Cluster Comput
DOI 10.1007/s10586-015-0501-5

Monte Carlo simulations of settlement dynamics in GPUs

Emmanuel N. Millán[1] · Silvana B. Goirán[2] · María Fabiana Piccoli[3] · Carlos García Garino[4] · Julieta N. Aranibar[2] · Eduardo M. Bringa[5]

Received: 20 February 2015 / Revised: 7 October 2015 / Accepted: 9 October 2015
© Springer Science+Business Media New York 2015

Abstract Recently, a Monte Carlo model was proposed in order to simulate settlement dynamics in drylands, including several environmental factors, and it was implemented as a serial CPU code. In this work we present a parallel implementation of that code using graphics processing units (GPU) and NVIDIA CUDA. The code was tested with two experiments, a *Baseline case* and a *Realistic case*. We take advantage of the GPU architecture to obtain significant speedups: ∼8× to ∼20× with the *Baseline case* in a NVIDIA Tesla C2050 versus a Phenom 1055T CPU. The *Realistic case* obtained ∼80× of speedup in the same hardware. The GPU performance of the code will allow the inclusion of additional factors affecting settlements and large grid sizes for detailed environmental degradation models.

Keywords Monte Carlo · Parallel processing · General purpose graphics processing units · Settlement dynamics

✉ Emmanuel N. Millán
emmanueln@gmail.com

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

[2] Instituto Argentino de Nivologia, Glaciologia y Ciencias Ambientales (IANIGLA), CONICET, CCT-Mendoza, Argentina and Facultad de Ciencias Exactas y Naturales, Universidad Nacional de Cuyo, Mendoza, Argentina

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

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

[5] CONICET and Facultad de Ciencias Exactas y Naturales, Universidad Nacional de Cuyo, Mendoza, Argentina

1 Introduction

Land use in arid and semiarid areas is conditioned by water limitations and vegetation characteristics. If water is not available for the development of irrigated agriculture, livestock production is one of the main economic activities in drylands, which sustains a third of the world population and 78 % of livestock worldwide [1,6]. This activity is experiencing changes related to climatic change and human activities. Population and economic growth, urbanization, and consumption patterns are shaping livestock production, with impacts on societies and environments, such as greenhouse gas emissions, nutrient cycles, land demands and degradation, and protein supply [11]. The challenge in order to produce sustainably depends on how we understand and managed the livestock sector and natural resources. In the Monte desert (Argentina), groundwater coupled ecosystems are used for subsistence livestock production developed in scattered settlements, which allows the coexistence of areas with high cover vegetation and rural communities [8]. Nonetheless, water access and improved infrastructure may increase population density and grazing pressures, with a higher risk of ecosystem degradation. An understanding of the feedbacks between natural resources and livestock settlements at present is crucial to estimate futures effects of environmental and socio-economic changes in the region. Different factors may affect the establishment of livestock settlements in arid environments. The hypothesis in which we based the modeling approach is that the availability of resources such as water, forest resources and access routes drive the establishment of settlements. We developed a Monte Carlo (MC) simulation to model the settlement dynamics in drylands (SeDD). This model includes six environmental drivers of settlements: surface and groundwater availability, vegetation type, existing settlements, access route, and

Published online: 28 October 2015

old river beds [14]. The model places settlements in areas where the environmental factors are suitable for these to thrive. In order to take into account social factors, our model includes a stochastic factor, a percentage of the settlements are placed at random, even in unfavorable regions. The model gradually reduces the suitability around established settlements, simulating vegetation degradation by the activity of the settlements. Due to the stochastic nature of the code, hundreds of simulations are needed in order to compute the required average behavior. In addition, because the probabilities associated with certain environmental factors are unknown, a parameter sweep is needed in order to find values that match field results. For instance, in Millán et al. [14], a parameter sweep was performed with a total of ∼11 million independent simulations (∼1.8 s each simulation), resulting in 12 days of execution time in a mini-cluster with three AMD FX-8350 microprocessors (84 GHz cores), where each CPU core executed one set of input parameters independently of the other CPU cores. This computational cost precludes the inclusion of additional environmental factors and increased resolution of the input maps (increasing the size of the simulated lattice), which would lead to more accurate results.

One possible way to obtain a faster performance for the MC SeDD code would be to port the code for parallel execution. Initially we perform some tests using OpenMP which is fast to implement. It resulted in a low efficiency as the number of CPU cores was increased: running in 6 CPU cores (AMD Phenom 1055T) would result in a 67 % efficiency, and running in 16 CPU cores (AMD Opteron 6272) would give an efficiency of 15.4 % for a lattice size of 2048 × 2048 cells. Other alternatives were to port from a serial environment to a distributed memory cluster using MPI, or to port to a hybrid CPU–GPU system. We decided to use GPUs only since other MC codes have been already implemented in GPUs obtaining excellent speed-ups. For instance, a 2D Ising MC model was developed with Multi-GPU acceleration by Block et al. [3], with 35× acceleration versus an optimized serial CPU implementation. As another example of MC in a 2D lattice, the q-state Potts model developed by Ferrero et al. [7] obtained 155× of speedup versus an optimized serial CPU code.

This work is organized as follows. In Sect. 2 we give a description of SeDD model and code, with details of the most time consuming functions. Next, we introduce the GPU implementation. In Sect. 3 we describe the hardware infrastructure in which the simulations are executed, and discuss the results obtained for two types of experiments, the Baseline case and the Realistic case. Finally, in Sect. 4 the conclusions are reported.

2 Material and methods

A complete description of the SeDD model and code structure is detailed in Millán et al. [14], and the source code is available at http://goo.gl/r0yzfJ. In the next subsection we give a short description of the model and source code.

2.1 Settlement dynamics model

The settlement dynamics in drylands (SeDD) model was developed to test which environmental factors drive the decision to establish new livestock settlements. Simulations of Settlement Dynamics have been implemented using Agents [13] and Multi-Agents [4] models in the past. Our SeDD model is based on the Monte Carlo model [2,5], which considers a 2D lattice with $N \times N$ cells. The suitability of a given cell in a 2D lattice to be settled is modeled based on the partial probabilities associated to each of the environmental factors. We assume that the inhabitants of the simulated area have knowledge of the environmental factors and where would be convenient to establish a new settlement. The model chooses in each time step the cell with the highest probability in the lattice, and if it is greater than a defined probability (given as an input parameter) the settlement is placed in the lattice. The model can also establish settlements at random. This stochastic component is included to account for social factors that may influence the placement of settlements and that are not included in the environmental factors in the model.

The simulation is composed of 4 stages: *Input, Setup, Evolve* and *Output*. The behavior of the model is represented by the following steps:

1. *Input stage* read the input parameters and map files for each environmental factor.
2. *Setup stage* for every cell in the lattice, calculate the minimum distances between that cell and roads, rivers and closest settlements.
3. *Evolve stage*
   Start evolution while $Step < TotalSteps$.
   (a) Decrease vegetation around all settlements.
   (b) Calculate the total probability for each cell in the lattice.
   (c) Select highest probability (*Pmax*), if more than one cell in the lattice has the same highest probability, choose one of them randomly.
   (d) If $Pmax > Pset$ (*Pset* being a predefined probability threshold), establish the settlement, otherwise,

choose a random cell in the lattice and a random probability (*Pran*), compare with a predefined random probability threshold (*Pthresh*) and establish settlement if $Pran > Pthresh$.

(e) If a new settlement was established, it is necessary to recalculate the distance to the closest settlement for each cell.

(f) *Output stage* write state of the simulation to output file.

(g) Increase *Step* and repeat steps from (a) to (g).

4. *Output stage* if $Step = TotalSteps$, write output maps to disk and finish.

In Eq. (1) it can be seen how the total probability of each cell in the lattice as stated in *Evolve Stage* (b) is computed, with k being any given cell of the $N \times N$ lattice:

$$PAll[k] = PVeg[k] * PRoad[k] * PSettl[k] * PWater[k] * PRiver[k] * Ppaleo[k] \quad (1)$$

In the next subsections we explain the implementation of the model for serial CPU execution and a parallel implementation for GPU. Both implementations follow the stages explained here but execute certain functions in serial in the CPU or in parallel in the GPU.

2.2 Serial implementation

The serial implementation follows the steps described in the previous subsection. The simulation is performed with a lattice size of $N \times N$ during t time steps. In the *Input* stage the following tasks are included: each environmental factor is read from a file stored in the disk and placed in RAM memory in an array of $N \times N$ size, and the input parameters needed to execute the simulation are read from the command line or from an input file. Each cell in the lattice could have a settlement, some vegetation, or belong to a river, road, etc.

Next, the simulation performs the *Setup* stage, two important functions are included in this stage: *calculate_min_distances()* and *set_probabilities()*. The first one calculates for each cell in the lattice the minimum distances to the closest road cell, river cell and settlement, storing it in an array for each of the three factors. Then, a probability is assigned to each cell in the lattice (*set_probabilities()* function). This probability is computed with the minimum distance to the environmental factors and a predefined weight (given as input) for different ranges of distances between the cell and roads, rivers, settlements, old river beds and groundwater depth.

The evolution of the system (*Evolve* stage) starts by decreasing the vegetation around each settlement up to a defined distance (*reduce_vegetation()* function). Then the function *calculate_PAll_BMC()* computes the total probability for each cell in the lattice as a result of multiplying the probabilities of all the environmental factors. The *select_kmax_BMC()* function searches for the highest probability (variable *Pmax*) in the lattice, if more than one cell has the same highest probability, one of them is selected randomly. The chosen probability (*Pmax*) is then compared with an input parameter (*Pset*), which functions as a threshold in order to limit the number of settlements placed by high probability (function *put_settlement_BMC()*). If $Pmax > Pset$ then the settlement is placed in the lattice. If a settlement was not placed by the high probability option ($Pmax > Pset$), a random cell is selected from the lattice, then a random probability (variable $Pran$) is generated and compared with a predefined threshold probability, $Pthresh$, which was given as an input parameter. If $Pran > Pthresh$, then a settlement is placed in that selected cell. As a result, $Pset$ controls settlement placing based on environmental variables, while $Pthresh$ gives random settlement placing possibly due to other variables not taken into account in this model, like certain social factors. Modifying $Pset$ and $Pthresh$ will result in significant changes in the number and distribution of settlements.

When a new settlement is established and before a new time step is started, a recalculation of distances and probabilities between cells and settlements is performed by the *recalc_min_distances()* function in order to increase the probabilities around newly installed settlements. Before the time step is finished, a part of the *Output* stage is performed, every n steps (n defined by an input parameter) an output function saves to disk the state of the system. Then, if the time step is less than the *Total Steps* variable, the *Evolve* stage is started again. Finally, the *Output* stage is executed, the output files that are written to disk include: the state of the vegetation and the settlements placed in the lattice.

We profiled the *Baseline case* (details of this experimental case are given in Sect. 3.3) with a lattice of size $N \times N$ with $N = 4096$ during 1000 steps with the *gprof* utility [9]. Three functions account for ∼96 % of the total execution time. The calculation of the total probability for each lattice cell takes 44 % of the execution time (function *calculate_PAll_BMC()* from the *Evolve* stage). The calculation from the setup stage of minimum distances between each lattice cell and the road and settlements takes ∼39 % (function *calculate_min_distances()*). The third function that recalculates the distance to the closest settlement for each cell of the lattice, which is executed each time a new settlement is established in the *Evolve* stage, takes ∼13 % of the total run time (function *recalc_pdist_settlements()*). These three functions are good candidates to port to the GPU because of their parallel nature, the calculations needed to compute the state for each cell at each time step are independent of the calculation of their neighbors.

We do not need double precision to store the probabilities of the environmental factors. We use single precision variables (*float* data type), with the benefit of saving RAM and Global memory and reducing CPU and GPU compute time. In the next section we discuss the parallel implementation of the SeDD code in GPUs.

2.3 Parallel implementation using GPUs

With the profiler results obtained in Sect. 2.2, we identified three functions that consume most of the simulation time. The calculations in each of the cells of the lattice for the functions *calculate_PAll_BMC()* and *recalc_pdist_settlements()*, both from the *Evolve* stage, and *calculate_min_distances()* from the *Setup* stage, are independent from calculations of their neighbors and can be mapped to the thread model provided by GPUs [15]. In order to minimize the amount of data to be copied to/from the Host memory from/to the GPU Global memory, we ported the entire simulation to the GPU, even the functions that were not time consuming in the CPU. Here we discuss the most relevant functions, and also the CUDA kernels that use the reduction technique [10] to avoid serialization of operations in the GPU.

The function *calculate_PAll_BMC()* in the CPU serial implementation sequentially multiplies the probability of each environmental factor for each of the cells in the lattice, and stores the result in an array. The computed total probability for each cell is independent of their neighbors, given that the information of distance to nearby settlements was already acquired by each individual thread. Therefore, each thread executed in the GPU can compute a lattice cell probability independently from other threads, without any inter thread communication. This function is implemented in the GPU with a single kernel called *cuda_kernel_calculate_PAll_BMC()*. Each thread reads from Global Memory the environmental factors, multiplies their probabilities and stores back in Global Memory the result. The use of Shared memory would not increase performance due to the fact that each value is read (from Global memory) and used only once. One of the advantages of using Shared memory is low latency compared with the latency of global memory, although Fermi and post-Fermi architectures cache accesses to global memory in L1 and L2 caches. Another advantage of Shared memory is coalesced access, although in GPUs with CC 2.0 or higher the impact on throughput with uncoalesced access to global memory is reduced due to the existence of caches [16].

The second function, *recalc_pdist_settlements()*, is called after a new settlement is placed in the lattice. This function updates, if needed, the minimum distance and probability from each cell to the new established settlement. The ported code to the GPU is composed of two CUDA kernels: *cuda_kernel_recalc_min_distances()* and *cuda_kernel_recalc_pdist_settlements()*. In the first kernel, each thread computes the distance to the newly placed settlement, if it is smaller than the stored distance (calculated from the setup stage or a call to this kernel from a previous step), is updated in Global memory. The second kernel, *cuda_kernel_recalc_pdist_settlements()*, updates the settlements probabilities of each cell according to the updated minimum distances. As in the kernel *cuda_kernel_calculate_PAll_BMC()*, there is no need to use Shared memory for these two kernels, each value read from Global memory is not reused or shared between the threads.

We ported to one CUDA kernel the function *calculate_min_distances()*, from the *Setup* stage. Each thread calculates and stores in Global memory the minimum distance to the closest road cell, settlement, river cell and old river bed cell.

We use the reduction technique with shared memory to search for the highest probability in the lattice [10] in the CUDA kernel *cuda_kernel_reduce_PAll_BMC()* and an atomic operation (*atomicMaxf()*) to reduce to a single value of probability. This kernel creates a shared memory array and each thread copies the total probability *PAll* corresponding to the associated cell of each thread, previously calculated in the *cuda_kernel_calculate_PAll_BMC()*. Next, it does a reduction finding the highest probability in the block and stores that probability in an array in Global memory. This process is repeated until a single value is obtained (variable *Pmax*).

If more than one cell has the same highest probability (*Pmax*), one of the cells is selected randomly. This is achieved by CUDA kernel *cuda_kernel_count_kmax_BMC()* using the *atomicAdd* operation to count the number of cells with the same highest probability. Then the *cuda_kernel_build_list_kmax_BMC()* is in charge of creating an array with the coordinates of all cells in the lattice with the same *Pmax*, and one of those coordinates is selected randomly with a random number generated in the CPU. There is no need to generate this number in the GPU, since this is at most a single random number for the whole lattice per step. We use the standard *rand()* function included in the *glibc* library with the current time (in nanoseconds) as a *seed* for each execution of the model.

Other works have tested GPU execution of different codes with various thread block sizes [12,18] and have found performance differences between them. For this reason, our code supports different block sizes for the thread block, and we tested the following dimensions: 64, 128, 192, 256, 512 and 1024 threads per block. In the Sect. 3.3 we discuss the obtained results.

We note that as an alternative to the implementation presented here, the use of libraries such as Thrust (http://thrust.github.io/) can help to decrease programming time and can offer a high performance solution to common parallel problems. Due to how the serial code implementation was coded,

it was required additional work to port the data structures to make them compatible with Thrust, therefore was decided to maintain the original code and to only write the necessary CUDA kernels to perform certain tasks in parallel.

3 Results and discussion

We executed two numerical experiments, the *Baseline case* (in Sect. 3.3) and the *Realistic case* (in Sect. 3.4). All the results shown in this section are an average of 10 simulations. In this section we first describe the numerical experiments, the hardware and software infrastructure, and then execute the experiments and discuss the obtained results.

3.1 Numerical experiments

We developed two types of numerical experiments in order to test performance: a *Baseline case* and a *Realistic case*. The *Baseline case* is used to test the correct functioning of the code and to be able to execute different lattice sizes without requiring the construction of new input maps for each environmental factor for the selected size.

In general, maps used in the simulation are generated using geographic information system (GIS) tools. However, the *Baseline case* uses only three types of factors which do not require GIS input maps to establish settlements: a road along the lattice diagonal, one type of vegetation that is decreased at each time step around settlements due to livestock grazing, and the distances amongst settlements. The *Baseline case* enables us to perform benchmarks with different lattice sizes and with an artificial set of input factors. We executed the *Baseline case* with four lattice sizes $N \times N$, with $N = 512, 1024, 2048, 4096$. These selected sizes would allow us to see how the code scales up to lattice sizes which could be finer than current map resolutions.

The second numerical experiment, the *Realistic case*, includes all factors and uses the real maps for the selected region, with a fixed lattice size. This set-up includes the following factors: water access through rivers and groundwater, five types of vegetation, access to paved roads and old river beds, and finally the distances amongst settlements. The simulation starts with the settlements that were present in the year 1928, and places settlements according to the aptitude levels present in each lattice cell. The *Realistic case* experiment enables us to understand the feedbacks between natural resources and livestock settlements, which is crucial to estimate future effects of environmental and socio-economic changes in the region. In our previous work [14], we executed the *Realistic case* with a lattice size of 150×150 cells, with a cell representing 750×750 meters, with a total area of 112.5×112.5 kilometers. Here we are able to increase the size of the lattice to 1250×1250 and with a cell representing 90×90 meters for the same region, increasing considerably the resolution of the lattice, which allows us to obtain more accurate results.

3.2 Hardware and software infrastructure

The simulations were executed in two hardware environments with the following characteristics:

- Workstation Phenom (denoted as "Phenom"): 2.8 GHz AMD Phenom II 1055T (released in 2010) 6 cores with 12 GB DDR3 of RAM memory. NVIDIA Tesla c2050 GPU (Fermi architecture, released in 2011), with 448 CUDA cores working at 1.15 GHz, and 3 GB memory. Slackware Linux 14.1 64 bit operating system with kernel 3.10.5, CUDA 6.5 and GCC 4.8.1.
- Two nodes of a Cluster at the Universidad Nacional de Cordoba (denoted as "Mendieta") with the same CPU and two different GPUs: two Intel Xeon E5-2680 v2 (Ivy Bridge microarchitecture, released in 2013) with 10 cores each running at 2.8 GHz and 64 GiB DDR3 at 1600 MHz. One node has a NVIDIA Tesla M2090 GPU (Fermi architecture, released in 2011) with 6 GiB of GDDR5 memory and the second node has one NVIDIA Tesla K20Xm GPU (Kepler architecture, released in 2012) with 6 GiB of GDDR5 memory. With Linux CentOS 6.5, kernel 2.6.32-504 and GCC 4.8.2.

We use the *GNU gprof* [9] software profiler in order to analyze the serial implementation of the code to identify the most time consuming functions. The GPU implementation was analyzed with the NVIDIA Visual Profiler 6.5 [17], a tool which provides a complete interface to profile CUDA applications, included with the CUDA development kit. All codes are compiled with *-O3* compiler optimizations and use CUDA 6.5 with NVIDIA driver version 340.29.

3.3 Baseline case

We executed the *Baseline case* with four lattice sizes $N \times N$ (with $N = 512, 1024, 2048, 4096$), during 1000 steps in two GPUs, the Tesla C2050 (Phenom workstation) and the Tesla k20x (Mendieta cluster node). The GPU execution was tested with 3 block sizes (128, 256 and 512), and the results can be seen in Fig. 1. In the Phenom workstation, the GPU execution with $blocksize = 128$ is between a $\sim$6 and $\sim$15 % faster than with a block size of 256 or 512 for all N sizes (Table 1). The speedups from GPU with block size 128 versus the CPU serial code in the Phenom Workstation go from 7.8× for $N = 512$ to 23.6× for $N = 4096$. The Intel Xeon CPU present in the Mendieta cluster is $\sim$3 times faster than the AMD Phenom for the *Baseline case* simulation. The GPU execution with $blocksize = 128$ is also faster than the block

Table 1 Percentage of performance improvement or decline comparing a blocksize of 128 compared with the blocksizes 256 and 512, for different linear lattice sizes (N), considering the *Baseline case* in the Tesla c2050 and Tesla k20x GPUs

N	Tesla c2050		Tesla k20x	
	128 versus 256	128 versus 512	128 versus 256	128 versus 512
512	6.6	8.7	−13.3	−14.4
1024	8.5	10.3	15.1	9.2
2048	7.2	8.7	7.4	14.4
4096	15.1	12.4	11.6	16.5

Table 2 *Baseline case* simulation with $N = 4096$

Stage	CPU	GPU	Speedup
Input	1.15	2.12	0.54
Setup	1235	5.67	217.8
Evolve	874.04	28.75	30.4
Output	57.83	59.22	0.97
Total	2168.02	95.76	22.64

Input, setup, evolve and *Output* time are shown. The reported time is in seconds with the average of 10 simulations executed in the Phenom Workstation. The GPU case is executed with a block size of 128

sizes 256 and 512, between a ∼7 and a ∼16 %, except for $N = 512$, where the block size 128 is slower than 256 and 512 block sizes, ∼13 and ∼14 % slower, respectively (Table 1). The speedups obtained in the k20x GPU versus the Xeon CPU in the Mendieta cluster go from 2.5× for $N = 512$ to 8.8× for $N = 4096$ (with block size 128). The wallclock time scales linearly with the lattice size, it should behave as N^2, and that is indeed what we find for the GPU, with a slightly worse scaling for the CPUs.

In Table 2 it can be seen the execution time and speedup for each stage of the simulation running in the Phenom CPU and the Tesla c2050 GPU (with block size 128), with $N = 4096$ during 1000 steps. The *Input* stage uses almost twice the time in the GPU than on CPU, an expected behavior considering the time required to perform memory allocation and transfer of the input arrays to the GPU. The *Setup* stage is where most of the speedup is obtained, particularly the *calculate_min_distances()* function, with ∼217× of speedup. The *Evolve* stage obtains a ∼30× speedup. And the *Output* time is slightly greater in the GPU simulation due to the need to copy the output arrays from the GPU to the CPU before writing the output files to disk. The time required to complete each of the phases of the simulation are also shown in Fig. 2 for different lattice sizes ($N \times N$ with $N = 512$, 1024, 2048 and 4096), in the Phenom workstation with a GPU block size of 128. In this figure it can be seen the improvement in performance for the *Setup* and *Evolve* phases between CPU and GPU simulations.

From the results obtained in Fig. 1 and Table 1, it can be seen that there is a difference in runtime when the block size is varied. For this reason we decided to perform a more in depth analysis of our code for a greater number of block sizes (64, 128, 192, 256, 512 and 1024) for two lattice sizes ($N = 1024$ and 2048). In Fig. 3 it can be seen the wallclock time normalized with the total lattice cells for the different block sizes. The best performance is obtained for $blocksize = 128$ for the two lattice sizes. Using the NVIDIA Visual Profiler we obtained the occupancy of each SM for a single CUDA kernel (*cuda_kernel_calculate_PAll_BMC()*). The values of occupancy for the $blocksizes = (64, 128, 192, 256, 512)$ are the following: 32, 62, 88, 88, and 87 % respectively. We obtained the best performance with an occupancy of 62 %, which is not the highest occupancy. This fact could be explained following the study by Ryoo et al. [18]: a small block size allows many thread blocks. Too many thread blocks would eventually lead to inefficient memory access, and the optimum block size is around 128 [12,18] for our type of simulation, and it is not necessarily the best block size for all the kernels executed in the simulation. In our code, this small block size results in more warps available to execute in each SM, which would hide the stalling effects of global memory latency or blocking operations by having enough independent warps available to execute in the SMs. The most time consuming functions in our code have Global memory access and are memory bound, for example, each thread of the *cuda_kernel_calculate_PAll_BMC()* kernel for the *Baseline case* has to access nine variables located in Global memory and perform two multiplications. Lowering the number of threads decreases the number of available warps ready to be executed and gives more blocks, in the case of the C2050 GPU, eight blocks can be executed per SM, with 128 threads per block there are 12 blocks per SM to be executed. The NVIDIA Profiler reports that for 256 threads per block the stall reason is "pipeline busy: the compute resources required by the instruction are not yet available". Even though the occupancy is at 88 % which is higher than for 128 threads per block at 62 % occupancy, the lower occupancy results in better performance as it can be seen in Figs. 1 and 3 when $blocksize = 128$. When the number of threads per

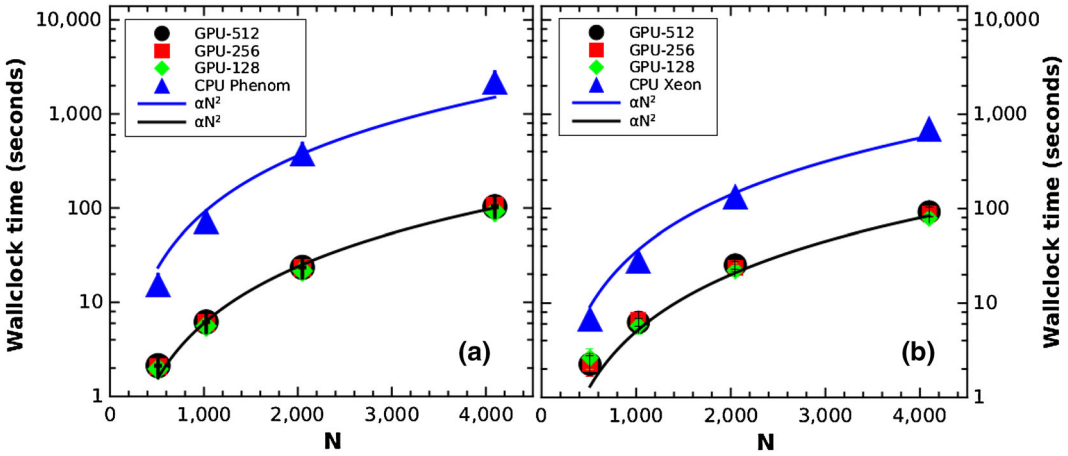


Fig. 1 Wallclock time (in seconds) for the *Baseline case* executed for different lattice sizes in **a** the Phenom workstation (Tesla c2050 GPU), and **b** a Mendieta Cluster node (Tesla k20x GPU). *Lines* indicate N^2 behavior

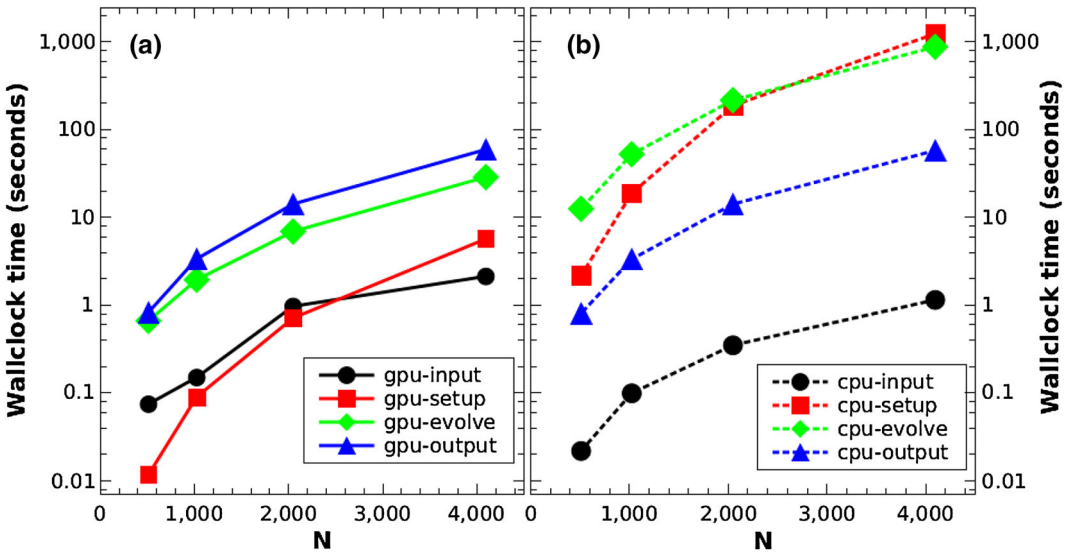


Fig. 2 Wallclock time (in seconds) for the *Baseline case* executed for different lattice sizes, divided into different contributions: *Input, Setup, Evolve* and *Output* time. **a** c2050 GPU, and **b** Phenom 1055t CPU

block decreases to 64 (Fig. 3), the running time increases, this issue is due to a memory latency problem because there is not enough active warps to hide the memory access cost. Also, it can be seen in Fig. 3 that the wallclock time for different block sizes has the same behavior for two different lattice sizes.

Communication and kernel compute time is shown in Table 3, using the NVIDIA Visual Profiler tool to inspect the performance of the GPU simulation. Communication and compute time scales as expected like N^2. The *Compute* time includes CUDA kernels executed in the *Setup* and *Evolve* stages, this is the reason why the kernel *Compute* time (Table 3) is greater to the *Evolve* stage time previously shown in Table 2 and Fig. 2.

3.4 Realistic case

The *Realistic case* simulation was executed for a lattice size of 1250 × 1250 in the two hardware infrastructures, using three GPUs. Results can be seen in Fig. 4. The difference in execution time present in the *Realistic case* experiment ($N = 1250$) versus the *Baseline case* ($N = 1024$, Fig. 1) is due to the environmental factors added to the *Realistic case*. We profiled the *Realistic case* experiment running in CPU with *gprof* in the Phenom workstation. The *calculate_min_distances()* function now uses the ∼90 % (∼867 s) of the total running time (∼959 s), the *calculate_PAll_BMC()* function requires ∼7 % (∼70 s) of the time, and finally the *recalc_pdist_settlements()* functions uses ∼1.5 % (∼16 s) of the total time.

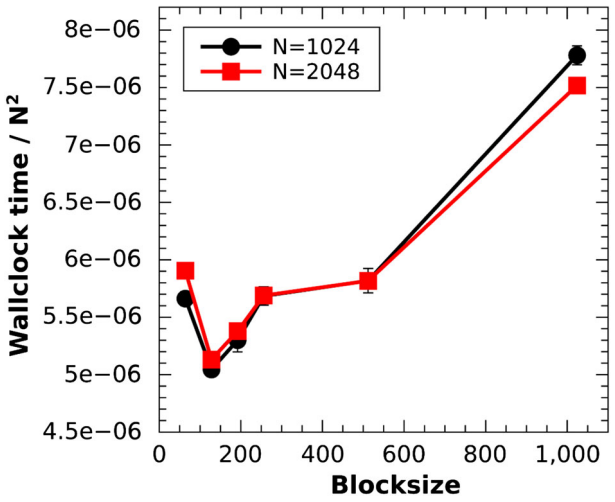


Fig. 3 Wallclock time normalized with total lattice cells ($seconds/N \times N$), considering the *Baseline case* for different block sizes ($N = 1024$ and $N = 2048$). Executed in the Phenom workstation

The GPU simulation obtains the best performance with $blocksize = 128$. The k20X GPU performs better than the c2050 and the m2090 GPUs, an expected behavior since the microarchitecture of the k20X (Kepler) is newer and it has more CUDA cores than the m2090 and c2050 (Fermi) microarchitecture. The performance of the NVIDIA m2090 GPU is very similar to the c2050 GPU present in the Phenom workstation. The CPU performance in the Xeon CPU (Mendieta cluster) is ∼5 times better than the performance of the Phenom CPU, which will result in smaller speedups for the simulation in the Mendieta cluster. The speedup obtained for the GPU versus CPU simulation in the Phenom workstation is ∼80× ($blocksize = 128$). For the k20X GPU the speedup versus the Xeon CPU is ∼23× ($blocksize = 128$). The m2090 GPU versus the Xeon CPU has a speedup of ∼20× ($blocksize = 128$). We consider that it is a fair comparison between the GPUs and CPUs utilized to perform the tests. The Phenom CPU is only one year older than the C2050 GPU, and the Xeon CPU is one year newer than the k20x GPU and two years newer than the m2090 GPU. For more details of the hardware used see Sect. 3.2.

As stated in the Introduction, we need to execute ∼100 simulations for each set of parameters. Running 100 simulations in the Phenom CPU of the *Realistic case* with $N = 1250$ will result in ∼26 h of execution time for one set of parameters, and we need to run multiple configurations of the same system. Executing the same 100 simulations in one GPU requires only ∼21 min. Therefore, the faster GPU code opens the possibility to add more relevant environmental factors with their corresponding set of parameters for the parameter sweep, like distance to unpaved roads, the topography of the terrain, etc.

Table 3 Communication and compute time (in seconds) for GPU simulations with four lattice sizes executed in the Phenom Workstation with $blocksize = 128$, during 1000 steps

Lattice size (N)	Communication time	Compute time
512	0.016	0.575
1024	0.048	2.1
2048	0.181	8.5
4096	0.714	38

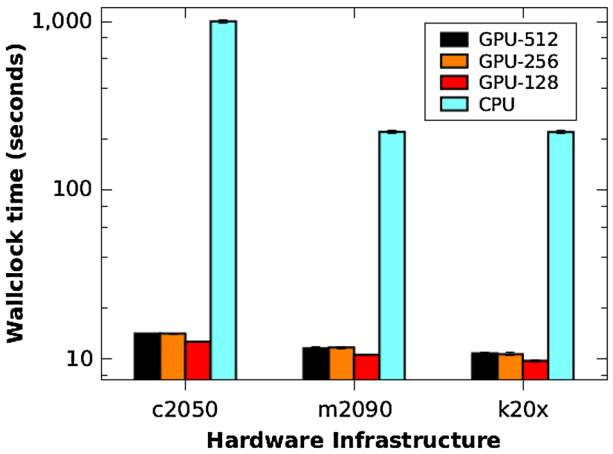


Fig. 4 Wallclock time for simulations of the *Realistic case* with a lattice size of 1250 × 1250, executed in three hardware infrastructures with three block sizes (128, 256, and 512)

We also tested the *Realistic case* with $N = 150$ during 1000 steps, the same lattice size we used in our previous work [14]. The obtained performance resulted in ∼4× of speedup in the Phenom Workstation with the Tesla C2050 GPU. This speedup is reasonable for a small lattice size due to the required time to prepare and copy the input data from the CPU to the GPU, which is one order of magnitude higher for the GPU simulation than the one required for the CPU simulation.

We aim to simulate the settlement distribution at a final time step, and not to present a "realistic" demographic time evolution as in Kohler et al. [13]. The results of our model approximate the spatial pattern of settlements at a regional scale, with higher densities and more aggregation near rivers and old river beds, and sparse settlements, with a random distribution, in areas inside the region without access to surface water, as observed in the real case. The simulated vegetation map resulting from the degradation of the vegetation around settlements has a similar spatial distribution of different vegetation classes as the remotely sensed (real case) vegetation. Degraded vegetation classes appear in simulation and real data in the NW and NE of the grid. Patches of vegetation with less vegetation cover are immersed in the area of historic woodlands in both, simulation and observation [14].

The model can reproduce the current settlement pattern because environmental factors and stochastic effects, suggesting that environmental features related on water availability have a strong effect on settlement spatial distribution in the studied area. Therefore, change in water availability and quality will modify settlement distribution and pressure on the environment. The model could be a useful tool to evaluate the effects of land use change (water provision, river flows), on settlement distribution and vegetation degradation in arid environments.

4 Conclusions

We developed a GPU Monte Carlo (MC) model in order to simulate the establishment of livestock settlements considering the environmental factors of the region. A 2D lattice is considered, and the settlements are placed in the lattice by using probabilities for each environmental factor and random numbers to account for social factors. The CPU implementation would take months of running in a desktop computer in order to sample the required parameter space with a greater lattice size. The GPU implementation enables the execution of such systems in a fraction of the time and results in speedups of up to $\sim 80\times$.

In order to parallelize the code, each lattice cell was assigned a given thread, which can evolve one step independently of any other threads, and without communication amongst threads. Of course, if a new settlement appears, each thread has to be given information of the distance to that settlement. An efficient *reduce* operation with shared memory [10] is used to find maximum probabilities for new settlements.

Two numerical experiments were executed aiming to test different optimizations of the code: a *Baseline case* with three artificial environmental factors and a *Realistic case* with six environmental factors. Three GPUs were used to execute the simulations: the Baseline case used a Tesla C2050, and a Tesla K20x; the Realistic Case added a third GPU, the Tesla M2090. We obtained speedups between $\sim 8\times$ and $\sim 20\times$ for the *Baseline case* in the Tesla C2050, for a Phenom CPU (2.8 GHz). Using the k20x GPU, we obtained from $\sim 2.5\times$ to $\sim 8\times$ of speedup versus a Xeon CPU (2.8 GHz). The *Realistic case* experiment is more complex and requires more computing time. In the Tesla C2050 the speedup obtained was $\sim 80\times$, while using the k20x GPU we obtained $\sim 23\times$ of speedup versus the Xeon CPU. The reduced speed-ups for the k20x are due to the comparison with a much faster CPU.

For a square 2-D lattice with size $N \times N$, scaling with lattice size followed the expected N^2 behavior. We tested several thread block sizes to find the optimum value for our simulation, and the best timing was achieved for a block size of 128, in agreement with results for other applications [12,18]. The occupancy of the GPU for different block sizes was also analyzed and we found that having a high occupancy did not give the best performance, a result that was also reported by Volkov in [19].

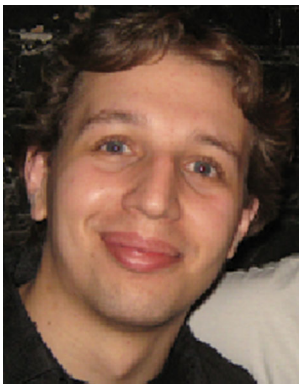
The performance improvement obtained with this new GPU simulation would allow the execution of larger lattice sizes, leading to increased resolution. In addition, more relevant environmental factors can be taken into account, including the possibility of a parameter sweep with many more dimensions. Such improvements are required for advanced simulations of settlements which would hopefully allow detailed planning of resource usage in arid environments.

Acknowledgments We acknowledge support from CONICET, ANPCyT grants (PICT-PRH-0092 and PICT-PRH 2703), and a SeCTyP UNCuyo grant. This work used the Mendieta Cluster from CCAD-UNC, that is part of SNCAD MinCyT, Argentina. We thank the anonymous reviewers for comments and suggestions which helped to improve the manuscript.

References

1. Asner, G.P., Elmore, A.J., Olander, L.P., Martin, R.E., Harris, A.T.: Grazing systems, ecosystem responses, AND global change. Annu. Rev. Environ. Resourc. **29**(1), 261–299 (2004). doi:10.1146/annurev.energy.29.062403.102142
2. Binder, K.: Monte Carlo and Molecular Dynamics Simulations in Polymer Science, vol. 20. Oxford University Press, New York (1995)
3. Block, B., Virnau, P., Preis, T.: Multi-GPU accelerated multi-spin Monte Carlo simulations of the 2D ising model. Comput. Phys. Commun. **181**(9), 1549–1556 (2010). doi:10.1016/j.cpc.2010.05.005
4. Bura, S., Gurin-Pace, F., Mathian, H., Pumain, D., Sanders, L.: Multiagent systems and the dynamics of a settlement system. Geogr. Anal. **28**(2), 161–178 (1996). doi:10.1111/j.1538-4632.1996.tb00927.x
5. Chan, V.W.K. (ed.): Theory and Applications of Monte Carlo Simulations. InTech (2013). doi:10.5772/45892
6. Corvalan, C., Hales, S., McMichael, A.J.: Ecosystems and Human Well-Being: Health Synthesis. World Health Organization, Geneva (2005)
7. 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
8. Goirán, S., Aranibar, J., Gomez, M.: Heterogeneous spatial distribution of traditional livestock settlements and their effects on vegetation cover in arid groundwater coupled ecosystems in the Monte desert (argentina). J. Arid Environ. **87**, 188–197 (2012). doi:10.1016/j.jaridenv.2012.07.011
9. Graham, S.L., Kessler, P.B., Mckusick, M.K.: Gprof. Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction—SIGPLAN 82 (1982). doi:10.1145/800230.806987
10. Harris, M., et al.: Optimizing parallel reduction in cuda. NVIDIA Dev. Technol. **2**(4) (2007)
11. Herrero, M., Thornton, P.K.: Livestock and global change: emerging issues for sustainable food systems. Proc. Natl. Acad. Sci. **110**(52), 20878–20881 (2013). doi:10.1073/pnas.1321844111

12. 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
13. Kohler, T.A., Bocinsky, R.K., Cockburn, D., Crabtree, S.A., Varien, M.D., Kolm, K.E., Smith, S., Ortman, S.G., Kobti, Z.: Modelling prehispanic pueblo societies in their ecosystems. Ecol. Model. **241**, 30–41 (2012). doi:10.1016/j.ecolmodel.2012.01.002
14. Millán, E., Goirán, S.B., Aranibar, J., Forconesi, L., García Garino, C., Bringa, E.M.: Evaluating the importance of environmental factors on livestock settlement spatial distribution in the monte desert with a Monte Carlo based model: settlement dynamics in drylands (SeDD). J. Arid Environ. (2015). arXiv:1507.07886. In revision
15. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. Queue **6**(2), 40 (2008). doi:10.1145/1365490.1365500
16. NVIDIA: NVIDIA CUDA C Programming Guide 4.2 (2012)
17. Nvidia visual profiler 6.5. http://docs.nvidia.com/cuda/profiler-users-guide/index.html#axzz3LbW336FP
18. 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
19. Volkov, V.: Better performance at lower occupancy. In: Proceedings of the GPU Technology Conference, GTC 10 (2010)



Emmanuel N. Millán received a BSc. in Software Engineering from the Universidad del Aconcagua, Argentina, in 2010. He is presently pursuing his Ph.D. thesis since 2011 under the supervision of Carlos García Garino, Eduardo M. Bringa and María Fabiana Piccoli. His thesis deals with the implementation of parallel problems in hybrid clusters including Graphics Processing Units (GPUs).

Silvana B. Goirán received a BSc. in Agricultural Engineer and has a Master of Science: Integrated planning for rural development and environmental management of the Universidad de Lleida, España. She is a Doctoral student in the Universidad Nacional de Cuyo, Argentina. Her research interests are: ecology, landscape ecology, degradation process and interaction between human and the environment, analysis of human impacts in ecosystem structure and function, and influence of natural resources in the patterns of human occupation. She is also an expert in remote sensing and GIS.

María Fabiana Piccoli received her Ph.D. degree from Universidad Nacional de San Luis (UNSL), Argentina, in 2005, and the Graduated in 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.

Julieta N. Aranibar received her Ph.D. in Environmental Sciences from University of Virginia, USA, in 2003. Currently holds a Researcher position within CONICET-IANIGLA, and also she is an Associate Professor at the Facultad de Ciencias Exactas y Naturales (FCEN, UNCuyo). Her areas of research are: biogeochemistry, livestock activity, ecology, groundwater-coupled ecosystems and biological soil crusts.

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