JCTC Journal of Chemical Theory and Computation

Simulating Lattice Spin Models on Graphics Processing Units

Tal Levy, Guy Cohen, and Eran Rabani*

School of Chemistry, The Sackler Faculty of Exact Sciences, Tel Aviv University, Tel Aviv 69978, Israel

Abstract: Lattice spin models are useful for studying critical phenomena and allow the extraction of equilibrium and dynamical properties. Simulations of such systems are usually based on Monte Carlo (MC) techniques, and the main difficulty is often the large computational effort needed when approaching critical points. In this work, it is shown how such simulations can be accelerated with the use of NVIDIA graphics processing units (GPUs) using the CUDA programming architecture. We have developed two different algorithms for lattice spin models, the first useful for equilibrium properties near a second-order phase transition point and the second for dynamical slowing down near a glass transition. The algorithms are based on parallel MC techniques, and speedups from 70- to 150-fold over conventional single-threaded computer codes are obtained using consumer-grade hardware.

I. Introduction

In most common cases, computer programs are written serially: to solve a problem, an algorithm is constructed and implemented as a serial stream of coded instructions. These instructions are executed on a central processing unit (CPU) on one computer. Momentarily disregarding specific cases for which modern CPU hardware is optimized, only one instruction may be executed at a time; on its termination, the next instruction begins its execution. On the other hand, parallel computing involves the simultaneous use of multiple computational resources. In recent years, massively parallel computing has become a valuable tool, mainly due to the rapid development of the graphics processing unit (GPU), a highly parallel, multithreaded, many core processor. However, due to the specialized nature of the hardware, the technical difficulties involved in GPU programming for scientific use stymied progress in the direction of using GPUs as general-purpose parallel computing devices. This situation has begun to change more recently, as NVIDIA introduced CUDA, a general purpose parallel computing architecture with a new parallel programming model and instruction set architecture (similar technology is also available from ATI but was not studied by us). CUDA comes with a software environment that allows developers to use C (C++, CUDA FORTRAN, OpenCL, and DirectCompute are now also supported[1]) for program development, exposing an application programming interface and language extensions that allow access to the GPU without specialized assembler or graphics-oriented interfaces. Nowadays, there are quite a few scientific applications that are running on GPUs; this includes quantum chemistry applications,[2,3] quantum Monte Carlo simulations,[4,5] molecular dynamics,[6–13] hydrodynamics,[14] classical Monte Carlo simulations,[15–17] and stochastic processes.[18–20]

In this work, we revisit the problem of parallel Monte Carlo simulations of lattice spin models on GPUs. Two generic model systems were considered:

(1) The two-dimensional (2D) Ising model[21] serves as a prototype model for describing critical phenomena and equilibrium phase transitions. Numerical analysis of critical phenomena is based on computer simulations combined with finite size scaling techniques.[22] Near the critical point, these simulations require large-scale computations and long averaging, while current GPU algorithms are limited to small lattice sizes[16] or to spin 1/2 systems.[17] Thus, in three-dimensions (3D), the solution of lattice spin models still remains a challenge in statistical physics.

(2) The North-East model[23] serves as a prototype for describing the slowing of glassy systems. The computational challenge here is mainly to describe correlations and fluctua-

* Corresponding author e-mail: rabani@tau.ac.il.

Table 1. Summary of NVIDIA's Different Architectures

	G80	GT200	Fermi
maximum number of multiprocessors	16	30	16
scalar cores per multiprocessor	8	8	32
double precision capability	none	30 FMA ops/clock	256 FMA ops/clock
special function units	2	2	4
warp schedulers per multiprocessor	1	1	2
shared memory	16 kB	16 kB	48 kB
concurrent kernels	1	1	up to 16

tions on very long time scales, as the system approaches the glass temperature (concentration).[23] As far as we know, simulations of facilitated models on GPUs have not been explored so far, but their applications on CPU have been discussed extensively.[24–27]

We develop two new algorithms useful to simulate these lattice spin models. For certain conditions, we obtain speedups of 2 orders of magnitude in comparison with serial CPU simulations. We would like to note that in many GPU applications, speedups are reported in Gflops, while in the present work, we report speedups of the actual running time of the full simulation. The latter provides a realistic estimate of the performance of the algorithms. This Article is organized as follows: Section II comprises a brief introduction to GPUs and summarizes their main features. Section III contains a short overview of lattice spin models and the algorithms developed. Section IV includes a short discussion of the random number generator used in this work. Section V presents the results and the comparisons between CPU and GPU performance. Finally, section VI concludes the Article.

II. GPU Architecture

To facilitate an understanding of GPU programming, we provide a sketch of NVIDIA's GPU device architecture. This is important for the development of the algorithms reported below, and in particular for understanding the logic and limitations behind our approach. By now, three generations of GPUs have been released by NVIDIA (G80, GT200, and the latest architecture codename "Fermi"). In Table 1 we highlight the main features and differences between generations.

A. Hardware Specifications. On a GPU device one finds a number of scalar multiprocessors (SMs). Each multiprocessor contains 8/32 (architecture dependent) scalar processor cores (SPs), a multithreaded instruction unit (MTIU), special function units for transcendental numbers and the execution of transcendental instructions such as sine, cosine, reciprocal, and square root, 32-bit registers, and the shared memory space. In general, unlike a CPU, a GPU is specialized for compute-intensive, highly parallel computation, exactly the task graphics rendering requires, and is therefore designed such that more transistors are devoted to data processing rather than data caching and flow control. This is schematically illustrated in Figure 1 and makes GPUs less general-purpose but highly effective for data-parallel computation with high arithmetic intensity. Specifically, they are opti-

mized for computations where the same instructions are executed on different data elements (often called single instruction multiple data, or SIMD) and where the ratio of arithmetic operations to memory operations is high. This puts a heavy restriction on the types of computations that optimally utilize the GPU, but in cases where the architecture is suitable to the task at hand it speeds up the calculations significantly.

B. Memory Architecture. NVIDIA's GPU memory model is highly hierarchical and is divided into several layers.

Registers. This is the fastest form of memory on the GPU. Usually automatic variables declared in a kernel reside in registers, which provide very fast access.

Local Memory. Local memory is a memory abstraction that implies "local" in the scope of each thread. It is not an actual hardware component of the SM. In fact, local memory resides in device memory allocated by the compiler and delivers the same performance as any other global memory region.

Shared Memory. This can be as fast as a register when there are no bank conflicts or when reading from the same address. It is located on the SM and is accessible by any thread of the block from which it was created. It has the lifetime of the block (which will be defined in the next subsection).

Global Memory. This is potentially about 400−600 times slower than register or shared memory. This module is accessible by all threads and blocks. Data transfers from and to the GPU are done from global memory.

Constant Memory. Constant memory is read only from kernels and is hardware optimized for the case when all threads read the same location (i.e., it is cached). If threads read from multiple locations, the accesses are serialized.

Texture Memory. Graphics processors provide texture memory to accelerate frequently performed operations. The texture memory space is cached. The texture cache is optimized for 2D spatial locality, so threads of the same warp that read texture addresses that are close together will achieve best performance. Constant and texture memory reside in device memory.

C. CUDA Programing Model. As has been mentioned, CUDA includes a software environment that allows developers to develop applications mostly in the C programming language. C for CUDA extends C by allowing the programmer to define C functions (kernels) that, when called, are executed N times in parallel by N different CUDA threads on the GPU scalar processors. Before invoking a kernel, the programmer needs to define the number of threads (N) to be created. Moreover, the programmer needs to decide into how many blocks these threads will be divided. When the kernel is invoked, blocks are distributed evenly to the different multiprocessors (hereby, having p multiprocessors requires a minimum of p blocks to achieve 100% utilization of the GPU). The blocks might be one-, two-, or three-dimensional with up to 512 threads per block. Blocks are organized into a one- or two-dimensional grid containing up to 65 535 blocks in each dimension. Each of the threads within a block that executes a kernel is given a unique thread ID. To differentiate between two threads from two different blocks,
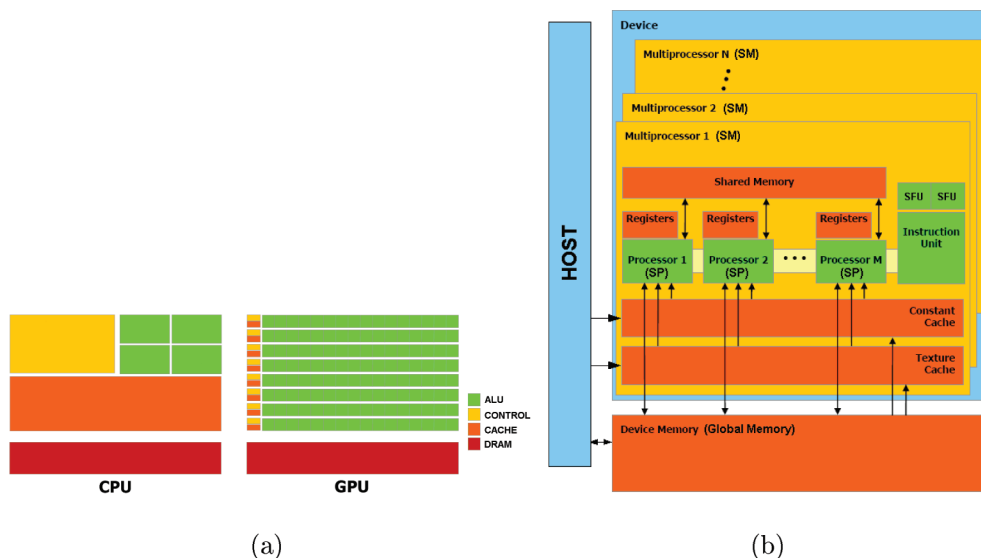
Figure 1. (a) A schematic delineation that illustrates the ratio of transistors devoted to data processing (ALUs) to those of data caching and flow control in CPUs and GPUs. This is a manifestation of the different philosophies between the "all purpose" CPU to the "dedicated purpose" GPU. (b) A visual hardware model that describes the memory architecture and hierarchy of a GPU device (courtesy of NVIDIA).

each of the blocks is given a unique ID as well. Threads within a block can cooperate among themselves by sharing data through shared memory. Synchronizing thread execution is possible within a block, but different blocks execute independently. Threads belonging to different blocks can execute on different multiprocessors and must exchange data through the global memory. There is no efficient way to synchronize block execution, that is, in which order or on which multiprocessor they will be processed; thus, an efficient algorithm should avoid communication between blocks as much as possible.

Much of the challenge in developing an efficient GPU algorithm involves determining an efficient mapping between computational tasks and the grid/block/thread hierarchy. The multiprocessor maps each thread to one scalar processor, and each thread executes independently with its own instruction address and register state. The multiprocessor SIMT (single instruction−multiple threads) unit creates, manages, schedules, and executes threads in groups of 32 called warps. When a multiprocessor is given one or more thread blocks to execute, it splits them into warps that get scheduled by the SIMT unit. The SIMT unit selects a warp that is ready to execute and issues the next instruction to the active threads of the warp. A warp executes one common instruction at a time, so full efficiency is realized when all threads of a warp agree on their execution path (on the G80 and GT200 architectures, it takes 4 clock cycles for a warp to execute, while on the Fermi architecture it takes only 1 clock cycle). If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path. When all paths terminate, the threads converge back to the same execution path. Branch divergence occurs only within a warp: different warps execute independently regardless of whether they are executing common or disjointed code paths. More information can be found in ref 1.

III. Lattice Spin Models

We consider a general lattice model of spins that are placed on a square lattice of dimensions $d = 1,2,3...$ The spins may be of any dimension and may acquire continuous or discrete values. In the applications reported below, we focus, for simplicity, on the case where the spins at lattice site i take discrete values of $s_i = \pm 1$, but this can be easily extended to any spin dimension and value. The interactions between the spins are given by the Hamiltonian:

$$H = \sum_{i \neq j} J_{ij} s_i s_j + \sum_i B_i s_i \qquad (1)$$

In the above equation, the first sum is usually carried over nearest neighbors only (which we will denote $\langle ij \rangle$). J_{ij} is the interaction parameter and may be constant, discrete, or continuous, and B_i is an external field. The above Hamiltonian can be used to study equilibrium properties as in the Ising model and spin glass models (Edwards−Anderson model,[28] Sherrington−Kirkpatrick model,[29] random orthogonal model,[30] etc.), or the dynamic behavior as in facilitated spin models (Fredrickson and Andersen,[31] Jackle−Eisinger North-East model,[23,32] etc.). Simulations of such systems are based on Monte Carlo techniques.[33] In most CPU implementations, the algorithms for equilibrium or dynamic simulations do not differ significantly. However, as will become clear below, they become very different when implemented on a GPU. Although the implementations described below are for two simple cases, the extension of our approach to the spin glass models mentioned above or to other facilitated spin models is straightforward.

A. Monte Carlo Simulation of the 2D Ising Model. The simplest 2D Ising model[22] describes N magnetic dipoles (or spins) placed on a 2D square lattice with one spin per cell. We limit the discussion to the spin 1/2 case, where each spin has only two possible orientations, "up" and "down". Each spin interacts with its nearest neighbors only, with a
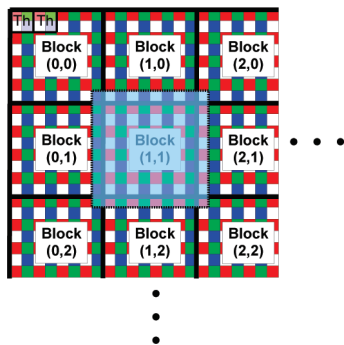
Figure 2. Illustration of the algorithm for an Ising model implemented on a GPU device. Every block handles a sublattice. Every thread is assigned 4 spin sites according to its ID and block. To improve memory access performance, each block copies its sublattice on to shared memory. Because a spin needs its 4 nearest neighbors to update, extra boundary sites are copied as well (blue-tinted square).

fixed interaction strength. In the absence of an external magnetic field, the Hamiltonian is given by

$$H = -J \sum_{\langle ij \rangle} s_i s_j \quad (2)$$

where $\langle ij \rangle$ represents a sum over nearest neighbors and J determines the energy scale. Monte Carlo simulation techniques based on the Metropolis algorithm[34] are perhaps the most popular route to obtain the thermodynamic properties of this model. In a CPU implementation of the Metropolis algorithm, a spin is selected at random, and an attempt to flip the spin is accepted with the Metropolis probability P_{acc} = min[1,exp($-\Delta H/T$)], where T is the temperature in units of energy. In the GPU algorithm developed here, we will take advantage of the fact that spins interact only with their nearest neighbors, and thus the problem can be divided into noninteracting domains. The generalization to the case of finite interacting regions is straightforward. The algorithm is as follows:

(1) Randomly initialize the lattice (this is done on the CPU).

(2) Copy lattice to the GPU.

(3) Divide the lattice into Q sublattices, each with P spins.

(4) A grid of Q thread-blocks is formed. Every thread-block contains $P/4$ threads.

(5) Every block copies a sublattice including its boundaries from the global memory to the shared memory. To avoid bank conflicts and save precious shared memory space, the short data type was used to form the lattice (Figure 2).

(6) Within the block, every thread is in charge of 4 spin sites (a sub block of 2 × 2). At first all red spins are updated (Figure 3a); that is, all threads are active. Once a thread finished updating the red spin, it continues to update its blue spin (Figure 3b). Because no native block synchronization exists, before updating the remaining spins, we make sure that all blocks finished the first two steps. This is done by recopying the data from the shared memory (boundaries excluded) onto the global memory and ending the kernel.

(7) Relaunch the kernel with the configuration generated in the previous step. Redo steps 5 and 6, only this time the green and white spins are being updated (Figure 3c).

(8) Recopy data to global memory.

This completes one Monte Carlo step (or one lattice sweep). In our implementation, we chose sublattices of 32 × 32 in size and blocks of 256 threads, as this choice turned out to be the most efficient. Using more threads does in fact reduce the time it takes the block to copy a sublattice to the shared memory, but then the ratio of arithmetic operations to memory operations is low and performance is poor. To obtain thermodynamic average properties, we use the fact the CPU and GPU can work in parallel and the lattice is copied from the device to the host from time to time, so averages can be calculated on the CPU while the GPU continues to sweep the lattice. We note in passing that similar algorithms have been proposed by Tobias et al.[16] and Block et al.[17] The former approach is restricted to lattices with up to 1024 × 1024 spins in 2D (assuming spins are stored as integer data type), whereas the algorithm presented in this work is applicable to larger systems, is designed for coalesced global memory access, and avoids bank conflicts, all of which make better use of the GPU architecture. The limit of system size in our approach is related to the size of the global memory on the GPU, which is typically on the order of 1−4 GB. This amounts to maximum system sizes of 20 000 × 20 000 to 40 000 × 40 000 spins. The algorithm of Block et al.[17] is applicable to much larger systems and deals with the issue of multi-GPU programming, but is currently restricted to spin 1/2 systems, while the algorithm presented in this work is suitable for the more generalized Potts model[35] and gives approximately the same speedups in comparison to an equivalent CPU code.

To acquire the correct thermodynamic equilibrium state, the chosen set of Monte Carlo moves must satisfy either detailed balance or the weaker balance condition. On the CPU, detailed balance is rigorously satisfied when randomly selecting a spin within the Metropolis algorithm. For the GPU, however, the approach we developed breaks detailed balance, and only balance is satisfied.[36,37] This is a sufficient condition to ensure that the algorithm correctly samples the Boltzmann distribution.

B. The North-East Model. The North-East model is based on the Ising model Hamiltonian with special constraints that are used to model facilitated dynamics.[23,31,38] The Hamiltonian is given by:

$$H = J \sum_{\langle i,j \rangle} s_i s_j + B \sum_i s_i \quad (3)$$

where s_i, J, B, and $\langle ij \rangle$ are described above. What makes this model different from the previous one is a constraint imposed on the transition probability to flip a spin. This probability is zero, unless the spin's upper (north) and right (east) neighbors point "up". In the latter case, one accepts a flip with the same Metropolis probability given by P_{acc} = min[1,exp($-\Delta H/T$)]. Thus, the thermodynamics of the model is the same as the Ising model, but the Monte Carlo dynamics generated by the above rule is quite different. In most applications reported in the literature, one takes $J = 0$. For $J \neq 0$, the dynamics generated by this model are richer and show an interesting re-entrant transition.[39] When $J = 0$, the thermodynamics is trivial, and the coupling between neigh-
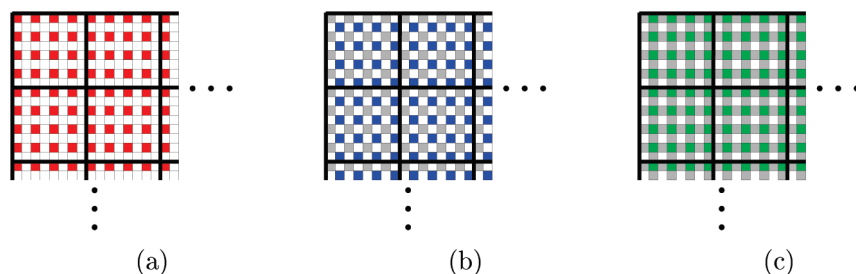
Figure 3. As native block synchronization is not available on a GPU, one must terminate the kernel's execution to achieve it. This is only necessary after the red and blue spins where updated (as can be clearly seen, they do not interact) and before moving on to the green and white spins, which in turn do interact with the aforementioned updated red and blue spins.

boring spins depends only on the aforementioned dynamical constraint. The lattice is initialized so spins point "up" with the probability c, which is also the equilibrium density of spins pointing "up", and can be expressed as $c = 1/z \, e^{-1/T}$, where the partition function is $z = e^{1/T} + e^{-1/T}$ and T is the unitless temperature. Because of the dynamical constraints, if c is too low, one finds domains of spins that are stuck. For a high value of c on the other hand, all spins are flippable. As a consequence, there is a critical concentration c^* below which the system is not ergodic. This transition from ergodic to nonergodic behavior is modeled by the spin−spin autocorrelation function:

$$\Phi(t) = \sum_i \frac{\langle s_i(t) \cdot s_i(0)\rangle - \langle s_i\rangle^2}{1 - \langle s_i\rangle^2} \tag{4}$$

where $\langle s_i\rangle = 2c - 1$ is the average spin polarization and $s_i(t)$ is the spin polarization at Monte Carlo step t for site i. We expect the function to decay to zero for an initial concentration $c > c^*$ and to decay to a finite value f (which is the fraction of spins that are stuck) for $c < c^*$. The CPU implementation of the North-East model is identical to that described for the Ising model, with the additional constraint for the flipping probability.

The GPU implementation for this model, similar as it may seem to the Ising model, is a bit cannier. We found out that applying the checkerboard algorithm (described in section III.A) does not yield the same relaxation dynamics as the serial CPU implementation. This is understandable, because this model imitates a diffusion process: for a spin to be able to change its configuration, it is necessary that its north and east neighbors point up. If they do not, they in turn will also need their neighbors to point up to be able and change their configuration. Equilibration takes place by an up spin diffusion from "north-east" to "south-west". By sequentially (instead of randomly) sweeping the lattice, we change the dynamics of this process. Such being the case, the corrected algorithm we developed is:

(1) Initialize the lattice so spins point up with probability c (on the CPU).

(2) Copy lattice to the GPU.

(3) Divide the lattice into Q sublattices, each with P spins.

(4) A grid of Q thread-blocks is formed. Every thread-block contains $P/4$ threads.

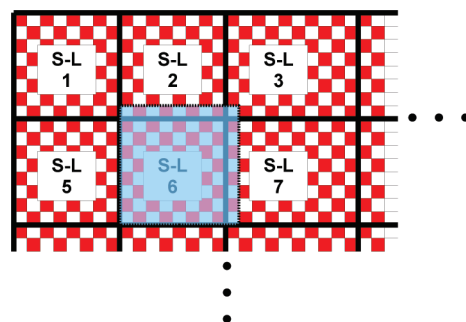(5) Blocks then randomly pick a sublattice (Figure 4) in such a way that two different blocks cannot pick the same

Figure 4. Although this figure looks similar to the Ising model implementation, in this case sublattices and spins are chosen randomly. This preserves the dynamical behavior in comparison to CPU algorithms. Note that when copying the data to the shared memory, only the upper and right boundaries are necessary.

sublattice. Every block copies a sublattice including its boundaries from the global memory to the shared memory.

(6) Within each block, λ threads concurrently pick λ spins from the sublattice randomly and update them. λ is chosen to be a small fraction of P.

(7) Synchronize the block.

(8) Reiterate steps 6 and 7 q times, such that $q \cdot \lambda = P/2$.

(9) Copy back data from shared memory to global memory (to allow block synchronization).

(10) Relaunch the kernel with the same configuration and redo steps 5−8.

(11) Recopy data to global memory.

This completes one lattice sweep. Again, we use the fact the CPU and GPU can work in parallel, and the lattice is copied from the device to the host from time to time to store the spin's configuration for the computation of the autocorrelation function.

The algorithm presented here does not preserve detailed balance nor the weaker balance condition, but because we are interested in its dynamics the given algorithm is correct. One should note that to obtain the correct dynamics an initialization of the parameter λ is needed. It is obvious that in the limit where $Q = \lambda = 1$ the algorithm is in fact serial and preserves detailed balance (results for this limit are given by the black line in Figure 5). We can now use this result as a reference and increase the number of threads and blocks that are working in parallel. There is an upper limit above which the results will diverge from the desired reference and the dynamics will no longer be correctly reproduced. Once
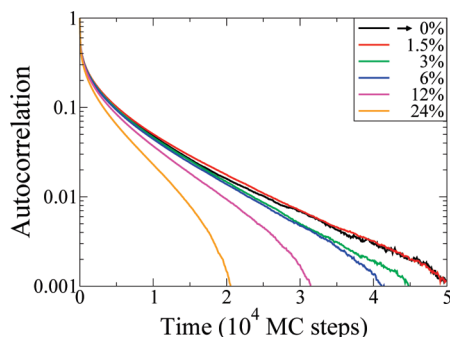
Figure 5. Convergence tests of the parameter λ for the North-East model at $c = 0.42$. The solid black line represents the reference value, which was obtained by running the simulation on a single thread−single block. This result is identical to the serial CPU result. The best performance is achieved for sublattices of 32×32 spins and for $\lambda = 16$ threads, that is, attempts to flip only $\sim$1.5% $\approx 16/(32 \times 32)$ of the spins inside a sublattice. One can see that for higher values of λ, the correlation diverges from the correct result.

λ has been evaluated, the simulation can be performed. In Figure 5 we provide a self-consistent test on the value of λ for the North-East model at $c = 0.42$.

It is possible to write an algorithm that maintains the weaker balance condition and at the same time preserves the dynamical relaxation of the model. The drawback of such an algorithm, however, is that it is $\sim$2.5$\times$ slower then the one presented above. The only changes are in steps 6−9:

(6) Within each block, λ threads concurrently pick λ red spins (Figure 4) from the sublattice randomly and update them. λ is chosen to be a small fraction of P.

(7) Synchronize the block and pick λ white spins from the sublattice randomly and update them.

(8) Copy back data from shared memory to global memory (to allow block synchronization).

(9) Relaunch the kernel with the same configuration and redo steps 6−8 three more times (this will complete one MC step).

IV. Pseudo Random Numbers Generation

Monte Carlo simulations rely heavily on the availability of random or pseudorandom numbers. In this work, random numbers were used to determine whether spin flips are accepted or rejected, in accordance with the Metropolis algorithm. As is widely known, the use of a poor quality PRNG (pseudo random numbers generator) may lead to inaccurate results.[40] In the course of this work, three different PRNGs were utilized:

(1) The first is L'Ecuyer with Bays−Durham shuffle and added safeguards.[41] This PRNG has a long period (2×10^{18}) and is easy to implement on a CPU. Unfortunately, porting it to GPUs causes a dramatic decrease in performance: this is mostly due to the fact that the implementation of this algorithm requires too many registers.

(2) Second is the "minimal" random number generator of Park and Miller.[41] This PRNG has a period of 2×10^9 and is portable to GPUs. In practice, however, it proved to work poorly, and the GPU simulation results were in poor agreement with reference values obtained with better PRNGs.

(3) Finally, we have the linear congruential random number generator (LCRNG).[41] This PRNG has a period of $10^6 - 10^9$ and provided good results even for very long runs. The LCRNG algorithm is very easy to port to the GPU and has the advantage of being very fast, requiring only a few operations per call.

In our implementations (section III), each thread used a separate LCRNG, creating its own sequence of pseudorandom numbers with a unique seed. The sequence (of thread i) is created as follows:

$$x_{j+1}^{Th_i} = (a \cdot x_j^{Th_i} + c) \bmod m \tag{5}$$

$$x_{j+1}^{Th_i} = abs\left(\frac{x_{j+1}^{Th_i}}{2^{31}}\right) \tag{6}$$

An appropriate choice of the coefficients is responsible for the quality of the LCRNG. We used[41] $a = 1\,664\,525$, $c = 1\,013\,904\,223$, and $m = 2^{32}$. The different seeds were created per thread according to

$$x_0^{Th_{i+1}} = (16\,807 \cdot x_0^{Th_i}) \bmod m \tag{7}$$

with $x_0^{Th_0} = 1$. Similar PRNGs were used for other GPU applications.[16,17]

(a)   (b)   (c)
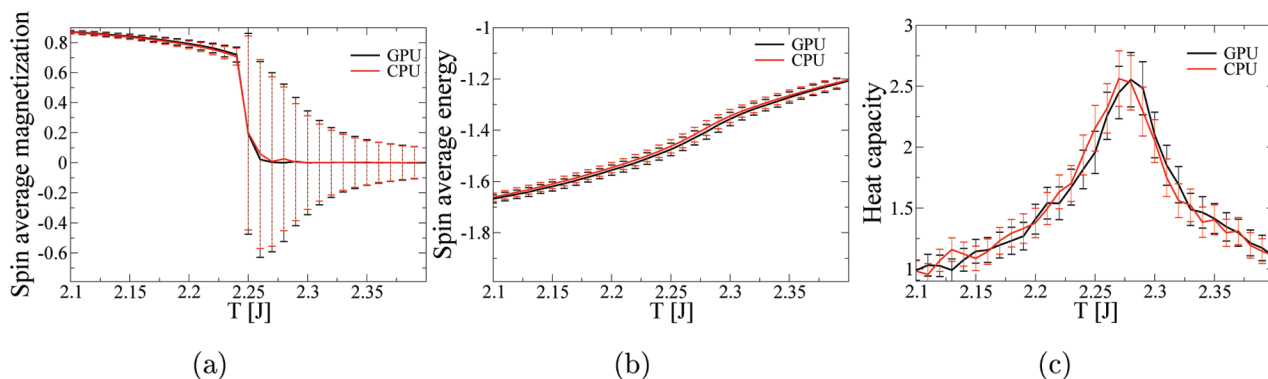
Figure 6. Comparison between the Monte Carlo simulation done on a CPU and on a GPU for a 128×128 spin−lattice. The temperature was stepwise reduced by 0.01 from $T = 2.4$ to $T = 2.1$. The critical temperature for this model is $T_c = 2/\log(1 + \sqrt{2})$. At each temperature, 2×10^7 sweeps through the lattice were performed, during which 10 000 different measurements were taken (after reaching equilibrium). In (a), we show the average spin magnetization, in (b) the average spin energy, and in (c) the heat capacity.
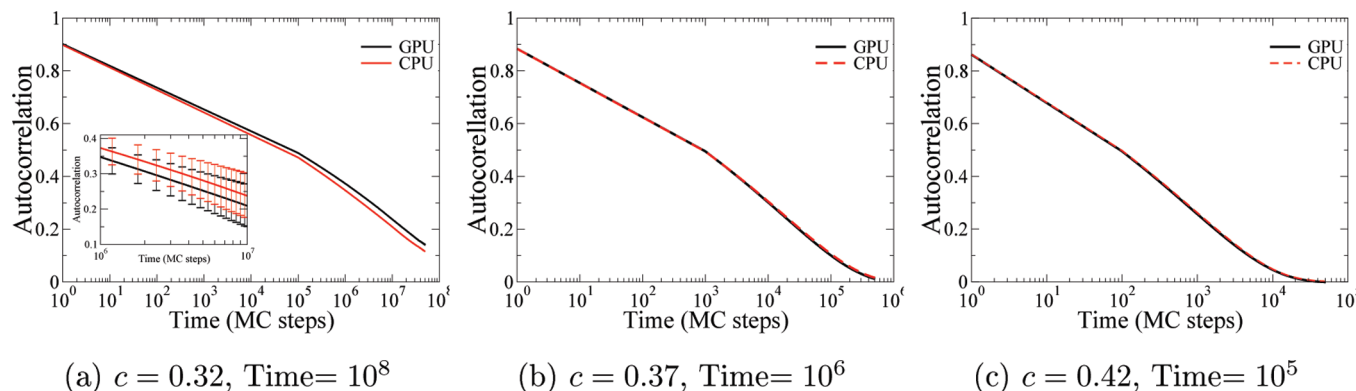
(a) $c = 0.32$, Time$= 10^8$ (b) $c = 0.37$, Time$= 10^6$ (c) $c = 0.42$, Time$= 10^5$

Figure 7. Comparison between spin−spin autocorrelation functions calculated from simulations done on a CPU and on a GPU for spin−lattices of different sizes and different initial concentrations. Results were obtained by averaging the autocorrelation functions of 30 different initial realization for every concentration. (a) Results for a 128×128 spin−lattice. As can be seen for low concentrations approaching c^*, relaxation time is very long (longer than the time we were willing to wait for the CPU results). Furthermore, at this concentration, more than 30 trajectories are required to average the autocorrelation function. (b and c) Results obtained from 512×512 spin−lattices.
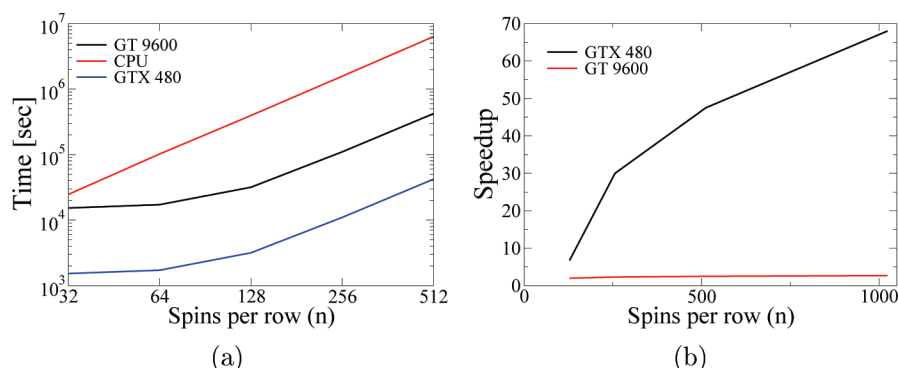

(a)   (b)

Figure 8. (a) Processing times for the 2D Ising model. Times are shown as a function of the number of spins n per row, which is related to the system size by $N = n^2$. A maximum acceleration factor of 15 was achieved (on the GT 9600GPU). We believe a higher value can be obtained for larger lattices. (b) Speedups of two different GPU cards versus the CPU serial implementation for the North-East model. Again, times are shown as a function of the number of spins n per row.

V. Results

For comparison, CPU codes were executed on a PC with Intel Core2 Duo E7400 @ 2.8GHz processor, Intel Raisin-City motherboard with Intel G41 chipset and Kingstone 2GB,800 MHz RAM (only a single core was used for the calculations). The operating system was CENTOS. Codes were compiled with an Intel C++ compiler (ICC) using all optimizations provided for best performance. Speedups reported here are given in terms of the complete application running time (from initialization until results are processed), rather than Gflops or spin updates per second. This choice is important, as it most closely describes the "real" gain in practical simulations by the use of a GPU rather than a CPU.

A. Ising Model. To verify the GPU implementation, we compared values of magnetization, energy, and heat capacity as a function of temperature (temperature was taken in energy units) between GPU and CPU versions. In Figure 6 are presented results from a 128×128 spin−lattice. We find the results to clearly agree. In terms of acceleration, we achieved a $15\times$ factor for lattices with 512×512 spins on the GT 9600 GPU (G80 architecture). Implementing the same code on the new GTX 480 GPU, we achieved a factor of $150\times$. This factor reduces to $2\times$ for small lattices ($32 \times$

32). The reason for this is that in small problems it becomes harder to hide memory access latencies, because there are not enough threads to execute between memory access operations. A theoretical analysis of our implementation shows that the GPU reaches full occupancy (a useful tool to check this is the "CUDA GPU Occupancy Calculator", which can be freely downloaded from ref 42). The 10-fold factor obtained by the GTX 480 in comparison with the GT 9600 is easily understood when taking into consideration that the GTX 480 has 15 SMs instead of 8 (on the GT 9600), 4 blocks can be active simultaneously instead of 3, and a warp executes in one clock cycle instead of 4. Thus, the relative speedup is $15/8 \cdot 4/3 \cdot 4/1 = 10$. From Figure 8a it is obvious that the GPU reaches full occupancy only for lattices bigger than 128×128. Note also that near the critical temperature, the fluctuations and noise increase. Because the present work is not concerned with determining the critical behavior, we have used the same number of Monte Carlo sweeps for all temperatures. Estimation of the critical behavior requires much longer runs, and perhaps also larger systems. In this respect, the GPU approach developed here provides the means to increase numerical accuracy by more than 1 order of magnitude (noise scales with the square root of the number

of MC steps), either by increasing the system size or by simulating longer Monte Carlo runs.

B. North-East Model. The North-East model has a critical concentration $c^* \approx 0.3$, below which the dynamics break ergodicity. In Figure 7 we show the results for spin−spin autocorrelation function for different concentrations above the critical value. Two lattice sizes were studied. Similar to the previous case, we find that the GPU results agree well with the CPU results, indicating that the proposed algorithm reproduces the correct dynamics. This is not trivial and depends on the value chosen for λ. Moreover, as pointed out above, the algorithm used for the Ising model fails to produce correct relaxation times. As the system approached the critical concentration, the dynamics become sluggish and the autocorrelation function decays slowly to zero.

In Figure 8b we compare the running times between the CPU and two GPU architectures. We achieved a $2\times$ factor on the GT 9600 GPU and a $\sim 70\times$ speedup running the same code on the new GTX 480 GPU. This factor reduces to $8\times$ for smaller lattices (128×128 and less). The GPU acceleration is nearly 2 orders of magnitude, implying that one can simulate the system closer to the critical density. This is important because the behavior of relaxation near the critical density is not necessarily universal, and thus extrapolations are often tricky.

VI. Conclusions and Summary

We have developed algorithms to simulate lattice spin models on consumer-grade GPUs. Two prototype models have been considered: the Ising model describing critical phenomena at equilibrium and the North-East model describing glassy dynamics. We showed that for equilibrium properties of lattice models an impressive speedup of $150\times$ can be achieved. To simulate the dynamics of such models, a more sophisticated approach was developed to preserve the dynamical rules and outcome. Our algorithm for the dynamic model reaches a $\sim 70\times$ factor in comparison to the serial CPU implementation for large system sizes. Although the algorithms were performed on specific models, we feel they can be easily extended to a larger class of similar systems.

Because the gain in computational power embodied by these results is in some cases 2 orders of magnitude, while the required GPU hardware is priced similarly to a CPU and can often be added to existing systems, the algorithms are certainly of interest. On the other hand, taking full advantage of it still requires savvy knowledge of the device and its capabilities and limitations, and the development of specific numerical algorithms. As the advantages of this useful technology become clear and knowledge about its implementation continues to build and spread, we hope it will become more accessible to a wide variety of computationally minded scientists.

References

(1) *NVIDIA CUDA, Compute Unified Device Architecture Programming Guide* Version 3.0, http://www.nvidia.com/object/cuda_develop.html (accessed 7/2010).

(2) Ufimtsev, I. S.; Martinez, T. J. *CiSE* **2008**, *10*, 26–34.

(3) Ufimtsev, I. S.; Martinez, T. J. *J. Chem. Theory Comput.* **2009**, *5*, 2619–2628.

(4) Anderson, A. G.; III, W. A. G.; Schröder, P. *Comput. Phys.* **2007**, *177*, 298–306.

(5) Meredith, J. S.; Alvarez, G.; Maier, T. A.; Schulthess, T. C.; Vetter, J. S. *Parallel Comput.* **2009**, *35*, 151–163.

(6) van Meel, J. A.; Arnold, A.; Frenkel, D.; Portegies Zwart, S. F.; Belleman, R. G. *Mol. Simul.* **2008**, *34*, 259–266.

(7) Stone, J. E.; Phillips, J. C.; Freddolino, P. L.; Hardy, D. J.; Trabuco, L. G.; Schulten, K. *J. Comput. Chem.* **2007**, *28*, 2618–2640.

(8) Anderson, J. A.; Lorenz, C. D.; Travesset, A. *J. Comput. Phys.* **2008**, *227*, 5342–5359.

(9) Davis, J.; Ozsoy, A.; Patel, S.; Taufer, M. In *Bioinformatics and Computational Biology*; Rajasekaran, S., Ed.; Springer Berlin: Heidelberg, 2009, Vol. 5462.

(10) Friedrichs, M. S.; Eastman, P.; Vaidyanathan, V.; Houston, M.; Legrand, S.; Beberg, A. L.; Ensign, D. L.; Bruns, C. M.; Pande, V. S. *J. Comput. Chem.* **2009**, *30*, 864–872.

(11) Genovese, L.; Ospici, M.; Deutsch, T.; Mehaut, J.-F.; Neelov, A.; Goedecker, S. *J. Chem. Phys.* **2009**, *131*, 034103.

(12) Demattè, L.; Prandi, D. *Briefings Bioinf.* **2010**, *11*, 323–333.

(13) Eastman, P.; Pande, V. S. *J. Comput. Chem.* **2010**, *31*, 1268–1272.

(14) Bernaschi, M.; Fatica, M.; Melchionna, S.; Succi, S.; Kaxiras, E. *Concurr. Comput.: Pract. Exper.* **2010**, *22*, 1–14.

(15) Lee, A.; Yau, C.; Giles, M. B.; Doucet, A.; Holmes, C. *ArXiv e-prints 0905.2441*, 2009.

(16) Preis, T.; Virnau, P.; Paul, W.; Schneider, J. J. *J. Comput. Phys.* **2009**, *228*, 4468–4477.

(17) Block, B.; Virnau, P.; Preis, T. *Comput. Phys. Commun.* **2010**, *181*, 1549–1556.

(18) Juba, D.; Varshney, A. *J. Mol. Graphics Modell.* **2008**, *27*, 82–87.

(19) Januszewski, M.; Kostur, M. *Comput. Phys.* **2010**, *181*, 183–188.

(20) Balijepalli, A.; LeBrun, T. W.; Gupta, S. K. *J. Comput. Inf. Sci. Eng.* **2010**, *10*, 011010.

(21) Onsager, L. *Phys. Rev.* **1944**, *65*, 117–149.

(22) Newman, M. E. J.; Barkema, G. T. *Monte Carlo Methods in Statistical Physics*; Oxford University Press: London, 1999.

(23) Reiter, J.; Mauch, F.; Jäckle, J. *Physica A (Amsterdam)* **1992**, *184*, 493–498.

(24) Garrahan, J. P.; Chandler, D. *Proc. Natl. Acad. Sci. U.S.A.* **2003**, *100*, 9710–9714.

(25) Jack, R. L.; Garrahan, J. P.; Chandler, D. *J. Chem. Phys.* **2006**, *125*, 184509.

(26) Garrahan, J. P.; Jack, R. L.; Lecomte, V.; Pitard, E.; van Duijvendijk, K.; van Wijland, F. *Phys. Rev. Lett.* **2007**, *98*, 195702.

(27) Chandler, D.; Garrahan, J. P. *Annu. Rev. Phys. Chem.* **2010**, *61*, 191–217.

(28) Edwards, S. F.; Anderson, P. W. *J. Phys. F: Met. Phys.* **1975**, *5*, 965.

(29) Sherrington, D.; Kirkpatrick, S. *Phys. Rev. Lett.* **1975**, *35*, 1792–1796.

(30) Marinari, E.; Parisi, G.; Ritort, F. *J. Phys. A: Math. Gen.* **1994**, *27*, 7615.

(31) Fredrickson, G.; Andersen, H. *Phys. Rev. Lett.* **1984**, *53*, 1244–1247.

(32) Jäckle, J.; Eisinger, S. *Z. Phys. B: Condens. Matter* **1991**, *84*, 115–124.

(33) Landau, D. P.; Binder, K. *A Guide to Monte Carlo Simulations in Statistical Physics*; Cambridge University Press: Cambridge, 2000.

(34) Metropolis, N.; Rosenbluth, A.; Rosenbluth, M.; Teller, R. A.; Teller, E. *J. Chem. Phys.* **1953**, *21*, 1087–1091.

(35) Wu, F. Y. *Rev. Mod. Phys.* **1982**, *54*, 235–268.

(36) Manousiouthakis, V. I.; Deem, M. W. *J. Chem. Phys.* **1999**, *110*, 2753–2756.

(37) Ren, R.; Orkoulas, G. *J. Chem. Phys.* **2006**, *124*, 064109.

(38) Nakanishi, H.; Takano, H. *Phys. Lett. A* **1986**, *115*, 117–121.

(39) Geissler, P. L.; Reichman, D. R. *Phys. Rev. E* **2005**, *71*, 031206.

(40) Ferrenberg, A. M.; Landau, D. P.; Wong, Y. J. *Phys. Rev. Lett.* **1992**, *69*, 3382–3384.

(41) Press, W. H.; Flannery, B. P.; Teukolsky, S. A.; Vetterling, W. T. *Numerical Recipes in C*; Cambridge University Press: New York, 1992.

(42) *CUDA GPU Occupancy Calculator* version 3.1, http://developer.nvidia.com/object/cuda_3_1_downloads.html (accessed 7/2010).