Peer-reviewed version available at Electronics 2020, 9, 189; doi:10.3390/electronics9010189

Article

Developing Efficient Discrete Simulations on Multi-Core and GPU Architectures

Daniel Cagigas-Muñiz[1], Fernando Diaz-del-Rio[1], M.R. López-Torres[1], F. Jiménez-Morales[2] and J.L. Guisado[1,*]

[1] Department of Computer Architecture and Technology, University of Seville, Spain; e-mail@e-mail.com
[2] Department of Condensed Matter Physics, University of Seville, Spain; e-mail@e-mail.com
* Correspondence: jlguisado@us.es

Abstract: In this paper we show how to efficiently implement parallel discrete simulations on Multi-Core and GPU architectures through a real example of application: a cellular automata model of laser dynamics. We describe the techniques employed to build and optimize the implementations using OpenMP and CUDA frameworks. We have evaluated the performance on two different hardware platforms that represent different target market segments: high-end platforms for scientific computing, using an Intel Xeon Platinum 8259CL server with 48 cores and also an NVIDIA Tesla V100 GPU, both running on Amazon Web Server (AWS) Cloud, and on a consumer-oriented platform, using an Intel Core i9 9900k CPU and an NVIDIA GeForce GTX 1050 TI GPU. Performance results are compared and analysed in detail. We show that excellent performance and scalability can be obtained in both platforms, and we extract some important issues that imply a performance degradation for them. We also found that current Multi-Core CPUs with large core numbers can bring a performance very near to that of GPUs, even similar in some cases.

Keywords: laser dynamics; parallel computing; cellular automatas; GPUs and Multi-Core processors performance

1. Introduction

Discrete simulation methods encompass a family of modeling techniques which employ entities that inhabit discrete states and evolve in discrete time steps. Examples include models with an intrinsic discrete nature, such as cellular automata (CA) and related lattice automata, like lattice gas automata (LGA) or lattice Boltzmann method (LBM), and also discretization of continuos models like many stencil-based partial differential equation (PDE) solvers and particle methods based on fixed neighbor lists. They are a powerful tool that has been widely used to simulate complex systems of very different kinds (in which a global behaviour results from the collective action of many simple components that interact locally) and to solve systems of differential equations.

To accurately simulate real systems, the quality of the computed results very often depends on the number of data points used for the computations and the complexity of the model. As a result, realistic simulations often involve too large runtime and memory requirements for a sequential computer. Therefore, efficient parallel implementation of this kind of discrete simulations is extremely important. But this type of discrete algorithms has a strong parallel nature, because they are composed of many individual components or cells that are simultaneously updated. They also have a local nature, since the evolution of cells is determined by strictly local rules, i.e. each cell only interacts with a number of neighboring cells. Thanks to this, they are very suitable to be implemented efficiently on parallel computers [1,2].

In this paper, we study the efficient parallel implementation of a real application of this type, a CA model of laser dynamics, on Multi-Core and GPU architectures, employing the most commonly used software frameworks for these platforms today: OpenMP and CUDA, respectively. In both cases, we describe code optimizations that can speed-up the computation and reduce memory usage. In either case we have evaluated the performance on two different hardware platforms that represent different target market segments: on a high-end chip intended for scientific computing or for servers and on a consumer-oriented one. In the case of the Multi-Core architecture, the performance has been evaluated on a dual socket server with 2 high-end Intel Xeon Platinum 8259CL processors (completing 48 cores between them) running on Amazon Web Server (AWS) Cloud, and also on a PC market Intel Core i9 9900k processor. For the GPU architecture, we present performance evaluation results on a high-end GPGPU NVIDIA Tesla V100 GPU running on AWS Cloud and on a consumer-oriented NVIDIA GeForce GTX 1050 TI. In all cases, we reported speedups compared to a sequential implementation. The aim of this work is to extract lessons that may be helpful for practitioners trying to implement discrete simulations of real systems in parallel.

The considered application uses cellular automata, a class of discrete, spatially-distributed dynamical systems with the following characteristics: spatial and temporal discrete character, local interaction and synchronous parallel dynamical evolution [3,4]. They can be described as a set of identical finite state machines (cells) arranged along a regular spatial grid, whose states are simultaneously updated by a uniformly applied state-transition function that refers to the states of their neighbors [5]. In the last decades, CA have been successfully applied to build simulations of complex systems in a wide range of fields, including physics (fluid dynamics, magnetization in solids, reaction-diffusion processes), bio-medicine (viral infections, epidemic spreading), engineering (communication networks, cryptography), environmental science (forest fires, population dynamics), economy (stock exchange markets), theoretical computer science, etc [6–8]. They are currently being very used, in particular, for simulations in geography (specially in urban development planning [9], future development of cities [10], and land use [11]) pedestrian or vehicular traffic [12,13], and bio-medicine (applied to physiological modeling, for example for cancer [14], or epidemic modeling [15]).

The application considered in this study is a cellular automata model of laser dynamics introduced by Guisado et. al., capable of reproducing much of the phenomenology of laser systems [16–19]. It captures the essence of laser as a complex system in which its macroscopic properties emerge spontaneously due to the self-organization of its basic components. This model is a useful alternative to the standard modeling approach of laser dynamics, based on differential equations, in situations for which the approximations considered for them are not valid, for instance for lasers ruled by stiff differential equations, lasers with difficult boundary conditions, or very small devices. The mesoscopic character of the model also allows to have results impossible to be obtained by the differential equations, such as studying the evolution of its spatio-temporal patterns.

In order to reduce the runtime of laser simulations with this model by taking advantage of its parallel nature, a parallel implementation of it for computer clusters (distributed-memory parallel computers), using the message-passing programming paradigm, was introduced in [20,21]. It showed a good performance on dedicated computer clusters [22] and also on heterogeneous non-dedicated clusters with a dynamic load balancing mechanism [23].

Due to the excellent ratio performance/price and performance/power of the Graphics Processing Units (GPUs), it is very interesting to implement the model on them. GPUs are massively parallel graphics processors originally designed for running interactive graphics applications, but that can also be used to accelerate arbitrary applications, what is known as GPGPU (General Purpose computation on GPU) [24]. They can run thousands of programming threads in parallel, providing speedups mainly from 10x to 200x compared to CPU (depending on the application and on the optimizations of its implementation), at very affordable prices. Therefore, GPUs have widespread use today in high performance scientific computing. Their architecture is formed by a number of multiprocessors, each

of them with a number of cores. All cores of a multiprocessor share one memory unit called shared memory and all multiprocessors share a memory unit called global memory.

A first version of a parallel implementation of the model for GPUs was presented in [25]. Even when that first implementation did not explore all the possible optimizations to boost the performance on that platform, it showed that the model could be successfully implemented on GPU. A speedup of 14.5 on a NVIDIA GeForce GTX 285 (a consumer-oriented GPU intended for low-end users and gamers) compared to an Intel Core i5 750 with 4 cores at 2.67 GHz was obtained. The GPU implementation described in the present paper differs from that previous one in that this new version has been carefully optimized to extract all possible performance from the GPU and his performance has been evaluated not only on a consumer-oriented GPU, but also on a Tesla scientific high-end GPU.

Another interesting parallel platform to implement discrete simulations today are Multi-Core processors. Since around 2005 all general-purpose CPUs implement more than one CPU (or "core") on the processor chip. For a decade, the number of cores in standard Intel x86 processors was modest (mainly from 2 to 8). But in the last years, there are high-end CPUs in the market including up to several dozen cores (now up to 18 cores for Intel Core i9 and up to 56 cores for Intel Xeon Platinum). Therefore, Multi-Core CPUs can start to be competitive with GPUs to implement parallel discrete simulations, specially taking into account that its parallelization with OpenMP is much easier than for GPUs. Therefore, we also present the first parallel implementation of the CA laser dynamics model for Multi-Core architectures and compare its performance on current high-end Multi-Core CPUs to the performance obtained on GPUs.

The remainder of the paper is organized as follows: Section 2 reviews the related work in the field of discrete simulations via cellular automata and their parallel implementation on Multi-Core and GPU Architectures. Section 3 describes the methodology employed in this work, trying to give useful indications to researchers interested in parallelizing efficiently their own codes. Section 4 presents the results and discusses their interpretation and significance. Finally, Section 5 summarizes the contents of this paper, the conclusions and indicates interesting future work.

2. Related work

Most parallel implementations of CA models on Multi-Core processors or GPUs were presented after 2007. In the case of Multi-Core processors, they became generalised only from 2005 onwards, and started to be used for parallel simulations in the following years. As regards GPUs, before 2007 there were few works devoted to the parallel implementation of cellular automata models on GPUs, because they had to adapt somehow their application to a shading language (a special purpose programming language for graphics applications), such as OpenGL. An example is the paper from Gobron et. al. [26], that studies a CA model for a biological retina obtaining a 20x speedup as compared to the CPU implementation. After the introduction in 2007 of CUDA (Compute Unified Device Architecture), a general purpose programming language for GPUs of the NVIDIA manufacturer, followed soon by a multi-platform one called OpenCL, the usage of GPUs in scientific computing exploded.

Let us review some relevant parallel implementations of CA models on Multi-Core CPUs and GPUs introduced from 2007 on.

Rybacki et. al. [27] presented a study of the performance of seven different very simple cellular automata standard models running on a single core processor, a multi core processor and a GPU. They found that the performance results were strongly dependent on the model to be simulated.

Bajzát et. al. [28] obtained an order of magnitude increase in the performance of the GPU implementation of a CA model for an ecological system, compared to a serial execution.

Balasalle et. al. [29] studied how to improve the performance of the GPU implementation of one of the simplest two-dimensional CAs —the game of life— by optimizing the memory access patterns. They found that carefull optimizations of the implementation can produce a 65% improvement in runtime from a baseline implementation. However, they did not study other more realistic CA models.

Special interest has been devoted to GPU implementations of Lattice Boltzmann methods, a particular class of CA. Some works have been able to obtain spectacular speedups for them. For instance, [30] reported speedups of up to 234x respect to single-core CPU execution without using SSE instructions or multithreading.

Gibson et. al. [31] presents the first thorough study of the performance of cellular automata implementations on GPUs and multi-core CPUs with respect to different standard CA parameters such as lattice and neighbourhood sizes, number of states, complexity of the state transition rules, number of generations, etc. They have studied a "toy application", the "game of life" cellular automaton in two dimensions and two multi-state generalizations of it. They employed the OpenCL framework for the parallel implementation on GPUs and OpenMP for multi-core CPUs. That study is very useful for researchers to help them choose the right CA parameters, when that is possible, taking into account their impact in performance. Also to help to explain much of the variation found in reported speedup factors from literature. Our present work is different and complementary to that study in the sense that the game of life is a toy model very useful to study the dependence of performance on general CA parameters, but it is also very interesting to study the parallelization and performance of a real application instead of a toy model such as the game of life, as we do in this work.

3. Materials and Methods

3.1. Cellular automaton model for laser dynamics simulation

We present parallel implementations for Multi-Core CPUs and for GPUs of the cellular automaton model of laser dynamics introduced by Guisado et. al. [16–18].

A laser system is represented in this model by a two-dimensional CA which corresponds to a transverse section of the active medium in the laser cavity.

Formally the CA model is made of:

a) A regular lattice in a two-dimensional space of $L \times L$ cells. Each lattice position is labelled by the indices (i, j). Also to avoid boundary problems and to best simulate the properties of a macroscopic system we use periodic boundary conditions.

b) The state variables associated with each node (i, j). In the case of a laser system we need two variables: one for the lasing medium $a_{ij}(t)$ and the other for the number of laser photons $c_{ij}(t)$. $a_{ij}(t)$ is a boolean variable: 1 represents the excited state of the electron in the lasing medium in cell (i, j) and 0 is the ground state. For the photons $c_{ij}(t)$ is an integer variable in the range $[0, M]$ where M is an upper limit, that represent the number of laser photons in cell (i, j). The state variables $a_{ij}(t)$ and $c_{ij}(t)$ represent "bunches" of real photons and electron, the values of which are obviously smaller than the real number of photons and electrons in the system and are connected to them by a normalization constant.

c) The neighborhood of a cell. In a cellular automata the state variables can change depending on the neighboring cells. In our model the *Moore neighborhood* is employed: the neighborhood of a cell consists of the cell itself and the eight cells around it at positions north, south, east, west, northeast, southeast, northwest and southwest.

d) The evolution rules that specify the state variables at time $t + 1$ in function of their state at time t. From a microscopic point of view the physics of a laser can be described by five procesess:

i) The pumping of the ground state of the laser medium to the excited state. In this way energy is supply to the lasing medium. This process is considered to be probabilistic: If $a_{ij}(t) = 0$ then $a_{ij}(t + 1) = 1$ with a probability λ.

ii) The stimulated emission by which a new photon is created when an excited laser medium cell surrounded by one or more photons decays to the ground state: If $a_{ij}(t) = 1$ and the sum of the values of the laser photons states in its neighboring cells is greater than 1, then $c_{ij}(t + 1) = c_{ij}(t) + 1$ and $a_{ij}(t + 1) = 0$.

iii) The non-radiative decaying of the excited state. After a finite time τ_a a excited laser medium cell will go to the ground state $a_{ij}(t+1) = 0$ without the generation of any photon.

iv) The photon decay. After a given time τ_c, photons will escape and its number will decrease by one unit $c_{ij}(t+1) = c_{ij}(t) - 1$.

v) Thermal noise. In a real laser system there is a thermal noise of photons produced by spontaneous emissions and they cause the initial start-up of the laser action. Therefore in our CA model a small number of photons less than 0.01% are added at random positions at each time step.

```
1: Initialize system
2: Input data
3: for time step = 1 to maximum time step do
4:    for each cell in the array do
5:       Apply noise photons creation rule (Fig. 2)
6:       Apply photon and electron decay and evolution of temporal variables
         (Fig. 3 )
7:       Apply pumping and stimulated emission rules (Fig. 4)
8:    end for
9:    Refresh value of c matrix with contents of c' matrix
10:   Calculate populations after this time step
11:   Optional additional calculations on intermediate results
12: end for
13: Final calculations
14: Output results
```

Figure 1. Pseudo code description of the main program for the CA laser model.

3.2. Sequential implementation of the model

The algorithmic description of the model using pseudo code is shown in Figs. 1 to 4. The main program is described in Fig. 1. The structure of the algorithm is based on a time loop, inside of which there is a data loop to sweep all the CA cells. At each time step, first the state of all the cells of the lattice is updated by applying the transition rules, and then the total populations of laser photons and electrons in the upper state are calculated by summing up the values of the state variables a_{ij} and c_{ij} for all the lattice cells. Because we are emulating a time evolution, the order of the transition rules for each time step can be switched. Of course, different orders get to slightly different particle quantities, but on the whole, CA evolution is similar. Fig. 2 defines the implementation of the noise photons creation rule. The photon and electron decay rules and the evolution of temporal variables are described in Fig. 3. Finally, Fig. 4 describes the implementation of the pumping and stimulated emission rules.

In order to simulate a parallel evolution of all the CA cells, we use two copies of the c_{ij} matrix, called c and c'. En each time step, the new states of c_{ij} are written in c' and the updated values of this matrix are only copied to c after finishing with all the CA cells. In the algorithmic description of the implementation of the model we have used two temporal variables, $\tilde{a}_{ij}$ and $\tilde{c}^k_{ij}$ as time counters, where k distinguishes between the different photons that can occupy the same cell. When a photon is created, $\tilde{c}^k_{ij} = \tau_c$. After that, 1 is subtracted to $\tilde{c}^k_{ij}$ for every time step and the photon will be destroyed when $\tilde{c}^k_{ij} = 0$. When an electron is initially excited, $\tilde{a}_{ij} = \tau_a$. After that, 1 is subtracted to $\tilde{a}_{ij}$ for every time step and the electron will decay to the ground level again when $\tilde{a}_{ij} = 0$.

1: {Introduce n_n number of photons in random positions}
2: **for** $n = 0$ to $n_n - 1$ **do**
3: {Generate two random integers in $(0, size - 1)$ interval}
4: $i \longleftarrow random\_number(0, L_x - 1)$
5: $j \longleftarrow random\_number(0, L_y - 1)$
6: {Look for first value of k for which $\tilde{c}_{ij}^{k} = 0$}
7: **while** $\tilde{c}_{ij}^{k} \neq 0$ and $k \leq M$ **do**
8: $k \longleftarrow k + 1$
9: **end while**
10: **if** $k \leq M$ **then**
11: {Create new photon}
12: $c'_{ij} \longleftarrow c'_{ij} + 1$
13: $\tilde{c}_{ij}^{k} \longleftarrow \tau_c$
14: **end if**
15: **end for**

Figure 2. Pseudo code diagram for the implementation of the noise photons rule.

3.3. Parallel Frameworks for Efficient CA Laser Dynamics Simulation

Algorithms described in previous Section 3.2 arise from a direct conversion of the systems of differential equations that represents the CA laser model. The efficient execution of these algorithms in parallel platforms to generate fast simulations of a bunch of different input parameters requires many specific considerations for each hardware platforms. To begin with, modern out-of-order execution superscalar processors achieves an almost optimal time execution of operations when operands reside in CPU registers. That is, they reach the so-called $data - flow - limit$ of the algorithm, being the most patent bottlenecks that of the real dependences among operations and the difficult branch predictions. In fact, taking some simple assumptions around these bottlenecks, some authors have proposed simple processor performance models that predicts computing times with enough accuracy [32]. Above this, when many operations cannot be executed over CPU registers, memory model is the other crucial factor.

In relation with our CA model, a simple inspection of the code and of the data evolution brings to light that memory usage is massive and that an elevated branch misprediction ratio is expected. First assertion is obvious: matrices that contains c_{ij}, $\tilde{c}_{ij}^{k}$, a_{ij}, $\tilde{a}_{ij}$ supposes many megabytes for those lattice widths that emulates a correct behavior of the laser dynamics. Second assertion comes mainly from two code features: the use of random values in many decisions representing the particle evolution, and the chaotic values that particle states take along the life of the simulation. Whereas this paper concentrates in a laser dynamics model, it is obvious that these features may be present in many CA simulations, mostly when cooperative phenomena are expected. What is more relevant, the existence of many branches (some of them in the form of nested conditional structures) in the "hot spot" zones implies that GPU implementations would suffer from an important deceleration. This is due to the inherent so-called thread divergence [33] that GPU compilers introduce in these cases, which is one the main reasons why the performance on these platforms diminishes.

Taking into account previous considerations, an accurate timing characterization of main sequential algorithm pieces was done. This analysis concludes that:

- More than 80% of the mean execution time is spent in stimulated emission and pumping rules. What is more, their execution times have a considerable variance: minimum times are around five times lower than maximum times. This asserts the effect of random values and the chaotic evolution of different cell particles.

```
1:  for j = 0 to L_y − 1 do
2:    for i = 0 to L_x − 1 do {CA lattice loop}
3:      if c_ij > 0 then {Apply photon decay rule}
4:        for k = 1 to M do
5:          {Substract 1 to every photon's lifetime}
6:          if c̃^k_ij > 0 then
7:            c̃^k_ij ⟵ c̃^k_ij − 1
8:            if c̃^k_ij = 0 then {One photon decays}
9:              c_ij ⟵ c_ij − 1
10:             c'_ij = c_ij
11:           end if
12:         end if
13:       end for
14:     end if
15:     if a_ij = 1 then {Apply electron decay rule}
16:       {Substract 1 to time of life of every excited electron}
17:       ã_ij ⟵ ã_ij − 1
18:       if ã_ij = 0 then
19:         {One electron decays}
20:         a_ij ⟵ 0
21:       end if
22:     end if
23:   end for
24: end for
```

Figure 3. Pseudo code diagram for the implementation of the photon and electron decay and evolution of temporal variables rules.

- Random number computation supposes around the 70% of the pumping rule time.

- The rest of time resides mainly in photon and electron decay. The oscillatory behavior of particle evolution during the stationary phase implies also a considerable variance in these times. This is even more exaggerated during transient evolution.

- Noise photon rule timing is negligible (in fact, its number of iterations is very much inferior than the rest of rules).

Previous facts make necessary the introduction of at least the next changes in both OpenMP and GPU code implementations (see *https://github.com/dcagigas/Laser-Cellular-Automata*):

- Of course, avoiding non re-entrant functions like simple random generators. Even more, although generating a seed for each thread should be enough to make random generation independent among threads, the deep inner real data dependences that random functions contain lasts in the mean longer than the rest of an iteration step. It is preferable an implementation similar to that of the *cuRAND* library, that is, a seed for each cell ij, which preserves a good random distribution while accelerates each step around a 40%.

- As only one electron per cell is allowed, suppressing the a_{ij} matrix. Thus, it is considered that if $\tilde{a}_{ij}$ is zero the electron is not excited; and it is excited elsewhere.

- Eliminating the refresh of c matrix (which supposes copying long matrices) with values of $\tilde{c}$ (line 9 of Fig. 1), by using pointers to these two matrices and swapping these pointers at the end of each iteration step.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 17 December 2019 doi:10.20944/preprints201912.0223.v1

Peer-reviewed version available at Electronics 2020, 9, 189; doi:10.3390/electronics9010189

```
1: for j = 0 to L_y − 1 do
2:   for i = 0 to L_x − 1 do {CA lattice loop}
3:     if a_ij = 0 then {Apply pumping rule}
4:       {Generate random number in (0,1) interval}
5:       ξ ⟵ random_number(0,1)
6:       if ξ < λ then {λ: pumping probability}
7:         {One electron is pumped}
8:         a_ij ⟵ 1
9:         ã_ij ⟵ τ_a
10:      end if
11:    else { (a_ij = 1)–>Apply stimulated emission rule}
12:      if neighbours(i,j) > δ then
13:        {Look for first value of k for which c̃^k_ij = 0}
14:        k ⟵ 1
15:        while c̃^k_ij ≠ 0 and k ≤ M do
16:          k ⟵ k + 1
17:        end while
18:        if k <= M then
19:          a_ij ⟵ 0
20:          ã_ij ⟵ 0
21:          c'_ij ⟵ c'_ij + 1
22:          c̃^k_ij ⟵ τ_c
23:        end if
24:      end if
25:    end if
26:  end for
27: end for
```

Figure 4. Pseudo code diagram for the implementation of the pumping and stimulated emission rules.

The source code of the different implementations and results achieved are available in *https://github.com/dcagigas/Laser-Cellular-Automata*. The source code is under GPL v3 license. Researchers can download and modify the code freely to run their own particular laser dynamic simulations.

3.3.1. OpenMP Framework

Previous improvements are quite easy to detect and to implement. However, there are further enhancements that speedup even more a CA simulation when running an OpenMP implementation over multicore platforms. As a result, apart from the OPENMP_NOT_OPTIMIZED version, an optimized one (called simply OPENMP) can be downloaded from the previous *github* page. For the sake of clarity, these further enhancements are grouped and listed in the following points. Moreover, they have been marked in the *github* source code with the symbol @.

- After a deeper examination of laser dynamics evolution, it was detected that very few cells contain more than one photon during the stationary evolution. Thus, the habitual matrix arrangement of variable $\tilde{c}^k_{ij}$, that is, storing consecutively the M values for each cell ij is switched by the following one. All the cells ij are stored consecutively for each of the M possible photons. In terms of the C++ code, this three-dimensional matrix is represented by: $\tilde{c}[M][L_x][L_y]$ (see $lifet\_f$ matrix in the code).

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 17 December 2019 doi:10.20944/preprints201912.0223.v1

Peer-reviewed version available at Electronics 2020, 9, 189; doi:10.3390/electronics9010189

The new arrangement implies that all elements of $\tilde{c}[0][][]$ are continuously used and then cached, but the rest of elements $\tilde{c}[1:M-1][L_x][L_y]$ are scarcely used, so they do not consume precious cache lines. On the contrary, if the habitual matrix arrangement had been used, it would have wasted many cache bytes (almost only 1 of each M elements would have been really utilized during the stationary period). The rest of code pieces where this matrix is manipulated are not decelerated by the new arrangement; e.g. very few cells generate a shifting from $\tilde{c}[k][L_x][L_y]$ to $\tilde{c}[k-1][L_x][L_y]$, $(k > 0)$, when a photon decays.

- While previous improvement avoids cache line wasting, memory consumption is another fundamental issue. The analysis of real values of the big code matrices leads to the conclusion that maximum values are small for most physical variables. Thus, instead of 32 bits per element (*unsignedint* variables in C++), real sizes in the optimized version have been reduced to unsigned short int and unsigned char whenever possible. More exactly, this supposed reducing memory size from: $32 \times L_x \times L_y \times (M+3)$ to $16 \times L_x \times L_y \times (M+1) + 8 \times 2$ (see $e, f1, f2, lifet\_f$ matrices in the optimized code).

- In order to promote loop vectorization, some conditional branches have been transformed into simple functions. For example, those conditional sentences that increment a counter q when a certain condition p is true have been written like $q+ = p$. This eases the task of the compiler when introducing SIMD instructions and predicative-like code and prevents many BTB (Branch Target Buffer) misprediction penalties because these conditions are difficult to predict (due to the random nature of particle state evolution).

- Loop splitting is another classical technique that reduces execution time when memory consumption is huge, and the loop manages several disjointed data. This occurs in the case of photon and electron decay rules, which have been separated into two different loops in the optimized version. This way, caches are not struggled with several matrixes thus preventing many conflict misses on them.

To sum up, previous optimizations achieve around a 2x speedup (see section 4) with respect to the basic one. It is worth to remark that both OpenMP versions give exactly the same particle evolution results.

Despite that random number computing has been considerable accelerated by using a seed for each cell (i, j), it continues to be the most time-consuming piece. A final improvement draws to an approx. additional 3 times speedup of the OpenMP simulation time: instead of computing random numbers during the simulation, generating a list of them previously and using this same list for all the desired simulations (e.g. if different parameters want to be tested like pumping probability, maximum electron and photon lifetimes, etc.).

Using a random list as an input for the model eases the checking of results for different platforms, because the output of the simulation must be exactly the same. More precisely, it is needed that a random number is stored in the list for each time step and for each cell.

However, this list should be enormous for the pumping rule, even if only a random true/false bit were stored for each time step and for each cell. For example, considering a simulation of 1000 steps, a lattice of 4096×4096 would occupy 2 GBytes. Because of this, this improvement has not been considered in the Result section. Nonetheless, the interested reader can test this optimization (note that big lattice sizes would overflow platform memory) simply by defining the constant $RANDOM\_VECT\_ENABLE$ in the *github* OpenMP codes. Defining this constant would generate the random numbers in advance while suppressing its computation during the simulation time.

3.3.2. CUDA Framework

The CUDA framework has three main kernels (i.e. CUDA functions written in C style code), the same as those of the OpenMP implementation. They are called for each time step sequentially. First the *PhotonNoiseKernel* produces new photons randomly, then the *DecayKernel* performs the electron and photon decay, and last the *PumpingKernel* does the pumping and stimulated emission. This order

can be altered but it must be the same as the one used in OpenMP. Otherwise, results could not be exactly the same.

There is one last kernel needed: *do_shannon_entropy_time_step*. Most of the variables that are needed to calculate the Shannon Entropy are stored in GPU global memory. Data transfers between computer host memory and GPU memory must be minimized because of its big latency. Despite that the calculations are not parallel, it is more convenient to perform time step Shannon Entropy calculations in GPU memory. There is also a final kernel called *finish_shannon_entropy_time_step* after the time/step loop. However, this last kernel has a low performance impact because it is executed only once.

Simulation parameters are defined in a header file; for example, the SIDE constant that determines the grid side of a simulation. In case of CUDA, and GPUs in general, memory size constraints are particularly important when comparing with computer workstations. The GPU global memory available is usually lower than that of a workstation. Thus, data structure types for electrons and photons are important for large grids. Matrix data structures grow by a factor of x4 for each SIDE increment, and x40 in case of the matrix that records photon energy values in each cell (*GPU_tvf*).

For example, with a grid SIDE of 8192 (2^{13}) and 4 GB of GPU global memory it is only possible to run the simulation if cells of *GPU_tvf* matrix variable are set to *char* type in C. As mentioned before, this variable is in charge of recording photon life time values in each grid cell. The *char* type is 8 bit size, so only initial photon life time values between 0 and 255 are allowed. The same happens with electron life time values. By default this constant value is set to *short int* (i.e. 16 bits) to allow higher values.

The CUDA programming environment and the latest NVIDIA architectures (Pascal, Turing and Volta) also have some restrictions related to integer atomic arithmetic operations. CUDA atomic arithmetic operations only allow the use of *int* data type but not the *short int* or *char*. In the *PhotonNoiseKernel* it is necessary to update new photons in the matrix data structures. Those updates are performed in a parallel way. To avoid race conditions, atomic arithmetic operations are needed when each GPU hardware thread updates a matrix photon cell (two hardware GPU threads could try to update the same cell at the same time). Therefore, it was necessary to use the *int* data type instead of *short int* or *char*, thus increasing the GPU memory size needed by these data structures.

CUDA framework has also one extra feature that can be enabled in the source files: the electron evolution output video. A .avi video file showing the electron evolution through the time steps can be produced. This feature involves the transfer of a video frame for each time step from GPU memory to host or computer memory. When activated and for a moderate grid side (1024 or above), the total execution time could be significantly high because of the latency between GPU and host memory transfers. This feature could also be adapted or modified to show photon evolution (nonetheless, electron and photon behaviours are very similar).

4. Results

We present here the performance evaluation results for the two architectures: Multi-Core and GPU. For each architecture we evaluated the performance on two different hardware platforms that are representative of different target market segments: a high-end chip intended for scientific computing or for servers and a consumer-oriented one.

We have tested that the simulation results of both parallel implementations reproduce the output of the original sequential one. As an example we show in Fig. 5 the time evolution of the total number of laser photons and the population inversion in the laser system for values of the parameter corresponding to a laser spiking regime. The results are the same as found in previous publications with a sequential implementation, as [21]. It is shown that when increasing the size of the CA lattice the results are smoother since the model reproduces better the macroscopic behavior of the system with a higher statistics.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 17 December 2019 doi:10.20944/preprints201912.0223.v1

Peer-reviewed version available at Electronics 2020, 9, 189; doi:10.3390/electronics9010189

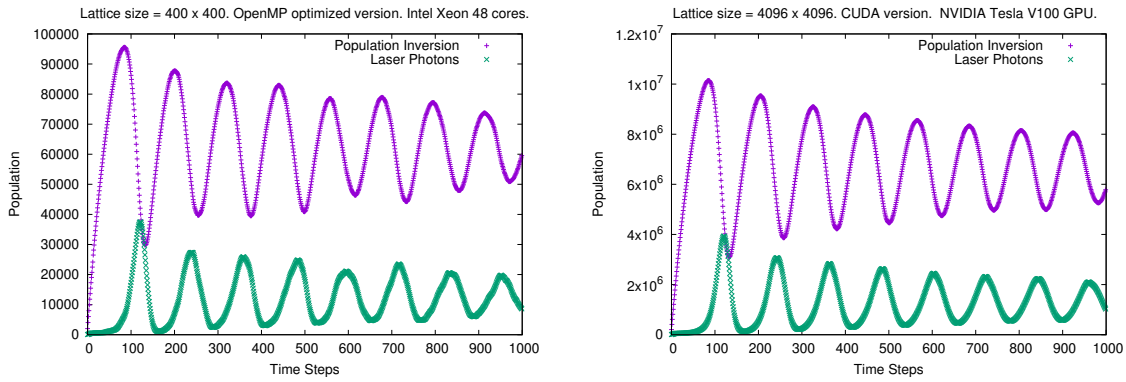

Figure 5. Output of the model for particular values of the system parameters corresponding to a laser spiking behavior. Parameter values: $\lambda = 0.0125$, $\tau_c = 10$, $\tau_a = 180$. The results are smoother for larger lattice sizes. (**Left:**) Sequential implementation with a lattice size of 400×400 cells. (**Right:**) Sequential implementation with a lattice size of 4096×4096 cells.

4.1. Multi-Core architecture

The Multi-Core architecture was executed and tested on the following two platforms.

4.1.1. High-end Multi-Core CPUs (48 cores)

We evaluated the performance on a high-performance server CPU running in the Cloud, using the Amazon Web Server(AWS) Infrastructure as a Service (IaaS) EC2 service. We run our performance test on a *m5.24xlarge* instance. It runs on a dual socket server with 2 Intel Xeon Platinum 8259CL processors with 24 physical cores each (completing 48 physical cores between them), running at a frequency of 2.50 GHz, with 35,75 MiB of cache memory. The total RAM memory was 373 GiB. Both processor sockets are linked by Ultra Path Interconnect (UPI), a high speed point-to-point interconnect link delivering a transfer speed of up to 10.4 GT/s.

4.1.2. Consumer-oriented Multi-Core CPU (8 cores)

The performance was evaluated on a PC with a Core i9 9900k processor and a total RAM memory of 16 GiB. The processor frequency was 3.6 GHz and the RAM memory was on a single channel running at 2400 MHz. This processor has 8 physical cores and each core has 2 hardware threads (completing a total of 16 Threads).

4.2. GPU architecture

The following two GPU chips were used to run and test the GPU architecture.

4.2.1. High-end GPU chip

We evaluated the performance on a *p3.2xlarge* instance of the Amazon Web Server(AWS) IaaS EC2 Cloud Computing service. It used a NVIDIA Tesla V100 GPU (Volta architecture) with 5,120 CUDA cores and a GPU memory of 16 GiB. The server used an Intel Xeon E5-2686v4 CPU with 4 physical cores running at 2.3 GHz. CPU and GPU were interconnected via PCI-Express Gen. 3, with a bandwidth of 32 GiB/s.

4.2.2. Consumer-oriented GPU chip

We used a consumer-oriented NVIDIA GeForce GTX 1050 Ti graphic card (Pascal architecture). This card has 768 CUDA cores running at 1290 MHz. The total memory is 4 GiB of GDDR5 type, with a maximum bandwidth of 1120 GiB/s.

4.3. Performance and scalability results

Fig. 6 shows the speedup of the parallel OpenMP implementation running on the Core i9 9900k PC (8 cores). In this case, speedups are fairly foreseeable. Firstly, for small lattice sizes most of matrices reside in CPU caches, thus achieving an excellent speedup versus the sequential (one thread) test until 8 threads. Launching 9 threads implies that a physical CPU must manage two threads (whereas the rest of CPUs, only one), thus causing the speedup to decrease. However, this problem diminishes for more threads. Finally, Core i9 simultaneous multi-threading begins to play a role from 9 threads up, hence speedups above 8 are reached for some tests when launching many threads.

On the other hand, for big lattices speedups are stuck according to the maximum RAM bandwidth, as predicted by the roof-line model [34]. In these cases, RAM memory bandwidth is the bottleneck that in fact determines program runtimes.

In the case of the the high-end server with two 2 Intel Xeon Platinum 8259CL CPUs (see Fig. 7), our parallel implementation shows an excellent speedup for large lattice sizes. The OpenMP optimized version reaches a speedup around $30x$ for 48 threads. The behaviors are similar to those obtained in the consumer-oriented hardware: a peak on the speedups is reached when launching the same number of threads than physical CPUs; then accelerations begin to decrease for some more threads, and finally speedups are recovered when the number of threads doubles the number of physical cores.

Nevertheless, for high number of threads, or more precisely, for small number of lattice rows per thread, the computation-to-communication scale begins to deteriorate speedups. Note that small lattices (Fig. 7 for lattice width $= 512$) exhibit patently this problem, whereas big lattice speedups are almost not deteriorated. This is a well-known effect when scientific applications are massively distributed [35]. Because the stimulated emission rule uses Moore neighborhood, the more threads in which we divide the lattice, the more communication between threads are needed, thus degrading the parallel performance.

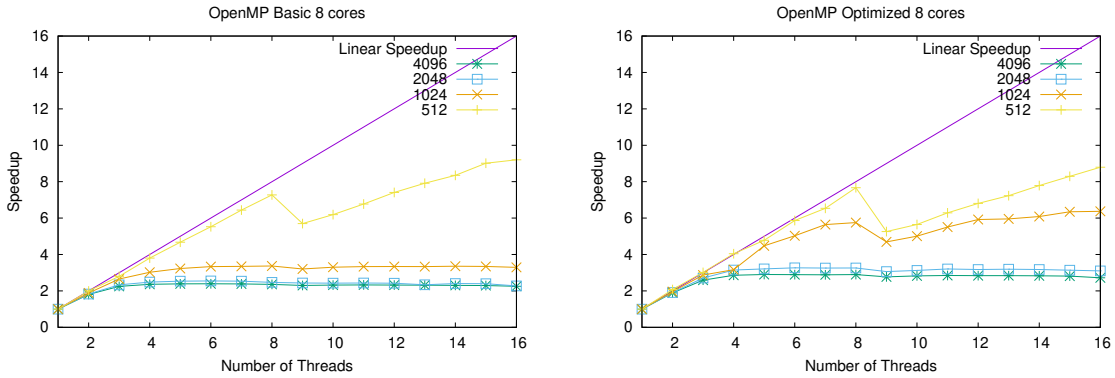

Figure 6. Speedup of the parallel OpenMP implementation running on a consumer-oriented CPU with 8 cores. (**Left:**) Basic implementation without in-depth optimization. (**Right:**) Fully optimized implementation.

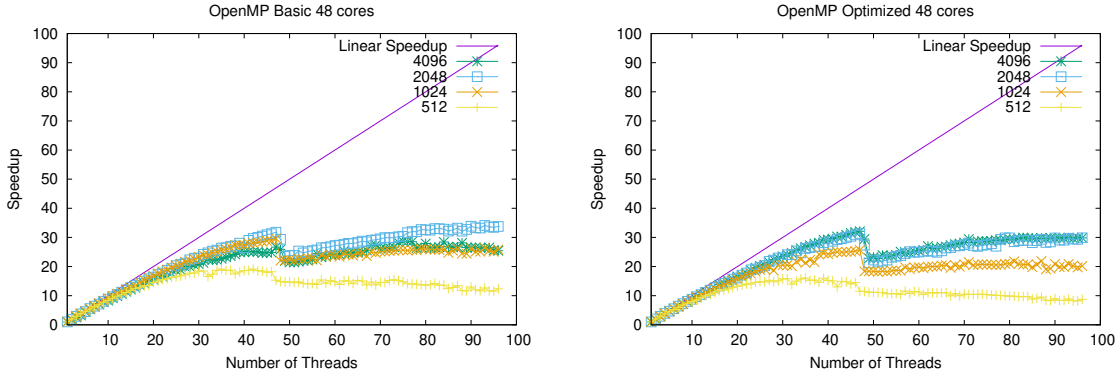

Figure 7. Speedup of the parallel OpenMP implementation running on a high-end dual-socket server with a total of 48 cores. (**Left:**) Basic implementation without in-depth optimization. (**Right:**) Fully optimized implementation.

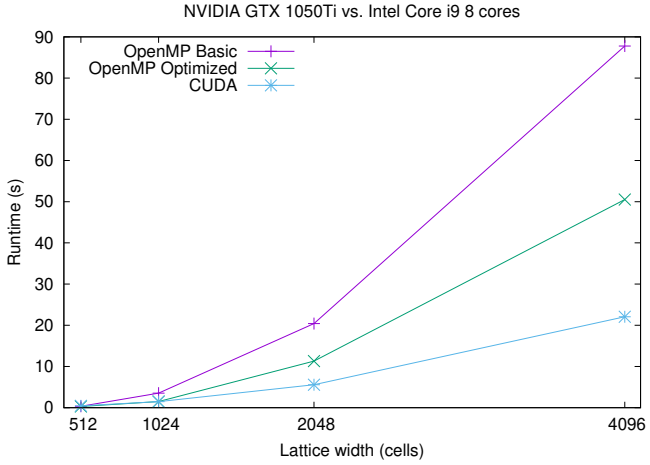

Figure 8. Runtime comparison between NVIDIA GTX 1050 Ti GPU and Intel Core i9 (8 cores) for different CA lattice sizes.

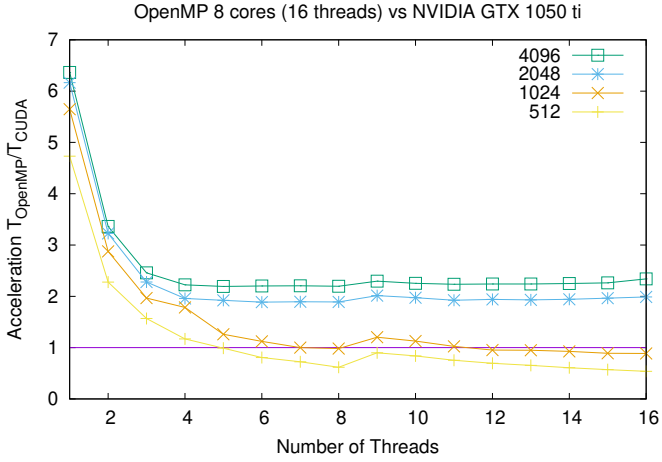

Figure 9. Comparison of OpenMP Optimized and CUDA times when using Intel Core i9 and NVIDIA GeForce GTX 1050 Ti resp.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 17 December 2019 doi:10.20944/preprints201912.0223.v1

Peer-reviewed version available at Electronics 2020, 9, 189; doi:10.3390/electronics9010189

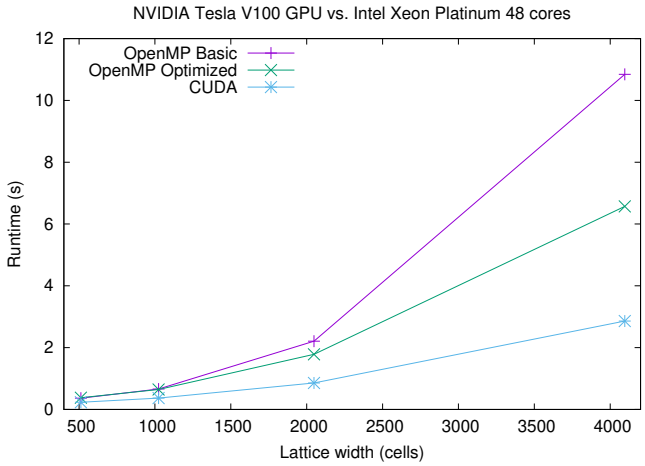

Figure 10. Runtime comparison between NVIDIA Tesla V100 GPU and Intel Xeon Platinum (dual socket with 48 cores in total) for different CA lattice sizes.

In Figs. 8 to 10 we show a runtime comparison between both CPU/GPU pairs, for different sizes and number of Multi-Core Threads. It is interesting to note that for CPU/GPU comparisons shown in Figs. 8, 9 and Fig. 10) CUDA implementation is, at most, 2.5x faster than the OpenMP Optimized version. In truth, only for big lattice sizes GPU platforms beat clearly CPU ones: as shown in Fig. 10, that happens only for a number of threads much smaller than the number of available physical cores of the CPU and for large system sizes. However, when using the 48 available cores of the CPU, the CUDA implementation is again only around 2.5x faster than the CPU. Even for small system sizes the runtime of CPU and GPU is similar.

An additional consideration plays in favor of classical CPU platforms. If a previously computed random list were used as an input for the model, (thus preventing the time spent in computing random numbers during simulation, see subsection 3.3.1), even OpenMP Optimized simulation times would be lower than that of CUDA versions. We recall here that, although this alternative is practical only for medium lattice widths, it speedups OpenMP around three times. This optimization does not favour so much GPU platforms.

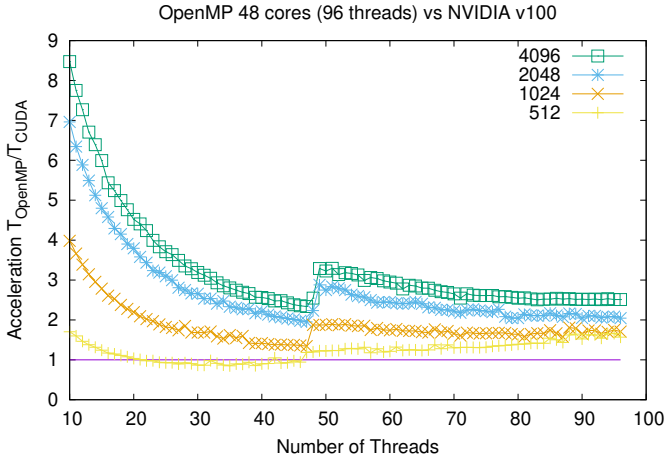

Figure 11. Comparison of OpenMP Optimized and CUDA times: Detail from 10 threads up when using Intel Xeon Platinum and NVIDIA Tesla V100 GPU, resp.

5. Conclusions and Future Lines

In some previous studies cited in the introduction and section 2, it was pointed out that the speedups achieved with GPU when comparing to single-core CPUs where above 10X and sometimes

above 100x. Even with these ratios, it is evident that a current multicore CPU may approach GPU performance. Moreover, this affirmation must be revised and carefully analyzed for Cellular Automata (CA) applications. In this paper, it is experimentally proved, using almost the same code for a laser dynamics CA (except for the necessary adaptation to each platform), that these distances have been significantly shortened. We conclude that nested conditional structures (in general, many branches in the "hot spot" zones) implies that GPU implementations would suffer from an important deceleration. Even for massive parallel data structures like CA, an approximately $3x$ speedup is achieved when using high performance computers and GPUs. The other factor that limits CPU and GPU performance for big lattice CAs is obviously the maximum RAM bandwidth, as predicted by the roofline model. If other variables were taken into account like price, TDP, source code maintenance, or easy and rapid software development, it is not clear that GPUs are always the best choice for an efficient parallelization of CA algorithms.

Future lines, provided the important conclusions obtained in this paper for the specific case of laser dynamics, include the extension of this analysis to generic Cellular Automata. This will be necessary in order to extract a simple but realistic model that allows to predict the performance on CPU and GPU platforms mainly as a function of the form of its state transition rules, its neighboring relations, the amount of memory, etc. It will be also interesting to investigate the efficient implementation and speedup obtained by implementing this model on Multi-GPUs.

Author Contributions: Conceptualization, D.C., F.D, and J.L.G.; methodology, D.C., F.D, M.R.L. and J.L.G.; software, D.C., F.D, M.R.L. and J.L.G.; validation, D.C., F.D, and J.L.G.; investigation, D.C., F.D, F.J. and J.L.G.; resources, D.C., F.D, and J.L.G.; writing–original draft preparation, D.C., F.D, F.J. and J.L.G.; writing–review and editing, D.C., F.D, and J.L.G..; visualization, D.C. and M.R.L.; supervision, J.L.G.; funding acquisition, D.C., F.J. and J.L.G.

Funding: This research was funded by the following research project of Ministerio de Economía, Industria y Competitividad, Gobierno de España (MINECO) and the Agencia Estatal de Investigación (AEI) of Spain, cofinanced by FEDER funds (EU): MABICAP (Bio-inspired machines on High Performance Computing platforms: a multidisciplinary approach, TIN2017-89842P). The work was also partially supported by the computing facilities of Extremadura Research Centre for Advanced Technologies (CETA-CIEMAT), funded by the European Regional Development Fund (ERDF). CETA-CIEMAT belongs to CIEMAT and the Government of Spain.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

References

1. Talia, D.; Naumov, N., Parallel cellular programming for emergent computation. In *Simulating Complex Systems by Cellular Automata*; Springer, 2010; pp. 357–384.
2. Bandini, S.; Mauri, G.; Serra, R. Cellular automata: From a theoretical parallel computational model to its application to complex systems. *Parallel Computing* **2001**, *27*, 539–553. doi:10.1016/S0167-8191(00)00076-4.
3. Wolfram, S. *Cellular automata and complexity*; Addison-Wesley: Reading, MA, 1994.
4. Ilachinski, A. *Cellular Automata: A Discrete Universe*; World Scientific: Singapore, 2001; p. 808.
5. Sayama, H. *Introduction to the Modeling and Analysis of Complex Systems*; Open SUNY Textbooks. 14: Geneseo, NY, 2015.
6. Chopard, B.; Droz, M. *Cellular Automata Modeling of Physical Systems*; Cambridge University Press: Cambridge, MA, USA, 1998.
7. Sloot, P.; Hoekstra, A. Modeling Dynamic Systems with Cellular Automata. In *Handbook of dynamic system modeling*; Fishwick, P., Ed.; Chapman & Hall/CRC, 2007; pp. (21) 1–6.
8. A.G., H.; Kroc, J.; Sloot, P., Eds. *Simulating Complex Systems by Cellular Automata*; Springer: Berlin, Heidelberg, 2010.
9. Gounaridis, D.; Chorianopoulos, I.; Koukoulas, S. Exploring prospective urban growth trends under different economic outlooks and land-use planning scenarios: The case of Athens. *Applied Geography* **2018**, *90*, 134–144. doi:10.1016/J.APGEOG.2017.12.001.

10. Aburas, M.M.; Ho, Y.M.; Ramli, M.F.; Ash'aari, Z.H. The simulation and prediction of spatio-temporal urban growth trends using cellular automata models: A review. *International Journal of Applied Earth Observation and Geoinformation* **2016**, *52*, 380–389. doi:10.1016/J.JAG.2016.07.007.
11. Liu, X.; Liang, X.; Li, X.; Xu, X.; Ou, J.; Chen, Y.; Li, S.; Wang, S.; Pei, F. A future land use simulation model (FLUS) for simulating multiple land use scenarios by coupling human and natural effects. *Landscape and Urban Planning* **2017**, *168*, 94–116. doi:10.1016/J.LANDURBPLAN.2017.09.019.
12. Qiang, S.; Jia, B.; Huang, Q.; Jiang, R. Simulation of free boarding process using a cellular automaton model for passenger dynamics. *Nonlinear Dynamics* **2018**. doi:10.1007/s11071-017-3867-5.
13. Tang, T.Q.; Luo, X.F.; Zhang, J.; Chen, L. Modeling electric bicycle's lane-changing and retrograde behaviors. *Physica A: Statistical Mechanics and its Applications* **2018**, *490*, 1377–1386. doi:10.1016/J.PHYSA.2017.08.107.
14. Monteagudo, A.; Santos, J. Treatment Analysis in a Cancer Stem Cell Context Using a Tumor Growth Model Based on Cellular Automata. *PLoS One* **2015**, *10*, e0132306. doi:10.1371/journal.pone.0132306.
15. Burkhead, E.; Hawkins, J. A cellular automata model of Ebola virus dynamics. *Physica A: Statistical Mechanics and its Applications* **2015**, *438*, 424–435. doi:10.1016/J.PHYSA.2015.06.049.
16. Guisado, J.; Jiménez-Morales, F.; Guerra, J. Cellular automaton model for the simulation of laser dynamics. *Physical Review E* **2003**, *67*, 66708.
17. Guisado, J.; Jiménez-Morales, F.; Guerra, J. Computational simulation of laser dynamics as a cooperative phenomenon. *Physica Scripta* **2005**, *T118*, 148–152.
18. Guisado, J.; Jiménez-Morales, F.; Guerra, J. Simulation of the dynamics of pulsed pumped lasers based on cellular automata. *Lecture Notes in Computer Science* **2004**, *3305*, 278–285.
19. Kroc, J.; Jimenez-Morales, F.; Guisado, J.L.; Lemos, M.C.; Tkac, J. Building Efficient Computational Cellular Automata Models of Complex Systems: Background, Applications, Results, Software, and Pathologies. *Advances in Complex Systems* **2019**, *22*, 1950013.
20. Guisado, J.; Fernández-de Vega, F.; Jiménez-Morales, F.; Iskra, K. Parallel implementation of a cellular automaton model for the simulation of laser dynamics. *Lecture Notes in Computer Science* **2006**, *3993*, 281–288. doi:10.1007/11758532_39.
21. Guisado, J.; Jiménez-Morales, F.; Fernández-de Vega, F. Cellular automata and cluster computing: An application to the simulation of laser dynamics. *Advances in Complex Systems* **2007**, *10*, 167–190.
22. Guisado, J.; Fernandez-de Vega, F.; Iskra, K. Performance analysis of a parallel discrete model for the simulation of laser dynamics. Proceedings of the International Conference on Parallel Processing Workshops. IEEE Computer Society, 2006, pp. 93–99. doi:10.1109/ICPPW.2006.62.
23. Guisado, J.; Fernández de Vega, F.; Jiménez-Morales, F.; Iskra, K.; Sloot, P. Using cellular automata for parallel simulation of laser dynamics with dynamic load balancing. *International Journal of High Performance Systems Architecture* **2008**, *1*, 251–259.
24. GPGPU. General-Purpose Computation on Graphics Hardware. http://gpgpu.org, as available on may 2012., 2012.
25. Lopez-Torres, M.; Guisado, J.; Jimenez-Morales, F.; Diaz-del Rio, F. GPU-based cellular automata simulations of laser dynamics. Proceedings of the XXIII Jornadas de Paralelismo; hgpu.org: Elche, 2012; pp. 261–266.
26. Gobron, S.; Devillard, F.; Heit, B. Retina simulation using cellular automata and GPU programming. *Machine Vision and Applications Journal* **2007**, *18*, 331–342.
27. Rybacki, S.; Himmelspach, J.; Uhrmacher, A. Experiments With Single Core, Multi Core, and {GPU}-based Computation of Cellular Automata. Advances in System Simulation, 2009. {SIMUL}'09. First International Conference on, 2009.
28. Bajzát, T.; Hajnal, E. Cell Automaton Modelling Algorithms: Implementation and Testing in {GPU} Systems. {INES} 2011, 15th International Conference on Intelligent Engineering Systems, 2011.
29. Balasalle, J.; Lopez, M.; Rutherford, M., Optimizing Memory Access Patterns for Cellular Automata on {GPU}s. In *GPU Computing Gems Jade Edition*; Elsevier - Morgan Kaufmann - NVIDIA, 2011; pp. 67–75.
30. Geist, R.; Westall, J., Lattice-Boltzmann Lighting Models. In *GPU Computing Gems, Emerald Edition*; Elsevier - Morgan Kaufmann - NVIDIA, 2011; pp. 381–399.
31. Gibson, M.J.; Keedwell, E.C.; Savić, D.A. An investigation of the efficient implementation of cellular automata on multi-core CPU and GPU hardware. *Journal of Parallel and Distributed Computing* **2015**, *77*, 11–25. doi:10.1016/j.jpdc.2014.10.011.

32. Jongerius, R.; Anghel, A.; Dittmann, G. Analytic Multi-Core Processor Model for Fast Design-Space Exploration. *IEEE TRANSACTIONS ON COMPUTERS* **2018**, *67*.
33. NVIDIA. CUDA C Best Practices Guide Version. Available in http://developer.nvidia.com/, Accessed in Dec. 2019.
34. Williams, S.; Waterman, A.; Patterson, D. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* **2009**, *52*, 65–76.
35. Hennessy, J.; Patterson, D. *Computer Architecture: A Quantitative Approach, 6th Edition*; Vol. 19, Elsevier-Morgan Kaufmann, 2017.