Cluster Comput
DOI 10.1007/s10586-017-0850-3

Performance analysis and comparison of cellular automata GPU implementations

Emmanuel N. Millán[1] · Nicolás Wolovick[2] · María Fabiana Piccoli[3] · Carlos García Garino[4] · Eduardo M. Bringa[1]

Received: 11 November 2016 / Revised: 28 March 2017 / Accepted: 29 March 2017

Abstract Cellular automata (CA) models are of interest to several scientific areas, and there is a growing interest in exploring large systems which would need high performance computing. In this work a CA implementation is presented which performs well in five different NVIDIA GPU architectures, from Tesla to Maxwell, simulating systems with up to a billion cells. Using the game of life (GoL) and a more complex variation of GoL as examples, a performance of 5.58e6 evaluated cells/s is achieved. The two optimizations most often used in previous studies are the use of shared memory and Multicell algorithms. Here, these optimizations do not improve performance in Fermi or newer architectures. The GoL CA code running in an NVIDIA Titan X obtained a speedup of up to ∼85 x and up to ∼230 x for a more complex CA, compared to an optimized serial CPU implementation. Finally, the efficiency of each GPU is analyzed in terms of cell performance/transistors and cell performance/bandwidth showing how the architectures improved for this particular problem.

Keywords Cellular automata · Graphics processing unit · Benchmarks · Performance analysis

✉ Emmanuel N. Millán
emillan@itu.uncu.edu.ar

[1] CONICET and FCEN, Universidad Nacional de Cuyo, Mendoza, Argentina

[2] Universidad Nacional de Córdoba, Córdoba, Argentina

[3] Universidad Nacional de San Luis, San Luis, Argentina

[4] ITIC and FING, Universidad Nacional de Cuyo, Mendoza, Argentina

1 Introduction

Cellular automata (CA) models have been used in several research areas because they involve simple models which can be used to simulate complex behavior. The CA are composed of a regular or irregular lattice of cells, where each cell interacts with his neighbors with a set of rules that define the evolution of the simulated system. There is a broad array of CA applications, including metallic materials [48], lattice Boltzmann systems [15,44], water flow [55], solidification of grain structures [9], urban grow modeling [4,57], heart simulations [8], wildfire spread [49], edge detection in images [46], etcetera.

High performance computing (HPC) has allowed solving problems of great magnitude which were inaccessible before the advent of innovative software taking advantage of powerful HPC hardware. Several HPC hardware solutions exist: dedicated computer clusters [56], specialized vector hardware such as Cray supercomputers or the cell broadband engine architecture, FPGA, Beowulf-type clusters and, recently hardware accelerators such as graphics processing units (GPU) and the Intel many integrated core (MIC) architecture. These last two options, GPUs and MICs, are attractive to perform HPC calculations because of their low implementation cost and high performance. GPUs have been used for research works in PCs and notebooks [47,50], small clusters [6] and they are present in 66 of the Top 500 Supercomputers List (June 2016 list [56]). Several research papers and benchmarks have demonstrated the high performance of GPUs compared to x86 microprocessors. Brown et al. [6] tested simulations with the LAMMPS Molecular Dynamics package and obtained speedups of ∼20 x using two GPUs and ∼9 x using 12 CPU cores against a serial CPU execution. Preis et al. [45] implemented a Monte Carlo simulation of the Ising model in GPUs, obtaining 35 x of speedup against one CPU code implementation, and more recently Ferrero et al. [16] obtained 155 x of speedup using one GPU against an optimized CPU serial code for the q-state Potts model.

The Game of Life (GoL) [18] is a well known CA simulation, where lattice cells can have a state of either dead or alive, and a set of simple rules define how the system changes after each generation or time step. The GoL has been studied and implemented in GPUs [1,20,43,50]. Perumalla et al. [43] presented an implementation using OpenGL and its performance was compared with a CPU implementation, and with the Repast [32] and NetLogo [60] CPU codes, achieving up to 16 x versus the CPU execution, and nearly 10^3-10^4 x versus the Repast code. Aaby and Perumalla et al. [1] expanded their previous work and developed a Multi-GPU/Multi-Node implementation of GoL to run in clusters of GPUs using CUDA [14], MPI and POSIX pthreads. They implemented a hiding latency technique to improve the performance of the GPU code by performing N updates of cells before communicating neighborhood data to other processes. This work obtained a speedup of over 30 x for Multi-GPU over a CPU code, and over 10^3 x of speedup against a CPU-based Java code. Rybacki et al. [50] used the James II framework (a Java based simulator, available at www.jamesii.org) to test different CA simulations, including GoL.

In previous works [29,30] different implementations of the Game of Life (GoL) [18] CA were shown. In [30] a preliminary version of the GoL CA was studied with different implementations: serial CPU, shared memory with OpenMP, distributed memory with MPI, GPU with CUDA, and a Multi-GPU implementation with MPI and CUDA. The shared memory implementation had poor performance compared with the distributed memory implementation, both running in the same multi-core workstation. Oxman et al. [41] also obtained good results with a distributed memory implementation compared with a shared memory version of the GoL CA. In Millán et al. [29] different optimizations were implemented and a performance analysis of the GoL CA for serial CPU and distributed memory implementations were studied using hardware counters [7]. The GPU code implemented in [30] obtained good speedups (20 x for the Multi-GPU code) but a study of different optimizations previously used was not carried out. Several optimizations for CA or Stencil codes have been implemented in the literature, such as shared memory [10,55], multicell [2], warp specialization [27], and loop unrolling/prefetching [51]. The use of the shared memory of the GPU presented a good starting point to improve the performance of CA and Stencil codes, as seen in [10,55], due to the high speed of shared memory compared with global memory. The introduction of a L1 cache in Fermi and post-Fermi GPU architectures might signal that it is no longer necessary to use shared memory to obtain good results [35]. Gibson et al. [19] and Topa et al. [54] tested the GoL CA in a Fermi GPU and obtained better results using global memory instead of shared memory. Maruyama et al. [27] developed a Stencil diffusion equation with several optimizations, included shared memory, in a Fermi and Kepler GPUs, with good results for the Kepler GPU using several optimizations techniques combined with shared memory.

The focus of this work is to benchmark and analyze the performance of four HPC GPU implementations of the GoL CA in five NVIDIA GPUs architectures, from the *Tesla G96* architecture to the *Maxwell GM200* architecture. A GPU baseline code and three implementations with commonly used optimizations techniques are tested to analyze how they behave in different GPU architectures. These benchmarks will allow the correct selection of which optimizations work well in the tested architectures, and they will also show that, for NVIDIA Fermi or newer architectures, a baseline GPU implementation results in greater speedups than two "classic" GPU optimizations. Several numerical experiments were designed to test different aspects of the GoL CA with the four implementations. Speedups of up to ∼230 x are obtained with the high range NVIDIA GeForce Titan X GPU. Also, two new metrics are used, which consists in the performance obtained normalized with the transistors count and the memory bandwidth of each of the tested GPUs.

This work is organized as follows. In Sect. 2 an introduction to the Game of Life CA is given, followed by the description of the Serial CPU and distributed memory codes. Next, the GPU implementations tested in this work are described. Section 3 includes details on the hardware and software used, with a description of the five experiments tested in this work. After this, results of the experiments and a discussion of the obtained results are presented, followed by the conclusions of this work.

2 Game of life

This section gives an introduction to the Game of Life and specifies the code implementations for serial CPU and distributed memory with MPI, with the GPU code implementations following.

The Game of Life (GoL) CA [18] has been studied in detail [20,41], and uses the Moore neighborhood (8 neighbors) [17] with periodic boundary conditions. The evolution of the CA is given by the states of each cell and of his neighbors, that can take two values, "alive" or "dead" (1 or 0 values respectively). At each time step, the CA checks the following rules for each cell of the lattice:

– Any cell with a state of alive that has less than two living neighbors will die in the next time step (isolation).
– Any cell with a state of alive that has two or three living neighbors will live in the next time step.

– Any cell with a state of alive that has more than three living neighbors will die in the next time step (overcrowding).
– Any dead cell that has exactly three living neighbors will be alive in the next time step (reproduction).

The Game of Life is a simple CA to evolve: to calculate the new state of each cell in the lattice it is only necessary to read nine values from memory, sum eight cell states, check the sum with three conditional statements, and write one value to memory. Therefore, this is not considered to be a compute intensive problem but a memory bound problem. Gibson et al. [20] tested the GoL CA in GPUs, concluding that a more compute intensive CA can obtain greater speedups than the GoL. In order to test this assumption, without using a significantly more complex CA algorithm such as reaction–diffusion [53] or Lattice Boltzmann (LB) [11], an additional numerical experiment was developed, where a compute intensive operation (the cosine function) was added to the GoL CA CPU serial and GPU codes. The objective is not to observe the behavior of the GoL with a cosine operation, but just to add compute time to the calculation of each cell to convert the problem from memory bounded to compute bounded.

Gibson et al. [20] also tested the effect of increasing the neighborhood radius from 1st to 5th neighbors. This was considered as an important test in the GPU codes implemented here, since it can increase memory pressure and the high memory bandwidth of the GPUs could obtain greater speedups against the CPU version. Increasing neighborhood size causes the GoL CA to decrease "activity" because of the increased possibility of encountering alive neighbors (overcrowding). Therefore, Gibson et al. [20] studied initial random probability distributions needed to guarantee activity up to a certain number of steps. These results by Gibson will be discussed further in Sect. 3.

In the next subsections, the serial and parallel implementations of the GoL are described including the implemented optimizations for GPU and CPU versions of the code.

2.1 CPU serial code and distributed memory implementations

In Millán et al. [29], CPU versions of GoL were implemented for CPUs, for both serial and MPI processing. Four serial implementations were tested: the *baseline* implementation, the *swap* version that switches the current state array with the next state array, the *one_grid* implementation that uses only one array to store (with *int* data type) the current and next state arrays, and the *one_grid_char* implementation that uses the *char* data type instead of *int* to store the states of the cells. The *one_grid* optimization resulted in the best speedup against the baseline code in an AMD FX-8350 4 GHz CPU (∼2.2 x) and is the version used here.

Three distributed memory implementations developed with MPI were tested in [29]: *baseline*, *one_grid*, and *one_grid* with *char* data type. Again, the *one_grid* version with the *integer* data type had the best performance, with speedups from ∼20 up to ∼75% against the MPI baseline code, and it is used here to compare with the GPU implementations.

This CA implementation added ghost columns and rows, called halo [23,29], simplifying the treatment of boundaries.

2.2 GPU code implementations

A description of GPU code implementations is given below. Four implementations were developed for this work: a GPU Baseline code that uses only global memory, two codes using shared memory, and a Multicell implementation. The shared memory versions differ mainly in their memory access pattern. Code snippets are not shown in this work because they would increase the number of pages considerably, although the entire source code for the GPU and CPU implementations is publicly available in http://goo.gl/9X7tcy. Other optimizations that could be analyzed in the future are: warp specialization [3,27], kernel specialization [31], ghost cell expansion [28,62] (specially for multi-GPU implementations), bit packing [21,41], and overlapping of calculations between CPU and GPU [52].

The performance of these four implementations will be affected by the different features of the GPU architectures. The features of each of the six GPUs (comprising five NVIDIA architectures) can be seen in Sect. 3.2. In Sect. 3.3, the numerical experiments are executed and the results of how each algorithm behaves in the different GPUs are discussed.

All GPU codes use three global kernels and two device functions. The first two global kernels are common to all implementations and they are in charge of implementing the periodic boundary conditions of the GoL CA by copying the halo rows (*copy_Rows()* kernel) and columns (*copy_Cols()* kernel) into the GPU global memory at the beginning of every time step [23]. The *copy_Rows()* kernel copies the first row of the lattice into the bottom halo row, next, the same kernel copies the last lattice row into the top halo row. Once this kernel finishes executing, the kernel *copy_Cols()* is in charge of copying the first left column (including the first and last halo cells) of the lattice into the right halo column, and the last right column (also including the first and last halo cells) of the lattice into the left halo column. A complete description and optimizations of the halo cell pattern can be found in ref. [23]. The third global kernel (*moveKernel()*) is specific to each GPU implementation and it is detailed in the following subsections. The two device functions are common to all implementations and they are in charge of adding the values of the neighbors of a given cell (*count_neighs()*) and to compute the next state of the cell by applying the GoL rule set (*check_rules()*). These two device functions are called from the *moveKernel()* global kernel. Once the *moveKernel()* has finished, the *next state array* is swapped with the *current state array* and a new time step is started. In the next subsections, a description of each of the GPU algorithms is given.

The memory bandwidth from the GPU global memory to the main memory was not taken into account because this CA does not need to copy data from/to the main memory once it has started evolving in the GPU. Access to main memory is of course necessary to output the CA configuration to storage for off-line analysis and visualization.

2.2.1 GPU baseline algorithm

The Baseline (named *GPU Baseline*) code calls the *moveKernel()* global kernel function that reads the state of the cells from the GPU global memory and computes the GoL rule set. It is a simple and straightforward implementation that uses one global CUDA kernel and the two device functions, *count_neighs()* and *check_rules()*. Each thread is responsible of computing the next state of a single lattice cell. Using the *count_neighs()* function, each thread reads from global memory the state (1 or 0 values) of his eight neighbors (Moore neighborhood), and sums its values. Next, *check_rules()* is called with the value obtained from *count_neighs()* and the rule set of the GoL CA is applied. Finally the next state of the cell is stored in the *next state array*. The *moveKernel()* global kernel function can be relatively easily modified to accommodate more complex CA.

2.2.2 Shared algorithm v1

The first shared memory version (named *Shared v1*) uses a simple approach, where each thread is responsible for copying one cell of the lattice from global memory to shared memory [10]. Threads that belong to the halo of each block of threads do not compute the rule set of the GoL. In Fig. 1 an example of this approach can be seen for a CA lattice of 64×64 cells and a GPU block size of $8 \times 8 = 64$ threads.

Each block has a 2D topology of 8×8 threads ($blockDim.x = 8$ and $blockDim.y = 8$), and each thread for every block is responsible for loading from global memory one cell and storing its value to a shared memory array. In the example shown in Fig. 1 cells surrounding the shared memory array are the halo (dark gray color), and these threads do not compute the GoL rule set. For this block size, halo cells represent ~44 % of the threads in the block, which will not perform any task beyond loading data from global memory to shared memory. Also, more blocks of threads will be needed to cover the entire CA lattice, as only 36 threads (green or light gray

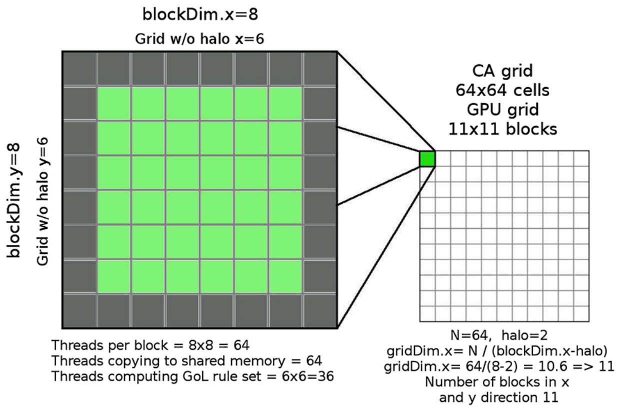

blockDim.x=8
Grid w/o halo x=6
blockDim.y=8
Grid w/o halo y=6
CA grid
64x64 cells
GPU grid
11x11 blocks
Threads per block = 8x8 = 64
Threads copying to shared memory = 64
Threads computing GoL rule set = 6x6=36
N=64, halo=2
gridDim.x= N / (blockDim.x-halo)
gridDim.x= 64/(8-2) = 10.6 => 11
Number of blocks in x
and y direction 11

Fig. 1 CA GPU shared memory implementation v1 *Shared v1*. Halo cells (*dark gray*) will only copy data from global to shared memory, and the remaining cells (*green* or *light gray* cells) will both copy data and compute the GoL rule set (Color figure online)

cells) of the 64 threads in each block will be computing the GoL. For a lattice of $N \times N$ cells, the blocks needed to calculate the entire lattice are $N/(blockDim.x - halo)$, and an example of this calculation can be seen in Fig. 1.

2.3 Shared algorithm v2

The second shared memory version (named *Shared v2*) uses a more complex pattern to access global memory, as implemented by Topa et al. [55]: each thread correspond to one lattice cell and is in charge of copying its state from global to shared memory. In addition, threads from the first to the fourth row of the 2D block of threads are in charge of copying halo cells to shared memory. Unlike the *Shared v1* implementation, here all threads of the block compute the GoL rule set. An example of this memory access pattern can be seen Fig. 2. Threads from the first/second row (green/cyan) of the block of threads are in charge of copying the top/bottom rows of the halo from global memory, and also of copying the corner top/bottom cells. The third/fourth (orange/blue) rows of threads are in charge of copying the left/right columns of the halo from global to shared memory. To calculate the number of blocks of threads in the GPU grid there is no need to take into account the number of halo rows or columns, as it was necessary with the *Shared v1* code. In the case shown in Fig. 2, with $N = 64$ and a block size of 64 threads (8×8 2D block), the number of blocks in each direction (for a 2D block grid) is $N/blockDim = 64/8 = 8\,blocks$ in each direction.

2.4 Multicell algorithm

The last implementation, named *Cells*, uses half the number of threads rather than use a thread for each single cell of the lattice, because each thread computes the state of two contiguous cell grids [2]. In Fig. 3 each thread computes the

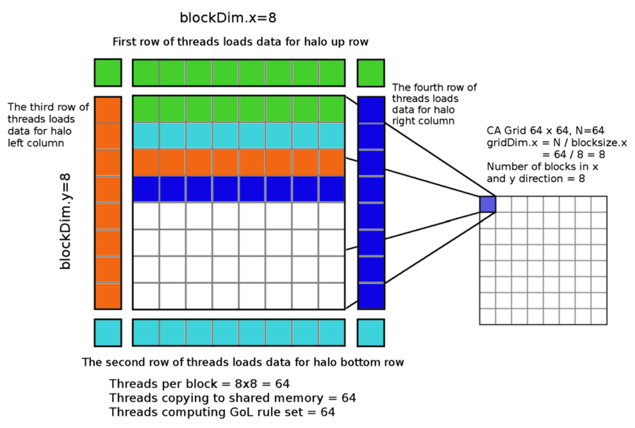

blockDim.x=8
First row of threads loads data for halo up row
The third row of threads loads data for halo left column
The fourth row of threads loads data for halo right column
CA Grid 64 x 64, N=64
gridDim.x = N / blocksize.x = 64 / 8 = 8
Number of blocks in x and y direction = 8
blockDim.y=8
The second row of threads loads data for halo bottom row
Threads per block = 8x8 = 64
Threads copying to shared memory = 64
Threads computing GoL rule set = 64

Fig. 2 CA GPU shared memory implementation v2 *Shared v2*. All threads in the block compute the GoL rule set. All threads copy from global to shared memory the state of a single cell. *Green threads* copy data from global to shared memory for the top row of the halo, *cyan* threads copy data for the bottom halo row, *orange* threads copy the first left halo column, and *blue* threads copy the right halo column (Color figure online)

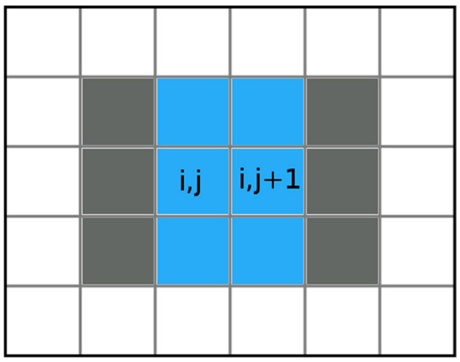


Fig. 3 GoL GPU cells implementation. Each thread computes cell states for two cells, (i,j) and (i,j+1). These two cells share the state of the *blue* cells, while *gray* cells are not shared between the two calculations (Color figure online)

next state for the cells (i,j) and (i,j+1), sharing six blue cells. This approach should diminish memory access, because each thread shares access to six cell states, from a total of 12 cells states needed by the thread (six gray cells are not shared). The number of cells computed by each thread can actually be changed in the code by an input parameter. In this work, results for two cells per thread are reported in detail, but tests for four cells per thread were also executed, without significant changes in performance, and will not be discussed further.

3 Results and discussion

In this section a series of five numerical experiments are executed in six GPUS from five different NVIDIA GPU architectures. First, this section details the hardware and software specifications used in this work, along a short description of the NVIDIA GPU architecture. Next, a description of the experiments performed is given. Numerical experiment results are discussed, and finally a summary of the results is given.

3.1 Hardware infrastructure and software specifications

GPU, MPI and Serial simulations were executed in four different environments: *FX-8350* from the Universidad Nacional de Cuyo (UNCuyo), *9400GT* from the Universidad Nacional de San Luis, *Opteron* from UNCuyo and *K20x* from the Universidad Nacional de Córdoba. In Tables 1 and 2 it can be seen the hardware and software specifications for each environment.

In the next subsection, a description of the NVIDIA GPU architecture is given to facilitate the discussion of the obtained results.

3.2 NVIDIA GPU architecture

The NVIDIA GPU architecture has been discussed in depth, including official documentation from NVIDIA [13,33–37,40], and research papers [5,26,42,59]. A description of the most relevant features of the GPUs used in this work can be seen in Table 3. References for each of the GPUs can be seen in the last row of the same table. The GPU model of execution is at the thread-level. GPUs group threads into blocks. These blocks are executed in Streaming Multiprocessors (SM), which includes processing units and memories, and their organization changes from architecture to architecture. In architectures previous to Fermi, the Streaming Multiprocessors are called SM, in Kepler SMX, and in Maxwell SMM. In the literature these naming differences are often disregarded, and the SM expression is used to refer to Streaming Multiprocessors in all architectures. This is the convention used here.

Table 3 give details about four features of GPUs which require additional explanations, involving Streaming Multiprocessors (SM): Registers/SM, Blocks/SM, Warps/SM and Threads/SM. The amount of registers that a SM can assign to running threads at a given time is defined by Registers/SM. The number of blocks that can be executed simultaneously in each SM is given by Blocks/SM. Next, the number of warps that can be schedule to execute in each SM is defined by Warps/SM. Finally, the number of threads that a SM can execute simultaneously is given by Threads/SM. The amount of resources a given kernel uses from these features is called occupancy.

Generally, occupancy has been used as an indicator for performance, with higher occupancy resulting in higher performance. Here, an example of how occupancy is calculated follows below. In the Tesla C2050 (with Fermi architecture)

Table 1 Hardware infrastructure used to execute the simulations

Name	CPU	RAM	GPUs
FX-8350	AMD FX-8350 8 cores at 4 GHz	32 GB	GeForce GTX Titan X (Maxwell GM200) 12 GB GeForce GTX 750ti (Maxwell GM107) 2 GB Tesla C2050 (Fermi GF100) 3 GB GeForce 210 (Tesla GT218) 1 GB
9400GT	Intel Core 2 Quad CPU Q9550 4 cores at 2.83 GHz	4 GB	GeForce 9400GT (Tesla G96) 0.5 GB
Opteron	Four AMD Opteron 6272 16 cores each (64 cores), at 2.1 GHz	128 GB	None
K20x	Two Intel Xeon E5-2680 v2 10 cores each (20 cores), at 2.8 GHz	64 GB	Tesla K20x (Kepler GK110) 6 GB

Table 2 Software specifications used to execute the simulations

Name	Linux version	Kernel version	OpenMPI	GCC	CUDA	NVIDIA Driver
FX-8350	Slackware 14.1 64 bit	3.10.17	1.8.1	4.8.2	6.5	349.16
9400GT	Debian 5.0.8 64 bit	2.6.26	NA	4.3.2	5	304.54
Opteron	Rocks 5.5 64 bit	3.10.46	1.8.1	4.8.1	NA	NA
K20x	CentOS 6.5 64 bit	2.6.32-504	NA	4.8.4	6.5	340.29

Table 3 Features of the NVIDIA GPUs used in this work

GPU	9400GT	GT210	C2050	K20x	GTX750	Titan X
Architecture	Tesla G96	Tesla GT218	Fermi GF100	Kepler GK110	Maxwell GM107	Maxwell GM200
Cores	16	16	448	2688	640	3072
Core clock	550 MHz	520 MHz	1.15 GHz	732 MHz	1.02 GHz	1 GHz
Transistors count	314 M	260 M	3100 M	7100 M	1870 M	8000 M
SM	2	2	14	14	5	24
SPs/SM	8	8	32	192	128	128
Registers/SM	8 K	16 K	32 K	64 K	64 K	64 K
Blocks/SM	8	8	8	16	32	32
Warps/SM	24	32	48	64	64	64
Threads/SM	768	1024	1536	2048	2048	2048
Shared/L1 Mem. size KB	16	16	48/16 or 16/48	48/16 or 32/32	64/24	96/24
Mem. bandwidth	12.8 GB/s	9.6 GB/s	144 GB/s	250 GB/s	86.4 GB/s	336.5 GB/s
Mem. type	DDR2	DDR3	GDDR5	GDDR5	GDDR5	GDDR5
Mem. width	128-bit	64-bit	384-bit	384-bit	128-bit	384-bit
Amount of memory	512 MB	1 GB	3 GB	6 GB	2 GB	12 GB
Compute Capability	1.1	1.2	2.0	3.5	5.0	5.2
References	[12,37,59]	[12,40,42]	[35]	[13,36]	[13,33]	[13]

only 32 K registers can be used simultaneously per SM (see Table 3). Using 128 threads per block, and if each thread needs 43 registers for a given kernel, this gives a total of 5504 registers per block. The number of blocks that can be assigned to one SM is $32K/5504\,registers = 5.8 = 5\,blocks$. The maximum number of blocks that a SM can execute simultaneously is given by $Blocks/SM = 8$, leaving 3 blocks that cannot be executed because the limit of maximum number

of registers used at a given time has been reached. Warps are groups of 32 threads, $5\,blocks \times 128\,threads/32 = 20\,warps$), and with this configuration only 20 warps are being executed from a possible 48 warp maximum. The total number of threads which are running simultaneously is $128\,threads \times 5\,blocks = 640\,threads$ of a maximum of 1536. This example would result in occupancy of $640 \times 100/1536 = 41$, and $\sim$59% of the SM is idle. If the number of registers per thread could be reduced to 30, by modifying the kernel code or by using the *-maxregcount* compiler flag to cause register spilling to global memory, a total of 3840 registers would be necessary for each block, given $32K/3240 = 8.3 = 8$ blocks executing simultaneously in a SM, giving $8\,blocks \times 128 threads/32 = 32\,warps$ running simultaneously and 1024 threads/SM ($8\,blocks \times 128\,threads$), resulting in occupancy of 66%. Although occupancy is considered a measure to take into account to improve performance, it has been shown by Volkov et al. [58] that low occupancy can sometimes give better performance than high occupancy. Also, Hong et al. [22] have discussed that to measure occupancy as the only metric to improve performance might not be sufficient.

Another important measure that is used to analyze performance is code divergence. A divergence appears when a given code-path is split between different threads inside a warp. For example, by reaching a conditional statement, threads might take different code-paths. GPUs solve this problem by serializing execution between the code-paths. Threads in the same warp join parallel execution after the divergence in the code finishes (after the conditional statement ends). If the percentage of divergence in the execution of a kernel is high, performance will likely drop for that kernel.

The Fermi architecture introduced L1 and L2 caches. Access to global memory is cached in L1 memory. This improves considerably the performance of codes which have uniform memory access, and reduces the need to use shared memory explicitly [19,54]. In the Kepler architecture this behavior was modified, and global memory access is cached in the L2 cache. To unify code behavior in post-Fermi architectures, the *-dlcm=ca* compiler flag was used to force Kepler and Maxwell GPUs to cache in L1, as the Fermi architecture does. For more information on Kepler see section 1.4.4.2. of [38], and for Maxwell see section 1.4.2.1. of [39]. The ECC memory feature was turned off in all the GPUs that supported.

GPUs from the Tesla architecture with a Compute Capability (CC) of 1.0–1.2 only support single precision floating point operations [12]. From CC 1.3 and onwards, double precision is also supported. For this reason, the numerical experiment reported here which involves *double* data types, uses single precision (*float*) in the 9400GT and GT210 GPUs.

The next subsection details the executed numerical experiments and their configuration.

3.3 Numerical experiments

Five experiments were designed to test different aspects of the CA running in GPUs.

First, different block sizes of threads were tested for the different GPU implementations. This is important because computational resources (GPU core registers, shared memory, L1 cache, etc.) vary for different block sizes, and there are no simple rules to find the optimum block size. Second, the performance of all the GPU implementations was tested in several NVIDIA GPUs from different NVIDIA architectures (Tesla, Fermi, Kepler and the latest Maxwell architecture). Third, the performance of the best GPU implementation of each GPU was compared to the distributed memory MPI code optimized in Millán et al. [29]. The fourth experiment involved increasing the radius of the neighborhood from 1st to 5th nearest neighbors, to increase the amount of data that each thread needs, from 9 (radius = 1) to 121 (radius = 5) number of neighbors. The GoL CA is a memory bound problem, where compute time is small compared with memory accesses time. For this reason, the last experiment adds a complex math operation (cosine) to the calculation of the state of each cell, in order to increase compute time. Executing these two types of CA, the classical GoL which is memory bound, and the complex modification of the GoL which is compute bound, would give an approximation of how the GPU implementations tested in this work might behave with other CAs.

A 2D lattice is used in all experiments, with a size of $N \times N$, N being 512, 1024, 2048, 4096, 8192 and 16384. All experiments are executed for 1000 steps. The initial random seed is the current time in nanoseconds given by the Linux command "*date +%N*". The initial random distribution of alive cells is 50 % for a neighborhood radius of 1. For the experiment with a neighborhood radius > 1 the initial distribution of alive cells used in this work is obtained from Gibson et al. [20], with the following values: $radius = 2$ with 25 %, $radius = 3$ with 14 %, $radius = 4$ with 10 % and $radius = 5$ with 4 %.

The executed code in all experiments is compiled with GCC (compiler optimizations -O3). In the case of the GPU implementations, the code is compiled with the maximum compute capability (CC) that each GPU supports. Results were averaged over five simulations. Standard deviation was generally below 1% in all cases and, therefore, error bars are not shown in the graphs. The largest lattice sizes ($N = 8192$ and $N = 16384$) could not be executed in some GPUs (9400GT, GT210 and GTX750) due to the amount of global memory each GPU has (0.5, 1 and 2 GB respectively). The next subsections show the five numerical experiments with a discussion of the obtained results, followed by a summary.

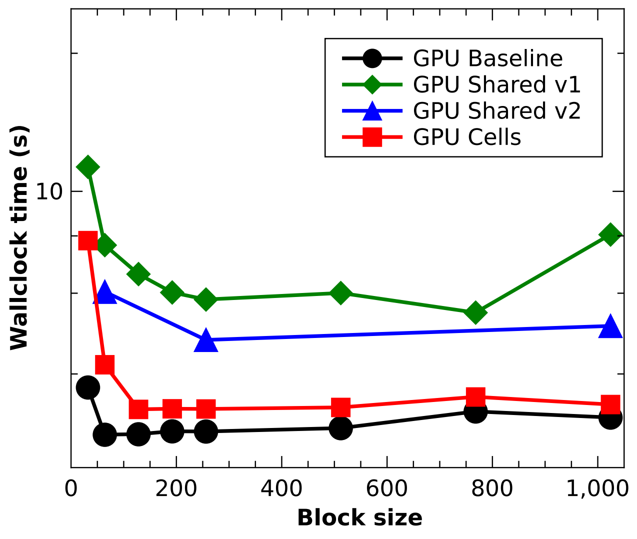


Fig. 4 Wall clock time of GPU simulations versus block size, for all GPU versions of the code, with lattice size=4096 × 4096, executed in the Titan X GPU

3.3.1 *Different block sizes*

The optimum block size of threads is generally selected by performing benchmarks. Previous studies have shown that performance of GPU codes varies with block size [22,51]. Execution time for the four GPU code implementations is shown in Fig. 4, for several 2D block sizes ($4 \times 8 = 32$, $8 \times 8 = 64$, $8 \times 16 = 128$, $12 \times 16 = 192$, $16 \times 16 = 256$, $16 \times 32 = 512$, $32 \times 24 = 768$ and $32 \times 32 = 1024$ threads per block). The best performance is obtained with *GPU Baseline*, except for a single marginal exception. All implementations display excellent timing for 256 (16×16 threads) threads per block and, therefore, all remaining experiments will be executed for that block size. In Fig. 4 it can be seen the results only for the Titan X GPU, and similar results were obtained for the C2050 and K20X GPUs.s

3.3.2 *Performance in different GPU architectures*

Numerical experiments shown in Fig. 5 test the performance of the four GPU implementations with GPUs from five different NVIDIA GPU architectures previously discussed in Sect. 3.1. The performance of the GPU implementations is similar for all GPUs, except for the 9400GT (pre-Fermi) GPU. The absence of L1 cache in pre-Fermi GPUs force the use of shared memory to improve memory access performance [10,55], and therefore improve overall performance in this memory bound problem. For this reason, the best performance for the 9400GT GPU was obtained with the GPU *Shared v1* and *Shared v2* implementations, reaching about ∼2 x of speedup compared with the *Baseline* code. The *Base-*

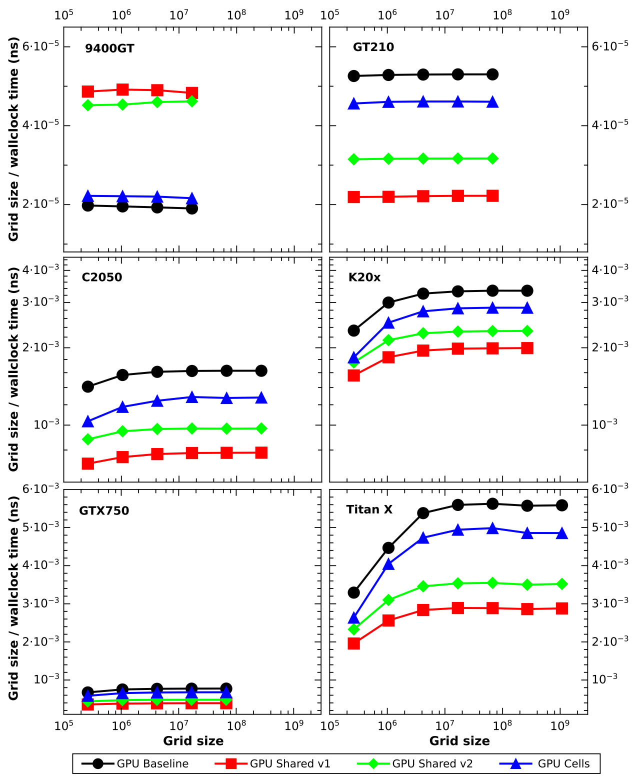


Fig. 5 GoL simulations for several GPU implementations, executed in five different NVIDIA GPUs architectures: Tesla G96 (9400GT), Tesla (GT210), Fermi (C2050), Kepler (K20x) and Maxwell (GTX750 and Titan X)

line code gives the best performance in Fermi and post-Fermi architectures. This is due to the implicit and automatic use of the L1 cache of global memory loads, a behavior also seen in [19,54] for CA simulations and in Maruyama *et al.* for Stencil Computation [27]. In the G96 architecture the *Shared v2* implementation is ∼1.05 x faster than *Shared v1*, while this changes to ∼1.2 x faster for post-G96 architectures. The *Multicell* implementation has a performance close to the *Baseline* code, but the results in this work do not show any benefit to implement a *Multicell* approach without shared memory for pre-Fermi architectures, as seen by Balasalle et al. [2]. The performance of all GPU implementations has a N^2 scaling with lattice size, which is the expected scaling for a good parallel implementation and for this type of problems.

3.3.3 *Benchmarks for different lattice sizes*

In a previous work by Millán et al. [29] the GoL was optimized for serial CPU and distributed memory with MPI execution. In this work, those codes are used to compare with the performance of the GPU implementations presented here. The best GPU implementations for each NVIDIA GPU can be selected from the previous experiment: the *Shared*

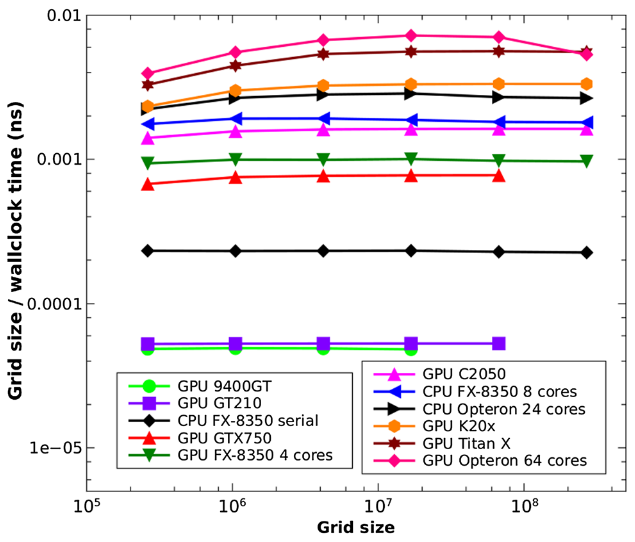


Fig. 6 Simulations for optimized MPI code and GPU implementations for different lattice sizes. The two low-end GPUs, 9400GT and GT210, are slower than a single core of the FX-8350 CPU. The Titan X GPU has a similar performance than 64 Opteron cores running an optimized GoL implementation

v2 code performs well in the 9400GT GPU, and the *GPU Baseline* code gives the best performance for the rest of the GPUs.

These numerical experiments use only those two GPU implementations to compare their performance against the optimized CPU Serial and MPI implementation running in two multicore workstations (FX-8350 and Opteron). The six GPUs (GeForce 9400GT, GeForce GT210, Tesla C2050, Tesla K20x, GeForce GTX750Ti and GeForce Titan X) are compared in Fig. 6 against two multicore CPUs, the AMD FX-8350 with 8 cores, and a cluster node with four AMD Opteron 6272 with 16 cores each (64 cores total).

The two oldest and low-end GPUs, the 9400GT and the GT210, have a similar performance and cannot match the performance of a single core of the FX-8350 CPU. The GTX750 GPU has a performance close to four cores of the FX-8350 CPU. The C2050 GPU compares well with eight cores of the FX-8350 CPU, the K20x GPU performs better than 24 CPU cores of the Opteron 6272. Finally, the Titan X GPU has a performance close to 64 cores of the Opteron cluster node. Gibson et al. [20] concludes that the GoL is a simple CA to compute for a GPU, a more complex CA that runs for many more steps will result in even better speedups. Also, it is important to consider that the MPI implementation used here has ∼20–∼75% better performance than a Baseline MPI implementation [29].

The next numerical experiment tests the behavior of the GPU implementations by increasing the radius of neighbors from 1 to 5.

3.3.4 Extended neighborhood simulations

The memory access pattern of the GoL CA retrieves the state of the neighbors of a given cell within a radius of 1 (8 nearest neighbors, or Moore neighborhood [17]), plus the state of the cell itself, and this gives that nine values are needed per cell. In Fermi and post-Fermi architectures, the L1 cache can be configured to cache global memory access, this will keep the state of cells most recently used by a thread in the cache memory.

This implies that using shared memory for a small neighborhood radius could not provide good performance in these GPUs, a result which was observed in the previous numerical experiments of Sects. 3.3.1 and 3.3.2, and in previous studies by Gibson et al. [19] and Maruyama et al. [27]. These tests show how each GPU implementation behaves as the number of neighbors increases. The results can be seen in Fig. 7, they show a steady increase in execution time as the neighborhood radius is increased.

To carry out this numerical experiment, the device function *count_neighs()* was modified. A simple approach to increase the neighborhood radius is to use *for* loops to count the values stored in neighbor cells. This will take two *for* loops (columns and rows) and one counter variable, as seen in the following CUDA C source code (*RADIUS* is a constant defined at compile time):

```
int ii, jj, count=0;
#pragma unroll
for (ii=-RADIUS; ii <= RADIUS; ii++){
#pragma unroll
for (jj=-RADIUS; jj <= RADIUS; jj++){
count += lattice[my_id + ii * size + jj];
}
}
count -= lattice[my_id];
```

This code is small and simple to read, but leads to a considerable performance drop. Using compiler flags to explicitly use loop unrolling did not improve performance. The source code for a radius of 1 used in previous numerical experiments did not make use of *for* loops to count the state of the neighbors. The for loops in *count_neighs()* were manually unrolled for all radius values, from 1 to 5. The speedup of the manually *unrolled for loops* implementation against the *for loop* implementation is the following: radius 1, 2.2 x faster; radius 2, 2.5 x faster; radius 3, 4 and 5, 2.6 x faster. Further analysis of the *PTX assembler* code is needed to find the reason why the automatic loop unrolling does not lead to better performance. For a radius of 5, each cells needs to access 121 neighbors states. Using the *Shared v2* code for this case nearly matches the performance of the *Baseline* code.

These results confirm that there is no need to use shared memory even when increasing the amount of cell states each thread needs by more than one order of magnitude. Table 4 shows the speedups of the three tested GPUs against the CPU serial code for each neighborhood radius. The C2050, K20x

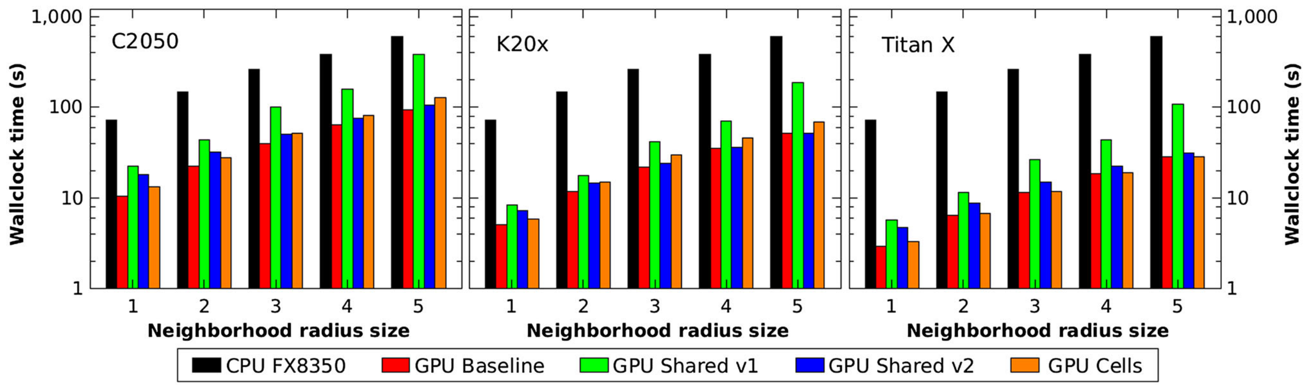


Fig. 7 GoL simulations for different GPU implementations with different neighborhood radius, executed in the C2050, K20x and Titan X GPUs, for a lattice size of 4096 × 4096 during 1000 steps (Color figure online)

Table 4 Speedups of the GPU baseline implementation against the CPU serial code, from execution times shown in Fig. 7, with N = 4096 and 1000 steps, executed in three GPUs: Tesla C2050, K20x, and Titan X

Radius	C2050	K20x	Titan X
1	6.8	14.2	24.3
2	6.6	12.4	23.1
3	6.5	11.8	22.8
4	5.9	10.7	20.3
5	6.3	11.5	21.1

and Titan X GPUs have an average speedup of ∼6 x; between ∼10 x and ∼14 x; and between ∼20 x and ∼24 x, respectively.

3.3.5 Computational intensive CA

The GoL CA is a memory bound problem, each cell needs to access nine values from memory and only eight sums are performed for a neighborhood radius of 1. Gibson et al. [20] concluded that the GoL is a simple CA which does not exploit the GPUs compute capabilities, and suggested that a more complex CA could achieve higher speedups than the GoL CA. To test the performance of the CA GPU implementations with a more compute intensive algorithm, a cosine operation of the sum of the neighbors of a cell was added to the rule set of the GoL. The cosine operation is one of the most costly operations that a GPU can perform (see section "5.4.1. Arithmetic Instructions" of the "CUDA C Programming Guide v7.0" document [13]).

Performance of the best GPU implementations for each of the NVIDIA GPUs can be seen in Fig. 8. From the numerical experiment in Sect. 3.3.3 it was seen that the two low end and oldest GPUs (the 9400GT and GT210 GPUs) could not match the performance of one core of the FX-8350 CPU executing the GoL CA. Here, these two GPUs do improve their performance and obtain a speedup of ∼2 x (in single precision, see Sect. 3.2 for more details) against the Serial CPU implementation running in the same FX-8350 CPU. The GTX750 achieves up to ∼33 x of speedup, the C2050 up to ∼75 x, the K20x up to ∼174 x and finally the Titan X up to ∼230 x, all against the serial CPU version running in the FX-8350 CPU. This compute intensive implementation executed in the two best GPUs compared with an MPI parallel execution in 8 cores of the AMD FX-8350 CPU gives a speedup ∼ 40 x for the Titan X GPU and ∼ 30 x for the K20x GPU (results not included in Fig. 8). Gibson et al. [20] assume that increasing the complexity of the CA gives a greater speedup than the one obtained with the GoL CA can be confirmed thanks to this numerical experiment. This is in part because the GPU platform implements the operation (sin/cos/tan) in the ISA while the CPU needs a table and a series computation.

Two new performance metrics are proposed to compare different GPUs architectures. The first one is seen in Fig. 9, for the obtained performance (Grid size/wallclock time) normalized with the amount of transistors present in each GPU. The performance of each GPU generation increases compared with the previous generations partly because of improvements in architecture, but also because of the increase in the number of transistors. For the case studied here, the performance obtained per transistor in the C2050 GPU is similar to the one obtained in the K20x GPU. In addition, the performance per transistor obtained for the GT210 GPU is slightly higher than the one obtained in the 9400GT, despite the fact that the speedups in Fig. 8 are very similar in these two GPUs. The increase in performance per transistor is due to the fact that the GT210 has less transistors than the 9400GT (260M for the GT210 compared with 314M for the 9400GT) and, therefore, improvements are due to architectural changes. As it was mentioned before, the GoL is a model in which the performance is bound by memory access. Because of this, in Fig. 10 the second performance metric can be seen, where the performance of each GPU is normalized

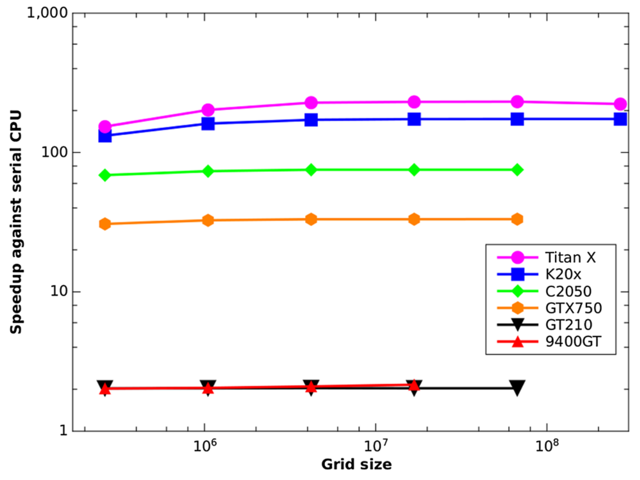


Fig. 8 Speedups of the Complex CA simulations for different lattice sizes, with the best implementation for each GPUs against an optimized Serial CPU implementation

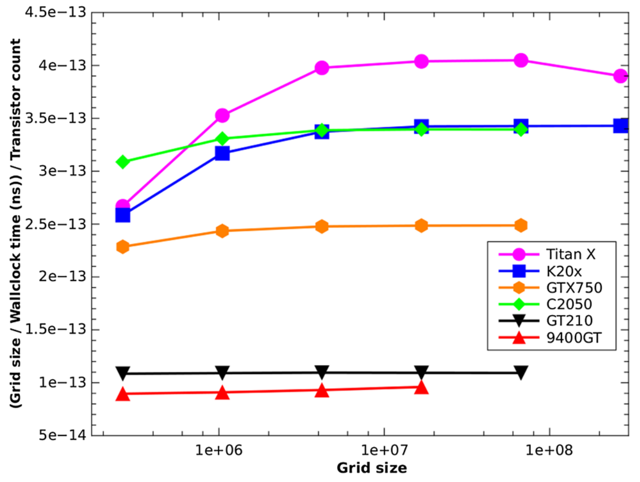


Fig. 9 Performance for the complex CA normalized with the transistors count present in each GPU

with respect to memory bandwidth. An interesting result can be seen when comparing the performance of the Titan X and K20x GPUs: the efficiency per GB of memory bandwidth is similar between these two GPUs. It can be concluded that, for this particular metric and for these type of problems, the efficiency of a K20x GPU is equivalent to a newer GPU like the Titan X, possibly because the memory width is the same, and because the performance in Fig. 8 is nearly the same.

3.3.6 Profiling the GPU baseline implementation

Using the NVIDIA profiler (by using *nvprof* from a Linux console or *nvvp* from a GUI) a more detailed view of the behavior of the implementations can be obtained. A brief analysis of the GPU Baseline implementation in the C2050 GPU is detailed here for a lattice size of 1024 × 1024 during 1000 steps with a block size of 256. Of the three global kernels (*moveKernel()*, *copy_Rows()* and *copy_Cols()*), *moveKernel()* takes ~98% of the compute time, leaving the remaining ~2% for the two kernels that copy the halo cells. A more in depth analysis could be made with the NVIDIA profiler to find where the GPU Baseline implementation can be improved.

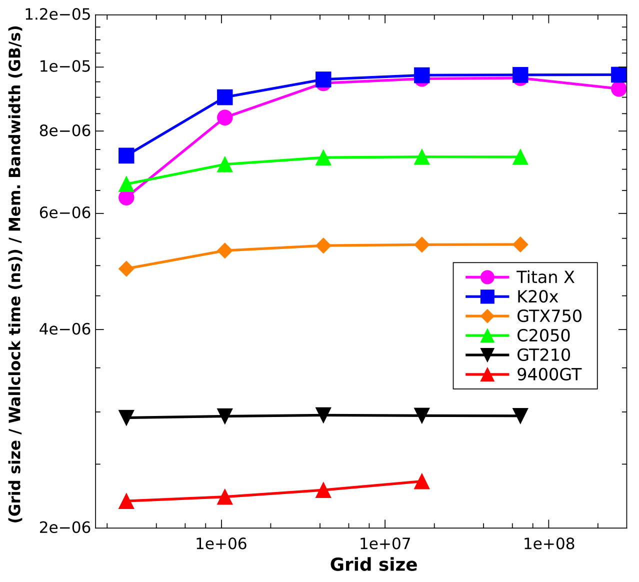


Fig. 10 Performance for the complex CA normalized with the memory bandwidth of each GPU

Table 5 Divergence as a function of number of steps, for the three conditional statements of the GoL rule set

Steps	Divergence (%)
10	(99, 98, 97)
10^2	(99, 99, 99)
10^3	(32, 43, 54)
10^4	(21, 46, 34)
10^5	(21, 34, 45)

There is a divergence in the code path which is inevitable when the GoL rule set has to be executed. The GoL rules can be expressed with three *if* conditional statements, in which a divergence occur. Table 5 shows divergence for these three conditional statements. As the CA reaches a steady state, divergences start to decrease, improving the CA performance. As Gibson et al. [20] stated, when running the CA for a considerable number of steps, GPU performance improves. It should be noted that divergence will vary for different initial conditions.

To quantify performance decrease due to this divergence, a test was carried out. The same simulation executed previously to build Table 5 was performed without checking GoL rules to remove the three divergence paths. Only a 20% time improvement was seen, which indicates that the divergence is not the main limiting performance factor.

Another approach to carry out performance analysis is to use the Roofline model [61]. This model provides insights on performance factors affecting CUDA kernels, and would be implemented in future studies.

3.4 Summary of results

In this section, a summary of the most important numerical results is presented:

1. Serial CPU code compared with GPU execution:
   - The Titan X GPU has a speedup of up to ∼85 x against a single AMD FX-8350 CPU core.
   - A low end and old GPU such as the NVIDIA GeForce 9400 or the GT210 cannot even match the performance of a single core of the same CPU.
2. GPU execution compared with parallel CPU with MPI execution:
   - The Titan X GPU performed similarly to 64 CPU cores AMD Opteron 6272.
   - The K20x GPU performed similarly to 24 CPU cores of the same AMD processor.
   - The Tesla C2050 GPU can be compared with 8 CPU cores AMD FX-8350.
3. The complex CA is compared between GPU and serial CPU execution:
   - The Titan X reaches up to ∼230 x of speedup against a single AMD FX-8350 CPU core.
   - The GeForce 9400 and the GT210 GPUs obtained ∼2 x of speedup against the same CPU.

A linear scaling with lattice size (where lattice size is N^2) was observed in all GPU implementations. The use of shared memory in GPUs from Fermi and onwards it is not necessary, at least for a simple access memory pattern like the one GoL has, even if the neighborhood radius is increased up to 5 (121 neighbors per cell).

One possible measure of performance for a CA could be given by the number of evaluated cells per second. For the Titan X, and a size of ∼1 billion cells during 1000 steps, executing five different simulations resulted in an average of 5.58e6 cells/s and in the FX-8350 GPU the performance drops to 0.22e6 cells/s. Another possible measure could be given by the wall clock time required for a given run, normalized to the number of cells and number of steps in that run. This is a general measure for molecular dynamics particle codes, where number of cells is replaced by the number of particles, and typical values fall around 1–500 microsec/step/particle [24]. In this work, the CA simulations obtain 0.17 nanosec/step/cell for a simulation with ∼1 billion cells and 1000 steps, running in the Titan X card, while the same simulation in the FX-8350 obtains 4.4 nanosec/step/-cell.

As a summary, the code presented here can be used to efficiently simulate more than a billion cells. This code presents excellent scaling with system size, and can reach speed-ups of up to more than two orders of magnitude when compared to an optimized CPU code. In addition, it can be easily adapted to follow various CA rules. Given all of this, together with the open-source nature of the code, the tools presented and analyzed in this paper can be useful to the scientific community using CA.

4 Conclusions and future work

This work presents several Cellular Automata (CA) GPU implementations with high performance in five different NVIDIA GPU architectures. The work focuses on the GoL CA, but also tests a more complex case. For the cases studied here, with a relatively simple memory access pattern, the use of shared memory in GPUs with architectures Fermi and newer does not provide any performance improvements. This is in agreement with results obtained by Gibson et al. [20] and Topa et al. [54] but only for the Fermi architecture. A comparison is made of two different shared memory implementations and a Multicell implementation with respect to a baseline GPU code. In the oldest architecture studied, the NVIDIA Tesla G96, shared memory implementations perform better than the baseline code with global memory access. On the other hand, in Fermi and post-Fermi architectures, the baseline code outperforms the rest of the implementations due to the automatic cache of global memory access in a L1 cache. Therefore, when access to global memory follows a simple pattern, as in the Game of Life CA, it is not necessary to use shared memory to improve performance. Multicell implementation does not improve performance either when there is a L1 cache in the GPU. Two new performance metrics, normalizing execution and system size with transistor count and with memory bandwidth, are also presented, clearly displaying architectural advances.

There might be additional optimizations which might improve performance at the cost of making this CA code significantly more complex. Maruyama et al. [27] concluded that some optimizations, like shared memory with blocking, might lead to significant coding time increase for a single kernel. The latest GPU architectures have improved simplifying memory access patterns, making source code easy to write and read, and decreasing programming time. The baseline code presented here is an example of this, being able to simulate extremely large systems and outperforming classical optimizations. This implementation would thus be useful for applications requiring efficient CAs, in areas as varied as chemistry and biology.

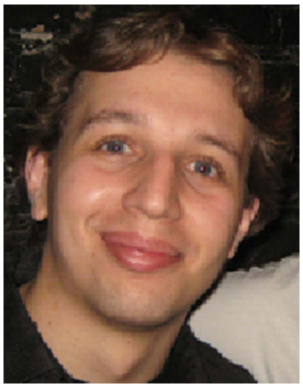
Future work will explore hybrid multi-GPU/multi-node [25] implementations of CA able to use all heterogeneous computing resources available, including CPU cores and GPUs, which communicate via MPI [1,30]. In addition, implementing the Roofline model [61] will help to improve analysis to identify possible optimization pathways to the CPU/GPU and hybrid implementations.

Acknowledgements The authors acknowledge support from CONICET, ANPCyT Grant PICT-2014-0696, and a SeCTyP UNCuyo Grant. This work used the clusters Mendieta from CCAD-UNC and ICB-ITIC from UNCuyo, which are part of the SNCAD network.

References

1. Aaby, B.G., Perumalla, K.S., Seal, S.K.: Efficient simulation of agent-based models on multi-GPU and multi-core clusters. In: Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques p. 29:1 (2010). doi:10.4108/icst.simutools2010.8822
2. Balasalle, J., Lopez, M.A., Rutherford, M.J.: Optimizing memory access patterns for cellular. In: Hwu, W. (ed.) GPU Computing Gems Jade Edition, pp. 67–75. Morgan Kaufmann, Amsterdam (2011)
3. Bauer, M., Cook, H., Khailany, B.: Cudadma. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on—SC 11 p. 12 (2011). doi:10.1145/2063384.2063400
4. Blecic, I., Cecchini, A., Trunfio, G.A.: Fast and accurate optimization of a GPU-accelerated ca urban model through cooperative coevolutionary particle swarms. Proc. Comput. Sci. **29**, 1631–1643 (2014). doi:10.1016/j.procs.2014.05.148
5. Brodtkorb, A.R., Dyken, C., Hagen, T.R., Hjelmervik, J.M., Storaasli, O.O.: State-of-the-art in heterogeneous computing. Sci. Program. **18**(1), 1–33 (2010)
6. Brown, W.M., Wang, P., Plimpton, S.J., Tharrington, A.N.: Implementing molecular dynamics on hybrid high performance computers—short range forces. Comput. Phys. Commun. **182**(4), 898–911 (2011). doi:10.1016/j.cpc.2010.12.021
7. Browne, S., Dongarra, J., Garner, N., London, K., Mucci, P.: A scalable cross-platform infrastructure for application performance tuning using hardware counters. In: ACM/IEEE 2000 Conference on Supercomputing, p. 42. IEEE (2000)
8. Campos, R.S., Lobosco, M., dos Santos, R.W.: A GPU-based heart simulator with mass-spring systems and cellular automaton. J Supercomput **69**(1), 1–8 (2014). doi:10.1007/s11227-014-1199-5
9. Carozzani, T., Gandin, C.A., Digonnet, H.: Optimized parallel computing for cellular automaton finite element modeling of solidification grain structures. Modelling Simul. Mater. Sci. Eng. **22**(1), 015,012 (2013). doi:10.1088/0965-0393/22/1/015012
10. Caux, J., Hill David, R., Siregar, P.: Accelerating 3D Cellular automata computation with GP-GPU in the context of integrative biology. In: Cellular Automata—Innovative Modelling for Science and Engineering, pp. 411–426. InTech (2011). https://hal.archives-ouvertes.fr/hal-00679045
11. Chen, S., Doolen, G.D.: Lattice Boltzmann method for fluid flows. Ann. Rev. Fluid Mech. **30**(1), 329–364 (1998). doi:10.1146/annurev.fluid.30.1.329
12. CUDA C Programming Guide, vol. 4.2. NVIDIA (2012). http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
13. CUDA C Programming Guide, vol. 7.0. NVIDIA (2015). http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
14. CUDA from NVIDIA. http://www.nvidia.com/cuda
15. Feichtinger, C., Habich, J., Kstler, H., Hager, G., Rde, U., Wellein, G.: A flexible patch-based lattice Boltzmann parallelization approach for heterogeneous GPU-CPU clusters. Parallal Comput. **37**(9), 536–549 (2011). doi:10.1016/j.parco.2011.03.005
16. Ferrero, E.E., De Francesco, J.P., Wolovick, N., Cannas, S.A.: q-state potts model metastability study using optimized GPU-based Monte Carlo algorithms. Comput. Phys. Commun. **183**(8), 1578–1587 (2012). doi:10.1016/j.cpc.2012.02.026
17. Ganguly, N., Sikdar, B.K., Deutsch, A., Canright, G., Chaudhuri, P.P.: A survey on cellular automata. Center for High Performance Computing, Dresden University of Technology (2003). http://citeseerx.ist.psu.edu/viewdoc/summary?, doi:10.1.1.107.7729
18. Gardner, M.: Mathematical games: the fantastic combinations of John Conway new solitaire game life. Sci. Am. **223**(4), 120–123 (1970)
19. Gibson, M.J., Keedwell, E.C., Savi, D.: Understanding the efficient parallelisation of cellular automata on CPU and GPGPU hardware. In: Proceeding of the Fifteenth Annual Conference Companion on Genetic and Evolutionary Computation Conference Companion—GECCO 13 Companion pp. 171–172 (2013). doi:10.1145/2464576.2464660
20. Gibson, M.J., Keedwell, E.C., Savi, D.A.: An investigation of the efficient implementation of cellular automata on multi-core CPU and GPU hardware. J. Parallel Distrib. Comput. **77**, 11–25 (2014). doi:10.1016/j.jpdc.2014.10.011
21. Hawick, K.A., Johnson, M.G.: Bit-packed damaged lattice potts model simulations with cuda and gpus. In: Proceedings of International Conferences on Modelling, Simulation and Identification, pp. 371–378 (2011)
22. Hong, S., Kim, H.: An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. SIGARCH Comput. Archit. News **37**(3), 152 (2009). doi:10.1145/1555815.1555775
23. Kjolstad, F.B., Snir, M.: Ghost cell pattern. In: Proceedings of the 2010 Workshop on Parallel Programming Patterns, p. 4. ACM, New York (2010)
24. LAMMPS: Lennard Jones Liquid Benchmark. http://lammps.sandia.gov/bench.html#lj
25. Lee, C., Ro, W.W., Gaudiot, J.L.: Boosting CUDA applications with CPU-GPU hybrid computing. Int. J. Parallel Program. **42**(2), 384–404 (2013). doi:10.1007/s10766-013-0252-y
26. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: Nvidia tesla: a unified graphics and computing architecture. IEEE Micro **28**(2), 39–55 (2008)
27. Maruyama, N., Aoki, T.: Optimizing stencil computations for NVIDIA Kepler GPUs. In: Größlinger, A., Köstler, H. (eds.) Proceedings of the 1st International Workshop on High-Performance Stencil Computations, pp. 89–95. Austria, Vienna (2014)
28. Meng, J., Skadron, K.: Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In: Proceedings of the 23rd International Conference on Conference on Supercomputing—ICS 09 pp. 256–265 (2009). doi:10.1145/1542275.1542313
29. Millán, E.N., Bederian, C., Piccoli, M.F., García Garino, C., Bringa, E.M.: Performance analysis of cellular automata HPC implementations. Comput. Electr. Eng. **48**, 12–24 (2015). doi:10.1016/j.compeleceng.2015.09.015
30. Millán, E.N., Martínez, P.C., Gil Costa, G.V., Piccoli, M.F., Printista, A.M., Bederian, C., García Garino, C., Bringa, E.M.: Parallel implementation of a cellular automata in a hybrid CPU/GPU environment. In: A. De Giusti (ed.) XVIII Congreso Argentino de

Ciencias de la Computación, pp. 184–193. Red de Universidades con Carreras en Informática RedUNCI (2013). ISBN 978-987-23963-1-2
31. Moore, N.: Kernel specialization for improved adaptability and performance on graphics processing units (gpus). Ph.D. thesis, Northeastern University Boston, MA (2012)
32. North, M.J., Collier, N.T., Ozik, J., Tatara, E.R., Macal, C.M., Bragen, M., Sydelko, P.: Complex adaptive systems modeling with repast simphony. Complex Adapt. Syst. Model. **1**(1), 3 (2013). doi:10.1186/2194-3206-1-3
33. NVIDIA: Whitepaper NVIDIA GeForce GTX 750 Ti, v1.1
34. NVIDIA: Whitepaper NVIDIA GeForce GTX 980, v1.1
35. NVIDIA: Whitepaper NVIDIAs Next Generation CUDA Compute Architecture: Fermi, v1.1
36. NVIDIA: Whitepaper NVIDIAs Next Generation CUDA Compute Architecture: Kepler GK110, v1.0
37. NVIDIA: Nvidia geforce 8800 gpu architecture overview. Technical brief, November 2006 (2006)
38. NVIDIA: Tuning Cuda Applications for Kepler, v7.0 (2015)
39. NVIDIA: Tuning Cuda Applications for Maxwell, v7.0 (2015)
40. NVIDIA: Nvidia geforce gtx 200 gpu architectural overview. Technical brief, May (2008)
41. Oxman, G., Weiss, S., Be'ery, Y.: Computational methods for conway's game of life cellular automaton. J. Comput. Sci. **5**(1), 24–31 (2014). doi:10.1016/j.jocs.2013.07.005
42. Papadopoulou, M.M., Sadooghi-Alvandi, M., Wong, H.: Microbenchmarking the GT200 GPU. Computer Group, ECE, University of Toronto, Technical Report (2009)
43. Perumalla, K.S., Aaby, B.G.: Data parallel execution challenges and runtime performance of agent simulations on gpus. In: Proceedings of the 2008 Spring Simulation Multiconference, SpringSim'08, pp. 116–123. Society for Computer Simulation International, San Diego, CA, USA (2008). http://dl.acm.org/citation.cfm?id=1400549.1400564
44. Pohl, T., Deserno, F., Thurey, N., Rude, U., Lammers, P., Wellein, G., Zeiser, T.: Performance evaluation of parallel large-scale lattice boltzmann applications on three supercomputing architectures. In: Proceedings of the ACM/IEEE SC2004 Conference p. 21 (2004). doi:10.1109/sc.2004.37
45. Preis, T., Virnau, P., Paul, W., Schneider, J.J.: GPU accelerated Monte Carlo simulation of the 2D and 3D ising model. J. Comput. Phys. **228**(12), 4468–4477 (2009). doi:10.1016/j.jcp.2009.03.018
46. RanjanNayak, D., Kumar Sahu, S., Mohammed, J.: A cellular automata based optimal edge detection technique using twenty-five neighborhood model. IJCA **84**(10), 27–33 (2013). doi:10.5120/14614-2869
47. Rapaport, D.: Enhanced molecular dynamics performance with a programmable graphics processor. Comput. Phys. Commun. **182**(4), 926–934 (2011). doi:10.1016/j.cpc.2010.12.029
48. Rauch, L., Madej, L., Spytkowski, P., Golab, R.: Development of the cellular automata framework dedicated for metallic materials microstructure evolution models. Arch. Civil Mech. Eng. **15**(1), 48–61 (2015). doi:10.1016/j.acme.2014.06.006
49. Russo, L., Russo, P., Vakalis, D., Siettos, C.: Detecting weak points of wildland fire spread: a cellular automata model risk assessment simulation approach. Chem. Eng. **36**, 253–258 (2014)
50. Rybacki, S., Himmelspach, J., Uhrmacher, A.M.: Experiments with single core, multi-core, and GPU based computation of cellular automata. In: First International Conference on Advances in System Simulation, 2009. SIMUL'09, pp. 62–67. IEEE (2009)
51. Ryoo, S., Rodrigues, C.I., Stone, S.S., Baghsorkhi, S.S., Ueng, S.Z., Stratton, J.A., Hwu, W.m.W.: Program optimization space pruning for a multithreaded gpu. In: Proceedings of the Sixth Annual IEEE/ACM International Symposium on Code Generation and Optimization—CGO 08 (2008). doi:10.1145/1356058.1356084
52. Shimokawabe, T., Aoki, T., Takaki, T., Yamanaka, A., Nukada, A., Endo, T., Maruyama, N., Matsuoka, S.: Peta-scale phase-field simulation for dendritic solidification on the tsubame 2.0 supercomputer. In: 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp. 1–11 (2011)
53. Smoller, J.: Shock waves and reaction-diffusion equations. In: Research Supported by the US Air Force and National Science Foundation, vol. 258. Springer, New York(Grundlehren der Mathematischen Wissenschaften, vol. 258), p. 600 (1983)
54. Topa, P.: Cellular automata model tuned for efficient computation on GPU with global memory cache. In: 2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pp. 380–383 (2014). doi:10.1109/pdp.2014.97
55. Topa, P., Młocek, P.: Using shared memory as a cache in cellular automata water flow simulations on gpus. Comput. Sci. **14**, 3 (2013)
56. Top 500 supercomputers, list of june 2016. http://www.top500.org/lists/2016/06/
57. Veerbeek, W., Pathirana, A., Ashley, R., Zevenbergen, C.: Enhancing the calibration of an urban growth model using a memetic algorithm. Comput. Environ. Urban Syst. **50**, 53–65 (2015). doi:10.1016/j.compenvurbsys.2014.11.003
58. Volkov, V.: Better performance at lower occupancy. In: Proceedings of the GPU Technology Conference, GTC, vol. 10 (2010)
59. Volkov, V., Demmel, J.: Benchmarking GPUs to tune dense linear algebra. 2008 SC—International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–11 (2008). doi:10.1109/sc.2008.5214359
60. Wilensky, U.: Netlogo. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL (1999). http://ccl.northwestern.edu/netlogo/
61. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Commun. ACM **52**(4), 65 (2009). doi:10.1145/1498765.1498785
62. Zhao, Y.: GPU accelerated computation and real-time rendering of cellular automata model for spatial simulation. J. Inform. Comput. Sci. **11**(12), 4453–4465 (2014). doi:10.12733/jics20104445



Emmanuel N. Millán received his Ph.D. degree from Universidad Nacional de San Luis (UNSL), Argentina in 2016, and a BSc. in Software Engineering from Universidad del Aconcagua, Argentina, in 2010. He is a researcher within CONICET. He is interested in the implementation of parallel problems in hybrid clusters including Graphics Processing Units (GPUs), with applications in Molecular Dynamics, Cellular Automata and Monte Carlo methods.

Nicolás Wolovick Ph.D. is leader of the GPGPU Computing Group at Universidad Nacional de Córdoba, he is also part of the University HPC Center (CCAD-UNC). He teaches CS curricula at his home university since 1996 in the areas of Computer Architecture, Parallelism and HPC.

María Fabiana Piccoli received her Ph.D. degree from Universidad Nacional de San Luis (UNSL), Argentina, in 2005, and the Computer Science degree from UNSL, Argentina, in 1995. She is a full Professor at the UNSL, and director of Departamento de Informática. She is interested in High Performance Computing, including Parallel and Distributed Computing.

Carlos García Garino received his Ph.D. degree from Universidad Politécnica de Cataluña, España, in 1993, and the Civil Engineering degree from UBA, Argentina, in 1978. He is a full Professor at the UNCuyo, and director of the ITIC-UNCuyo. He is interested in Computer Networks, Distributed Computing and Computational Mechanics.

Eduardo M. Bringa received his Ph.D. in Physics from the University of Virginia (USA) in 2000. He was staff member at Lawrence Livermore National Laboratory (LLNL, USA), and currently holds a Principal Researcher position within CONICET, and also a Full Professor position at Facultad de Ciencias Exactas y Naturales (FCEN, Universidad Nacional de Cuyo). He leads the group on Simulations in Materials, Astrophysics and Physics (SIMAF, https://sites.google.com/site/simafweb/).