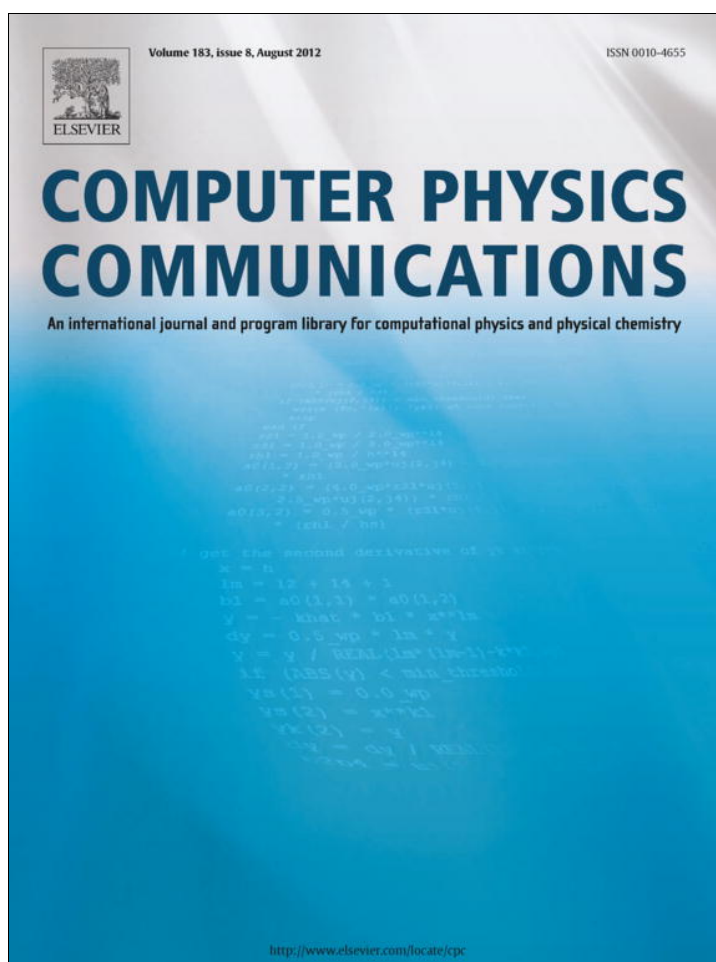
Provided for non-commercial research and education use.
Not for reproduction, distribution or commercial use.



This article appeared in a journal published by Elsevier. The attached copy is furnished to the author for internal non-commercial research and education use, including for instruction at the authors institution and sharing with colleagues.

Other uses, including reproduction and distribution, or selling or licensing copies, or posting to personal, institutional or third party websites are prohibited.

In most cases authors are permitted to post their version of the article (e.g. in Word or Tex form) to their personal website or institutional repository. Authors requiring further information regarding Elsevier's archiving and manuscript policies are encouraged to visit:

http://www.elsevier.com/copyright

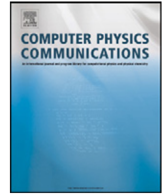
Author's personal copy

Computer Physics Communications 183 (2012) 1578–1587

Contents lists available at SciVerse ScienceDirect

Computer Physics Communications

www.elsevier.com/locate/cpc



q-state Potts model metastability study using optimized GPU-based Monte Carlo algorithms

Ezequiel E. Ferrero [a], Juan Pablo De Francesco [b], Nicolás Wolovick [b], Sergio A. Cannas [b,c,*]

[a] *CONICET, Centro Atómico Bariloche, 8400 San Carlos de Bariloche, Río Negro, Argentina*
[b] *Facultad de Matemática, Astronomía y Física, Universidad Nacional de Córdoba, Ciudad Universitaria, 5000 Córdoba, Argentina*
[c] *Instituto de Física Enrique Gaviola (IFEG-CONICET), Ciudad Universitaria, 5000 Córdoba, Argentina*

ARTICLE INFO

Article history:
Received 1 January 2011
Received in revised form 2 February 2012
Accepted 25 February 2012
Available online 3 March 2012

Keywords:
Monte Carlo
GPU
CUDA
Potts model
Metastability

ABSTRACT

We implemented a GPU-based parallel code to perform Monte Carlo simulations of the two-dimensional q-state Potts model. The algorithm is based on a checkerboard update scheme and assigns independent random number generators to each thread. The implementation allows to simulate systems up to $\sim 10^9$ spins with an average time per spin flip of 0.147 ns on the fastest GPU card tested, representing a speedup up to 155×, compared with an optimized serial code running on a high-end CPU.

The possibility of performing high speed simulations at large enough system sizes allowed us to provide a positive numerical evidence about the existence of metastability on very large systems based on Binder's criterion, namely, on the existence or not of specific heat singularities at spinodal temperatures different of the transition one.

© 2012 Elsevier B.V. All rights reserved.

1. Introduction

The tremendous advances allowed by the usage of numerical simulations in the last decades have promoted these techniques to the status of indispensable tools in modern Statistical Mechanics research. Notwithstanding, many important theoretical problems in the field still remain difficult to handle due to limitations in the available computational capabilities. Among many others, typical issues that challenge the numerical treatment concern systems with slow dynamics (i.e., dynamical processes that involve very different time scales) and/or strong finite size effect, which require fast simulations of a very large number of particles. Some typical examples we may cite are spin glass transitions [1], glassy behavior [2,3] and grain growth [4]. In such kind of problems the state of the art is usually launched by novel numerical approaches or extensive computer simulations. In this sense, the advent of massive parallel computing continuously opens new possibilities but, at the same time, creates a demand for new improved algorithms. In particular, the usage of GPU cards (short for Graphics Processing Units) as parallel processing devices is emerging as a powerful tool for numerical simulations in Statistical Mechanics systems [5–10], as well as in other areas of physics [11–13].

These GPUs have a Toolkit that abstracts the end-user from many low-level implementation details, yet all the typical problems of concurrency exist and they are magnified by the massive amount of (virtual) threads it is capable to handle. An extremely fine grained concurrency is possible and advised thanks to the Single Instruction Multiple Thread (SIMT) model. Therefore, any non-trivially independent problem requires a correct concurrency control (synchronization), and the lack of it hinders correctness in a much dramatic way than current 4 or 8-way multicore CPU systems. The other challenge apart from correctness is performance, and here is where the algorithm design practice excels. Taking into account internal memory structure, memory/computation ratio, thread division into blocks and thread internal state size, can boost the algorithm performance ten times from a trivial implementation [14]. It is also customary to give an approximation of the speedup obtained from a CPU to GPU implementation in terms of "$N\times$", even though, as we discuss later, this number will always depend on the corresponding efforts devoted to optimally programming for each architecture.

In this work we focus on GPU-based Statistical Mechanics simulations of lattice spin systems. In particular, we study the metastability problem in the ferromagnetic q-state Potts model [15] in two dimensions when $q > 4$. While this phenomenon is clearly observed in finite size systems, its persistence in the thermodynamics limit is still an unsolved problem and subject of debate [16–20]. In an earlier work, Binder proposed a numerical criterion to determine whether metastability remains in the thermodynamic limit or not, based on the scaling properties of the average energy in the vicinity of the transition temperature [16]. However, the narrow range of temperature values of the metastable region requires

* Corresponding author at: Facultad de Matemática, Astronomía y Física, Universidad Nacional de Córdoba, Ciudad Universitaria, 5000 Córdoba, Argentina.
E-mail address: cannas@famaf.unc.edu.ar (S.A. Cannas).

0010-4655/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.cpc.2012.02.026

Author's personal copy

high precision calculations for the criterion to work. Hence, to reduce finite size bias and statistical errors down to an appropriated level, large enough system sizes are needed. The computation capabilities required to carry out such calculations in a reasonable time were unavailable until recently.

We developed an optimized algorithm to perform Monte Carlo numerical simulations of the q-state Potts model on GPU cards. This algorithm allowed us to simulate systems up to $N = 32768 \times 32768 \sim 1.073 \times 10^9$ spins with a lower bound time of 0.147 ns per spin flip using an NVIDIA GTX 480 Fermi card, and in terms of speedup, we obtained 155× from an optimized CPU sequential version running on an Intel Core 2 Duo E8400 at 3.0 GHz.

What is remarkable about the speedup is that it allowed us to explore bigger systems, simulate more iterations, explore parameters in a finer way, and all of it at a relatively small cost in terms of time, hardware and coding effort. With this extremely well performing algorithm we obtained a positive numerical evidence of the persistence of metastability in the thermodynamic limit for $q > 4$, according to Binder's criterion.

The paper is structured as follows. In Section 2 we briefly review the main properties of the Potts model and the particular physical problem we are interested in. In Section 3 we introduce the simulation algorithm and in Section 4 we compare the predictions of our numerical simulations against some known equilibrium properties of the model to validate the code. In Section 5 we check the performance of the code. In Section 6 we present our numerical results concerning the metastability problem. Some discussions and conclusions are presented in Section 7.

2. The *q*-state Potts model

2.1. The model

The q-state Potts model [15] without external fields is defined by the Hamiltonian

$$H = -J \sum_{\langle i,j \rangle} \delta(s_i, s_j) \tag{1}$$

where $s_i = 1, 2, \ldots, q$, $\delta(s_i, s_j)$ is the Kronecker delta and the sum runs over all nearest neighbors pairs of spins in a Bravais lattice with N sites. Being a generalization of the Ising model ($q = 2$), this model displays a richer behavior than the former. One of the main interests is that the two-dimensional ferromagnetic version ($J > 0$) exhibits a first order phase transition at some finite temperature when $q > 4$, while for $q \leqslant 4$ the transition is continuous [15]. Hence, it has become a paradigmatic model in the study of phase transitions and their associated dynamics, like for instance, domain growth kinetics [21–25] and nucleation as an equilibration mechanism [17,26,27].

Some equilibrium properties of the two-dimensional model are known exactly, which allows numerical algorithms testing. We list here some of them that are used for comparison with the numerical results in the present work. For instance, the transition temperature for the square lattice in the thermodynamic limit is given by [28]

$$\frac{k_B T_c}{J} = \frac{1}{\ln(1 + \sqrt{q})} \tag{2}$$

where k_B is the Boltzmann constant. Hereafter we will choose $k_B/J = 1$. Considering the energy per spin $e = \langle H \rangle / N$, in the thermodynamic limit the latent heat for $q > 4$ is [28]

$$e_d - e_o = 2\left(1 + \frac{1}{\sqrt{2}}\right) \tanh \frac{\Theta}{2} \prod_{n=1}^{\infty} (\tanh n\Theta)^2 \tag{3}$$

where $\Theta = \operatorname{arccos} \sqrt{q}/2$ and

$$e_d = \lim_{N \to \infty} \frac{1}{N} \lim_{T \to T_c^+} \langle H \rangle, \tag{4}$$

$$e_o = \lim_{N \to \infty} \frac{1}{N} \lim_{T \to T_c^-} \langle H \rangle. \tag{5}$$

Also

$$e_d + e_o = -2(1 + 1/\sqrt{q}) \tag{6}$$

from which the individual values of e_d and e_o can be obtained [29].

The order parameter is defined as

$$m = \frac{q(N_{max}/N - 1)}{q - 1} \tag{7}$$

where $N_{max} = \max(N_1, N_2, \ldots, N_q)$, being N_i the number of spins in state i. At the transition the jump in the order parameter (for $q > 4$) is given by [30]

$$\Delta m = 1 - q^{-1} - 3q^{-2} - 9q^{-3} - 27q^{-4} - \cdots. \tag{8}$$

2.2. Metastability

The problem of metastability in the infinite size q-state Potts model (for $q > 4$) is an old standing problem in statistical mechanics [16,31,24,32,18,20]. It has also kept the attention of the Quantum Chromodynamics' (QCD) community for many years [33,34,31,19,35], because it has some characteristics in common with the deconfining (temperature driven) phase transition in heavy quarks.

Metastability is a verified fact in a finite system. It is known [17,24,18] that below but close to T_c the system quickly relaxes to a disordered (paramagnetic) metastable state, with a life time that diverges as the quench temperature T approaches T_c (see for example Fig. 4 in Ref. [20]). This state is indistinguishable from one in equilibrium in the sense of local dynamics, namely, two times correlations depend only on the difference of times, while one time averages are stationary [18].

Nevertheless, the existence of metastability in the thermodynamic limit is still an open problem [18]. In Ref. [16] Binder studied static and dynamic critical behavior of the model (1) for $q = 3, 4, 5, 6$. Using standard Monte Carlo procedures he obtained good agreement with exact results for energy and free energy at the critical point and critical exponents estimates for $q = 3$ in agreement with high-temperature series extrapolations and real space renormalization-group methods. When analyzing the $q = 5$ and 6 cases he realized that the transition is, in fact, a very weak first order transition, where pronounced "pseudocritical" phenomena occur. He studied system sizes from $N = 16 \times 16$ up to $N = 200 \times 200$, and observation times up to $10^4 MCS$ (a Monte Carlo Step MCS is defined as a complete cycle of N spin update trials, according to the Metropolis algorithm). Within his analysis he was unable to distinguish between two different scenarios for the transition at $q \geqslant 5$ due to finite size effects taking place at the simulations. He proposed two self-avoiding possible scenarios for the transition. In the first one the energy per spin reaches the transition temperature with a finite slope both coming from higher and lower temperatures, thus projecting metastable branches at both sides of the transition that end at temperatures T_{sp}^+ and T_{sp}^- both different from T_c. In the second scenario, the energy reaches T_c with an infinite slope which would imply a first order phase transition with a true divergence of the specific heat at T_c.

On the other hand, other approaches based on different definitions of the spinodal temperatures predict, either the convergence of the finite size spinodal temperatures to T_c [17,19] or a convergence to limit values different from but closely located to T_c [20].

Author's personal copy

3. Optimized GPU-based Monte Carlo algorithm for the *q*-state Potts model

We developed a GPU-based code to simulate the two-dimensional Potts model, using classical Metropolis dynamics on square lattices of size $N = L \times L$ sites with periodic boundary conditions. For the spin update we partition lattice sites in two sets, the whites and the blacks, laid out in a framed checkerboard pattern in order to update in a completely asynchronous way all the white cells first and then all the black ones (given that the interactions are confined to nearest neighbors). This technique is also know as the Red–Black Gauss–Seidel [36]. We analyzed equilibrium states of systems ranging from $N = 16 \times 16$ to $N = 32\,768 \times 32\,768$ ($2^{15} \times 2^{15} \simeq 1.073 \times 10^9$ spins).

The typical simulation protocol is the following. Starting from an initial ordered state ($s_i = 1\ \forall i$) we fix the temperature to $T = T_{min}$ and run t_{tran} to attain equilibrium, then we run t_{max} taking one measure each δt steps to perform averages. After that, we keep the last configuration of the system and use it as the initial state for the next temperature, $T = T_{min} + \delta T$. This process is repeated until some maximum temperature T_{max} is reached. We repeat the whole loop for several samples to average over different realizations of the thermal noise. In a similar way we perform equilibrium measurements going from T_{max} to T_{min} starting initially from a completely random state.

3.1. GPU: device architecture and CUDA programming generalities

In 2006, NVIDIA decided to take a new route in GPU design and launched the G80 graphics processing unit, deviating from the standard pipeline design of previous generations and transforming the GPU in an almost general purpose computing unit. Although this decision could have been driven by the gaming community asking for more frames per second, NVIDIA took advantage of his General Purpose Graphics Processing Units (GPGPU), and in 2007 they launched the CUDA SDK, a software development kit tailored to program its G80 using C language plus minor extensions. The G80 hardware and the CUDA compiler quickly proved to have an extremely good relation in terms of GFLOPS per watt and GFLOPS per dollar with respect to the CPU alternatives in the application field of numerical algorithms.

The architecture has evolved two generations, GT200 in 2008 and 2009, and the GF100 in 2010, also known as the Fermi architecture. All of them share the same Single Instruction Multiple Thread (SIMT) concurrency paradigm in order to exploit the high parallelism (up to 480 computing cores) and the high memory bandwidth (up to 177 GBps). The SIMT model is a convenient abstraction that lies in the middle of the SIMD (Single Instruction Multiple Data) and MIMD (Multiple Instruction Multiple Data), where the first reigned in the 80's with the vector computers, and the later is the commonplace of almost every computing device nowadays, from cellphones to supercomputers.

Using SIMT paradigm, the parallel algorithm development changes greatly since it is possible to code in a one-thread-per-cell fashion. The thread creation, switching and destruction have such a low performance impact that doing a matrix scaling reduces to launch one kernel per matrix cell, even if the matrix is $32\,768 \times 32\,768$ of single precision floating point numbers summing up 1 GThread all proceeding in parallel. In fact, for the implementation, the more threads the better, since the high memory latency to global memory (in the order of 200 cycles) is hidden by swapping out warps (vectors of 32 threads that execute synchronously) waiting for the memory to become available.

It is important to emphasize the role of blocks in the SIMT model. Threads are divided into blocks, where each block of threads has two special features: a private shared memory and the ability to barrier synchronize. Using these capabilities, the shared memory can be used as a manually-managed cache that in many cases greatly improves the performance.

We used the GTX 280, GTX 470 and GTX 480 boards. The relevant hardware parameters for these boards are shown in Table 1.

The improvements of the Fermi architecture lay on the new computing capabilities (improved Instruction Set Architecture – ISA), the doubling of cores, the inclusion of L1 and L2 cache, increased per-block amount of parallelism and shared memory.

As every modern computing architecture the memory wall effect has to be relieved with a hierarchy of memories that become faster, more expensive and smaller at the top. The bottom level is the global memory, accessible by every core and having from 1 GB to 1.5 GB of size[1] and a latency of 200 cycles. The next level is the shared memory, that is configurable 16 KB or 48 KB per block having a latency of only 2 cycles. At the top there are 32 K registers per block. There are also texture and constant memory, having special addressing capabilities, but they do not bring any performance improvement in our application. The Fermi architecture has also incorporated ECC memory support to eventually deal with internal data corruption.

The programming side of this architecture is a "C for CUDA", an extension of the C Programming Language [37] that enables the *host* processor to launch *device* kernels [38]. A kernel is a (usually small) piece of code that is compiled by `nvcc`, the NVIDIA CUDA Compiler, to the PTX assembler that the architecture is able to execute. The kernel is executed simultaneously by many threads, organized in a two-level hierarchic set of parallel instances indexed as $(grid, block)$ (a grid of thread blocks). Internally each grid and block can be divided up to two dimensions for the grid and three dimensions for the block, in order to establish a simple thread-to-data mapping. Special variables store the thread position information of block and thread identifier (bid, tid) that distinguishes the threads executing the kernel.

It is interesting to note that although the unit of synchronous execution is a warp of 32 threads, the threads inside a warp may diverge in their execution paths (occurrence of bifurcations), at the cost of having to re-execute the warp once for each choice taken. Needless to say that in general this impacts negatively in the performance and has to be avoided.

The present code is divided in two main functions: spin update and energy and magnetization computation. The first function is implemented in host code by the function `update` and this comprises calling the device kernel `updateCUDA` once updating white cells and next updating black cells in a checkerboard scheme. The energy and magnetization (and their related moments) summarization is done by `calculate` that calls the kernel `calculateCUDA` and two more auxiliary kernels: `sumupECUDA` and `sumupMCUDA`.

3.2. Random number generator

The Potts model simulation requires a great amount of random numbers. Namely, each cell updating its spin needs one integer random number in $\{0, \ldots, q-1\}$ and possibly a second one in the real range $[0, 1)$ to decide the acceptance of the flip. Hence, a key factor to performance is using a good parallel random number generator.

Given the great dependence in terms of time (it has to be fast) and space (small number of per-thread variables), we find Multiply-With-Carry (MWC) [39] ideal in both aspects. Its state is

[1] These values apply to consumer graphics cards. The Tesla HPC line incorporates up to 6 GB of memory (e.g. Tesla C2070), that is configurable to be ECC in order to improve reliability.

Author's personal copy

Table 1
Key features about NVIDIA GTX 280, GTX 470, and GTX 480 graphic cards.

Board model	GTX 280	GTX 470	GTX 480
Available	Q2 2008	Q1 2010	
GPU	GT200	GF100	
CUDA capability	1.3	2.0	
CUDA cores	240	448	480
Processor clock	1.30 GHz	1.22 GHz	1.40 GHz
Global memory	1 GB	1.25 GB	1.50 GB
Memory bandwidth	141.7 GBps	133.9 GBps	177.4 GBps
L1 cache	N/A	16 KB–48 KB	
L2 cache	N/A	768 KB	
Max # of threads per block	512	1024	
Shared memory per block	16 KB	48 KB–16 KB	
Max # of registers per block	16 384	32 768	

only 64 bits, and obtaining a new number amounts to compute $x_{n+1} = (x_n \times a + c_n) \bmod b$, where a is the multiplier, b is the base, and c_n is the carry from previous modulus operation. We took the implementation from the CUDAMCML package [40] that fixes $b = 2^{32}$ in order to use bit masks for modulus computation.

For independent random number sequences, MWC uses different multipliers, and they have to be *good* in the following sense: $a \times b - 1$ should be a safeprime, where p is a safeprime if both p and $(p-1)/2$ are primes. Having fixed $b = 2^{32}$, the process of obtaining safe primes boils down to test for primality of two numbers $goodmult(a) \equiv prime(a \times 2^{32} - 1) \wedge prime((a \times 2^{32} - 2)/2)$. It is important to remark that the nearer to 2^{32} is a the longer the period of the MWC (for a close to its maximum, the period is near to 2^{64}), therefore it is always advisable to start looking for *goodmult* down from $2^{32} - 1$.

We limit the amount of independent random number generators (RNG) to $512^2/2 = 131\,072$ that is slightly lower than the 150 000 good multipliers that CUDAMCML gives in its file `safe_primes_base32.txt`. The state needed comprises 12 bytes per independent RNG, totalizing 1.5 MB of global memory, less than 0.15% of the total available in the GTX 280. We consider this a good trade-off between independence in number generation and memory consumption. This design decision is crucial in the parallelization of the spin update function, as we frame the lattice in rectangles of 512×512, to give each thread an independent RNG.[2] Moreover, this implies that the larger the lattice, the more work will be done by a single thread.

It is important to remark we are well below the RNG cycle even for the largest simulations.

3.3. Spin update

On top of the checkerboard division we have first to frame the lattice in rectangles of 512×512 in order to use the limited amount of independent RNG (Fig. 1, left). This implies launching two consecutive kernels (black/white) of $512 \times 512/2$ threads, typically organized into a grid of 32×16 blocks of 16×16 threads. The second step comprises the remapping of a two-dimensional stencil of four points in order to save memory transfers. The row–column pair (i, j) is mapped to $(((i + j) \bmod 2 \times L + i)/2, j)$, and this allows to pack all white and all black cells in contiguous memory locations improving locality and allowing wider reads of 3 consecutive bytes (Fig. 1, right).

We encode each spin in a byte, allowing simulations with $q \leqslant 256$ and $L^2 \leqslant$ *available RAM*. Since some extra space is needed for the RNG state and for energy and magnetization summarization, this upper bound is not reached. The biggest simulation we achieve is $L = 32\,768$, $q = 45$ for the GTX 480.

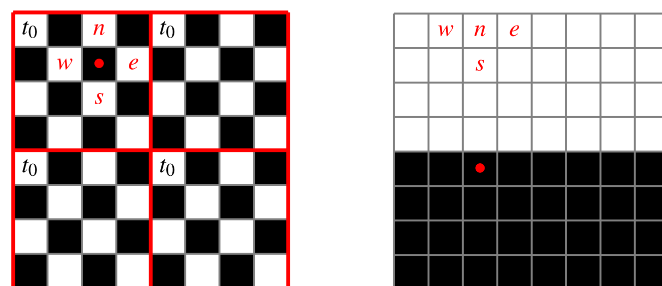


Fig. 1. (Color online.) On the left: an 8×8 checkerboard framed in 4×4 (red marking), the cells updated by thread t_0 are singled out, we also marked the north, east, south and west neighbors of cell •. On the right: packed checkerboard showing first half of whites, where the neighboring cells n, e, s, w are marked, also in the second half of black cells • is singled out.

It is important to remark that shared memory is not used, since we could not improve performance and it hindered readability of the code. Texture memory techniques were not used for the same reasons.

3.4. Computation of energy and magnetization

During the evolution of the system we extract periodically two quantities: energy, Eq. (1), and magnetization, Eq. (7). The kernel responsible for this job is `calculateCUDA`. It first partitions the cells into CUDA blocks. In each block we have easy access to barrier synchronization and shared memory among its threads. Each block within its cells adds the local energies and accumulates in a partial vector $(n_1, n_2, \ldots, n_q)$ the number of spins in each state. This is performed in shared memory using atomic increments to avoid race conditions. After that, those blocks' results are added up in parallel using a butterfly-like algorithm [38] by kernels `sumupECUDA` and `sumupMCUDA`, but none of the known optimizations [41] are applied, since it implies obfuscating the code for a marginal global speedup. Previous kernels end up with up to approximately a thousand partial energies and vectors of spin counters, that are finally added in the CPU.

It has to be noticed that device memory consumption in this part is linear not only in N, but also in q.

3.5. Defensive programming techniques and code availability

Writing scientific code that is maintainable, robust and repeatable is of utmost importance for the fields of science where computer simulation and experimentation is an everyday practice [42].

[2] For system sizes smaller than $N = 512^2$ we use smaller frames, and then, fewer RNG. But 512×512 is the standard framing choice for most of the work.

Author's personal copy

Table 2
Comparison between calculated and known exact values of e_o, e_d, and Δm at the transition for different values of q. Results were obtained from averages over 10 samples of linear system size $L = 2048$ and equilibration and measurements times of at least 5×10^5 MCS each one.

q	$-e_o$		$-e_d$		Δm	
	Exact	Calculated	Exact	Calculated	Exact	Calculated
6	1.508980...	1.51(2)	1.307516...	1.306(1)	0.677083...	0.674(2)
9	1.633167...	1.6332(5)	1.033499...	1.0334(5)	0.834019...	0.8338(4)
15	1.765905...	1.7659(2)	0.750492...	0.7509(4)	0.916693...	0.9167(3)
96	1.960306...	1.96030(3)	0.243817...	0.24382(4)	0.989247...	0.98924(2)

CUDA coding in particular is hard, not only in creating the algorithms, choosing a good block division and trying to take advantage of all its capabilities, but also, in the debugging and maintenance cycle. Debugging tools are evolving rapidly, for example there is a memory debugger `cuda-memcheck` that is shipped with current CUDA SDK. Nevertheless, we would rather adhere to some passive and active security measures within our code to make it easier to understand and modify, and at the same time, to make it robust in the sense of no unexpected hangs, miscalculations or silent fails.

Among passive security measures, we use assertions (boolean predicates) related to hardware limitations like the maximum of 512 threads per block. Other use of the assertions is checking for the integer representation limitations: given the computing power that GPGPU brings, lattices of $32\,768 \times 32\,768$ are feasible to simulate, and integer overflow could be a possibility, for example when computing the total energy. Assertions were also used to enforce preconditions on algorithm running, for example, the spin updating cannot do well if L is not multiple of the frame size. We also check every return condition of CUDA library calls and kernels, in order to lessen the asynchrony of error detection in CUDA. The same practice is used in standard library calls for file handling.

Active security measures are also taken. We use tight types in order to detect problems in compile time. We also decorate parameters and variable names with `const` modifiers where applicable. For pointer immutable parameters we forbid the modification of pointed data as well as the pointer itself. The scope of automatic variables is as narrow as possible, declaring them within blocks, in order to decrease the namespace size in every line of code. We put in practice the simple but effective idea of using meaningful variable names in order to improve the readability.

We also adhere to the practice of publishing the code [43] in the line of [5,6,9], since it benefits from community debugging and development. It can be found on [44].

4. Algorithm checking

In order to validate our CUDA code we run some typical simulations to measure well established results.

First we calculate the energy per spin e and magnetization m above and below the transition temperature, by cooling (heating) from an initially disordered (ordered) state. The behaviors of e and m as functions of T for different values of q are shown in Fig. 2. From these calculations we obtain the values of the energy (e_d and e_o) and magnetization jump Δm at the exact transition temperature (see Section 2). Results are compared with exact values in Table 2.

We can see a very good agreement between data and exact results. It's worth noting that the data from Table 2 is not the result of extrapolations of some finite size analysis, but the values from curves in Fig. 2 at the transition itself. Since we measure one point each ΔT in temperature, cooling and heating procedures won't necessary lead to a point measured exactly at T_c. So, we have to interpolate points close to T_c to deduce the corresponding values of e_o, e_d and m_o at T_c. The differences obtained from interpolations

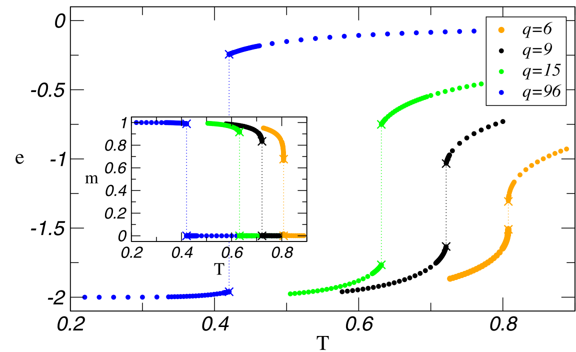


Fig. 2. (Color online.) Equilibrium energy per spin e and magnetization m (inset) versus temperature for $q = 9, 12, 15, 96$. Exact values at the transition point from Eqs. (3), (6) and (8) are marked as crosses. Data comes from averages over 10 samples of linear system size $L = 2048$. Error bars are smaller than the symbol size.

using points separated by ΔT and points separated by $2 \times \Delta T$ determine the estimated errors.

We also calculate the fourth order cumulant of the energy [45, 46]

$$V_L = 1 - \frac{\langle H^4 \rangle}{3\langle H^2 \rangle^2} \tag{9}$$

as a function of the temperature for $q = 6$ and different system sizes. As it is well known, V_L is almost constant far away from the transition temperature and exhibits a minimum at a pseudo critical temperature

$$T_c^*(L) = T_c + \frac{T_c^2 \ln(q e_o^2 / e_d^2)}{e_d - e_o} \frac{1}{L^d}. \tag{10}$$

In Fig. 3b we show $T_c^*(L)$ vs. $1/L^2$ for $q = 6$. The extrapolated value of $T_c^*(L)$ for $L \to \infty$, 0.8078 ± 0.0002 agrees with the exact value $T_c = 0.8076068\ldots$ within an accuracy of the 0.025%.

Let us emphasize that, as it is well known, it's very difficult to get good measures of cumulants with a single-spin flip MC algorithm. In order to get reliable averages of the cumulant minimum location, one should guarantee a measurement time long enough to let the system overcome the phase separating energy barrier back and forward several times. Moreover, the characteristic activation time to overcome the barrier increases both with q and L (it increases exponentially with L). For instance, simulation times of the order 10^7 for each temperature are needed to obtain a good sampling for $q = 6$ and $L = 256$.

In addition, we test our code for the $q = 2$ (Ising) case. Fig. 4 shows the susceptibility of the order parameter calculated as

$$\chi = \frac{N}{T}\left[\langle m^2 \rangle - \langle m \rangle^2\right]. \tag{11}$$

The extrapolated value of the pseudo critical temperature $T^*(L)$ (defined as the location of the susceptibility maximum) for

Author's personal copy

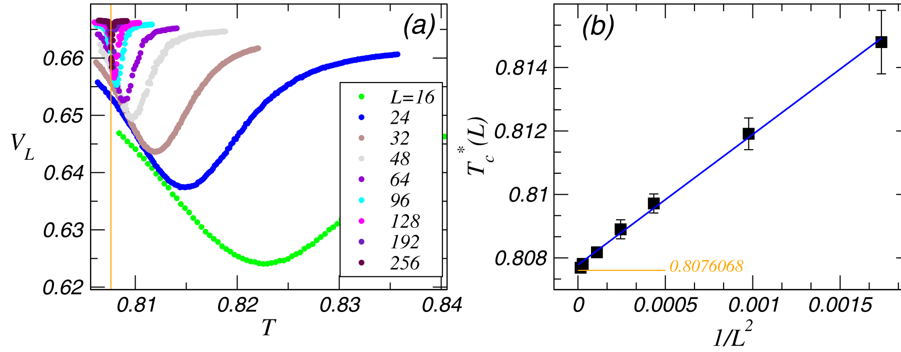


Fig. 3. (Color online.) Finite size scaling of the fourth order cumulant for $q = 6$. (a) V_L as a function of temperature for different system sizes. Averages were taken over several samples ranging from 300 to 400 for small system sizes down to 50 and 20 for $L = 128$ and $L = 256$. The orange line indicates the analytically predicted location of the minimum in the thermodynamic limit. (b) Pseudo critical temperature T_c^* vs. $1/L^2$. Error bars, estimated from the uncertainty when locating the minimum of V_L, are shown only when larger than the symbol size.

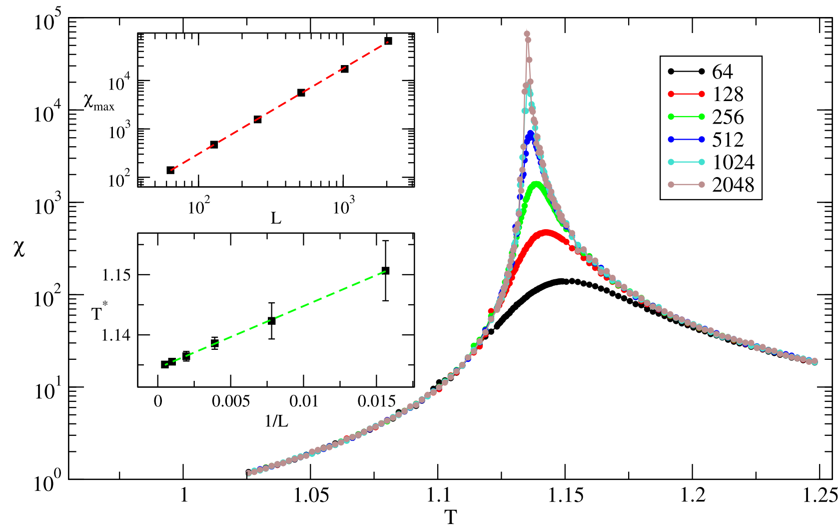


Fig. 4. (Color online.) Finite size scaling of the susceptibility for $q = 2$. Main plot: χ as a function of temperature for different system linear sizes. Averages were taken over several samples ranging from 300 for small system sizes down to 50 and 15 for $L = 1024$ and $L = 2048$, respectively. We have used equally equilibration and measurement times of $2 \times 10^5 MCS$, measuring quantities each $10MCS$, thus totalizing averages over 6×10^6 to 3×10^5 as we increase the system size. Upper inset: Maximum value of the susceptibility peak χ_{max} vs. L. Error bars, estimated from the uncertainty when evaluating the maximum, are smaller than the symbol size. Lower inset: Pseudo critical temperature $T^*(L)$ vs. $1/L$. Error bars, estimated from the uncertainty when locating the position of the maximum, are shown only when larger than the symbol size.

$L \to \infty$, 1.1345 ± 0.0001, agrees with the exact value[3] $T_c(q = 2) = 1.134592\ldots$ within an accuracy of the 0.009%. Even more, if we plot the maximum value of χ against the linear size L it is expected to observe a finite size scaling of the form $\chi_{max} \sim L^{\gamma/\nu}$ [47], where γ and ν are the exactly known critical exponents for the 2D Ising model. We obtain such scaling with a combined exponent $\gamma/\nu = 1.77 \pm 0.02$, in a good agreement with the exact value $\gamma/\nu = \frac{7/4}{1} = 1.75$.

5. Algorithm performance

The first step towards performance analysis is the kernel function calling breakdown. In this case, it is done using CUDA profiling capabilities and some scripting to analyze a 2.9 GB `cuda_profile_0.log` file produced after 12.6 hours of computation. The parameters used for this profiling are $q = 9$, $N = 2048 \times 2048$, $T_{min} = 0.721200$, $T_{max} = 0.721347$, $\delta T = 10^{-5}$, $t_{tran} = 10^5 MCS$, $t_{max} = 10^4 MCS$ and $\delta t = 500 MCS$.

The profile shows that there are approximately 32 millions of calls to `updateCUDA` and just a few thousands to the other three kernels. Since the individual GPU time consumptions of each kernel are comparable, the only relevant kernel to analyze is `updateCUDA`.

To analyze the kernel `updateCUDA` we sweep L in the range from 512 to 32 768 in powers of two, measuring the average execution time of the kernel and normalizing it to nanoseconds per spin flip.

We compare the three GPUs, using the same machine code (Compute Capability – CC 1.3, generated by NVCC 3.2),[4] and the same video driver (driver version 260.19). We also compare the GPUs performance with a CPU implementation. For this version, we tried to keep the structure of the CUDA code, in order to compare the execution of the same physical protocol on each architecture. We replaced the calls to CUDA kernels with loops running over all the spins in the same checkerboard scheme, we used the same MWC random number generator. We also added some optimizations to improve the CPU performance like creating a precomputed table of Boltzmann weights for the spinflip acceptance

[3] It should be remembered that $J_{\text{Potts}} = 2J_{\text{Ising}}$ if we compare our Hamiltonian (1) with the usual Ising Hamiltonian, thus giving a $T_c(q = 2)$ which is a half of the commonly appearing in Ising model works.

[4] Using CC 2.0 ISA does not bring any performance improvement.

Author's personal copy

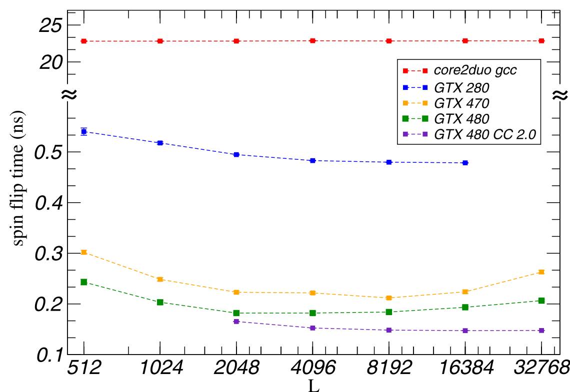


Fig. 5. (Color online.) Spin flip time in nanoseconds vs. lattice size running on an Intel Core 2 Duo E8400@3.0 GHz CPU, and running on GTX 280, GTX 470 and GTX 480 NVIDIA GPUs. Averages are performed over 400 runs for the GPUs and 60 runs for the CPU. Error bars are smaller than symbol sizes when not showed.

for each simulated temperature, since the CPU have no mechanism for hiding memory latency and the impact of any floating-point unit (FPU) computation is noticeable. We run the CPU code against a Core 2 Duo architecture (E8400 – Q1 2008) using GCC 4.4.5 with carefully chosen optimization flags.[5]

We also vary q in the set $\{6, 8, 9, 12, 15, 24, 48, 96, 192\}$. We don't find any significant variation of the performance with q, except in the $q = 2^k$ cases for the GTX 280, where the compiler obtains slight performance advantages using bitwise operators for modulus operation. The Fermi board has an improved modulus, rendering that difference imperceptible.

The profiling measurement is done in the GPU cases using CUDA profiling capabilities that gives very precise results, avoiding any code instrumentation. For the CPU version it is necessary to instrument the code with simple system calls to obtain the wall time. In order to make the measurement independent of the temperature range covered, given that the transition temperature (and therefore the flip acceptance rate) changes with q, we choose a deterministic write, i.e. we always write the spin value irrespective if the spin changes its state respect of its previous state or not. Writing the spin value only when it changes its state, brings a slight performance improvement around 2% in the general case.

In Fig. 5 we can see that the curve corresponding to the CPU implementation is flat around[6] 22.8 ns, showing no dependence of the averaged spin flip time with system size. For GPU cases, instead, we do have variations with respect to L. The slowest card is the GTX 280, with spin flip times in the range [0.48 ns, 0.54 ns] which are 47× to 42× faster than those of the CPU code. The GTX 470 has a variation between 0.21 ns and 0.30 ns, giving a speedup between 108× and 76×. The fastest card is the GTX 480 with spin flip times in [0.18 ns, 0.24 ns] achieving a speedup from 126× to 95×. There is also another curve corresponding to a specifically tuned version for the GTX 480 card[7] and CC 2.0, obtaining 155× (0.147 ns) for the fastest case. It is important to notice that even when using newer CPU architectures like Nehalem (X5550 – Q1 2009) the spin flip time only drops 2 ns in the best case respect to the Core 2 Duo, and that Intel C++ Compiler (ICC) cannot do any better than that.

Nevertheless, it should be noted that better CPU implementations could be possible, since most appropriate implementations for each architecture could be quite different from each other. For example, lower times can be attained for CPU using typewriter update scheme instead of a checkerboard one. For that reason, we hold the idea that a good measure to compare performances between GPU implementations is the "time per spin flip", and the speedup respect to a CPU implementation is just additional illustrative information.

The variations for the GPU cards are due to two competing factors in the loop of the update kernel. One is strictly decreasing with L and is related to the amount of global memory movements per cell. Since there is one RNG for each thread, the global memory for the RNG state is retrieved one time in the beginning and stored in the end, therefore the larger the L, this single load/store global memory latency is distributed into more cells. The second factor is increasing in L and is given by the inherent overhead incurred by a loop (comparison and branching), that for $L = 32\,768$ amounts to 4096 repetitions.

We also frame at 256×256 and 1024×1024, obtaining a 25% of performance penalty for the former, and a performance increase of 2% in the later. This gives us more evidence that the framing at 512×512 is an appropriate trade-off between memory consumption by the RNG and the speed of the code.

Although there are divergent branches inside the code, even for deterministic cell writes (the boolean "or" operator semantics is shortcircuted), eliminating all divergent branches doing an arithmetic transformation does not bring any performance improvement. This shows the dominance of memory requests over the integer and floating point operations, and the ability of the hardware scheduler in hiding the divergent branch performance penalty in between the memory operations.

To our knowledge this is the first time the Potts model is implemented in GPUs, so there is no direct performance comparison. There exist previous works that deal with similar problems and that report performance measurements. Preis et al. [5] implemented a 2D Ising model in GPUs, they reported a speedup of 60× upon their CPU implementation using a GTX 280. Their implementation has the disadvantage that the system size is limited by the *maximum number of threads per block* allowed (enforcing $L \leqslant 1024$ on GT200 and $L \leqslant 2048$ on GF100). Later on, Block, Virnau and Preis [6] simulated the 2D Ising model using multi-spin coding techniques obtaining 0.126 ns per spin flip in a GT200 architecture. Weigel [9,10] has also considered the 2D Ising model, obtaining a better 0.076 ns per spin flip [48] on the same architecture, which is improved to 0.034 ns per spin flip on a Fermi (GF100) architecture. Moreover, this was obtained with a single-spin coded implementation; however this gain is partially due to the use of a multi-hit technique updating up to $k = 100$ times a set of cells while others remain untouched. Notwithstanding, Weigel obtains [10] 0.13 ns per spin flip for the update without multi-hit and multi-spin, which is comparable with the result of the multi-spin coded version in [6]. Performance results on the 3D Ising model are also available [5,10]. The Heisenberg spin glass model is simulated on a GPU in Ref. [7], and for this floating point vector spin, they achieve a 0.63 ns per spin flip update on a GF100 architecture. Implementations of the Heisenberg model are also reported in [10] with times per spin flip down to 0.18 ns on a Fermi architecture, representing impressive speedups (up to 1029×). Recently, a GPU parallelization for the GF200 architecture was implemented in the Cellular Potts Model [49] with ∼80× speedup respect to serial implementations.

[5] Compiler options `-O3 -ffast-math -march=native -funroll-loops`.

[6] It's worth mentioning that in order to compare this value with CPU implementations of the Ising model (e.g., 8 ns in [10]), one should take into account that the Potts model update routine requires an extra random number to choose where to flip the spin. In addition, using MWC doesn't provide the fastest execution times; other RNGs as LCG-32 give better times but not completely reliable results [10] due to their short period. For the sake of completeness, we report that eliminating one random number toss and using LCG-32 instead of MWC we obtain a spin flip time of 14.5 ns for our CPU implementation.

[7] Each block is filling the maximum 1024 threads, we also disable L1 cache for a (free) slight performance improvement: compiler options `-Xptxas -dlcm=cg -Xptxas -dlcm=cg`.

Author's personal copy

We also conduct end-to-end benchmarks of a small simulation ($q = 9$, $L = 1024$, # of samples $= 3$, $T_{min} = 0.71$, $T_{max} = 0.73$, $\delta T = 0.002$, $t_{tran} = 2000$, $t_{max} = 8000$, $\delta t = 50$). We obtain 193 s for the GTX 280 and 8115 s for the Intel Core 2 architecture, with a global speedup of 42×, very similar to the speedup reported by the microbenchmarks. The coincidence of microbenchmarks and end-to-end benchmarks results reaffirms the fact that all the optimization efforts should go to the update kernel `updateCUDA`.

6. Metastability in the *q*-state Potts model

Based on Binder's criterion described in Section 2 we analyze the existence of metastability for $q > 4$ as the system size increases. From Fig. 2 we see that for large enough values of q the energy branches attain the transition temperature from both sides with a finite slope, even with a relatively poor temperature resolution. As q decreases, a closer approach to T_c is needed in order to distinguish whether a true singularity at T_c is present or not, since the spinodal temperatures are expected to be located very close to [20] T_c.

A power law divergence of the specific heat at T_c would imply the following behavior

$$e_{T<T_c} = e_o - A^-(1 - T/T_c)^{1-\alpha_-}, \tag{12}$$

$$e_{T>T_c} = e_d - A^+(1 - T_c/T)^{1-\alpha_+} \tag{13}$$

with $\alpha_-, \alpha_+ > 0$.

On the other hand, if well defined metastable states occur, the energy could be represented in terms of a specific heat diverging at pseudospinodal temperatures T^+_{sp}, T^-_{sp}

$$e_{T<T_c} = e^-_{sp} - A^-\left(1 - T/T^+_{sp}\right)^{1-\alpha_-}, \tag{14}$$

$$e_{T>T_c} = e^+_{sp} - A^+\left(1 - T^-_{sp}/T\right)^{1-\alpha_+}. \tag{15}$$

If divergences for the specific heat occur at the pseudospinodals, we should see exponents $\alpha_- = \alpha_+ \approx 0$ in Eqs. (12) and (13), since Eqs. (14) and (15) imply finite slopes at T_c.

We measure equilibrium curves for $e_{T<T_c}$ ($e_{T>T_c}$) starting from an ordered (disordered) initial state and performing a cooling (heating) procedure approaching T_c, as described in Section 3. The results are presented in Figs. 6 and 7. In both figures a crossover of the curve's slope as we approach T_c can be observed for all values of q. Close enough to T_c, the curves for $q = 9, 15, 96$ show exponents which are indistinguishable from 1, consistently with the existence of metastability and divergences at spinodal temperatures different from T_c, at least for $q \geqslant 9$.

As pointed out by Binder [16], to observe the crossover (if it exists at all) a temperature resolution at least $\Delta T = T_c - T^-_{sp}$ for the high energy branch (or $\Delta T = T^+_{sp} - T_c$ for the low energy branch) is needed, where $\Delta T \equiv |T - T_c|$. A numerical estimation of the lower spinodal temperature predicted by Short Time Dynamics [20] is given by

$$\frac{T_c - T^-_{sp}}{T_c} \simeq 0.0007\left(\ln(1 + q - 4)\right)^{2.81}. \tag{16}$$

The vertical dashed lines in Fig. 6 correspond to $T = T_c + \Delta T(q)$, as predicted from Eq. (16) according to the previous criterion. The coincidence with the crossover points for all values of q shows a complete agreement between the present results and those from Short Time Dynamics calculations. To attain the desired temperature resolution the system size has to be large enough, since finite size rounding errors are expected to decay as $1/L$ [16,45]. This is illustrated in the inset of Fig. 6 for the particular case $q = 9$, where a strong finite size effect is observed for $L = 128$. A rough estimation of the minimum size required to reduce the error $L \approx 1/\Delta T$

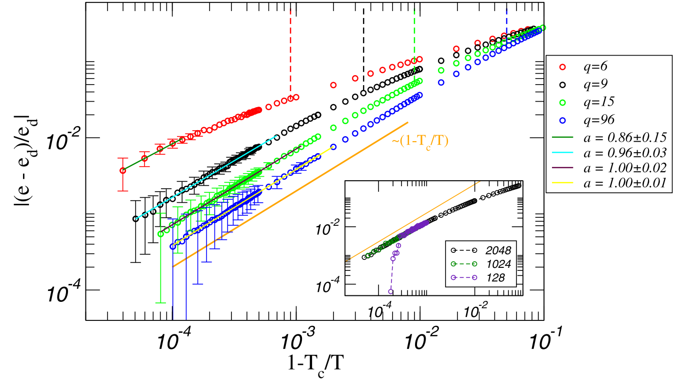


Fig. 6. (Color online.) Log–log plot of energy differences versus temperatures $T > T_c$ for various q. Data correspond to averages over 20 samples of systems size $L = 2048$, equilibration times ranging from $5 \times 10^4[MCS]$ to $2 \times 10^5[MCS]$ and measurement times of $5 \times 10^4[MCS]$, with sampling every $100[MCS]$. Error bars were estimated considering a 90% confidence interval (only some representative error bars are shown for clarity). Full color lines are power-law fits of the form $|(e - e_d)/e_d| = A(1 - T_c/T)^a$ (resulting exponents a are showed in the labels). Dashed vertical lines of different colors correspond to $T = T_c + \Delta T(q)$, with $\Delta T = T_c - T^-_{sp}$ and T^-_{sp} from Eq. (16). The inset shows $q = 9$ curves for different system sizes, the full orange curve indicates the slope 1.

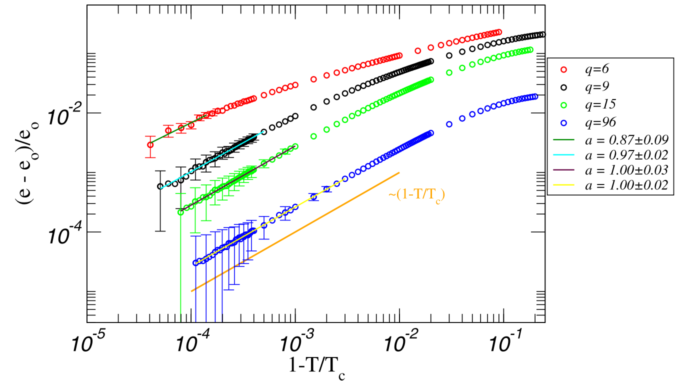


Fig. 7. (Color online.) Log–log plot of energy differences versus temperatures $T < T_c$ for various q. Data correspond to averages over 20 samples of systems size $L = 2048$, equilibration times ranging from $5 \times 10^4[MCS]$ to $2 \times 10^5[MCS]$ and measurement times of $5 \times 10^4[MCS]$, with sampling every $100[MCS]$. Error bars were estimated considering a 90% confidence interval (only some representative error bars are shown for clarity). Full color lines are power-law fits of the form $(e - e_o)/e_o = A(1 - T/T_c)^a$ (resulting exponents a are shown in the labels).

predicts $L = 400$. We see that this finite size effect is suppressed for sizes $L \geqslant 1000$. Moreover, further increase of the system size does not change the behavior of the curves close to T_c.

We have no estimations for T^+_{sp} for arbitrary values of q, but a close look to the curves in Fig. 2 suggest that T^+_{sp} is closer to T_c than T^-_{sp} is. This is consistent with the behavior observed in Fig. 7, where crossovers occur closer to T_c than in Fig. 6.

Our results for $q = 6$ are not conclusive. For instance, in the high energy branch we observe the previously discussed crossover, but the slope changes from 0.6 to 0.8. Such variation is of the same order of the fitting error below the crossover. This is because statistical fluctuations in the energy become very important at the required temperature resolution level ($\Delta T/T_c \leqslant 10^{-4}$), as can be seen in Fig. 6. Hence, to obtain a clear answer a very large sample size (one can roughly estimate ∼2000) and probably a larger system size are needed. In fact, we performed simulations with a sample size 50 (for $L = 2048$), without any improvement in the results. We even simulate systems of $L = 8192$ with a sample size on the order of 10 with no appreciable change.

Author's personal copy

The situation is more difficult for the low energy branch, where no clear evidence of crossover is observed (see Fig. 7). However, one could expect the existence of an upper spinodal temperature T_{sp}^{+} located closer to T_c than the lower one T_{sp}^{-} and therefore a higher temperature resolution (together with larger system and sampling sizes) would be needed to elucidate whether there is metastability or not.

7. Discussion

We implemented a CUDA-based parallel Monte Carlo algorithm to simulate the Statistical Mechanics of the q-state Potts model. The code allows a speedup (compared with an optimized serial code running on a CPU) from 42× in the GTX 280 card up to 155× in a GTX 480, with an average time per spin flip of 0.54 ns down to 0.147 ns respectively. Those times are of the same order of previous implementations in the simpler case of the Ising model, without the usage of sophisticated programming techniques, such as multi-spin coding. Besides the speedup, the present algorithm allows the simulation of very large systems in very short times, namely $\sim 10^9$ spins with an average time per *MCS* of 0.15 s. Such performance is almost independent of the value of q. The key factors to achieve those numbers is the per-thread independent RNG that is fast and takes only a few registers, the framing scheme that increases the amount of computation done by each thread and at the same time it bounds the number of independent RNG needed, and finally the cell-packing mapping that orders the memory access.

The possibility of performing high speed simulations at large enough system sizes allowed us to study the metastability problem in the two-dimensional system based on Binder's criterion, namely, on the existence or not of specific heat singularities at spinodal temperatures different from the transition one (but very close to). Our results provide a positive numerical evidence about the existence of metastability on very large systems, at least for $q \geqslant 9$.

Even when our results for $q = 6$ suggest the same behavior as for larger values of q, they could also be consistent with the absence of metastability. Hence, one cannot exclude the existence of a second critical value $4 < q^* \leqslant 9$ such that metastability disappears when $4 < q < q^*$.

Although the present implementation was done for a two-dimensional system with nearest neighbors interactions (checkerboard update scheme), its generalization to three-dimensional systems and/or longer ranged interactions is feasible, but some features should be adjusted. For the generalization to the 3D case, the checkerboard scheme defining two independent sub-networks persists, however the cell-packing scheme should be updated conveniently. For the 2D case with first and second neighbors interactions, there are nine independent sub-networks to update instead of two. The combination of both generalizations is direct.

The present implementation is based on the simplest single-spin flip algorithm namely, Metropolis. Its extension to more sophisticated single-spin flip algorithms (see for example Refs. [50, 51]) is also straightforward and represents an interesting prospective in the field. In particular, temperature reweighting [52] or other histogram-based techniques (see for example [47]) can be implemented by keeping track of the energy changes at each spin flip for each step, instead of making the calculation of the energy over the whole system at each step. This kind of tracking could be done without loose of performance by implementing a parallel accumulation of local energy changes *on-the-fly* taking advantage of the GPU's hierarchic memory scheme.

Besides its theoretical interest, the large-q Potts model (or minor variations of it) is widely used for simulating the dynamics of a large variety of systems, such as soap bubbles and foam [53,54], grain growth [55,56], gene segregation [57], biological cells [58], tumor migration [59], image segmentation [60], neural networks [61] and social demographics behavior [62,63]. The present implementation of the Potts model on GPUs, or easy modifications of it, would result helpful for some of the above cited applications. The possibility of simulating bigger systems and having results faster than usual should be welcomed in the statistical physics community. Our CUDA code is available for download and use under GNU GPL 3.0 at our Group webpage [44].

Acknowledgements

We thank C. Bederián for very useful suggestions. We would also like to thank A. Kolton and C. Sánchez for kindly giving access to a GTX 470 and a GTX 480 respectively, for benchmarks and simulations. Fruitful discussions and suggestions from O. Reula and M. Bellone are also acknowledged. This work was partially supported by grants from FONCYT/ANPCYT (Argentina), CONICET (Argentina), SeCyT, Universidad Nacional de Córdoba (Argentina) and NVIDIA professor partnership program.

References

[1] K.H. Fischer, J.A. Hertz, Spin Glasses, Cambridge University Press, 1993.
[2] W. Kob, in: Lecture Notes for "Slow Relaxations and Nonequilibrium Dynamics in Condensed Matter", in: Les Houches Session, vol. LXXVII, 2003, p. 199.
[3] K. Binder, W. Kob, Glassy Materials and Disordered Solids: An Introduction to Their Statistical Mechanics, World Scientific Publishing, Singapore, 2005.
[4] L.F. Cugliandolo, Phys. A: Stat. Mech. Appl. 389 (2010) 4360–4373.
[5] T. Preis, P. Virnau, W. Paul, J.J. Schneider, J. Comput. Phys. 228 (2009) 4468–4477.
[6] B. Block, P. Virnau, T. Preis, Comput. Phys. Comm. 181 (2010) 1549–1556.
[7] M. Bernaschi, G. Parisi, L. Parisi, Comput. Phys. Comm. 182 (2011) 1265–1271.
[8] K. Hawick, A. Leist, D. Playne, Int. J. Parallel Progr. (2010) 1–19.
[9] M. Weigel, Comput. Phys. Comm. 182 (2011) 1833–1836.
[10] M. Weigel, arXiv:1101.1427, 2011.
[11] F. Herrmann, J. Silberholz, M. Bellone, G. Guerberoff, M. Tiglio, Classical Quantum Gravity 27 (2010) 032001.
[12] J. Tickner, Comput. Phys. Comm. 181 (2010) 1821–1832.
[13] M. Clark, R. Babich, K. Barros, R. Brower, C. Rebbi, Comput. Phys. Comm. 181 (2010) 1517–1528.
[14] S. Ryoo, C.I. Rodrigues, S.S. Baghsorkhi, S.S. Stone, D.B. Kirk, W. mei W. Hwu, in: Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (23th PPOPP'2008), ACM SIGPLAN, 2008, pp. 73–82.
[15] F.Y. Wu, Rev. Modern Phys. 54 (1982) 235.
[16] K. Binder, J. Stat. Phys. 24 (1981) 69.
[17] J. Meunier, A. Morel, Eur. Phys. J. B 13 (2000) 341.
[18] A. Petri, M. Ibañez de Berganza, V. Loreto, Philos. Mag. 88 (2008) 3931–3938.
[19] A. Bazavov, B.A. Berg, S. Dubey, Nucl. Phys. B 802 (2008) 421–434.
[20] E. Loscar, E. Ferrero, T. Grigera, S. Cannas, J. Chem. Phys. 131 (2009) 024120.
[21] J. Vinals, M. Grant, Phys. Rev. B 36 (1987) 7036.
[22] G.S. Grest, M.P. Anderson, D.J. Srolovitz, Phys. Rev. B 38 (1988) 4752–4760.
[23] C. Sire, S. Majumdar, Phys. Rev. E 52 (1995) 244.
[24] E. Ferrero, S. Cannas, Phys. Rev. E 76 (2007) 031108.
[25] M.P.O. Loureiro, J.J. Arenzon, L.F. Cugliandolo, A. Sicilia, Phys. Rev. E 81 (2010) 021129.
[26] S. Rutkevich, Int. J. Mod. Phys. 13 (2002) 495.
[27] B. Bauer, E. Gull, S. Trebst, M. Troyer, D.A. Huse, J. Stat. Mech. Theory Exp. 2010 (2010) P01020.
[28] R.J. Baxter, J. Phys. C 6 (1973) L445.
[29] T. Kihara, Y. Midzuno, T. Shizume, J. Phys. Soc. Japan 9 (1954) 681.
[30] R.J. Baxter, J. Phys. A 15 (1982) 3329.
[31] A. Velytsky, B.A. Berg, U.M. Heller, Nucl. Phys. B (Suppl.) 119 (2003) 861–863.
[32] M. Ibañez de Berganza, V. Loreto, A. Petri, arXiv:0706.3534, 2007.
[33] H. Meyer-Ortmanns, Rev. Modern Phys. 68 (1996) 473–598.
[34] F. Karsch, S. Stickan, Phys. Lett. B 488 (2000) 319–325.
[35] C. Bonati, M. D'Elia, arXiv:1010.3639, 2010.
[36] W. Press, et al., Numerical Recipes in C, second edition, Cambridge University Press, 1992.
[37] B.W. Kernigham, D.M. Ritchie, The C Programming Language, Prentice Hall, Inc., Englewood Cliffs, NJ, 1978.
[38] D.B. Kirk, W. mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, Morgan Kaufmann Publishers, 2010.
[39] G. Marsaglia, J. Mod. Appl. Stat. Methods 2 (2003) 2–13.
[40] S. Andersson-Engels, Erik Alerstam, Tomas Svensson, CUDAMCML User manual and implementation notes, 2009.

Author's personal copy

[41] M. Harris, Optimizing cuda, SC07 Tutorial, 2007.
[42] Z. Merali, Nature 467 (2010) 775–777.
[43] N. Barnes, Nature 467 (2010) 753.
[44] http://www.famaf.unc.edu.ar/grupos/GPGPU/Potts/CUDAPotts.html, 2010, Q-State Potts model for CUDA site.
[45] W. Janke, Phys. Rev. B 47 (1993) 14757–14770.
[46] K. Binder, Rep. Progr. Phys. 60 (1997) 487.
[47] D.P. Landau, K. Binder, A Guide to Monte Carlo Simulations in Statistical Physics, third edition, Cambridge University Press, 2009.
[48] http://www.cond-mat.physik.uni-mainz.de/~weigel/research/gpu-computing, Simulating spin models on GPU site, 2010.
[49] J.J. Tapia, R.M. D'Souza, Comput. Phys. Comm. 182 (2011) 857–865.
[50] D. Loison, C.L. Qin, K.D. Schotte, X.F. Jin, Eur. Phys. J. B 41 (2004) 395–412.
[51] H. Suwa, S. Todo, Phys. Rev. Lett. 105 (2010) 120603.
[52] A.M. Ferrenberg, R.H. Swendsen, Phys. Rev. Lett. 63 (1989) 1658.
[53] J.A. Glazier, D. Weaire, J. Phys.: Condens. Matter 4 (1992) 1867–1894.
[54] S. Sanyal, J.A. Glazier, J. Stat. Mech. Theory Exp. 2006 (2006) P10008.
[55] D. Weaire, J.A. Glazier, Mater. Sci. Forum 94–96 (1992) 27–38.
[56] G.L. Thomas, R.M.C. de Almeida, F. Graner, Phys. Rev. E 74 (2006) 021407.
[57] K.S. Korolev, M. Avlund, O. Hallatschek, D.R. Nelson, Rev. Modern Phys. 82 (2010) 1691–1718.
[58] F. Graner, J.A. Glazier, Phys. Rev. Lett. 69 (1992) 2013–2016.
[59] S. Turner, J.A. Sherratt, J. Theoret. Biol. 216 (2002) 85–100.
[60] F. Bentrem, Centr. Eur. J. Phys. 8 (2010) 689–698.
[61] V. Kryzhanovsky, in: Artificial Neural Networks – ICANN 2008, vol. 5164, 2008, pp. 72–80.
[62] C. Schulze, Centr. Eur. J. Phys. 16 (2005) 351–355.
[63] V.A. Traag, J. Bruggeman, Phys. Rev. E 80 (2009) 036115.