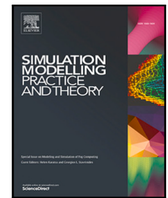
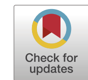
Simulation Modelling Practice and Theory 118 (2022) 102519

Contents lists available at ScienceDirect

Simulation Modelling Practice and Theory

journal homepage: www.elsevier.com/locate/simpat





Efficient simulation execution of cellular automata on GPU

Daniel Cagigas-Muñiz *, Fernando Diaz-del-Rio, Jose Luis Sevillano-Ramos, Jose-Luis Guisado-Lizar

Department of Computer Architecture and Technology, Universidad de Sevilla, Avenida Reina Mercedes s/n, 41012 Sevilla, Spain

ARTICLE INFO

Keywords:
Cellular automata
Parallel computing
Performance optimization
Stencil computation
Graphics Processing Units

ABSTRACT

Graphics Processing Units (GPUs) can be used as convenient hardware accelerators to speed up Cellular Automata (CA) simulations, which are employed in many scientific areas. However, an important set of CA have performance constraints due to GPU memory bandwidth. Few studies have fully explored how CA implementations can take advantage of modern GPU architectures, mainly in the case of intensive memory usage. In this paper, we make a thorough study of techniques (stencil computing framework, look-up tables, and packet coding) to efficiently implement CA on GPU, taking into account its detailed architecture. Exhaustive experiments to validate these implementation techniques for a number of significant memory-bounded CA are performed. The CA analysed include the classical Game of Life, a Forest Fire model, a Cyclic cellular automaton, and the WireWorld CA. The experimental results show that implementations using the presented techniques can significantly outperform a baseline standard GPU implementation. The best performance results of all known implementations of memory bounded CA were obtained. Moreover, some of the techniques, like look-up tables or temporal blocking, are indeed relatively easy to implement or to apply when the transition rules are simple. Finally, detailed descriptions and discussions of the indicated techniques are included, which may be useful to practitioners interested in developing high performance simulations in efficient languages based on CA on GPU.

1. Introduction

Cellular Automata (CA) are mathematical models that evolve in discrete time steps, composed of elements called "cells" that change their state at each time step according to some rules, by taking into account the state of the neighbouring cells. CA are commonly used in complex and/or dynamic system modelling and simulation, mainly in the fields of physics, biology, and computer science [1–4].

A sequential implementation of a cellular automaton using slow interpreted languages can be enough in certain situations in which the output time response is not critical, the CA model evolution rules are simple, the CA data are relatively small, and/or experiments do not need to be repeated many times. In these cases, many practitioners just prototype for slow CPU performance nonparallel programming languages like Python, Matlab, or Octave.

However, there are many real-world applications of CA simulation models that demand a high level of computational capability because their CA model is complicated in terms of evolution rules or data usage, for real-time (or time-constrained) applications, or for parametric studies in which the same experiment must be executed many times under different initial configurations (in

* Corresponding author.
E-mail addresses: dcagigas@us.es (D. Cagigas-Muñiz), fdiaz@us.es (F. Diaz-del-Rio), jlsevillano@us.es (J.L. Sevillano-Ramos), jlguisado@us.es (J.-L. Guisado-Lizar).

https://doi.org/10.1016/j.simpat.2022.102519
Received 8 January 2022; Received in revised form 22 February 2022; Accepted 24 February 2022
Available online 5 March 2022
1569-190X/© 2022 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

which case it is critical to accelerate every individual experiment). Therefore, these applications of CA require the usage of parallel computing techniques. Examples involve scientific problems as diverse as fluid dynamics [5], cancer growth [6], 3D metal forming [7], solidification in materials science [8], or urban land use simulations [9], among many others. It is therefore interesting to analyse some programming fundamentals on how CA implementations on modern computing platforms can be improved when performance is important.

Computer-based implementations of CA are inherently parallel because each cell can be processed independently at each time step. With the arrival in 2007 of programming languages for Graphical Processing Units (GPUs) such as Cuda and later OpenCL, the design and implementation of CA experienced a new impetus. The internal architecture of GPUs has many more computational units (processors) than a CPU. This feature makes GPUs ideal hardware accelerators for implementing CA. There is much room for performance improvement, especially for memory-bounded CA, such as the GoL. Memory-bounded CA are cellular automata that use more time in memory accesses than in computing cell states for the next time step. This means that the GPU arithmetic computational units are not being efficiently used.

At each time step of CA, cell computations can be performed in parallel by several processors without the need for communication or synchronization among them. Several experimental works can be found in the scientific literature about the performance benefits of using GPUs over CPUs (see Section 2).

There is, however, one problem when coding performance-sensitive algorithms using GPUs. GPU manufactures do not provide access to the machine code. For example, in the case of NVIDIA Cuda programming language, only Parallel Thread Execution (PTX) is provided. PTX is a low-level parallel thread execution virtual machine and instruction set architecture (ISA) [10]. Although the syntax of PTX is similar to that of an assembly level programming language, there is not a one to one correspondence between a PTX instruction and a GPU machine code instruction. Moreover, GPU's instruction set architecture (ISA) and/or instruction encoding changes from architecture to architecture. Another important related problem are compilers. In the case of NVIDIA and Cuda, the compiler is a 'black box' that applies code optimization. Nevertheless, it is not possible to determine whether compiler optimizations are always appropriate.

Therefore, there is no way of knowing exactly how the high-level code (Cuda or OpenCL) is actually executed on the GPU. Only some general guidelines based on the GPU architecture are indicated by manufacturers. This makes experimentation, profiling, and testing necessary to ensure effective code improvement. CA are not an exception despite its apparent simplicity.

Getting the maximum performance from GPU implementations is not a trivial task. Knowledge on low level hardware architecture details is needed to take advantage of modern GPU full capabilities. Even GPU programmers do not always squeeze the full potential of their hardware. Concurrent and/or parallel programming has obvious advantages, but it is always more complex to implement. However, as mentioned before, high performance is desired.

For a better comprehension of the main bottleneck involved in GPU performance, our analysis is clearly extensible to higher dimensional problems. Nonetheless, many code optimization techniques described in this study can be extended to three-dimensional CA more or less directly. Three-dimensional CA are another good example of massive data structures that need an efficient GPU implementation for a better performance.

The contributions of this paper can be summarized in three points:

1. An analysis of which GPU architecture aspects influence most in the performance of CA implementations.
2. An analysis of different new algorithms and/or techniques to improve baseline CA implementations performance on GPUs, breaking the current GPU performance barrier for some class of CA.
3. The implementation and validation (with experimental results) of the new algorithms and/or techniques proposed. As a result, a new highly efficient implementation technique is proposed for memory bandwidth bounded CA.

This article is organized as follows. Related works are discussed in Section 2. A generic CA baseline implementation on GPU is described in Section 3. Performance issues related to the implementation of CA on GPU architectures are analysed in Section 4. Section 5 presents new coding techniques for higher performance, which are based on previous analysis. The proposed techniques are tested using some CA models in Section 6. The experimental results are discussed in Section 7. Finally, conclusions and future work are outlined in Section 8.

2. Related works

There have been some studies on the application of GPUs for CA in the scientific literature. Most of them are focused on highlighting the benefits of computing CA on GPUs versus CPUs. Examples can be found in [11,12], and [13]. In these studies, there is little discussion about GPU optimization algorithms and/or techniques applicable to CA.

In addition, they usually focus on only one cellular automaton: the Game of Life (GoL). GoL was created by Conway [14] in 1970 and it is the most referenced and well-known cellular automaton. Unfortunately, these references did not provide data on more (complex) CA models. GoL is considered a 'toy model' cellular automaton that is often far from real dynamic models. An exception to this practice can be found, for example, in [15] where laser dynamics is modelled by using a cellular automaton employing high performance multiprocessors and GPUs.

Several algorithms and issues that influence the CA performance in GPUs were studied in [16] using again GoL cellular automaton as an example. Issues such as the influence of shared memory on GPUs and the thread block size on Cuda were analysed. In particular, this work did not find any influence of the block size on the algorithms used; thus these authors set always the block size to 32×32 threads. They also tested an algorithm called "multicell algorithm", in which each GPU thread processes two cells. Although it can

```
matrix_variable_declaration (A);
fill_matrix_with_initial_values (A);
allocate_memory_in_GPU (p_current_state, p_next_state);
transfer_Host_to_GPU (A, p_current_state);
step = 0;
while step < MAX_STEPS do
    compute_step (p_current_state, p_next_state);
    swap (p_current_state, p_next_state);
    step = step + 1;
end
transfer_GPU_to_Host (p_current_state , A);
print_or_save_results (A);
```

Fig. 1. Pseudo code description of a generic cellular automaton program in GPU.

be beneficial in other types of algorithms or models, it did not improve the results of the baseline GoL implementation. This study provided several interesting results but left significant room for further research to improve the CA performance on modern GPUs. No solutions to improve the baseline GoL implementation using GPUs were provided.

Previously, there were also some nonspecific studies in [17]. However, some of the results and conclusions provided were totally opposed to [16]. This is due to the fact that the authors in [17] worked with old NVIDIA graphic cards (Fermi architecture). These graphics cards were the first generation of full Cuda programmable GPUs that did not have any kind of cache memory in their internal architecture. As it will be seen in later sections, this issue is absolutely critical to understand the performance of CA on modern GPUs.

3. Cellular automata model

The typical bi-dimensional CA (Cellular Automata) workflow on a GPU consists of first declaring two grids (arrays) A and B in memory. The first grid A represents the current state of the CA and the second grid B the state after a time or computation step. Note that using only one grid for the current and next state would produce incorrect values for GPU execution, since a thread could modify a cell state before the other thread has read this state. The content of A (current state) is prefilled with the initial data of the CA on the host computer (usually a multiprocessor). Then, its content is transferred to the GPU memory. From that moment on, both grids (arrays) are only accessed in the GPU memory.

In each time iteration or computation step, the grid B (next state) is calculated based on the content of matrix A (current state). At the end of the computation step, the roles of A and B are exchanged to prepare the next time iteration. In programming languages like C/C++, this implies the use of pointers (*p_current_state*, *p_next_state*). Cuda and OpenCL are based on C/C++ because they are programming languages that focus on code efficiency. Once the last iteration or computation step has been completed, the final contents of the grid containing the current state of the CA must be transferred from the GPU (device) memory to the CPU (host) memory. Fig. 1 shows a pseudo-algorithm that summarizes a typical CA implementation on a GPU.

The *compute_step* function from Fig. 1 implies the execution of at least one kernel on the GPU. A kernel is nothing more than a function executed within the GPU. One of the critical aspects of obtaining good performance is minimizing transfers between the GPU and the host computer as they involve memory latency. This is why memory transfers between GPU and CPU are usually limited to the beginning and the end of the CA simulation. For the same reason, the number of iterations or computation steps performed should be relatively high to take advantage of the parallel architecture of the GPU. The use of relatively small A and B grids of data, the continuous transfer of information between the GPU and CPU, and/or a few time steps can result in an important decrease of performance.

4. Analysis of CA performance

As stated in the Introduction, there is much room for time enhancement in memory-bounded CA. A preliminary analysis was done by us in [18], where we show that the GoL implementations are far from taking full advantage of current architectures using the roofline model [19]. This model is an excellent tool for detecting promptly potential bottlenecks and performance issues. Going forward, in this section, the two factors that have a major influence on the CA performance are to be analysed in more depth.

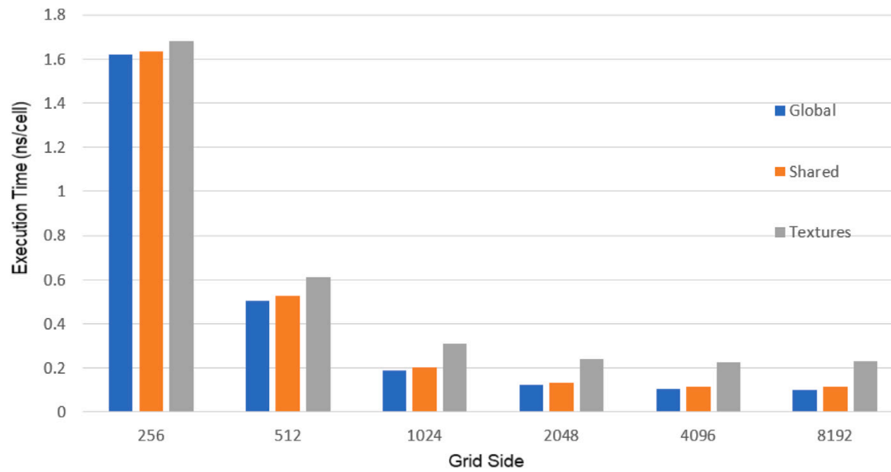


Fig. 2. Comparison of performance of GPU memory types (global, shared and texture memories) in GoL cellular automaton and different grid sizes. A NVIDIA GTX 1650 TI GPU (Turing architecture) was used.

4.1. GPU memory model management

The objective in this subsection is to determine which GPU memory management is more effective for CA implementation. GPUs have several types of memory and it is responsibility of the programmer to determine its management. This first analysis will provide a first approach of how CA should be coded.

To do this, several CUDA implementations of GoL (Game of Life) cellular automaton were tested. There are only two states in GoL: dead or alive. Alive cells remain alive if there are two or three alive neighbour cells. Dead cells become alive with three alive neighbour cells. In any other case, cells become dead or remain dead. Moore neighbourhood (up, down, left, right, and four diagonals) is taken into account. There are some variations of this cellular automaton, including even 3D versions. The GoL is the base for much more complex CA.

Cuda is the GPU programming language used in every experiment and implementation of this paper. It is the native programming language of the NVIDIA manufacturer and it is the predominant one currently. The concepts explained here are perfectly valid and analogous to other GPU languages such as OpenCL. From the programmer's point of view, cuda allows the use of three main types of memory: registers (local memory), shared memory, and global memory.

The general strategy for achieving a good acceleration/performance in GPUs is to try to avoid access to global memory as much as possible. Copying data from global memory to shared memory or using textures are alternatives to avoid many global memory accesses. Textures are an additional type of memory present in NVIDIA GPUs that are specifically geared towards image processing. Thus, three implementations of the GoL based on [20] were tested: using global memory, using shared memory, and using textures. To study the effect of memory, these three memory models have also been tested for different CA (grid) sizes. The summary of the results obtained are shown in Fig. 2 for a total of 1000 time steps.

The overall conclusion that can be drawn from these first results is that although textures are ideal for processing 2-dimensional images, they are not so ideal for implementing bi-dimensional CA. As for the implementation of the GoL using global vs. shared memory at the thread block level, theory indicates that using shared memory can reduce access to global memory if data sharing among threads is carefully programmed. The global memory data bus acts as a bottleneck in case all threads want to access it. However, it is interesting to remark that GoL experimental results in Fig. 2 do not confirm this statement. Results using only global memory are slightly worse than those of the shared memory case. This effect is more noticeable for larger (grid) CA.

A detailed analysis of the CA model (see previous Section 3) and the GPU architecture itself can provide the key to explain this behaviour. The cellular automaton grid (data matrix) that has the current state of the CA only needs to be read from the global memory for each time/computation step. In addition, this data structure is not modified during the entire computation step. This data, like the automatic variables defined in a thread block, are stored in cache memory and in registers within a Multiprocessor Streaming (SM). Moving data that is read from global memory to shared memory in a block of threads creates an unnecessary overhead because that information is already cached either in registers or in the 1 or 2 cache memory level.

Write memory access has a similar behaviour. The cell information processed by a block of threads is written only once and in a different area of the global memory. Therefore, moving the information from global memory to shared memory in each block of threads, and then again to global memory, does not reduce the data traffic to global memory. It only generates a greater overhead by having to move data unnecessarily from one type of memory to another. Cache memory is again a key component because it groups writes in blocks without programmer intervention. Shared memory is useful to gain performance only when threads in a block have to share data frequently over the same memory area (i.e., read/write several times to the same data in the same time step), before writing the results to global memory. This is not the case of the GoL and many CA.

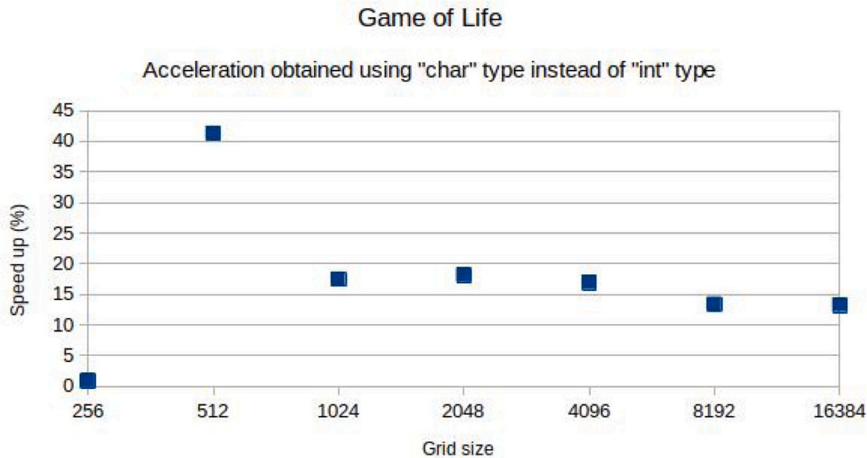


Fig. 3. Acceleration obtained when changing the GoL cellular automaton cell data type from "int" to "char". Fifteen independent executions (tests) for each grid size and data type were performed. The mean time in milliseconds was taken. An NVIDIA GTX 1650 TI graphics card was used. Only the main GoL Cuda kernel was measured using the NVIDIA Profiler. The main Cuda kernel run time is between 74% (256 grid size) and 99% (16384 grid size) of the total program execution time on the GPU.

As a first conclusion, it can be stated that, having enough internal registers and a good cache memory system (which is being improved with each new GPU generation), CA implementations tend to read and write each cell only once from the global memory in each time/computation step. This has also a positive aspect: programming using only global memory is easier, and therefore Cuda code is cleaner. The negative effect of using shared memory when programming CA in the absence of cache memory was also briefly commented in [16].

4.2. Cell size

Another factor to study involves reducing cell size. In the experiments of Section 4.1, the data type "int" was used for each cell. That means 32 bits per cell in memory. This allows to code up to 2^{32} possible states per cellular automaton (grid) cell. In the case of the GoL, only two states per cell are possible (live or dead). Therefore, an 8-bit data type "char" per cell is enough for the GoL cellular automaton and many CA. This reduces memory usage by 4 times with a consequent decrease in memory latency. There are studies of GoL on multiprocessors such as [21] in which cells are encoded with a single bit. Nonetheless, apart from the technical complexity, this solution is not generalizable to other CA. Because Cuda (and OpenCL) is based on C/C++ programming language, it is not possible to use data types smaller than 8 bytes ("char" type).

The Fig. 3 shows the acceleration factor obtained by changing the cell type from "int" to "char" in the GoL cellular automaton. It can be clearly seen that:

1. The acceleration is very irregular for small grid sizes: 1% for a 256×256 grid size and 41% for a 512×512 grid size.
2. Then, the performance improvement tends to be constant for large grids. This is determined by memory bandwidth constraints and GPU functional units.

The maximum acceleration obtained is low in proportion to the decrease in memory usage. GPUs are optimized to work with 32-bit floating point data, which is the same size as the "int" data. Although the 8-bit "char" data type represents a decrease of x4 data in memory, the maximum performance obtained is below 20% for large grid sizes (see Fig. 3). GPUs have functional units designed for *float* or *double* data types that involve 32 or 64 bits respectively. The 8 bit *char* data type is not well suited for the GPU functional units, and can have negative effects in relatively small CA grids as it can be observed in Fig. 3. Nevertheless, in the case of large CA, codifying each CA cell with a smaller data size variable reduces the overall data size and therefore the performance should be better. This gives a hint to readdress memory accesses by using standard and wide variables, as it will be presented in Section 5.2.

5. Techniques for runtime-efficient coding of CA

Based on the previous works mentioned in Section 2, CA model description of Section 3, and the analysis results of Section 4, some techniques for promoting CA efficient performance are proposed. Some of these techniques have been used before, but no study has been carried out to extend them to generic CA on GPUs.

5.1. Look-up tables

An interesting performance improvement consists of trying to reduce or eliminate selective structures ($if - then - else$). GPUs execute the same instructions for all threads (basic computational units) in a warp or block. This style of programming, called single-instruction multiple-thread (SIMT), works well when all threads take the $if - then$ path at once. Otherwise, several passes of the block/warp execution are required. One pass will be needed for those threads that follow the $then$ part and another pass for those that follow the $else$ part. These passes are sequential to each other, thus incrementing the execution time [22]. In general, the $if - then - else$ or $switch - case$ selective structures appear naturally when coding CA rules and it is convenient to suppress them. Programmers usually do not take this issue into account when coding CA.

An alternative is to use what it is going to be defined in this study as "Look-Up Tables" (LUTs) [18]. These tables are data structures that have as input the current state of a cell (row coordinates) and the state and/or number of neighbouring cells (column coordinates). They are coded as bi-dimensional arrays. For instance, in the case of the GoL, this involves using a matrix of 2 rows by 9 columns. The rows represent the two possible states of a cell (live or dead) and the columns represent the number of neighbouring live cells. The content of this matrix indicates the state at which the cell should change in the next time/computation step.

LUTs coding and management is very dependent on the cellular automaton. Nevertheless, in the case of CA with intensive memory usage, the transition rules between states tend to be relatively simple (i.e., need low computational resources). LUTs can be simplified in this set of CA and are relative easy to code. This issue will be analysed in detail with practical CA in Section 6.

5.2. Packet coding

A plausible solution to the limitation imposed by the GPU bandwidth is to access memory only in $float$ or $double$ data chunks. Therefore, each thread computes more than one cell. In this way, the computational load of each thread is increased. As mentioned in Section 2, a somewhat related but different approach was analysed and named "multicell algorithm" in [16], in which each thread (cell) accesses to its neighbours sequentially (see Fig. 4, middle). However, memory accesses were done without taking into account the GPU memory data bus architecture. As a consequence, these authors obtained slightly worse results even than the GoL Baseline version.

Although the original idea was correct, because the ratio of read accesses per cell is reduced from 9/1 (3×3 for a cell) to 12/2 (4×3 for two contiguous cells), the GPU architecture is still not really used efficiently. When a thread processes two cells of a CA, it produces two different (non-coalesced) access requests to the memory (see Fig. 4, middle).

In contrast, it is possible to read/write a set of cells packed in just one memory access (see Fig. 4, bottom). Therefore, memory access requests can be reduced by x2, x4, or more.

To develop this algorithm, several cells must be codified in a "supercell" whose size is that of the Functional Unit bus width. In the case of programming languages like C/C++ (and cuda), the maximum data size available is 64 bits. The ideal scenario, however, would be to utilize even higher data sizes.

Fig. 5 shows how to code 8 cells of the same type in a supercell for the case of GoL computation. It can be observed that, although 8 neighbouring supercells must be read to calculate the new states of a particular supercell, central cells (that is, all except the first and the last one) only need to access cells from the upper and lower supercells.

This technique, which is going to be called "Packet Coding" in this paper, has been scarcely used and for very specific cases. To the authors' knowledge, the unique deep study in the field of CA is [21] and only multiprocessors were used. In [21] the GoL was coded with 1 bit per cell for an CPU implementation. The performance results were remarkable; however, as mentioned in Section 4.2 this solution is not extensible to every CA. Experiments using Packet Coding for different CAs are described in Section 6.

5.3. Temporal blocking

A third approach for improving CA performance on GPUs is to take advantage of the studies that have been made in stencil computing [23]. Stencil computation has been used in many applications and ranges from simulation to machine vision, machine learning applications, or partial differential equation (PDE) solving. This has led to a great deal of research.

A stencil code consists of updating the elements of an array (2D or 3D) based on a fixed pattern. These array elements are also often called cells. The similarities with CA are obvious and GPUs have also been used to achieve higher performance [24,25].

Part of the solution to the low CA performance is to improve the utilization of the GPU computer units. In stencil computing, the CA grid is usually divided into subplanes of the same size (named tiles in stencil computation) and each subplane or tile is processed by a thread block. At each time/computation step, all tiles of the grid are fully processed.

Going further, another more complex technique for stencil computation is temporal blocking, which enhances the temporal reuse of data, and thus reduces the number of data transfers from/to the global GPU memory. It is based on a time-tiled execution, but the operations from several consecutive time steps are combined to exploit data reuse in memory [26].

However, the cell values that are in the edges (ghost or halo zones) depend on other tiles. In the case of GPUs, this means a dependency between thread blocks. This implies that in temporal blocking there must be periodic synchronization between threads. In Fig. 6 a basic scheme of how temporal blocking could be applied to a CA divided into 4 tiles is shown. This technique applied to 3-dimensional stencils is called 3.5D blocking in [27].

Global memory bandwidth reduction is thus achieved by combining several time/computation steps of the same tile to be executed consecutively. Intermediate data reside in registers, shared memory, or cached memory. The problem when using GPUs

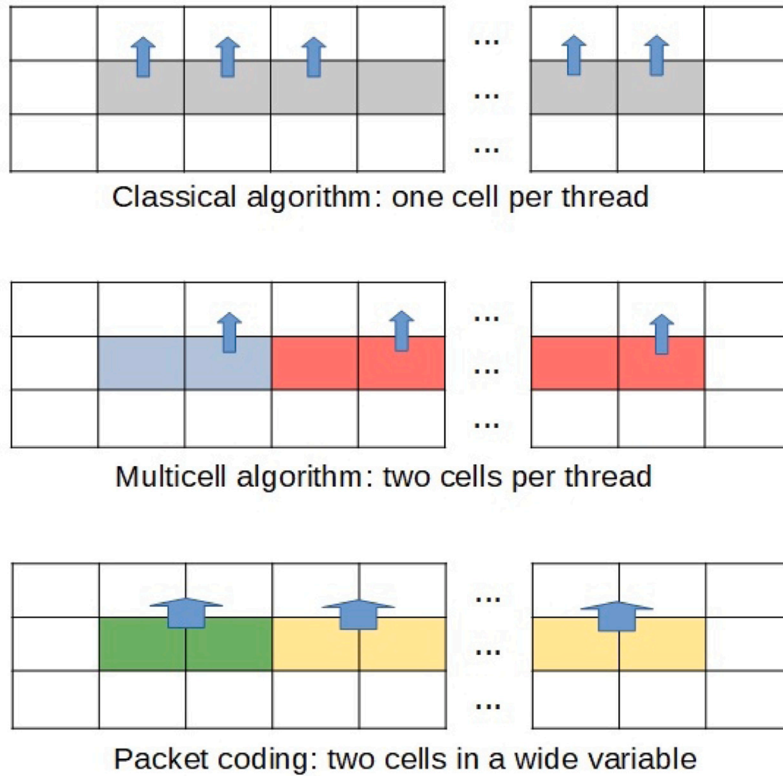


Fig. 4. Memory access patterns comparison for different CA codifications when accessing the North neighbours of a warp of cells (likewise for the rest of neighbours). From up to bottom: in a classical implementation looking for the north cell provokes a coalesced access to a set of consecutive bytes; on the contrary, the multicell coding introduces non-coalesced accessing (only one out of two elements in the case of a two-cell per thread implementation); finally, the novel packet coding reproduces again coalescing accesses but for wider element sizes.

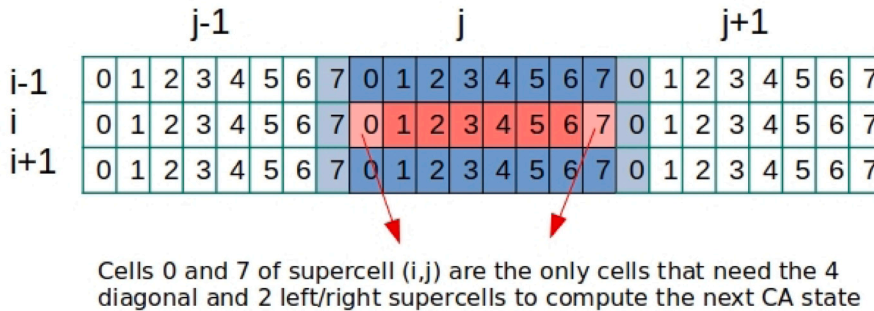


Fig. 5. Packet coding example: one supercell codes 8 cells, and 8 supercells are needed to compute a new supercell in a cellular automaton using Moore neighbourhood.

is that, unlike CPUs, its implementation is more complicated than spatial blocking [28]. For example, for every GPU, it is necessary to calculate and ensure that a thread block has enough shared and local memory to save data for calculating a group of time/computation steps. Consequently, examples of stencil computation in GPUs and temporal blocking are rare.

To apply this technique to CA (especially to the most complex ones), automatic code generators should be used. The two environments that do this in GPUs efficiently are STENCILGEN [29] and AN5D [30]. In addition, the AN5D currently has the best performance results in stencil computing and GPUs and it is publicly available for use.

AN5D improves the 3.5D blocking algorithm obtaining the best performance in stencil computation for both single and double precision floating point. Only a *pragma* directive is necessary to indicate which loop must be translated into Cuda code. In AN5D, the C code that is translated into Cuda code must be composed only of three 'for' loops and a variable assignment. It is not possible to include selective structures (if-else) or previous instructions that have dependencies on the cell values. AN5D must

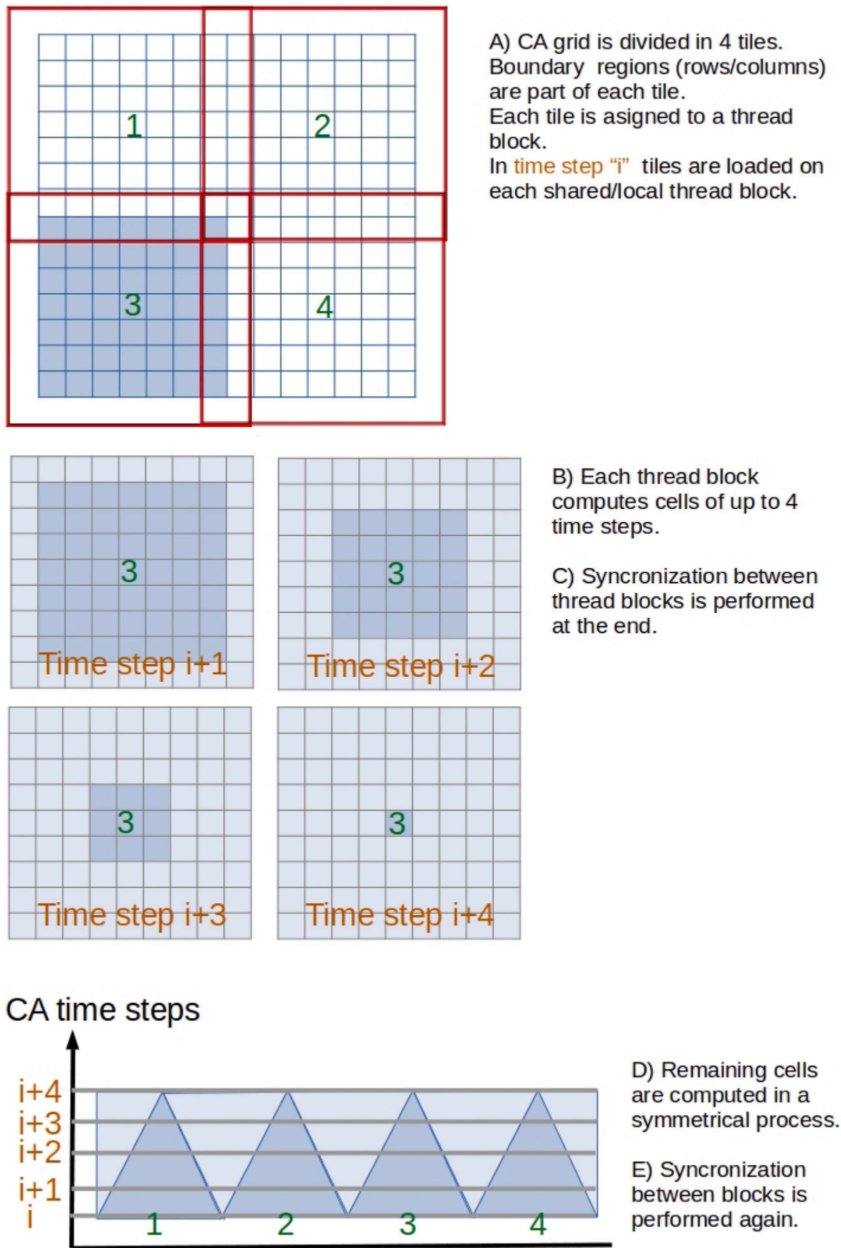


Fig. 6. Basic example of overlapped tiling (temporal blocking) in a GPU.

resolve dependencies at compiling time. For all these reasons, look-up tables (see Section 5.1) must be used too. Therefore, the C sequential code must be codified as shown below, so that AN5D can be compiled to cuda:

```
    ...
#pragma scop
    for (int t = 0; t < TIMESTEPS; t++)
      for (int i = 1; i < SIZE - 1; i++)
        for (int j = 1; j < SIZE - 1; j++) {
           grid[(t+1)%2][i][j] =
                lookup_table [grid[t%2][i][j]]
                [ (grid[t%2][i-1][j] +
                grid[t%2][i+1][j] +
```

Table 1
Mean execution time and standard deviation (left and right values between parenthesis in each table cell) of 15 independent runs in milliseconds of the Game of Life (GOL) cellular automaton for different GPU implementations: baseline, baseline using a look-up table to code rules, temporal blocking using AN5D framework (with default options), packet coding using 32 bits per cell (4 subcells of 8 bits) and packet coding using 64 bits per cell (8 subcells of 8 bits). Best case for each algorithm/technique and each grid size is presented in bold.

Grid size	Baseline Cuda implementation	Look-up table	AN5D (Temporal blocking)	32 bits packet coding (4 subcells per cell)	64 bits packet coding (8 subcells per cell)
256	6.2 (0.0)	5.8 (0.0)	25.7 (0.1)	**3.7 (0.0)**	3.9 (0.0)
512	20.9 (0.4)	20.4 (0.3)	34.9 (0.2)	**8.9 (0.0)**	10.5 (0.2)
1024	88.5 (9.7)	87.4 (9.7)	82.3 (7.5)	39.6 (0.3)	**38.6 (0.8)**
2048	325.2 (11.6)	305.0 (0.9)	264.8 (1.8)	128.1 (8.0)	**114.9 (9.3)**
4096	1299.2 (2.6)	1234.4 (1.2)	1036.0 (8.8)	488.5 (1.5)	**436.0 (1.4)**
8192	5300.5 (16.6)	5052.2 (16.4)	4228.5 (18.2)	1977.6 (8.6)	**1770.2 (10.8)**
16384	21303.0 (60.8)	20324.4 (58.6)	17066.8 (71.8)	7960.2 (31.4)	**6996.2 (27.2)**

```
                grid[t%2][i][j-1] +
                grid[t%2][i][j+1] +
                grid[t%2][i-1][j-1] +
                grid[t%2][i-1][j+1] +
                grid[t%2][i+1][j-1] +
                grid[t%2][i+1][j+1]) ];
        }
#pragma endscop
    ...
```

The variable assignment inside the three loops calculates the new cell state based on neighbour cells. The main data structure used should be only a 3 dimensional array. First array dimension contains the current and new grids. The grids swap their roles in each time step. Second and third array dimensions contain the rows and columns.

6. Experimental results

In this section, the algorithms and techniques proposed before are verified experimentally. Four representative CA that have been proposed in the scientific bibliography are selected. Two CA have Moore neighbourhood and two have Von Neuman neighbourhood. The number of states ranges from 2 to 15. Rules are diverse and the NVIDIA profiler indicates memory bandwidth overhead when using the standard baseline implementation version.

In the next subsections, the results obtained for each cellular automaton are described. For all experiments in this paper, a NVIDIA GTX 1650 TI graphics card was used in a computer with an AMD Ryzen 5 3600 processor. Linux-Mint 20.04 operating system, NVCC 10.1, and GCC 9.3 compilers were part of the software configuration. Cuda programming language was used to code CA. Every test of this article was executed 15 times and the average and standard deviation are presented.

Experiments (Cuda implementations) use different CA grid sizes. From 256 × 256 cells to 16384 × 16384 cells. Results are detailed in tables instead of graphics to appreciate the performance differences in small grid sizes. The baseline Cuda implementation result is always placed at the first table column. This is the reference implementation when comparing with the other results. Only the CA main Cuda kernel is measured using the NVIDIA Profiler in each case. Initial procedures like setting up the original grid states, are not taken into account. The main CA kernel computes the time steps. It represents between 72% (256 × 256 grid) and 99% (16384 × 16384 grid) of the total execution time for the baseline Cuda implementation. For all experiments, 1024 time steps were used. The source code of all experiments is available at [31].

6.1. Game of life

The Game of Life (GoL) cellular automaton was the first proposed cellular automaton. This simple cellular automaton is ideal to make a first test on the algorithms and techniques proposed in the sections before. In Table 1 there is a summary of the results obtained.

Results in Table 1 include GoL Cuda implementation using only LUTs, AN5D framework, packet coding of 32 bits per cell, and packet coding of 64 bits per cell. The average execution times in the baseline and LUTs versions are very similar. The AN5D version has better execution time but only for 1024 × 1024 grid size and above. In the case of 16384 × 16384 GoL grid (the biggest grid), there is a speed up of 25% when comparing with GoL baseline execution time. Packet coding implementations present the best results: a speed-up above 300% is achieved for the largest CA. However, there is almost a 14% time improvement when using the 64 bit packet coding algorithm (8 subcells per cell) instead of the 32 bit version (4 subcells per cell). Therefore, the next CA will use this packet coding version.

A Kruskal–Wallis test [32] was performed to determine whether or not there was a statistically significant difference between the median run times of the different techniques or algorithms used. Each technique or algorithm was applied to 7 different grid sizes (from 256 to 16384) as shown in Table 1.

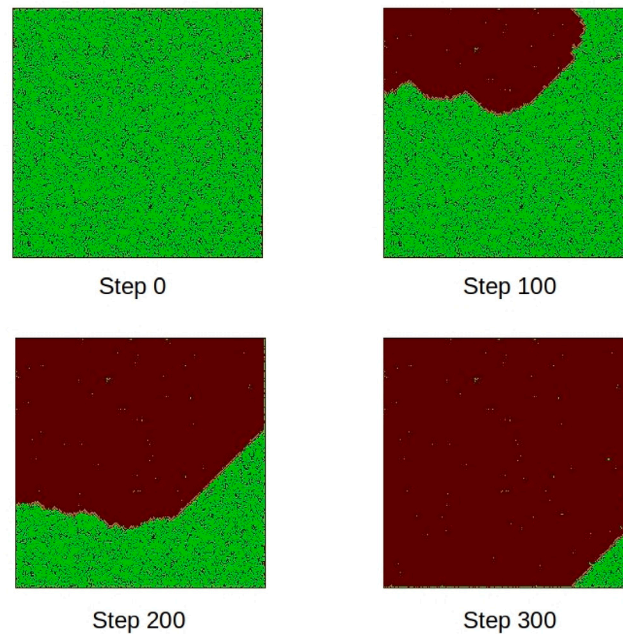


Fig. 7. Example of Forest Fire cellular automaton evolution for a 256 × 256 grid and 300 time steps. Fire cells are showed in red colour, tree cells in green colour, empty cells in brown colour and ash cells in grey colour.

The test revealed that the behaviour of all algorithms/techniques used was the same (H = 0.475, p = 0.924). That is, there was no statistically significant difference in behaviour between two or more of the tested algorithms. Similar results were obtained in the rest of the CA tested (See Sections 6.2–6.4):

- Forest Fire (H = 0.255, p = 0.968)
- Cyclic Cellular Automaton (H = 1.241, p = 0.871)
- WireWorld (H = 0.475, p = 0.924)

A series of Wilcoxon signed-rank tests [33] were also performed to determine whether there were differences between the baseline version of GoL and the other algorithms/techniques used (pair-wise comparisons). The tests revealed that there was a statistically significant difference in the mean execution time between the baseline version and the 32-bit packet coding (z = 0.0, p = 0.015625), 64-bit packet coding (z = 0.0, p = 0.015625) and Look-Up Table (z = 0.0, p = 0.015625) versions. For AN5D, the difference was not as significant (z = 5.0, p = 0.15625). This is primarily because AN5D behaves worse than the other algorithms/techniques for small grid sizes.

6.2. Forest fire

The Forest Fire cellular automaton is used in simulating how a fire spreads in a forest. There are many other models based on CA that study how a fire is propagated (see [34,35] for instance). The implementation selected here is based on Ref. [36]. This model has been chosen because it complies with temporal blocking and AN5D. The transition rules are: a "tree cell" with at least one "fire cell" neighbour fire becomes a "fire cell". A "fire cell" becomes "ash cell". An "ash cell" becomes "empty cell" if it does not have a neighbour "fire cell". An "empty cell" remains empty. Fig. 7 shows an example of a Forest Fire cellular automaton execution.

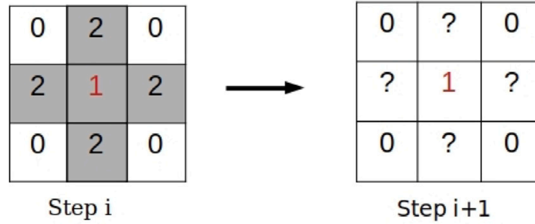
In this model, LUT and AN5D implementations were considered. Forest Fire uses Von Neumann neighbourhood (4 cells) instead of Moore neighbourhood (8 cells) used in GoL. This issue is important when coding look-up tables in AN5D.

If an "empty cell" is labelled with 0, a "tree cell" with 1, and an "ash cell" with 2, then a "fire cell" should be labelled with 9 to implement the cellular automaton rules in a look-up table. It must be taken into account that AN5D only allows arithmetic operations (sums) of neighbour cells when calculating state transitions. Thus, when using Von Neumann neighbourhood, the state $i+1$ value must be higher than 4 x state i value, if any rule (state transition) of i (or lower than i) depends on detecting a state $i+1$. Fig. 8 illustrates this issue using two examples of cell transitions in the Forest Fire cellular automaton.

In conclusion, the LUT in Forest Fire cellular automaton has 9 rows and 9 × 4 columns. Rows between 4th and 8th are not used. In the worst case, a cell can have 4 "fire cells" so 36 columns (9×4) in the array are needed in a Cuda implementation.

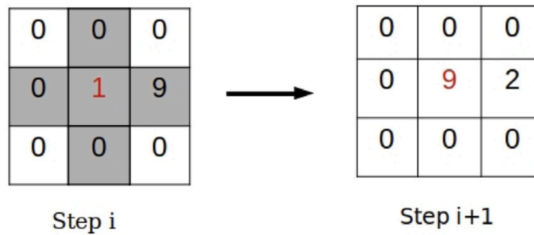
There is an alternative to this oversized look-up table: use a compacted array. Instead of adding neighbour states/values, a direct neighbour state check can be performed. Thus, look-up tables can be simplified. In case of Forest Fire this can be done just detecting a "fire cell".

Case A: neighbour cells/state sum value is below fire cell/state value



“Tree cell” remains “tree cell” because there is is not any neighbour “fire cell”: 2+2+2+2 < 9

Case B: neighbour cells/state sum value is equal or above fire cell/state value



“Tree cell” turns into a “fire cell” because there is one neighbour “fire cell” : 0+0+0+9 ≥ 9

Fig. 8. An example of Forest Fire cellular automaton cell/state transitions. “Fire cell” state must be 9 value in order to easily detect (only using neighbour sums) whether there is a neighbour “fire cell” or not.

Table 2
Mean execution time and standard deviation (left and right values between parenthesis in each table cell) of 15 independent runs in milliseconds of the Forest Fire cellular automaton for different GPU implementations: baseline, baseline using a look-up table, baseline using a compact look-up table (and an additional operation to detect a “fire cell”), and temporal blocking using AN5D framework (with default options). Best case for each algorithm/technique and each grid size is presented in bold. .

Grid size	Baseline Cuda implementation	Look-up table	Compact look-up table	AN5D (Temporal blocking)
256	5.0 (0.0)	**5.0 (0.0)**	6.0 (0.0)	18.7 (0.2)
512	17.5 (0.2)	**17.5 (0.2)**	21.5 (0.3)	20.1 (0.2)
1024	80.5 (5.3)	81.8 (4.0)	90.3 (10.0)	**59.1 (0.3)**
2048	289.5 (5.7)	281.1 (11.0)	320.1 (0.5)	**210.1 (2.7)**
4096	1152.0 (10.0)	1145.1 (12.6)	1289.8 (1.8)	**677.3 (7.8)**
8192	4655.6 (50.9)	4714.0 (36.4)	5275.1 (9.4)	**2588.1 (20.0)**
16384	18692.4 (253.5)	18023.8 (114.8)	23371.4 (66.2)	**10775.4 (558.7)**

The Cuda code of the main kernel could be similar to this:

```
...
cell = grid[i][j];
is_fire = grid[i-1][j]==FIRE || grid[i+1][j]==FIRE ||
          grid[i][j-1]==FIRE || grid[i][j+1]==FIRE;
new_cell = lookup_table[cell][is_fire];
...
```

The compact look-up table using this approach is reduced to 4 rows (one for each cell state) and 2 columns (0 if there is no “fire cell” or 1 if there is at least a “fire cell”). The cost of this memory usage reduction yields an extra computation time for each cell. Because each cell has to perform 4 comparisons and 3 OR (sum) operations.

In Table 2 it can be seen the results obtained for the Forest Fire implementation. The compact look-up table version has the worst results. The extra computational time needed for each cell does not compensate the look-up table memory reduction. In this

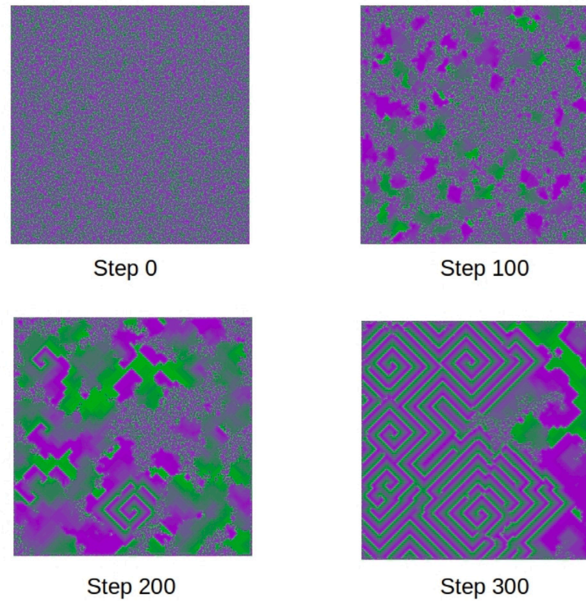


Fig. 9. Example of Cyclic Cellular Automaton evolution for a 256 × 256 grid and 300 time steps. There are 15 different cells/states. Cell colours range from purple light to green light.

Table 3
Mean execution time and standard deviation (left and right values between parenthesis in each table cell) in milliseconds of 15 independent runs of the Cyclic Cellular Automaton for different GPU implementations: baseline, baseline using a look-up table, packet coding using 16 bits per cell (2 subcells per cell), packet coding using 32 bits per cell (4 subcells per cell) and packet coding using 64 bits per cell (8 subcells per cell). Best case for each algorithm/technique and each grid size is presented in bold.

Grid size	Baseline Cuda implementation	Compact look-up table	16 bits packet coding (2 subcells per cell)	32 bits packet coding (4 subcells per cell)	64 bits packet coding (8 subcells per cell)
256	11.7 (0.4)	5.5 (0.0)	8.0 (0.0)	**3.7 (0.0)**	6.4 (0.0)
512	41.4 (0.7)	19.4 (0.3)	26.1 (0.1)	**10.1 (0.1)**	13.1 (0.3)
1024	136.4 (10.1)	87.1 (8.3)	97.4 (9.9)	**42.1 (0.4)**	42.3 (0.3)
2048	514.6 (6.8)	303.2 (12.1)	330.2 (1.0)	132.1 (9.3)	**127.7 (13.9)**
4096	2102.3 (30.2)	1236.2 (11.4)	1328.8 (9.9)	528.1 (8.5)	**499.7 (11.8)**
8192	8302.9 (67.7)	4887.5 (51.9)	5264.6 (17.8)	1974.1 (8.4)	**1858.9 (36.0)**
16384	33602.7 (110.2)	21761.4 (80.4)	20818.0 (51.6)	8033.1 (141.2)	**7289.8 (127.1)**

case, the small memory reduction is not significant. The original look-up table implementation improves again the Cuda baseline solution. Nevertheless, as in GoL, the improvement is still not significant. On the other hand, the AN5D solution can speed-up the baseline solution by almost 80% (8192 × 8192 grid size). This is a relevant improvement when comparing to baseline GoL and AN5D GoL versions where up 25% of speed-up was obtained.

The Wilcoxon signed-rank tests performed in this cellular automaton showed that there was a statistically significant difference in the mean execution time between the baseline version and the compact look-up table (z = 0.0, p = 0.015625). There was no statistically significant differences in the mean execution time between the baseline version and the look-up table (z = 0.0, p = 0.015625). It can be observed that the absolute time values for each grid size are just slightly better in case of the look-up table algorithm/technique when comparing to the baseline version. The AN5D was near to have also a significant difference in the mean execution time (z = 3.0, p = 0.078125). Again, the abnormal behaviour of AN5D for small grid sizes is the main cause of this result.

6.3. Cyclic cellular automaton

The Cyclic Cellular Automaton was created by David Griffeath in 1991 [37]. Applications of Cyclic CA range from sensor networks [38] to crystallization process simulations [39].

Transition rules are again simple: an i cell state becomes an $i+1$ cell state if there is a $i+1$ neighbour cell state. There are one, two-and three-dimensional versions of Cyclic CA.

Here a 15 state bi-dimensional Cyclic Cellular Automaton was tested. Von Neumann neighbourhood is again considered. In Fig. 9 it can be seen an evolution during 300 time steps. The spiral patterns that form the Cyclic Cellular Automaton after a period of time can be observed clearly.

Table 4
Mean execution time and standard deviation (left and right values between parenthesis in each table cell) in milliseconds of 15 independent runs of the WireWorld Automaton for different GPU implementations: baseline compression (2 cells per byte), baseline using a look-up table, temporal blocking using AN5D framework (with default options) and packet coding using 64 bits per cell (8 subcells per cell). Best case for each algorithm/technique and each grid size is presented in bold. (*) No time measurement was possible because NVIDIA profiler limitations.

Grid size	Baseline Cuda implementation	Compression (2 cells per byte)	Look-up table	AN5D (Temporal blocking)	64 bits packet coding (8 subcells per cell)
256	6.1 (0.0)	4.7 (0.0)	6.0 (0.1)	24.0 (0.3)	**4.0 (0.0)**
512	20.8 (0.0)	13.7 (0.1)	20.5 (0.2)	26.3 (0.4)	**11.1 (0.0)**
1024	86.1 (8.0)	51.2 (0.2)	89.3 (9.4)	87.9 (2.6)	**40.5 (0.1)**
2048	311.9 (7.6)	183.7 (5.3)	319.4 (8.0)	287.5 (8.4)	**125.9 (11.8)**
4096	1240.4 (3.0)	722.6 (1.9)	1230.8 (22.7)	961.2 (9.8)	**460.4 (3.1)**
8192	5176.8 (18.4)	3069.8 (27.7)	5067.6 (114.6)	(*)	**1881.0 (11.7)**
16384	21746.7 (145.1)	12428.2 (305.9)	20498.8 (79.3)	(*)	**7721.0 (36.6)**

Experiments applied to this cyclic cellular automaton include using a look-up table and packet coding in three versions: 16, 32, and 64 bits. The 64 bit version has again better performance than the 32 bits for large CA. The 16-bit version results are interesting. They show that even a low packing coding rate (2 subcells per cell) improves the baseline version. Nevertheless, the performance is far from the 32 and 64 bit versions. It is again experimentally proved the importance to fit the data memory access to the GPU functional units (32 bits for *float* and 64 bits for *double*). The results are summarized in Table 3.

The look-up table was compacted in this case. The uncompacted look-up table using the same codification explained in the Forest Fire would be extremely big (its size grows exponentially due to 15 states) and it would be difficult to code and to initialize. On the contrary, the compact look-up table size results to have just 16 × 2 bytes and the resulting code is clear and easy to implement. Performance using an uncompacted look-up table is worse because very big look-up tables interfere in the cache access. Thus, this Cyclic Cellular Automaton version uses a code similar to Forest Fire cellular automaton with a compact look-up table like this:

```
#define N 15
...
cell = grid[i][j];
transition = grid[i-1][j]==(cell+1)%N ||
             grid[i+1][j]==(cell+1)%N ||
             grid[i][j-1]==(cell+1)%N ||
             grid[i][j+1]==(cell+1)%N;
new_cell = lookup_table[cell][transition];
...
```

In contrast to the GoL and Forest Fire, the implementation with a compact look-up table does show a significant speed-up improvement (up to 70% for large grid sizes and more than x2 for small grid sizes) when compared to the Cuda baseline version. Nonetheless, it is the packet coding implementation that boosts the performance above 450% in case of large grids (16384 × 16384).

The Wilcoxon signed-rank tests performed for the cyclic cellular automaton revealed that there was a statistically significant difference in the mean execution time between the baseline version and every algorithm/technique used: look-up table ($z = 0.0$, $p = 0.015625$), packet coding 16 bits ($z = 0.0$, $p = 0.015625$), packet coding 32 bits ($z = 0.0$, $p = 0.015625$) and packet coding 64 bits ($z = 0.0$, $p = 0.015625$).

6.4. Wireworld

The Wire World cellular automaton was created by Brian Silverman in 1987. It is well suited to simulate electronic circuits and is Turing-complete. It was even used to simulate a complete computer [40].

This cellular automaton uses Moore neighbourhood. There are four states or cell types: empty, wire, electron head, and electron tail. Rules are: "empty cells" remain empty. An "electron head cell" turns always into an "electron tail cell". An "electron tail cell" turns always into a "wire cell". A "wire cell" turns into a "electron head cell" if exactly one or two of the neighbouring cells are "electron head cell". Otherwise remains wire.

For test purposes, a concentric set of intercommunicated wire squares are simulated. At step 0, a set of random "electron head cells" are placed in "wire cells". In Fig. 10 there are four cellular automaton screenshots for a 1024 step simulation. It can be observed that the number of electron cells grow in each time step and they almost cover the whole wire circuit at the end of the simulation.

In this last experimental case, every proposed technique is tested and compared. There is also a new implementation: compression. CA were tested until now using a *char* type (8 bits) per cell.

In Table 4 experiment results performed using the Wire World Cellular Automaton are showed. The compression version scales very well: there is almost a 50% of speed up improvement when comparing to the baseline version. Now the look-up table version has again lower execution time than the baseline version, but is far from the compressed implementation performance. The AN5D (temporal blocking) framework provides better results than the look-up table and baseline versions for 1024 × 1024 and 2048 × 2048

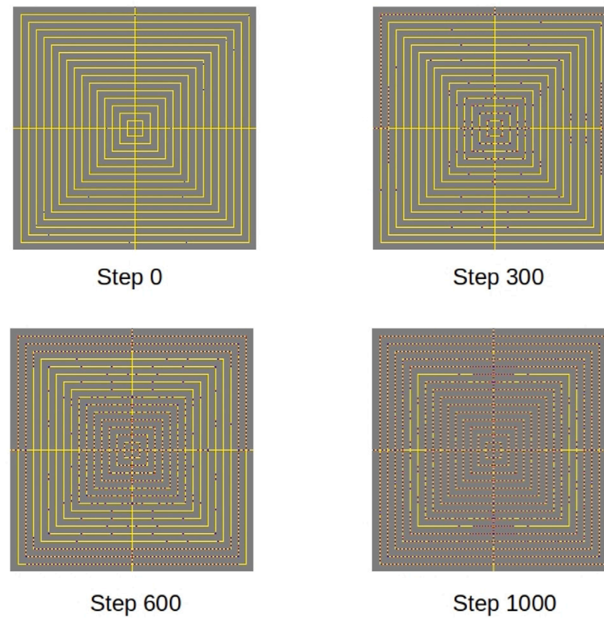


Fig. 10. An example of a WireWorld cellular automaton evolution for a 256 × 256 grid and 1000 time steps. Empty cells are showed in grey colour, wire cells are showed in yellow colour, electron tails are showed in red colour, and electron heads are showed in blue colour.

grid sizes. Nevertheless, the results for 8192 × 8192 and 16384 × 16384 sizes could not be obtained due to the limitations of the NVIDIA profiler.

Finally, the Wilcoxon signed-rank tests were again performed for the Wire World cellular automaton. Results showed that there was a statistically significant difference in the mean execution time between the baseline version and: the compressed version ($z = 0.0$, $p = 0.015625$), and packet coding 64 bits ($z = 0.0$, $p = 0.015625$). The AN5D could not be compared due to the lack of the results for the 4096, 8192, and 16384 grid sizes.

Packet coding technique has the best performance results for each grid size, even when compared with the compression implementation.

7. Discussion

Evidently, due to the vast types of CA, a finite set of experimental CA tests can never be exhaustive. However, the main aspects of 'classical CA' are captured in this paper and they allow to draw some important conclusions on the run-time efficiency of their execution over GPUs. To the best of our knowledge, no one has so far been able to improve the baseline implementations on GPUs of the four CA shown here. This can be extended very probably to many other CA because, firstly, there have not been many efforts in the current literature to improve the runtime efficiency on GPU and, secondly, some of the necessary codifications are actually very unconventional. The baseline GPU implementation of a CA can be beaten by several improvements that allow to come closer to the hardware constraints (especially that of memory bandwidth) imposed by the platform. In this work, we have achieved to outperform the baseline standard Cuda runtime using three different techniques. Going further, we think that our results can be generalized to the majority of memory bounded CA, using some of the techniques shown here.

Translating CA rules into a look-up table always improves the baseline solutions, because the common 'if-then-else' set of structures (used to code CA rules) can have an important performance penalty for a big number of states (above 4). Of course, a study of how to codify CA rules into a look-up table must be thoroughly done, mainly to avoid excessive comparison, logical and multiple indexing operations.

However, for CA with a small number of states (between two and four), like GoL, Forest Fire, and Wire World, the benefit of using a look-up table is not large (see Tables 1–3). Even a negative performance impact can be seen in these cases if a compact look-up table is used to reduce the size of the table arrays at the cost of a higher computation time, as shown in Forest Fire results in Table 2). On the contrary, CA with a bigger number of states, like the cyclic cellular automaton, can achieve up to a 50% of speed-up (see Table 3) even when look-up tables are compacted. The complexity of implementing a look-up table depends on the cellular automaton, but it is essential when using the AN5D framework (temporal blocking, see Section 5.3).

Temporal blocking has never been used before to implement CA solutions in GPUs. The reason is probably the implementation complexity and/or the lack of meta-compilers to generate Cuda and/or OpenCl temporal blocking code. First, the experimental results obtained in this study prove that temporal blocking always improves CA baseline solutions. As expected, this performance improvement is only noticeable in large CA grids: above 1024 × 1024 in the cases of GoL and Wire World, and above 512 × 512 in the

case of Forest Fire. Temporal blocking loses its potentiality when CA grids can be cached in L1 or L2 cache memory. Second, AN5D is the only publicly available framework to implement temporal blocking in stencil computation. Unfortunately, this framework is not flexible enough to implement any cellular automaton. However, the implementation simplicity of AN5D (only adding *pragmas* to a sequential C code) makes it an interesting option. Further research should be addressed to create additional temporal blocking frameworks for CA.

Packet coding is by far the best performance coding solution for memory bandwidth bounded CA. Baseline implementation results are always improved for every cellular automaton tested and for any grid size. Performance improvements can achieve up to 5x acceleration. This solution takes advantage of the fact that GPU data buses are designed to access large data chunks in parallel. In Section 6 it has been experimentally proved that it is much better to issue one 64 bit data bus access than eight 8 bit data bus accesses. In the later case, the memory access coalescing of 8 hardware threads (cells) is not done properly and the threads compete for the data bus, while there is only one data access in the former case. Data bus management is therefore critical. Even CA with half grid size in memory (compressed CA) can get worse performance results if the data are not managed in an efficient way (see Comparison of Wire World compression and packet coding results in Table 4). Through our study, it can be clearly concluded that a CA implemented in GPU with: *a*) simple transition rules, *b*) a relative big number of different states (above 4), and *c*) Von Neuman neighbourhood, can obtain essential performance improvements using packet coding techniques.

In addition, another technique can speed up even more the execution due to the evident memory reduction that it would introduce: compressing the cell state into the minimum number of bits. Compression is for some CA easy to apply if it does not complicate the rule coding. Reviewing 'classical bi-dimensional CA' described in the scientific bibliography, it can be concluded that most of them need a maximum of 16 states or below. These CA include all examples used in this paper plus Brian's Brain, Codd's cellular automaton, Langton's Loop, CoDi, and Langton's ant, among others. A good exception to these CA is the original Von Neumann original cellular automaton (29 states) and some variations like Nobili's Cellular Automaton (32 states). Therefore, using a 4 bit coding scheme for every cell can be enough for a big CA set. That implies a 50% of memory usage reduction and less memory-related contingencies. This implementation case can be an even better reference than the baseline Cuda version when comparing the goodness of other techniques and/or algorithms proposed in this study (see Wire World results in Table 4).

It must be remarked that these techniques may not be necessarily beneficial when applied to CA with complex rules (i.e., high computational load per hardware thread). This is clear for the packet coding technique because a single thread must perform the computational work of several cells. In this case, a baseline solution where each hardware thread computes a single cell may be more efficient. Meanwhile, other techniques like using look-up tables or temporal blocking could still obtain better results.

Finally, despite the diversity of the CA tested here, their diverse implementation techniques, and the variability of their initial conditions, it has been demonstrated in Section 6 that there are no statistical differences between the median run times of the different techniques or algorithms. This is a consequence of the high degree of parallelization of the simulation models over a GPU, which gives them a strong scaling, i.e., given one of this CA model having a data size X and a runtime Y, then increasing the data size to 4*X implies an approximate runtime of 4*Y. The results of the Kruskal–Wallis test reflect this fact. On the contrary, when making a pairwise comparison through the Wilcoxon signed-rank test between the baseline implementation and each of the other algorithm results, it is proved that, in general, there are statistical differences (improvements). These statistical differences are a consequence of the improvements of the algorithms/techniques used.

8. Conclusions and future work

This paper presents the first (to the authors' knowledge) thorough study of the application and effectiveness of certain optimizations and techniques (stencil computing framework, look-up tables, and packet coding) to improve the performance of memory bandwidth-limited CA on GPU.

It was shown that it is possible to approach the limit imposed by GPU memory bandwidth on some CA implementations using these techniques. In addition, some of them, such as look-up tables or temporal blocking, are not technically difficult to implement when the transition rules are simple.

The performance improvements obtained can be quite significant, which is especially true for large CA. In this sense, the performance of three-dimensional CA implementations on GPU can take important advantage of these improvements and future research is needed.

Additionally, future work should address the effect of these optimizations on energy consumption and instantaneous power. As discussed in [41], memory accesses and computing units are the two main energy consumption components in GPU devices. In this paper, we have considered different CA which are mostly memory-bounded applications, and, accordingly, the presented optimization techniques focus mainly on reducing memory accesses. By means of these optimizations, it is expected that the instantaneous power would decrease as fewer memory accesses would be made. Likewise, the energy consumption would decrease due to both the lower power and the execution time reduction. Further studies should confirm these hypotheses.

Finally, further studies are also needed to advance in the state of the art of frameworks that generate automatic GPU implementations of CA using temporal blocking, and in the combination of techniques described and tested in this study to achieve even higher performance ratings.

Acknowledgements

This study was funded by the research project of Ministerio de Economía, Industria *y* Competitividad, Gobierno de España (MINECO), Spain and the Agencia Estatal de Investigación (AEI) of Spain, cofinanced by FEDER funds (EU): Par-HoT (Parallel Data Processing based on Homotopy Connectivity: Applications to Stereoscopic Vision and Biomedical Data, PID2019-110455GB-I00) and CIUCAP-HSF:US-1381077.

References

[1] S.F. Judice, Lattice gas cellular automata for fluid simulation, in: Encyclopedia of Computer Graphics and Games, Springer International Publishing, Cham, 2018, pp. 1–8, http://dx.doi.org/10.1007/978-3-319-08234-9_184-1.
[2] B. Arca, T. Ghisu, G.A. Trunfio, GPU-accelerated multi-objective optimization of fuel treatments for mitigating wildfire hazard, J. Comput. Sci. 11 (2015) 258–268, http://dx.doi.org/10.1016/j.jocs.2015.08.009.
[3] R. Lubas, J. Was, J. Porzycki, Cellular automata as the basis of effective and realistic agent-based models of crowd behavior, J. Supercomput. 72 (6) (2016) 2170–2196, http://dx.doi.org/10.1007/s11227-016-1718-7.
[4] J. Kroc, F. Jiménez-Morales, J.L. Guisado, M.C. Lemos, J. Tkáč, Building efficient computational cellular automata models of complex systems: background, applications, results, software, and pathologies, Adv. Complex Syst. 22 (05) (2019) 1950013.
[5] K.R. Tubbs, F.T.-C. Tsai, GPU accelerated lattice Boltzmann model for shallow water flow and mass transport, Internat. J. Numer. Methods Engrg. 86 (3) (2011) 316–334, http://dx.doi.org/10.1002/nme.3066, URL https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.3066.
[6] A.G. Salguero, A.J. Tomeu-Hardasmal, M.I. Capel, Dynamic load balancing strategy for parallel tumor growth simulations, J. Integr. Bioinform. 16 (1) (2019) 20180066, http://dx.doi.org/10.1515/jib-2018-0066.
[7] M. Sitko, K. Banas, L. Madej, Scaling scientific cellular automata microstructure evolution model of static recrystallization toward practical industrial calculations, Materials 14 (2021) (2021) http://dx.doi.org/10.3390/ma14154082.
[8] B. Jelinek, M. Eshraghi, S. Felicelli, J.F. Peters, Large-scale parallel lattice Boltzmann-cellular automaton model of two-dimensional dendritic growth, Comput. Phys. Comm. 185 (3) (2014) 939–947, http://dx.doi.org/10.1016/j.cpc.2013.09.013.
[9] C. Xia, H. Wang, A. Zhang, W. Zhang, A high-performance cellular automata model for urban simulation based on vectorization and parallel computing technology, Int. J. Geogr. Inf. Sci. 32 (2) (2018) 399–424, http://dx.doi.org/10.1080/13658816.2017.1390118.
[10] A. Kerr, G. Diamos, S. Yalamanchili, A characterization and analysis of PTX kernels, in: 2009 IEEE International Symposium on Workload Characterization, IISWC, 2009, pp. 3–12, http://dx.doi.org/10.1109/IISWC.2009.5306801.
[11] M.J. Gibson, E.C. Keedwell, D.A. Savić, An investigation of the efficient implementation of cellular automata on multi-core CPU and GPU hardware, J. Parallel Distrib. Comput. 77 (2015) 11–25, http://dx.doi.org/10.1016/j.jpdc.2014.10.011.
[12] S. Rybacki, J. Himmelspach, A. Uhrmacher, CPU and GPU based simulation of cellular automata - a performance comparison, in: Proceedings of the 1st SIMUL, 2009, pp. 62–67.
[13] E. Millán, P. Martínez, G. Gil Costa, M. Piccoli, A. Printista, C. Bederian, C. García Garino, E. Bringa, in: A. De Giusti (Ed.), Parallel implementation of a cellular automata in a hybrid CPU/GPU environment, XVIII Congreso Argentino de Ciencias de la Computación, 2013, pp. 184–193.
[14] E.R. Berlekamp, J.H. Conway, R.K. Guy, Winning Ways for Your Mathematical Plays, second ed., A K Peters/CRC Press, New York, USA, 2001.
[15] C. Daniel, F. Diaz-del Rio, M. López-Torres, F. Jiménez-Morales, J.L. Guisado, Developing efficient discrete simulations on multicore and GPU architectures, Electronics 9 (2020) 189, http://dx.doi.org/10.3390/electronics9010189.
[16] E. Nicolas, N. Wolovick, F. Piccoli, C. Garcia Garino, E. Bringa, Performance analysis and comparison of cellular automata GPU implementations, Cluster Comput. 20 (2017) (2017) http://dx.doi.org/10.1007/s10586-017-0850-3.
[17] W.-m.W. Hwu, GPU Computing Gems Jade Edition, first ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.
[18] F. Diaz-del Rio, D. Cagigas-Muñiz, J.-L. Guisado-Lizar, J.-L. Sevillano-Ramos, Efficient parallel implementation of cellular automata and stencil computations in current processors, in: P. Nicopolitidis, S. Mistra, L. Yang, B. Zeigler, Z. Ning (Eds.), Advances in Computing, Informatics, Networking and Cybersecurity - a Book Honoring Prof. Mohammad S. Obaidat's Significant Scientific Contributions, Springer-Nature, 2022, pp. 1–29.
[19] G. Ofenbeck, R. Steinmann, V. Caparros, D.G. Spampinato, M. Püschel, Applying the roofline model, in: 2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, 2014, pp. 76–85, http://dx.doi.org/10.1109/ISPASS.2014.6844463.
[20] A. Simpson, Oak ridge leadership computing facility, URL https://github.com/olcf/game_of_life_tutorials/tree/master/CUDA.
[21] G. Oxman, S. Weiss, Y. Be'ery, Computational methods for Conway's game of life cellular automaton, J. Comput. Sci. 5 (2013) (2013) http://dx.doi.org/10.1016/j.jocs.2013.07.005.
[22] D.B. Kirk, W.-m.W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, Morgan Kaufmann Publishers, Burlington, MA, 2010.
[23] P.S. Rawat, M. Vaidya, A. Sukumaran-Rajam, A. Rountev, L. Pouchet, P. Sadayappan, On optimizing complex stencils on GPUs, in: 2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS, 2019, pp. 641–652.
[24] A. Schäfer, D. Fey, High performance stencil code algorithms for GPGPUs, Procedia Comput. Sci. 4 (2011) 2027–2036, http://dx.doi.org/10.1016/j.procs.2011.04.221.
[25] J. Holewinski, L.-N. Pouchet, P. Sadayappan, High-performance code generation for stencil computations on GPU architectures, in: Proceedings of the 26th ACM International Conference on Supercomputing, New York, NY, USA, 2012, pp. 311–320, http://dx.doi.org/10.1145/2304576.2304619.
[26] P. Rawat, Optimization of Stencil Computations on GPUs (Electronic Thesis Or Dissertation), Ohio State University, 2018.
[27] A.D. Nguyen, N. Satish, J. Chhugani, C. Kim, P. Dubey, 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs, in: SC, IEEE, 2010, pp. 1–13.
[28] K. Hou, H. Wang, W.-c. Feng, Gpu-UniCache: Automatic code generation of spatial blocking for stencils on GPUs, in: Proceedings of the Computing Frontiers Conference, in: CF'17, Association for Computing Machinery, New York, NY, USA, 2017, pp. 107–116, http://dx.doi.org/10.1145/3075564.3075583.
[29] P.S. Rawat, M. Vaidya, A. Sukumaran-Rajam, M. Ravishankar, V. Grover, A. Rountev, L.-N. Pouchet, P. Sadayappan, Domain-specific optimization and generation of high-performance GPU code for stencil computations, Proc. IEEE 106 (11) (2018) 1902–1920.
[30] K. Matsumura, H. Zohouri, M. Wahib, T. Endo, S. Matsuoka, AN5D: automated stencil framework for high-degree temporal blocking on GPUs, in: International Symposium on Code Generation and Optimization, 2020, pp. 199–211, http://dx.doi.org/10.1145/3368826.3377904.
[31] D.C.-M. niz, Cellular automata software repository, URL https://github.com/dcagigas/GPU-Cellular-Automata.
[32] E. Ostertagova, O. Ostertag, J. Kováč, Methodology and application of the Kruskal-Wallis test, Appl. Mech. Mater. 611 (2014) 115–120.
[33] D. Rey, M. Neuhäuser, Wilcoxon-signed-rank test, international encyclopedia of statistical science, in: International Encyclopedia of Statistical Science, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 1658–1659, http://dx.doi.org/10.1007/978-3-642-04898-2_616.
[34] J.G. Freire, C.C. DaCamara, Using cellular automata to simulate wildfire propagation and to assist in fire management, Nat. Hazards Earth Syst. Sci. 19 (1) (2019) 169–179, http://dx.doi.org/10.5194/nhess-19-169-2019, URL https://nhess.copernicus.org/articles/19/169/2019/.
[35] Y. Zhao, D. Geng, Simulation of forest fire occurrence and spread based on cellular automata model, in: ICAIIS 2021: 2021 2nd International Conference on Artificial Intelligence and Information Systems, Chongqing, China, May 28 - 30, 2021, ACM, 2021, pp. 304:1–304:6, http://dx.doi.org/10.1145/3469213.3471332.
[36] L. Hugo, P. Hugo, P. Thomas, AutoCelle in C++, URL https://github.com/hugofloter/CelularAutomaton.
[37] D. Griffeath, Self-organizing two-dimensional cellular automata: 10 still frames, in: Designing Beauty: The Art of Cellular Automata, Springer International Publishing, Cham, 2016, pp. 1–12, http://dx.doi.org/10.1007/978-3-319-27270-2_1.
[38] K. Kwak, Y. Baryshnikov, E. Coffman, Cyclic cellular automata: A tool for self-organizing sleep scheduling in sensor networks, in: Proceedings - 2008 International Conference on Information Processing in Sensor Networks, IPSN 2008, 2008, pp. 535–536, http://dx.doi.org/10.1109/IPSN.2008.69.
[39] R. González-García, G. Castanon, H.E. Hernández Figueroa, 2D photonic crystal complete band gap search using a cyclic cellular automaton refination, Photon. Nanostruct.: Fundam. Appl. 12 (2014) (2014) http://dx.doi.org/10.1016/j.photonics.2014.09.003.
[40] V. Gladkikh, A. Nigay, Wireworld++: A cellular automaton for simulation of nonplanar digital electronic circuits, Complex Systems 27 (2018) (2018).
[41] C. Luo, R. Suda, A performance and energy consumption analytical model for GPU, in: 2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing, 2011, pp. 658–665, http://dx.doi.org/10.1109/DASC.2011.117.