GPU-based cellular automata simulations of laser dynamics

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

Resumen—
We present a parallel implementation for Graphics Processing Units (GPUs) of a model based on cellular automata (CA) to simulate laser dynamics. A cellular automaton is an inherent parallel type of algorithm that is very suitable to simulate complex systems formed by many individual components which give rise to emergent behaviours. We exploit the parallel character of this kind of algorithms to develop a fine-grained parallel implementation of the CA laser model on GPUs. A good speedup of up to 14.5 over a sequential implementation running on a single core CPU has been obtained, showing the feasibility of this model to run efficient parallel simulations on GPUs.

Palabras clave— **GPU, CUDA, parallel computing, cellular automata, laser dynamics, modelling and simulation, laser physics.**

I. Introduction

Cellular Automata (CA) are a class of fully discrete, spatially-distributed dynamical systems that are characterized by local interaction and synchronous parallel dynamical evolution [1, 2]. They are a powerful tool to describe, understand and simulate complex systems in which a global behaviour results from the collective action of many simple components that interact locally.

In recent years, CA have been successfully applied to build simulations of complex systems in many different fields of science and technology: 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), etc [3, 4, 5].

One of the fields for which the CA approach can be used is laser physics. Guisado et. al. have introduced a CA-based model for simulating laser dynamics, showing that it can reproduce much of the phenomenology of laser systems [6, 7, 8]. This model can be very useful as an alternative to the standard modelling approach —differential equations— in different situations such as lasers ruled by stiff differential equations, difficult boundary conditions, or very small devices for which the approximations considered for the differential equations are not valid.

The execution of complex systems simulations using CA models has large runtime requirements because a large system with many interacting cells must be used. The reason is that global and collective properties cannot be deduced from its simpler components, but emerge from the evolution and interaction of many elements [9, 10].

However, a cellular automaton is a distributed type of algorithm with an inherent parallel nature, because it is composed of many individual components or cells that are simultaneously updated, and also with a local nature, since the evolution of the cells is determined by strictly local rules. These characteristics make them ideally well suited to be implemented very efficiently on parallel computers.

In order to exploit this parallelism in the case of the CA-based model of laser dynamics, a parallel implementation for distributed-memory parallel computers was introduced [11, 12]. It was found that the parallel implementation offers a good performance running on dedicated computer clusters [13] and also on heterogeneous non-dedicated clusters with a dynamic load balancing mechanism [14].

However, it is not always easy to have inmediate access to a large computer cluster. On the other hand, in the last half decade, Graphics Processing Units (GPUs) have revolutionized the landscape of high performance scientific computing. GPUs are massively parallel processors which are capable of running thousands of programming threads in parallel. Depending on the application, they are offering a 10 to 100 times speedup at price points extremely affordable. For that reason, in the present work, we present a parallel implementation of the CA-based model of laser dynamics for GPUs.

GPUs are traditionally used for interactive graphics applications, but their characteristics have made it possible to use them to accelerate arbitraty applications, what is usually known as GPGPU (General Purpose Computation on GPU) [15]. The architecture of a GPU is formed by a number of multiprocessors, each of them with a number of cores. All of the cores in a multiprocessor share a memory unit called shared memory and all of the multiprocessors share a memory unit called global memory.

II. Related work

There are not many works previous to 2007 that study the implementation of cellular automata on GPUs. They used a shading language such as OpenGL, a special programming language intended for graphics applications. Thus the programming was very difficult since the developer had to somehow adapt his/her application to a graphics language. For example the paper from Gobron et. al. [16] studies a CA model for a biological retina obtaining a 20x speedup as compared to the CPU implemen-

[1]Dpto. de Arquitectura y Tecnología de Computadores, Universidad de Sevilla, Seville, Spain, e-mail: rlopez@atc.us.es, jlguisado@us.es, fdiaz@atc.us.es.
[2] Dpto. de Física de la Materia Condensada, Universidad de Sevilla, Seville, Spain, e-mail: jimenez@us.es.

tation.

The introduction in 2007 of CUDA (Compute Unified Device Architecture), a general purpose programming language for GPUs of the NVIDIA manufacturer, was a milestone that boosted the usage of GPUs in scientific computing. Soon after, another one called OpenCL was introduced, which is multiplatform. They both offer the flexibility of a general purpose language so that the programming of arbitrary applications is much more easy. Therefore, there has been more CA implementations from that date. We will refer to some of them.

Rybacki et. al. [17] studied 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 whether GPU implementations are useful or not depends strongly on the model to be simulated.

Bajzát et. al. [18] studied the GPU implementation and peformance of a CA model for an ecological system.

Balasalle et. al. [19] have 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 showed that a carefully optimized implementation can give up to a 65% improvement in runtime from a baseline implementation, but they did not study other CA models.

GPU implementations have been specially investigated for Lattice Boltzmann methods, a particular class of CA, obtaining dramatic speedups of up to 234x over single-core CPU execution without using SSE instructions or multithreading [20].

III. Cellular automaton model for laser dynamics simulation

We have developed a GPU-based implementation of the cellular automaton model of laser dynamics introduced by Guisado et. al. [6, 7, 8]. In this model, a laser system is represented by a two-dimensional CA that corresponds to a transverse section of the active medium in the laser cavity. The cellular space is a two-dimensional square lattice of $N_c = L \times L$ cells with periodic boundary conditions.

Two variables $a_i(t)$ and $c_i(t)$ are associated with each node of the CA. The first one, $a_i(t)$, represents the state of the electron in node i at time t: if $a_i(t) = 0$ the electron is in the laser ground state and if $a_i(t) = 1$ it is in the upper laser state. The second variable, $c_i(t) \in \{0, 1, 2, ..., M\}$, represents the number of photons in node i at time t. A large enough upper value of M is taken to avoid saturation of the system.

The state variables $a_i(t)$ and $c_i(t)$ represent "bunches" of real photons and electrons. Their values are obviously smaller than the real number of photons and electrons in the system and are connected to them by a normalization constant.

The *Moore neighborhood* is employed. Each cell has nine neighbours: The cell itself, its four nearest neighbours (at positions north, south, east and west) and the four next neighbours (at positions northeast, southeast, northwest and southwest).

The evolution of the system is governed by a set of transition rules, which represent the different physical processes taking place in a laser system at the microscopic level:

- Rule 1. Pumping: If $a_i(t) = 0$ then $a_i(t+1) = 1$ with a probability λ.
- Rule 2. Stimulated emission: If $a_i(t) = 1$ and the sum of the values of the laser photons states in the nine neighbor cells is greater than a certain threshold (1 in our model), then $c_i(t+1) = c_i(t) + 1$ and $a_i(t+1) = 0$.
- Rule 3. Photon decay: A finite life time τ_c is assigned to each photon when it is created. The photon will be destroyed τ_c time steps after it was created.
- Rule 4. Electron decay: A finite life time τ_a is assigned to each electron that is promoted from the ground level to the upper laser level. That electron will decay to the ground level again τ_a time steps after it was promoted, if it has not yet decayed by stimulated emission.

In addition, a small number of photons in the laser mode are introduced in the system in random positions at every time step. To this end, a small number N_n of cells ($< 0.01\%$ of total) with randomly chosen positions are selected and $c_i(t+1) = c_i(t) + 1$ is applied for them. These random photons simulate spontaneous emission as well as thermal contributions and are responsible —as in real lasers— of the initial start-up of the laser action.

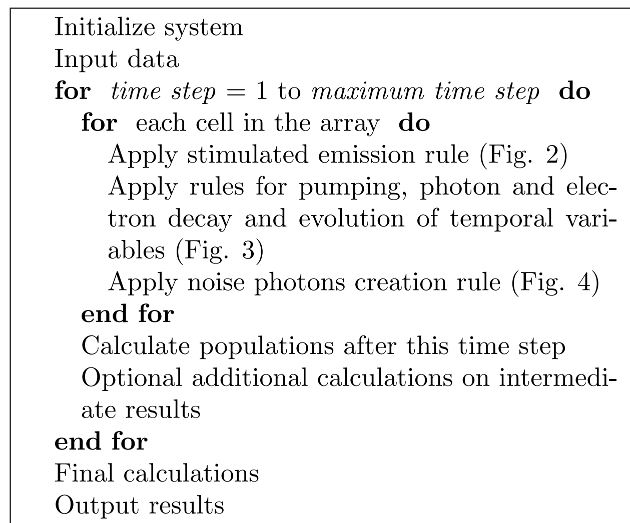
```
Initialize system
Input data
for time step = 1 to maximum time step do
  for each cell in the array do
    Apply stimulated emission rule (Fig. 2)
    Apply rules for pumping, photon and electron decay and evolution of temporal variables (Fig. 3)
    Apply noise photons creation rule (Fig. 4)
  end for
  Calculate populations after this time step
  Optional additional calculations on intermediate results
end for
Final calculations
Output results
```

Fig. 1. Pseudo code diagram for the implementation of the main program for the CA laser model.

IV. Sequential implementation of the model

The main structure of the CA laser model algorithm is shown in Fig. 1. After initializing the system, there is a time loop. At each time step, firstly the state of all the cells of the lattice are updated by

applying the transition rules, and secondly the total populations of laser photons and electrons in the upper state are calculated by summing up the values of the state variables $a_{\vec{r}}(t)$ and $c_{\vec{r}}(t)$ for all the cells.

The implementations of the CA rules are described in the algorithms shown in Figs. 2 to 4. In particular, Fig. 2 describes the implementation of the stimulated emission rule. This rule uses a function that calculates the sum of laser photons in the neighbourhood of a cell, including the effect of periodic boundary conditions.

```
for j = 0 to L_y − 1 do
  for i = 0 to L_x − 1 do {CA lattice loop}
    if a_ij = 1 then
      if neighbours(i, j) > δ then
        {Look for first value of k for which c̃^k_ij = 0}
        k ⟵ 1
        while c̃^k_ij ≠ 0 and k ≤ M do
          k ⟵ k + 1
        end while
        if k <= M then
          a_ij ⟵ 0
          ã_ij ⟵ 0
          c'_ij ⟵ c'_ij + 1
          c̃^k_ij ⟵ τ_c + 1
          {τ_c + 1 is assigned because 1 is substracted in the decay loop}
        end if
      end if
    end if
  end for
end for
{Refresh value of c matrix with contents of c' matrix}
for j = 0 to L_y − 1 do
  for i = 0 to L_x − 1 do {CA lattice loop}
    c_ij ⟵ c'_ij
  end for
end for
```

Fig. 2. Pseudo code diagram for the implementation of the stimulated emission rule.

In the algorithmic description of the implementation of the model, L_x and L_y represent the width of the lattice in the x and y directions. Two indices i and j are used explicitly instead of a vector $\vec{r} = (i, j)$ to indicate the location of a cell. Thus, the state variable $a_{\vec{r}}$ for the cell located at $\vec{r} = (i, j)$ is represented as a_{ij} and $c_{\vec{r}}$ is represented as c_{ij}. Two temporal variables, $\tilde{a}_{ij}$ and $\tilde{c}^k_{ij}$, are used as time counters, where k distinguishes between the different photons that can ocupy the same cell. When a photon is created, $\tilde{c}^k_{ij} = \tau_c$. After that, 1 is substracted 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 substracted to $\tilde{a}_{ij}$ for every time step and the electron will decay to the ground level again when $\tilde{a}_{ij} = 0$.

Fig. 3 describes the implementation of the pumping, photon and electron decay and the evolution of the temporal variables rules. Finally, the implementation of the noise photons creation rule is described in Fig. 4.

```
for j = 0 to L_y − 1 do
  for i = 0 to L_x − 1 do {CA lattice loop}
    if c_ij > 0 then {Apply photon decay rule}
      for k = 1 to M do
        {Substract 1 to every photon's lifetime}
        if c̃^k_ij > 0 then
          c̃^k_ij ⟵ c̃^k_ij − 1
          if c̃^k_ij = 0 then {One photon decays}
            c_ij ⟵ c_ij − 1
            c'_ij = c_ij
          end if
        end if
      end for
    end if
    if a_ij = 1 then {Apply electron decay rule}
      {Substract 1 to time of life of every excited electron}
      ã_ij ⟵ ã_ij − 1
      if ã_ij = 0 then
        {One electron decays}
        a_ij ⟵ 0
      end if
    else if a_ij = 0 then {Apply pumping rule}
      {Generate random number in (0, 1) interval}
      ξ ⟵ random_number(0, 1)
      if ξ < λ then {λ: pumping probability}
        {One electron is pumped}
        a_ij ⟵ 1
        ã_ij ⟵ τ_a
      end if
    end if
  end for
end for
```

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

V. Parallel implementation of the model for GPUs

Before describing the parallel implementation of the CA laser dynamics model, let us briefly review some concepts from general purpose programming on GPUs (GPGPU Programing) and in particular of nVidia's CUDA architecture. Unlike CPUs, designed to run many different programs with heterogeneous data, GPUs are designed to run the same set of operations on a large number of homogeneous data (ie. pixels of an image) repetitively: SIMD (Single Instruction, Multiple Data) model. In CUDA, these repetitive operations are organized into a special type of function called a kernel. Usually these functions are designed to process a single element of

{Introduce n_n number of photons in random positions}
for $n = 0$ to $n_n - 1$ **do**
{Generate two random integers in $(0, size-1)$ interval}
$i \longleftarrow random_number(0, L_x - 1)$
$j \longleftarrow random_number(0, L_y - 1)$
{Look for first value of k for which $\tilde{c}_{ij}^k = 0$}
while $\tilde{c}_{ij}^k \neq 0$ and $k \leq M$ **do**
$k \longleftarrow k + 1$
end while
if $k \leq M$ **then**
{Create new photon}
$c'_{ij} \longleftarrow c'_{ij} + 1$
$\tilde{c}_{ij}^k \longleftarrow \tau_c$
end if
end for

Fig. 4. Pseudo code diagram for the implementation of the noise photons rule.

a homogeneous set of data (for example an element of a vector integer, of a float, of a complex structure, etc.).

When a CUDA kernel is executed, the GPU launches many threads as elements must be processed, and each of these threads run an "instance" of the kernel. If the number of items exceeds the maximum number of threads that the GPU can support, each thread will process more than one element. CUDA groups the threads in sets of three dimensions called blocks. These blocks are also grouped in sets of three dimensions called grids. The total number of blocks per grid dimension and the number of threads per block dimension are set when the kernel is launched, depending on the data size and the characteristics of the problem to be solved. There are several global variables that are used to identify which element should be processed by each thread. One of them (blockIdx) indicates in which block a thread is allocated. Another one, called threadIdx, is the thread identifier. With this information, and knowing the number of threads per block dimension, we can write a formula to calculate the index or indexes of the elements to be processed.

Taking into account that matrices are stored in memory row after row contiguously as if they were a vector, we have chosen to use only one of the dimensions of grids and blocks for our implementation of laser simulation. In this way we can simplify the formula that calculates the index or indexes, as shown in Fig. 5.

The main program algorithm is very simple as shown in Fig. 6. It consists of initializations, memory allocation of data on the GPU, and a temporal loop that launches three kernels which perform the cellular automaton simulation. This code is executed on the CPU in order to avoid racing conditions, because CUDA does not provide a synchronization mechanism of threads from different blocks. Global memory keeps its contains (which are shared

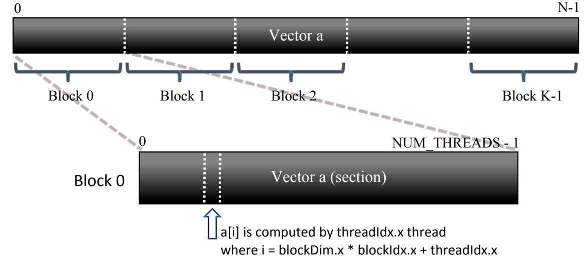

Fig. 5. Determination of the element to be processed by a thread.

between all threads of any block) between successive calls to the kernels in CUDA. It is therefore possible to write the algorithm as described. The synchronization problem has led us to write the simulation in three kernels, to avoid race conditions that would occur if we try to count, for example, the number of photons in the vicinity while another thread is performing the decay of photons.

Initialize system
Input data
Allocate GPU memory for grids
Inizialize random numbers seeds for each cell
for *time step* = 1 to *maximum time step* **do**
{blocks $=L_x$ * L_y / threads}
Call phothon decay kernel <<<blocks, threads>>>
Call pumping and electron decay kernell <<<blocks, threads>>>
Call noise photons creation kernel <<<1, 1>>>
end for
Copy results (populations) from GPU to CPU
Optional additional calculations on intermediate results
Final calculations
Output results

Fig. 6. Pseudo code diagram for the CUDA implementation of the main program for the CA laser model.

The first one of the kernels simulates the photon decay rule. The second kernel simulates electron decay, pumping, stimulated emission and count of electrons and photons. The third one is responsible for generating the random photons noise. The code of the first two kernels is a slightly modified version of the bodies of the sequential loops in the equivalent algorithms shown in Figs. 2 and 3. The main program runs the temporal loop, while the iterations through the grid are performed by thousands of threads that are created on the GPU. In order to avoid the need of synchronization to prevent collisions when different threads access the same cell (race conditions again), the third kernel is launched with a single block containing a single thread that executes sequentially the operations described in Fig. 4.

An efficient implementation of the algorithm used to count the number of photons and electrons in each temporal iteration, necessary to extract statis-

tical data from the simulation, is very important for the final performance of the GPU simulation. This is a typical problem—a parallel reduction of values stored in a vector—that has been solved previously in various ways. We have chosen a binary tree reduction algorithm based in the work of Sengupta et. al. [21].

This algorithm performs a reduction of vector portions in each of the blocks, using shared memory and synchronization barriers between the threads of a block. As a result, as many partial sums as the number of blocks used in the simulation are obtained. When all the blocks have finished computing these partial sums, they are reduced to GPU global memory using the atomic addition operation. This operation is the main bottleneck in the performance of our simulation.

VI. Performance evaluation

The performance has been evaluated in two different systems whose characteristics are shown in Table I. Each system is configured including GPU technology from the same period as its CPU. This allows us to compare the performance improvement of GPU versus CPU on equal terms, as well as to follow the evolution of CPUs and GPUs performance.

TABLA I
Tested systems characteristics

	System 1	System 2
CPU (Intel)	Core2 Quad Q6600	Core i5 750
Clock (GHz)	2.40	2.67
Cores	4	4
L1 Cache (KB)	4 x 32	4 x 32
L2 Cache (KB)	2 x 4096	4 x 256
L3 Cache (KB)	-	8192
Memory Type	DDR2 (Single Ch.)	DDR3 (Dual Ch.)
Memory (GB)	2	4
GPU (nVidia)	9500 GT	GTX 285
Stream Proc.	32	240
Core clock (MHz)	550	648
Shaders clock MHz)	1350	1476
Memory clock (MHz)	400	1242
Memory (MB)	512	1024
Bus Width (bits)	128	512

The running time spent in processing each cellular automata cell for one time iteration is shown in Fig. 7, for different values of the CA lattice side. The figure shows the timing of sequential and parallel algorithms executed in each of the test system (t_{CPU} and t_{GPU}). The performance of the sequential algorithm is reduced with the CA size because of a finite cache memory size effect: as the CA size increases, the distance between neighboring cells grows, increasing the probability that the cache line in which these cells were stored has been used for another portion of the system.

The speedup $S = t_{GPU}/t_{CPU}$ obtained by each GPU over his CPU in both tested systems for different sizes of lattice side is shown in Fig. 8. In both

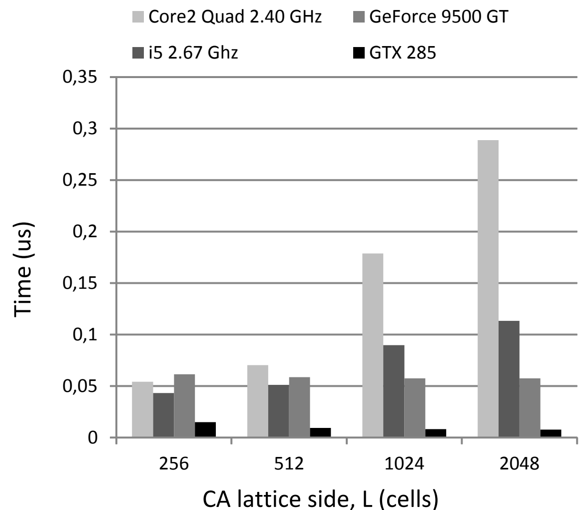

Fig. 7. Running time (in microseconds) spent in processing each cellular automata cell for one time iteration, for different values of the CA lattice side.

systems it is observed that the speedup increases with the size of the problem due to the aforementioned finite cache memory size effect. We can also see how the acceleration GPU/CPU in a few years has increased for our case study. A maximumum speedup value of 14.5 has been obtained.

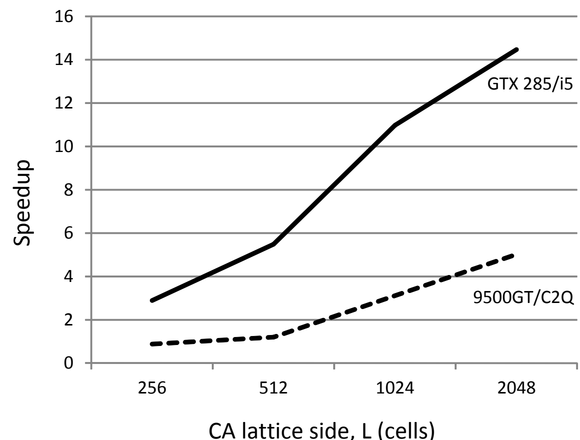

Fig. 8. Speedup obtained by each GPU over his CPU in both test systems for different sizes of CA lattice side.

VII. Conclusions and future work

A parallel implementation for GPUs of a discrete model of laser dynamics using cellular automata (CA) has been presented. This kind of model is an alternative to the standard description based on differential equations that offer advantages in different situations in which they can not be applied. Our GPU parallel implementation exploits the inherent parallelism of cellular automata to obtain a speedup of up to 14.5 over a sequential implementation running on a single core CPU.

After proving that this model can be successfully implemented on GPU with a good speedup, in the future we will try to further optimize our implementation to obtain higher performance gains.

References

[1] S. Wolfram, *Cellular automata and complexity*, Addison-Wesley, Reading, MA, 1994.

[2] A. Ilachinski, *Cellular automata. A discrete Universe*, World Scientific, Singapore, 2001.

[3] Bastien Chopard and Michel Droz, *Cellular Automata Modeling of Physical Systems*, Cambridge University Press, Cambridge, MA, USA, 1998.

[4] P. M. A. Sloot and A. G. Hoekstra, "Modeling dynamic systems with cellular automata," in *Handbook of dynamic system modeling*, P. A. Fishwick, Ed., pp. (21) 1–6. Chapman & Hall/CRC, 2007.

[5] P. M. A. Sloot A. G. Hoekstra, J. Kroc, Ed., *Simulating Complex Systems by Cellular Automata*, Springer, 2010.

[6] J. L. Guisado, F. Jiménez-Morales, and J. M. Guerra, "Cellular automaton model for the simulation of laser dynamics," *Physical Review E*, vol. 67, no. 6, pp. 066708, 2003.

[7] J. L. Guisado, F. Jiménez-Morales, and J. M. Guerra, "Computational simulation of laser dynamics as a cooperative phenomenon," *Physica Scripta*, vol. T118, pp. 148–152, 2005.

[8] J. L. Guisado, F. Jiménez-Morales, and J. M. Guerra, "Simulation of the dynamics of pulsed pumped lasers based on cellular automata," *Sixth International conference on Cellular Automata for Research and Industry, ACRI 2004. Lecture Notes in Computer Science*, vol. 3305, pp. 278–285, 2004.

[9] D. Talia and N. Naumov, *Simulating Complex Systems by Cellular Automata*, chapter Parallel cellular programming for emergent computation, pp. 357–384, Springer, 2010.

[10] S. Bandini, G. Mauri, and R. Serra, "Cellular automata: from a theoretical parallel computational model to its application to complex systems," *Parallel Computing*, vol. 27, no. 5, pp. 539–553, 2001.

[11] J. L. Guisado, F. Fernández de Vega, F. Jiménez-Morales, and K. Iskra, "Parallel implementation of a cellular automaton model for the simulation of laser dynamics," *International Conference on Computational Science, ICCS 2006. Lecture Notes in Computer Science*, vol. 3993, pp. 281–288, 2006.

[12] J. L. Guisado, F. Jiménez-Morales, and F. Fernández de Vega, "Cellular automata and cluster computing: An application to the simulation of laser dynamics," *Advances in Complex Systems*, vol. 10, no. Suppl. No. 1, pp. 167–190, 2007.

[13] J. L. Guisado, F. Fernández de Vega, and K. Iskra, "Performance analysis of a parallel discrete model for the simulation of laser dynamics," in *2006 International Conference on Parallel Processing, ICPP 2006, Workshops*. 2006, pp. 93–99, IEEE Computer Society.

[14] J. L.. Guisado, F. Fernández de Vega, F. Jiménez-Morales, K. A. Iskra, and P. M. A. Sloot, "Using cellular automata for parallel simulation of laser dynamics with dynamic load balancing," *International Journal of High Performance Systems Architecture*, vol. 1, no. 4, pp. 251–259, 2008.

[15] "Gpgpu. general-purpose computation on graphics hardware.," http://gpgpu.org, as available on may 2012., 2012.

[16] S. Gobron, F. Devillard, and B. Heit, "Retina simulation using cellular automata and GPU programming," *Machine Vision and Applications*, vol. 18, pp. 331–342, 2007.

[17] S. Rybacki, J. Himmelspach, and A. M. Uhrmacher, "Experiments with single core, multi core, and GPU-based computation of cellular automata," in *Advances in System Simulation, 2009. SIMUL'09. First International Conference on*, 2009.

[18] T. Bajzát and E. Hajnal, "Cell automaton modelling algorithms: Implementation and testing in GPU systems," in *INES 2011, 15th International Conference on Intelligent Engineering Systems*, 2011.

[19] J. Balasalle, M. A. Lopez, and M. J. Rutherford, *GPU Computing Gems Jade Edition*, chapter Optimizing Memory Access Patterns for Cellular Automata on GPUs, pp. 67–75, Elsevier - Morgan Kaufmann - NVIDIA, 2011.

[20] R. Geist and J. Westall, *GPU Computing Gems, Emerald Edition*, chapter Lattice-Boltzmann Lighting Models, pp. 381–399, Elsevier - Morgan Kaufmann - NVIDIA, 2011.

[21] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for GPU computing," in *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, 2007, pp. 97–106.