C to Cellular Automata
and Execution on CPU, GPU and FPGA

Jens Drieseberg, Christian Siemers
Department of Informatics, Clausthal University of Technology
Julius-Albert-Str. 4, 38678 Clausthal-Zellerfeld, Germany
Phone +49 5323 72 7101, 7117, Fax +49 5323 72 7149
E-Mail {jens.drieseberg, christian.siemers}@tu-clausthal.de

Abstract—Over the last decades Cellular Automata (CA) have become more and more present in solving general-purpose problems, but the main issue is how to map a problem to a Cellular Automata model. Special languages were developed for programming such models, but learning a new programming language is very time consuming. Furthermore software developers have to keep in mind the specific structure of Cellular Automata when designing a new algorithm. In this paper we present a method to generate Cellular Automata models from standard C code. The code is transcoded by mapping the complete algorithm written in C to a Cellular Automata model that may be compiled for CPU, GPU and even FPGA without further user interaction.

Keywords—Cellular Automata Models & Algorithms, Fine-Grained Parallel Architectures and FPGA, Efficient Architectures and Implementations

I. INTRODUCTION

CA are well known and studied computing models. They are used in different science areas like biology, chemistry, physics and so on. Introduced by John von Neumann [10] and Konrad Zuse [11] CA models are mostly used to simulate different complex systems. CA consist of a field of connected cells and a set of rules that define the next state of each cell depending on its actual state, as well as the states of the cells in its (local) neighbourhood. The main advantage of CA is that the cells can be updated in parallel, therefore CA systems are suitable for multi/many-core architectures and hardware implementations.

The classic CA model was enhanced by Rolf Hoffmann et al. [4] to Global Cellular Automata (GCA). GCA are not limited to a local neighbourhood, they can have connections across the field of the automata. Furthermore the neighbourhood does not have to be static, it is possible to change the neighbours dynamically and each cell can have its own special rule. They showed that the GCA model can be used for general problems, not only for simulation, and developed a special programming language for the GCA model, called Global Cellular Automata Language (GCA-L) [5]. Moreover they compared a software implementation of the GCA model to an FPGA solution [3] and demonstrated that the hardware implementation is significantly faster.

The Trend/jTrend Project is another example of using CA models for solving general purpose problems. Hui-Hsien Chou et al. [2] developed a Java based CA simulator and a special language to describe the cell rules and the neighbourhood between the cells.

There are many more CA simulators and applications, but all have in common that they must be programmed in a CA matter. The user has to know the structure and has to define rules for the cells and their neighbourhood connections.

Martin Mortensen [6] studied also the usage of CA models for general-purpose problems, but his idea differs from the other approaches. Instead of using special CA constructs that must be defined by a programmer, he developed a method to convert algorithms from a high-level programming language called Tiny Imperative Programming Language (TIP) to CA models and developed a Java based simulator to evaluate them. Therefore, also existing algorithms can be converted to CA without rewriting them.

We are following his approach for the code analysis but generate different CA models that are optimized for the compilation/simulation step for different systems. Furthermore our approach generates less cells and communication messages between the cells while running the algorithm.

Instead of an explicit analysis to identify parallelism in the C algorithms as it is done in [1], the parallel execution of independent code parts is a side product of the CA behaviour.

In the following section we will describe how C code can be transformed to a CA model with parallelization of independent code blocks and automatic handling of synchronization problems. In Section 3 we give an overview of the cell structure and functionality. In Section 4 we present a method to implement the CA models in software and to generate VHDL code from the models for efficient implementation in FPGA logic. Finally we make a comparison between the different implementations in Section 5.

II. C TO CA MODEL

In this section we describe how C code can be mapped to CA models. Currently our algorithm supports a subset of the ANSI C language. Pointers, arrays and structs are not supported yet, as well as recursions and global variables. These specific C constructs are part of our future research.
As explained in Section 1, we are mainly using the method from Martin Mortensen [6] for the code analysis that is required to generate the CA models. But distinct from [6] our method generates low-level functionality cells for an easy implementation in software and especially in hardware. The important step in the analysis is the transformation of the standard C code to a static single assignment form (SSA form). The SSA form is a common intermediate code form used in many compilers for their optimization process. This form offers the opportunity to easily detect data dependencies and insert data flow control structures.

Figure 1 shows an example for the transformation of a C function that contains an if statement. The SSA form is not ANSI C code and cannot be compiled from a standard C compiler. Especially the variable definitions in the if block that are used outside the local block, might be irritating for a C programmer, but all variables are seen global in the SSA form for a complete analysis. In the SSA form each variable is assigned once and the \( \phi \) functions control read access to the variables. They decide, based on the conditional path that is evaluated during the algorithm, which variable must be chosen at the point of the \( \phi \) function. In the example it is easy to see that \( a1 \) must be returned, because \( a0 \) is smaller than 5, but how does \( \phi2 \) decides this?

When following the algorithm flow it can be detected which variable must be chosen, but the CA model has no sequential algorithm flow. The \( \phi2 \) function could be evaluated at the same time as the if statement and the assignment of \( a0 \). Each \( \phi \) function has at least one argument and for each of its arguments a priority and a use condition rule. The use condition of an argument defines when the value of the argument can be used and contains the conditional path that has to be evaluated before using it. \( \phi \) functions with use conditions control the program flow and therefore automatically handle synchronization problems in the parallel execution of the CA.

In our example the use condition for \( a0 \) is that the if statement must be evaluated to false. \( a1 \) has no use condition meaning that it can be used whenever it is evaluated (contains a valid value). If more than one use condition is true, the priority of the arguments comes into scope.

One question might be: Why does a0 gets an “if must be false” condition instead of a1 gets an “if must be true” condition as it is done in the C code? The reason is: When a0 is uncon-
but cleared when an outer loop clears its inner loop. The values of reload variables can be used for the next iteration or outside the loop. Their only function is to store exactly one final value of a variable at the end of an iteration. This value is again chosen by a \( \phi \) function from all values that the variable can have in the loop.

There is another problem with this solution. Caused by the parallel execution wrong values could be read at the beginning of loops. This could occur for \( \phi \) functions that read a variable from the last loop iteration. If the evaluation of the variable is independent of other parts, the value of the actual iteration could be assigned to the variable before the read access is done. Therefore the \( \phi \) function reads a wrong value. Figure 3 shows this situation. In Figure 3(a) \( x2 \) could be evaluated before \( \phi i0 \) and therefore \( y1 \) gets a wrong value, because the evaluation of \( x1 \) and \( x2 \) does not depend on the evaluation of \( y1 \). In [6] this is handled by inserting a \( \text{load variable} \) in front of the \( \text{reload variable} \) to sequentialize the variable access. But this is not necessary in every loop and leads to additional overhead. Our algorithm analyses the loop and inserts a \( \text{load variable} \) at the beginning of the loop only in case if the variable is read before it is written, which saves a lot of cells in the final CA model. Figure 3(b) shows the sequentialized code. The correct value for \( y1 \) is now assigned whatever the execution order is.

Loops make it also necessary to block the evaluation of \( \phi \) functions that are used outside a loop and depend on variables in the loop (\( \text{reload variables} \)) unless the loop is evaluated completely. Otherwise the \( \phi \) functions are evaluated when the variables are assigned in the first loop iteration. To achieve this blocking we could add a "while must be false" condition to each argument in the \( \phi \) functions. But instead we add the condition for the \( \phi \) function itself, which reduces the overhead and guarantees that the \( \phi \) functions are not evaluated before the loop ends.

Finally the following algorithm performs the mapping from C code into a CA model:

1) parse the C code and generate the corresponding syntax tree,
2) introduce \( \text{reload variables} \) and \( \text{load variables} \) (if necessary) for all variables that are assigned in loops,
3) create the SSA form of the syntax tree:
4) a) introduce a unique variable for each assignment,
   b) introduce a \( \phi \) function for every read access to a variable,
   c) analyse which values are possible at the point of each \( \phi \) function,
   d) set the use conditions of the arguments in the \( \phi \) functions,
5) perform a depth-first traversal on the resulting tree and create a corresponding CA cell for each node,
5) create the connections between the CA cells resulting from the tree and from the variable dependencies.

### III. CELL STRUCTURE AND FUNCTIONALITY

Our CA is basically a GCA with static neighbourhood and 13 different cell types that vary in their intrinsic rule to generate their next state. When counting all different logic and arithmetic operation types there are more cells, but we distinguish only \( \text{unary} \) and \( \text{binary operation} \) cells. The cellular behaviour of all \( \text{unary} \) and all \( \text{binary operation} \) cells is the same. \( \text{Unary operation} \) cells get the value from their child, perform an operation on it and send the new value to their parent. \( \text{Binary operation} \) cells get the values from their children, combine them using their specific operation (+, −,...) and send the combined value to their parent.

The main difference to a standard GCA is that we use an asynchronous CA, meaning that the new state of a cell can influence the update process of another cell in the same generation. The cell generation is the number of update processes that are executed since the CA is started. Updating all cells of the CA is synonymical to change the CA generation from \( i \) to \( i + 1 \).

Asynchronous updates cannot happen in a fully parallel architecture like a FPGA, but they are possible when the model is evaluated by a system with less processing units than cells in the CA model, like a CPU. It is an advantage of the architecture. In multi/many-core systems the cores work in parallel and the update of the whole field of cells is done very fast. If one cell waits for the data of another cell it is less important for the performance, because the other cell is working at the same time. In a single-core system one cell is updated after the other and it is waste of time to wait for the

```
1 int x0 = 0;
2 int y0 = 0;
3 while( ... )
4 {
5   int y1 = phi0( x0, x2 ) + 1;
6   int x1 = ... ;
7   int x2 = phi1( x1 ); //reload variable
8 }
```

(a) Code Without Load Variable

```
1 int x0 = 0;
2 int y0 = 0;
3 while( ... )
4 {
5   int x1 = phi0( x0, x3 ); //load variable
6   int y1 = phi1( x1 ) + 1;
7   int x2 = ... ;
8   int x3 = phi2( x2 ); //reload variable
9 }
```

(b) Code With Load Variable

Figure 3. Introducing \textit{Load Variables} To Sequentialize The Program Flow
next update to get another cells data that is already evaluated in the actual generation. When using asynchronous updates, the new data of a neighbour can be used when it is evaluated, instead of waiting for the next cell generation. Caused by the mapping of the C code to its SSA form and the explicit clearing of loop variables, asynchronous updates can be done without further attention.

The cells are mainly generated from a binary tree structure (see Section 2) and so the most cells have three neighbours: One parent and two children. Only three cell types have global connections through the automata. *Function cells* have an additional connection to the cell that stores the return value. *Argument cells* have a global connection to read the value from their associated *variable cells*. *Path condition cells* have to check if another cell is evaluated and if the state of the cell corresponds to a state that is defined at compile time. This is required by C language constructs like *if*, *else* and *while* that split the program flow into different ways (see φ functions in Section 2).

Figure 4 shows an example of a C function that is ported to a CA model. The middle of the figure illustrates the neighbourhood connections of the cells and the right image shows the final CA model.

The following steps are executed when the CA model is evaluated. Items with the same number are processed in parallel:

1) all cells are in the idle state and waiting for evaluate messages that start their specific process (cell rule),
2) when the CA is started, the function cell (func) sets its state to computing and sends an evaluate message to its children,
3) the two assignment cells (=0,=1) set their state to computing and send an evaluate message to their right child,
4) the operation cells (+,*) set their state to computing and send evaluate messages to the children,
5) a) the value cells (1,2,3) set their state to evaluated and send an evaluation complete message to their parent,
   b) the phi cell (phi) sets its state to computing and sends an evaluate message to its child,
6) a) the operation cell (+) takes the values from its children (because of the evaluation complete messages), sets its own value to their sum, sets its state to evaluated and sends an evaluation complete message to its parent,
   b) the argument cell (arg) sets its state to computing and checks if its associated variable is evaluated, at this moment this is not the case, therefore the cell does nothing,
7) the assignment cell (=0) gets an evaluation complete message from its right child, sets its own value to the value from the right child and sends an evaluate message to its left child,
8) the variable cell (var0) takes the value from its parent as its own value, sets its state to evaluated and sends an evaluation complete message to its parent,
9) a) the assignment cell (=0) sets its state to evaluated and sends an evaluation complete message to its parent,
   b) the argument cell recognizes that the variable cell (var0) is evaluated, it takes the value, sets its state to evaluated and sends an evaluation complete message to its parent,
10) the phi cell takes the value, sets its state to evaluated and sends an evaluation complete message to its parent,
11) the operation cell (*) takes the values from its children, sets its own value to their product, sets its state to evaluated and sends an evaluation complete message to its parent,
12) the assignment cell (=) stores the value and sends an evaluate message to its left child,
13) the variable cell (var) sets its value to the value from the parent, sets its state to evaluated and sends an evaluation complete message to its parent,
14) the assignment cell (=) sets its state to evaluated and sends an evaluation complete message to its parent,
15) the function cell sets its value to the value from variable cell (var) and its state to evaluated.

Although the original C algorithm is purely sequential, independent parts in the CA model are automatically evaluated in parallel. Binary operations can evaluate both sides of the operator at the same time and the two assignment cells are evaluated in parallel as far as they could (waiting for the evaluation of var), which could not be expressed in C code. The parallelism in the CA model is intensified the more the algorithm grows and the used data is independent from other parts.

IV. IMPLEMENTATION

Because of the strict partitioning of the cells and the low level functionality each cell has to handle, the implementation of each cell type is not solely complex. All cells have in common that they have a state and can change their state depending on the states and messages of connected cells. Furthermore all cells have a parent cell (except of the main function cell) and most of them have two children. When their update method is called, all cells check if their parent or their children send a message and react accordingly.

The CPU and GPU implementation of the cells are nearly the same, only the call of their update process is different. We developed our algorithm in C++ and used NVIDIA’s parallel computing architecture CUDA [7] for the GPU implementation. The CUDA compiler generates individual executables for the CPU based and the GPU based program, only the partition management must be written twice, not the code for the cells. All cells are stored as C++ objects in a 2-dimensional array as shown in Figure 4(c). When executed on the CPU, every cell update must be called individual in a loop (parallelized with OpenMP [8] for better performance with multi-core CPUs). For the GPU update only one call for the whole CA is necessary, because CUDA manages the threads itself.

The program performed by the CPU and GPU can be seen as a virtual machine that evaluates the CA model. When storing the CA model to a file it is possible to write such a virtual machine in nearly any programming language and therefore it can be executed on an arbitrary system.

V. RESULTS

Table I shows some implementation results for three different C algorithms. The two different values for the algorithms result from the optimization strategies used in the ISE for the synthesis and implementation. For the first value of each algorithm we used a global strategy that optimizes the timing and for the second value we used an area optimization method.

The cells column contains the number of generated cells in the CA model. The register column contains the number of registers that are used in the FPGA and the LUT column expresses the number of used lookup tables. For the Virtex-5 69,120 of both are available. The last column shows the maximal achievable frequency for the actual setting (timing estimation by ISE). One clock cycle corresponds to one update of the whole CA.

The first example is the binary greatest common divisor (gcd) algorithm published by Josef Stein [9]. The implemented
Table II
Execution Times for Different Target Systems

<table>
<thead>
<tr>
<th>name</th>
<th>CPU</th>
<th>GPU</th>
<th>FPGA (speed)</th>
<th>FPGA (area)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcd</td>
<td>6 ms</td>
<td>22 ms</td>
<td>0.0016 ms</td>
<td>0.0024 ms</td>
</tr>
<tr>
<td>loop</td>
<td>35 ms</td>
<td>180 ms</td>
<td>0.0181 ms</td>
<td>0.0199 ms</td>
</tr>
<tr>
<td>loop unrolled</td>
<td>22 ms</td>
<td>17 ms</td>
<td>0.0018 ms</td>
<td>0.0018 ms</td>
</tr>
</tbody>
</table>

Figure 5. Loop Example

```c
int func( int a, int b )
{
    return a + b;
}

int main()
{
    int x, y;
    int res = 0;
    for ( x = 0; x < 10; x++ )
        for ( y = 0; y < 10; y++ )
            res += func( x, y );
    return res;
}
```

Table III
CA Generations to Evaluate the Algorithms

<table>
<thead>
<tr>
<th>name</th>
<th>CPU</th>
<th>GPU</th>
<th>FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcd</td>
<td>380</td>
<td>433</td>
<td>652</td>
</tr>
<tr>
<td>loop</td>
<td>4100</td>
<td>4779</td>
<td>7459</td>
</tr>
<tr>
<td>loop unrolled</td>
<td>325</td>
<td>345</td>
<td>629</td>
</tr>
</tbody>
</table>

In example one and two, the CPU implementation outperforms the GPU implementation, because it benefits more from the asynchronous cell updates as described in Section 3. This is also reflected by the lower generation count in Table III. For algorithms were only few data dependencies exists, as in the third example with the unrolled loop, the GPU implementation is faster. The cells rarely have to wait for each other and the threads can be used advisable. In this case the higher number of cell generations is covered by the high grade of parallel working cells.

The FPGA implementation outperforms the CPU and GPU implementation as expected. The timings of the FPGA implementation correspond to the values of the GPU implementation divided by a factor of approximately 10,000. The number of CA generations is the highest of all, which is caused by the fully parallel architecture. All cell updates are triggered from the same clock and therefore no asynchronous data exchange is possible. Although our graphic card contains hundreds of cores, not all of them work in parallel and asynchronous updates influence the data exchange between the cells. Therefore the number of generations is lower than in the FPGA implementation. As a whole the FPGA implementation is about 2,000 to 13,000 times faster then the corresponding CPU/GPU programs.

VI. CONCLUSION

We presented a new approach to generate CA models from existing C code algorithms. Using this method the main advantage of a CA, its fully parallel architecture, can be used to solve general-purpose problems, without considering the specific structure of the CA. Developers do not have to design rules for each cell and define neighbourhood connections by themselves, it is automatically performed by transcoding the source code. The generated CA models can be evaluated on nearly any system and we showed three different implementations, two software implementations as virtual machines for the CPU and the GPU and a hardware implementation that uses automatically generated VHDL code for FPGA synthesis. We showed that the hardware implementation outperforms the software implementations by an order of magnitude on three different examples.
REFERENCES


