Implementing APL-like data parallel functions on a GCA machine

Johannes Jendrsczok[1], Rolf Hoffmann[1], Patrick Ediger[1], and Jörg Keller[2]

[1] TU Darmstadt, FB Informatik, FG Rechnerarchitektur
Hochschulstraße 10, D-64289 Darmstadt
{jendrsczok, hoffmann, ediger}@ra.informatik.tu-darmstadt.de
[2] FernUniversität in Hagen, Fakultät für Mathematik und Informatik
Universitätsstr. 1, D-58084 Hagen
Joerg.Keller@FernUni-Hagen.de

Abstract. We show how data parallel operations can be implemented on a Global Cellular Automaton (GCA). For this purpose we implement a number of frequently used functions of the language APL. APL has been chosen because of the ability to express matrix and vector operations in a short and concise manner. It is shown how various operations on vectors and matrices can be executed through the execution of appropriate GCA generation computations. Finally the architecture of the underlying GCA machine is presented which interprets GCA instructions in a pipelined manner and using several memories.

1 Introduction

The GCA (Global Cellular Automaton) model [1–3] is an extension of the classical CA (Cellular Automaton) model [4]. 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 GCA model is a generalization 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 "global" neighbors are not fixed; they can be changed by the local rule from generation to generation. Thereby the range of parallel applications for the GCA model is much wider than for the CA model.

The CA model is suited to all kinds of applications with local communication. Typical applications for the GCA model are – besides all CA applications – graph algorithms, hypercube algorithms, logic simulation, numerical algorithms, communication networks, neuronal networks, games, and graphics.

The general aim of our research (project "Massively Parallel Systems for GCA" supported by Deutsche Forschungsgemeinschaft) is the hardware and software support for this model [5].

In this paper we show how APL instructions on matrices and vectors can be implemented by a hardware architecture which supports the GCA model. APL (*A programming Language*) was defined 1963 by K. Iverson. His book [6] is a rich source for complex parallel operators. APL is a data parallel language, which allows describing algorithms in a very concise form. Data parallel languages such as Fortran 90 are wide in use for scientific codes, requiring high performance general-purpose computers. As a GCA can be implemented very efficiently on an FPGA, we map data parallel functions on a GCA. The GCA can be used to provide a lower cost alternative platform providing high performance for such computations.

The Global Cellular Automaton model. A GCA consists of an array of cells. Each cell contains a data field d and one ore more pointer fields p. A cell k dynamically establishes a link to any other cell (the *global cell*) identified by a pointer p. The local information (d, p)

and the global information (d^*, p^*) is input of the functions f_d and f_p which compute the next data and the next pointer respectively. All cells are updated in parallel in a synchronous way. The assignment symbol "$\Leftarrow$" is used to indicate the synchronous updating. The function f_d and f_p may further depend on the *space index* k of the cell (corresponding to the *cell identifier* used in the following sections) and central control information. Typical central control information is the generation counter t, an additional operation code, address offsets or general parameters. The operation principle of a *basic* GCA can be described by the following rules:

$$d \Leftarrow f_d(d, p, d^*, p^*, k, control),\ p \Leftarrow f_p(d, p, d^*, p^*, k, control).$$

Note that $d = D[k]$, $p = P[k]$, $d^* = D[P[k]]$ and $p^* = P[P[k]]$ if the arrays D and P are used to store the cells.

A GCA can also be *condensed* to the form

$$p = f_p(d, k, control),\ d \Leftarrow f(d, d^*, p, k, control).$$

The assignment symbol "=" indicates that a value is assigned to the temporary pointer variable p which may be a wire or a temporary register in a hardware implementation. We will use the condensed model in the following sections because this is sufficient to implement data parallel functions.

2 Mapping of APL variables and APL operators

APL variables are scalars, vectors or matrices of integers, reals, booleans or characters. In this paper we assume for simplicity that the elementary data type is integer. It is assumed that all variables are stored as GCA cell arrays in a common cell memory. The memory cells are identified by their identifiers (or addresses) ranging from 0 to $N-1$ if the capacity of the memory is N cells. Thus a cell can be accessed through its unique cell identifier (Cell-ID).

A central object memory table is used apart from the cell memory to store general information for each variable which can be either a scalar, or a vector or a matrix. This information is the same for every variable X and consists of three entries:

- $start(X)$ the first cell identifier of the variable X (the start position),
- $rows(X)$ the number of rows,
- $cols(X)$ the number of columns.

The introduction of an object table is useful because it allows managing a set of cell objects. Also the information in the object table is variable and it can be adopted to the problem or for dynamic memory allocation during runtime. If the object information is fixed on the other hand, in the hardware design the variables can be treated as constants in order to minimize the resources, e. g. by the use of specialized functions.

In order to compute the identifier (linear index, address) of a cell in the memory, the id function was defined. For a given element (y, x) of matrix B its identifier is calculated by

$$id(B(y, x)) = (y \cdot cols(B) + x) + start(B).$$

The following operations have been chosen as examples of frequently used matrix operations in order to show their implementation on a condensed GCA:

- Transposition of a matrix
- Indexing function for rows and columns
- General indexing

– Horizontal and vertical rotation

We assume that all objects including the target have the required size and are already allocated in the memory and object table. For all following operations it is only necessary to calculate an update of the cells of the target structure. The other cells remain unchanged.

Transposition of a matrix. This operation exchanges the values at the position (i, j) with the values at the position (j, i). The following APL instruction realizes this function:

```
MC ← ⍉ MB // Targetmatrix ← ⍉ Sourcematrix
```

Only one generation is needed to achieve the transpose of the source matrix in the GCA field. The values of the source matrix at the position (j, i) are copied to the cells at the position (i, j) in the target matrix.

MC: $p = id(MB(j, i))$, $d \Leftarrow d^*$

It is also possible to use the same matrix as source and target (`MA ← ⍉ MA`). The cells can directly be reordered in the cell array after having set the cell pointers properly. Note that also the number of rows and the number of columns in the object table have to be interchanged.

$cols(MA) \Leftarrow rows(MA)$, $rows(MA) \Leftarrow cols(MA)$
MA: $p = id(MA(i \cdot cols(MA) + j\%rows(MA), i \cdot cols(MA) + j/rows(MA)))$, $d \Leftarrow d^*$

Indexing function for rows. As an example for this operation the following APL instruction is used:

```
MA ← MB[IV;] // Targetmatrix ← Sourcematrix[Indexvector;]
```

The index vector IV is interpreted as a matrix with one column. It contains row indices. The i^{th} element of IV defines the index of the row of the source matrix which is assigned to the i^{th} row of the target matrix. A typical application for this operation is the permutation of rows.

In the condensed GCA model this operation can be executed in two generations. In the first generation the contents of IV are copied into each column of MA. In the second generation this value d is used to compute the cell identifiers of the cells in row d of the source matrix. The values of the selected source matrix rows are then copied into the rows of the target matrix.

1^{st} Generation: MA: $p = id(IV(i, 0))$, $d \Leftarrow d^*$
2^{nd} Generation: MA: $p = id(MB(d, j))$, $d \Leftarrow d^*$

Indexing function for columns. The instruction for this operation reads in APL as follows:

```
MA ← MB[;IV] // Targetmatrix ← Sourcematrix[;Indexvector]
```

This operation works like the previous one, only that columns and rows are interchanged. A typical application is the permutation of columns.

1^{st} Generation: MA: $p = id(IV(j, 0))$, $d \Leftarrow d^*$

2^{nd} Generation: MA: $p = id(MB(i,d))$, $d \Leftarrow d^*$

General indexing function. The indexing functions can be generalized using an index matrix IM. Thereby the rows or columns can individually be indexed. This operation is not directly supported by APL. It can be implemented with two generations.

1^{st} Generation: MA: $p = id(IM(i,j))$, $d \Leftarrow d^*$
2^{nd} Generation for permuting the rows: MA: $p = id(MB(d,j))$, $d \Leftarrow d^*$
2^{nd} Generation for permuting the columns: MA: $p = id(MB(i,d))$, $d \Leftarrow d^*$

Horizontal rotation. In this operation each row of the source matrix undergoes a cyclic left shift. In APL the instruction looks like this:

```
MA ← S ϕ MB // Targetmatrix ← Scalar ϕ Sourcematrix
```

Two generations are needed to execute this operation. In the first generation the column positions of the global cells (S positions to the right from the actual cell) are computed using the scalar value. The result is saved in the cells d of the target matrix. In the second generation the global cells located in MB are accessed through $id(MB(i,d))$ and copied to MA.

1^{st} Generation: MA: $p = id(S(0,0))$, $d \Leftarrow (j + d^*)\%cols(MA)$
2^{nd} Generation: MA: $p = id(MB(i,d))$, $d \Leftarrow d^*$

Vertical rotation. In the vertical rotation each column of the source matrix undergoes a cyclic shift upwards.

```
MA ← S ⊖ MB // Targetmatrix ← Scalar ⊖ Sourcematrix
```

As for the horizontal rotation two generations are needed to execute this operation. In the first generation the row positions of the cells to be accessed are computed. The shifted values of the source matrix are then copied to MA in the second generation.

1^{st} Generation: MA: $p = id(S(0,0))$, $d \Leftarrow (i + rows(MA) - d^*)\%rows(MA)$
2^{nd} Generation: MA: $p = id(MB(d,j))$, $d \Leftarrow d^*$

3 Architecture of a GCA machine

In this section we present an architecture of a GCA machine. The GCA machine interprets GCA instructions for the condensed GCA model. The described APL instructions can easily be executed on this machine which was verified with a simulator written in Java. The format of the GCA instructions is $(opc\_a, opc\_b, target\_object, source\_object)$.

The semantic of the instructions corresponds mainly to different kinds of generation computations used in the previous sections.

The first design is a sequential system (Fig. 1) with a pipeline structure to show the basic functionality and to describe the necessary components of the system. A central issue will be how to avoid congestion to the object table. If the data layout can be computed during compilation of the APL program, then all references to the object table can be compiled into the instructions and the object table need not be referenced at all. If on the other hand, the object table changes during the runtime, as for a transposition, then replication may be a possible choice.

The machine consists of

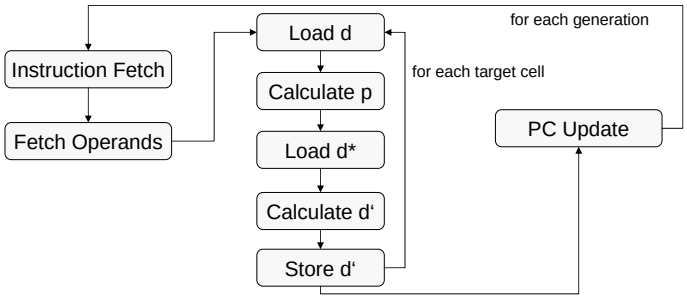

Fig. 1. Controller state machine

- an instruction memory IM,
- an object table memory OM,
- the execution pipeline with the cell memories R, S and the result memories R', S'.

In order to read in parallel the cell data and the neighbor cell data within two different pipeline stages the data memory is mirrored. Thus the memory R is used to read the cell data and the memory S is used to read the cell data of the global cell. In order to implement the synchronous updating scheme, the results are first saved in the buffer memories R' and S'. Thereby the old values of the cells are always available during the calculation of one generation and are not overwritten in between. The cell memories (R, S) and (R', S') are interchanged after each generation.

Before the calculation of a new generation begins, the next instruction is fetched from the instruction memory. Then the object information of the two objects (start position, rows and columns) is fetched from the object memory. This access can be performed sequentially in two cycles as shown in (Fig. 2) or in one cycle using a dual port memory. Additionally the internal operation codes for the calculation of the new data d (OPC_d) and new pointer p (OPC_p) are set.

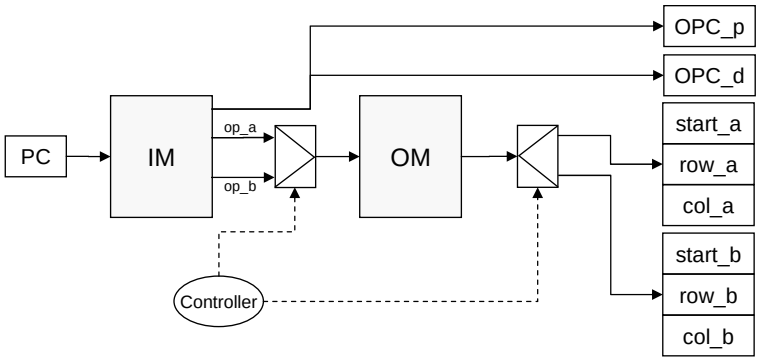

Fig. 2. Instruction and operand fetch

The calculation of the new cell data is performed within five pipeline stages (Fig. 3). First the cell data is read out of the cell memory R. With the value of d, the current Cell-ID (ID), the operation code (OPC_p) and the values of the object memory it is possible to calculate the pointer p to the global cell. In the next pipeline stage the cell data d^* of the neighbor cell is read from the cell memory S using the pointer p. After this step the calculation of d can be performed. In the last pipeline stage the new cell's content is saved in the memories R' and S'. After the latency the pipeline operates on five cells in parallel. Thereby a new result can be computed in each cycle. In order to increase the level of parallelism it is possible to

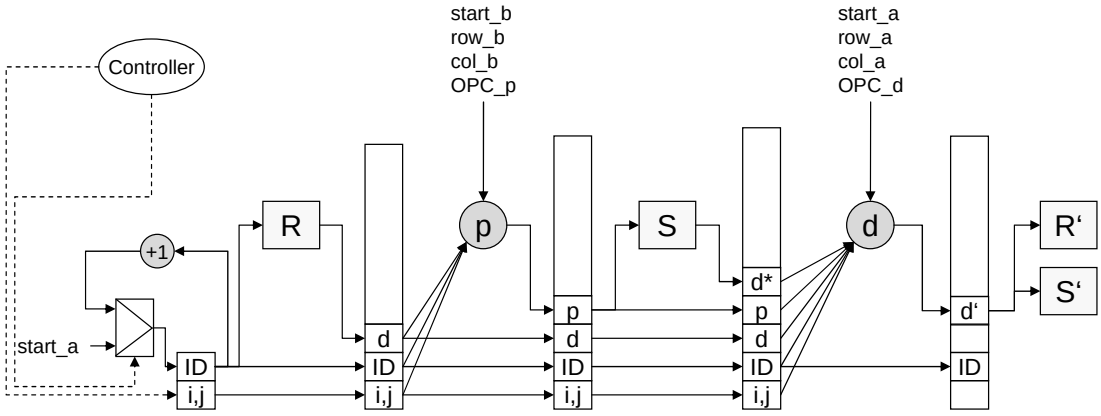

Fig. 3. Pipeline for the cell calculation

use several pipelines in parallel as shown previously [7, 8]. Furthermore it is even possible to implement a GCA fully in parallel as shown in [3].

4 Conclusion

Data parallel operations which are a characteristic of languages like APL or Fortress can easily be implemented on a GCA. Typical APL operations were implemented on the condensed GCA model using one or more generations per operator. A GCA machine was defined which executes GCA instructions in a pipeline manner using several memories. Each GCA instruction controls the execution of a specific generation computation taking into account the start addresses and number of rows/columns of the objects. The degree of parallelism can be increased by using several pipelines in parallel due to the properties of the model (massively parallel, no write conflicts). In addition the pipeline can be modified in a way that the number of generations to execute a data parallel operation is minimized.

References

1. Hoffmann, R., Völkmann, K.P., Waldschmidt, S.: Global Cellular Automata GCA: An Universal Extension of the CA Model. In: Worsch, Thomas (Editor): ACRI 2000 Conference. (2000)
2. Hoffmann, R., Völkmann, K.P., Waldschmidt, S., Heenes, W.: GCA: Global Cellular Automata. A Flexible Parallel Model. In: PaCT '01: Proceedings of the 6th International Conference on Parallel Computing Technologies, London, UK, Springer-Verlag (2001) 66–73
3. Heenes, W.: Entwurf und Realisierung von massivparallelen Architekturen für Globale Zellulare Automaten. PhD thesis, Technische Universität Darmstadt (2006)
4. von Neumann, J.: Theory of Self-Reproducing Automata. University of Illinois Press, Urbana and London (1966)
5. Heenes, W., Hoffmann, R., Jendrsczok, J.: A multiprocessor architecture for the massively parallel model GCA. In: International Parallel and Distributed Processing Symposium (IPDPS), Workshop on System Management Tools for Large-Scale Parallel Systems (SMTPS). (2006)
6. Iverson, K.E.: A programming language. Wiley, New York (1962)
7. Heenes, W., Völkmann, K.P., Hoffmann, R.: Architekturen für den globalen Zellularautomaten. In: 19th PARS Workshop, Gesellschaft für Informatik (GI). (2003) 73–80
8. Hoffmann, R., Heenes, W., Halbach, M.: Implementation of the Massively Parallel Model GCA. In: PARELEC, IEEE Computer Society (2004) 135–139