCA: A massively parallel model. In: IPDPS 2003, Nice, France, April 22-26 (2003)
4. Heenes, W., Hoffmann, R., Jendrsczok, J.: A multiprocessor architecture for the massively parallel model GCA. In: IEEE Proceedings of 20th International Parallel & Distributed Processing Symposium, IPDPS/SMTPS 2006, Rhodes Island, Greece, April 25-29. IEEE, Los Alamitos (2006)
5. Jendrsczok, J., Ediger, P., Hoffmann, R.: The Global Cellular Automata Experimental Language GCA-L1. Technical Report RA-1-2007, Technische Universität Darmstadt (2007)
6. Jendrsczok, J., Ediger, P., Hoffmann, R.: A Scalable Configurable Architecture for the Massively Parallel GCA Model. International Journal of Parallel, Emergent and Distributed Systems (IJPEDS) 24(4), 275–291 (2009)
7. Jendrsczok, J., Hoffmann, R., Ediger, P.: A Generated Data Parallel GCA Machine for the Jacobi Method. In: 3rd HiPEAC Workshop on Reconfigurable Computing, Paphos, Cyprus, January 25, pp. 73–82 (2009)
8. Jendrsczok, J., Hoffmann, R., Lenck, T.: Generated Horizontal and Vertical Data Parallel GCA Machines for the N-Body Force Calculation. In: Berekovic, M., Müller-Schloer, C., Hochberger, C., Wong, S. (eds.) ARCS 2009. LNCS, vol. 5455, pp. 96–107. Springer, Heidelberg (2009)
9. Schäck, C., Heenes, W., Hoffmann, R.: A Multiprocessor Architecture with an Omega Network for the Massively Parallel Model GCA. In: Bertels, K., Dimopoulos, N., Silvano, C., Wong, S. (eds.) SAMOS 2009. LNCS, vol. 5657, pp. 98–107. Springer, Heidelberg (2009)
10. Schäck, C., Heenes, W., Hoffmann, R.: Network Optimization of a Multiprocessor Architecture for the Massively Parallel Model GCA. In: Mitteilungen - Gesellschaft für Informatik e. V., Parallel-Algorithmen und Rechnerstrukturen, Wolfgang Karl and Rolf Hoffmann and Wolfgang Heenes, vol. 26, pp. 48–57 (Dezember 2009)
11. Dymond, P., Ruzzo, W.: Parallel RAMs with owned global memory and deterministic context-free language recoginition. In: Kott, L. (ed.) ICALP 1986. LNCS, vol. 226, pp. 95–104. Springer, Heidelberg (1986)