GPU-based cellular automata simulations of laser dynamics

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

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, crypography), 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 immediate 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 arbitrary 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-
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 \( N_x = L \times L \) 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, \( c_i(t) \in \{0, 1, 2, \ldots, M\} \), 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 \( \lambda \).
- **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 \( \tau_c \) is assigned to each photon when it is created. The photon will be destroyed \( \tau_c \) time steps after it was created.
- **Rule 4. Electron decay:** A finite life time \( \tau_a \) 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 \( \tau_a \) 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 \( (< \, 0.01\% \, \text{of total}) \) 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.

---

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

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

---

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 \( a_{ij} \) and \( c_{ij} \) 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\((i, j)\) > \( \delta \) then
        \( k \leftarrow 1 \)
        while \( \tilde{c}_{ij} \neq 0 \) and \( k \leq M \) do
          \( \tilde{c}_{ij} \leftarrow k + 1 \)
          \( k \leftarrow 1 \)
        end while
        if \( k <= M \) then
          \( a_{ij} \leftarrow 0 \)
          \( \tilde{a}_{ij} \leftarrow 0 \)
          \( \tilde{c}_{ij} \leftarrow \tilde{c}_{ij} + 1 \)
          \( \tilde{\tau}_c \leftarrow \tau_c + 1 \)
          \{\( \tau_c + 1 \) is assigned because 1 is substracted in the decay loop\}
        end if
      end if
    end if
  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}
    \( c_{ij} \leftarrow \tilde{c}_{ij} \)
  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 \( a_{ij} \) for the cell located at \( \vec{r} = (i, j) \) is represented as \( a_{ij} \) and \( c_{ij} \) is represented as \( c_{ij} \). Two temporal variables, \( a_{ij} \) and \( \tilde{c}_{ij} \), are used as time counters, where \( k \) distinguishes between the different photons that can occupy the same cell. When a photon is created, \( \tilde{c}_{ij} = \tau_c \). After that, 1 is substracted to \( \tilde{c}_{ij} \) for every time step and the photon will be destroyed when \( \tilde{c}_{ij} = 0 \). 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
        \{Subtract 1 to every photon’s lifetime\}
        if \( \tilde{c}_{ij} > 0 \) then
          \( \tilde{c}_{ij} \leftarrow \tilde{c}_{ij} - 1 \)
        end if
        if \( \tilde{c}_{ij} = 0 \) then
          \{One photon decays\}
          \( c_{ij} \leftarrow c_{ij} - 1 \)
          \( c_{ij} \leftarrow c_{ij} \)
        end if
      end for
    end if
  end for
{Generate random number in \((0, 1)\) interval}
\( \xi \leftarrow \text{random\_number}(0, 1) \)
if \( \xi < \lambda \) then {\( \lambda \): pumping probability}
  \{One electron is pumped\}
  \( a_{ij} \leftarrow 1 \)
  \( \tilde{a}_{ij} \leftarrow \tau_a \)
end if
else if \( a_{ij} = 0 \) then {Apply pumping rule}
  \{Generate random number in \((0, 1)\) interval\}
  \( \xi \leftarrow \text{random\_number}(0, 1) \)
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 Programming) 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
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 contents (which are shared 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.

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.

<table>
<thead>
<tr>
<th>Tested systems characteristics</th>
<th>System 1</th>
<th>System 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU (Intel)</td>
<td>Core2 Quad Q6600</td>
<td>Core i5 750</td>
</tr>
<tr>
<td>Clock (GHz)</td>
<td>2.40</td>
<td>2.67</td>
</tr>
<tr>
<td>Cores</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>L1 Cache (KB)</td>
<td>4 x 32</td>
<td>4 x 32</td>
</tr>
<tr>
<td>L2 Cache (KB)</td>
<td>2 x 4096</td>
<td>4 x 2096</td>
</tr>
<tr>
<td>L3 Cache (KB)</td>
<td>-</td>
<td>8192</td>
</tr>
<tr>
<td>Memory Type</td>
<td>DDR2 (Single Ch.)</td>
<td>DDR3 (Dual Ch.)</td>
</tr>
<tr>
<td>Memory (GB)</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>GPU (nVidia)</td>
<td>GeForce 9500 GT</td>
<td>GTX 285</td>
</tr>
<tr>
<td>Stream Proc.</td>
<td>32</td>
<td>240</td>
</tr>
<tr>
<td>Core clock (MHz)</td>
<td>550</td>
<td>648</td>
</tr>
<tr>
<td>Shaders clock (MHz)</td>
<td>1350</td>
<td>1476</td>
</tr>
<tr>
<td>Memory clock (MHz)</td>
<td>400</td>
<td>1242</td>
</tr>
<tr>
<td>Memory (MB)</td>
<td>512</td>
<td>1024</td>
</tr>
<tr>
<td>Bus Width (bits)</td>
<td>128</td>
<td>512</td>
</tr>
</tbody>
</table>

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

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


