International Journal of Modern Physics C
Vol. 23, No. 8 (2012) 1240002 (11 pages)

DOI: 10.1142/S0129183112400025

World Scientific
www.worldscientific.com

SIMULATING SPIN MODELS ON GPU: A TOUR

MARTIN WEIGEL

Institut für Physik, Johannes Gutenberg-Universität Mainz
D-55099 Mainz, Germany

Applied Mathematics Research Centre
Coventry University, Coventry, CV1 5FB, England
weigel@uni-mainz.de

Received 24 October 2011
Accepted 12 November 2011
Published 21 July 2012

The use of graphics processing units (GPUs) in scientific computing has gathered considerable momentum in the past five years. While GPUs in general promise high performance and excellent performance per Watt ratios, not every class of problems is equally well suitable for exploiting the massively parallel architecture they provide. Lattice spin models appear to be prototypic examples of problems suitable for this architecture, at least as long as local update algorithms are employed. In this review, I summarize our recent experience with the simulation of a wide range of spin models on GPU employing an equally wide range of update algorithms, ranging from Metropolis and heat bath updates, over cluster algorithms to generalized ensemble simulations.

Keywords: Spin models; Monte Carlo simulations; GPU computing; cluster algorithms; generalized-ensemble simulations.

PACS Nos.: 75.50.Lk, 64.60.Fr.

1. Introduction

The use of Monte Carlo simulations in statistical physics has been an impressive success story.[1] While, in the early days, computer simulations of simple systems such as spin models could hardly compete with the elaborate perturbative techniques such as the ϵ expansion and high-temperature series,[2] the vast increase in available computer power together with a series of ingenious improvements of simulation technology has turned the simulational techniques into the method of choice, and often even the only feasible approach, for a wide range of problems ranging from critical phenomena, over studies of surfaces and restricted geometries, to nucleation and a wealth of nonequilibrium processes.

This success notwithstanding, there is still a range of problems whose insatiable appetite for more computer power generates a strong demand for the design and

construction of ever more efficient computers. Such problems include systems with quenched disorder[3–5] whose rugged free-energy landscapes lead to slow dynamics with exploding autocorrelation times and, on top of that, require averages over large numbers of disorder configurations to cope with the lack of self-averaging of important quantities. Similarly, biopolymers as well as structural glasses suffer from a many-valleyed energy landscape. For the simulation of such systems, in particular, special purpose computers are being built[6] and new computational architectures are being tried.

A rather successful advance in this direction in recent years has been the use of graphics processing units (GPUs) for general-purpose scientific computing.[7,8] Having been designed for real-time rendering of realistic 3D graphics scenes in computer games, they are characterized by a massively parallel architecture consisting of many relatively simple compute units. As they lack the sophisticated control logic and excessive on-die caches commonly found in todays CPUs, most of the die space is devoted to arithmetic logic units (ALUs) used for doing actual computations. If an application manages to constantly feed the ALUs with data, therefore, a GPU can deliver a total floating point performance vastly exceeding that of a current CPU. This observation is the basis of using GPUs for general purpose computations. Programming GPUs for such purposes has become feasible with the advent of appropriate language extensions such as NVIDIA CUDA and OpenCL. Good performance, however, can only be expected if the special control flow and organization of GPU devices is taken into account on developing the code.

In the following, I discuss how efficient GPU implementations can be achieved for simulations of spin models, and how these codes perform compared to implementations on current CPUs. In Sec. 2, I give some background on the architecture of the NVIDIA GPUs used in the present work. Section 3 is devoted to the discussion of simulations using local updates such as Metropolis and heatbath applied to discrete and continuous ferromagnetic spin models as well as the Ising spin glass. In Sec. 4, I discuss GPU implementations of cluster algorithms for spin-model simulations, while Sec. 5 is devoted to multicanonical (MUCA) and Wang–Landau (WL) simulations. Finally, Sec. 6 contains my conclusions.

2. The NVIDIA Architecture

General-purpose computing on GPUs relies on high-performance GPU devices as currently provided by NVIDIA or ATI. Different programming environments are available, the most prominent ones being NVIDIA CUDA and, more recently, OpenCL. As the implementations discussed here reach back to times before the advent of OpenCL, all codes make use of NVIDIA CUDA and, consequently, are restricted to run on NVIDIA GPUs. While the terminology of describing GPUs is, therefore, specific to NVIDIA, the general GPU layout is quite generic and, with some modifications, also applies to ATI boards. Figure 1 shows a schematic representation of the general architecture of a current GPU. The chip contains a number

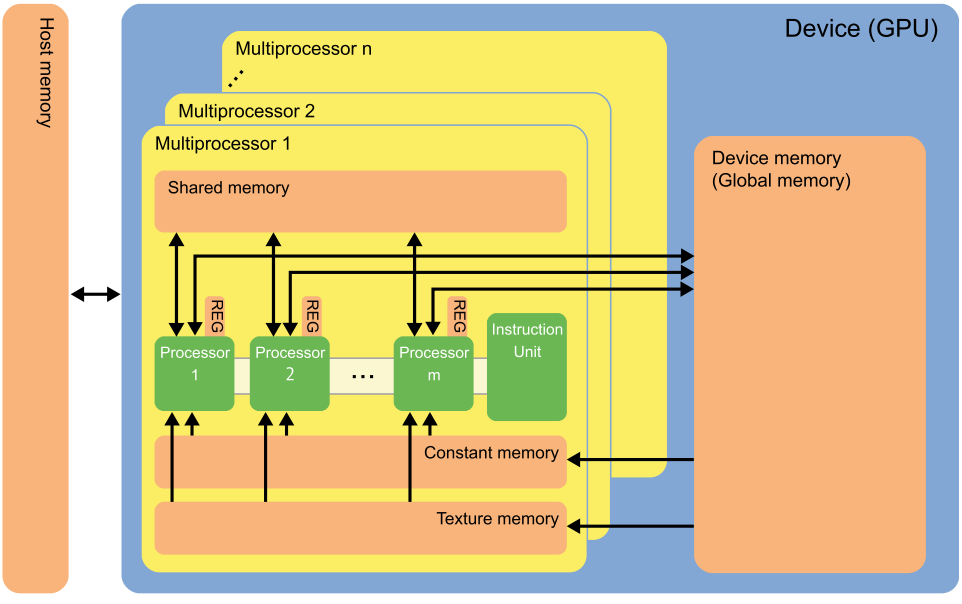

Fig. 1. (Color online) Schematic representation of the architecture of current GPUs.

of multiprocessors each composed of a number of parallel processing units. The systems based on the GT200 architecture used in this study feature 30 multiprocessors of eight processors each, while the boards of the more recent Fermi generation currently feature 14−16 multiprocessors at 32 cores. The systems come with a hierarchy of memory layers with different characteristics:

- *Registers*: each multiprocessor is equipped with several thousand registers with local, zero-latency access;
- *Shared memory*: processors of a multiprocessor have access to a small amount (16 KB for Tesla, 48 KB for Fermi) of on chip, small latency shared memory;
- L1 *and* L2 *caches*: 16/48 kB L1 cache and 768 kB L2 cache;
- *Global memory*: large amount (currently up to 6 GB) of memory on separate DRAM chips with access from every thread on each multiprocessor with a latency of several hundred clock cycles;
- *Constant and texture memory*: read-only memories of the same speed as global memory, but cached;
- *Host memory*: cannot be accessed from inside GPU functions, relatively slow transfers.

Since the processing units of each multiprocessor are designed to perform identical calculations on different parts of a dataset, flow control for this single instruction multiple data (SIMD) type of parallel computations is rather simple. It is clear that this type of architecture is near ideal for the type of calculations typical for computer

graphics, namely rendering a large number of triangles in a 3D scene or the large number of pixels in a 2D projection in parallel.

The organization of processing units and memory outlined in Fig. 1 translates into a combination of two types of parallelism: the processing units inside of each multiprocessor work synchronously on the same data set (vectorization), whereas different multiprocessors work truly independent of each other (parallelization). The corresponding programming model implemented in the CUDA framework[9] is outlined schematically in Fig. 2: computations on GPU are encapsulated in functions (called "kernels") which are compiled to the GPU instruction set and downloaded to the device. They are executed in a two-level hierarchic set of parallel instances ("execution configuration") called a "grid" of thread "blocks." Each block can be thought of as being executed on a single multiprocessor unit. Its threads (up to 512 for the GT200 architecture and 1024 for Fermi cards) access the same segment of shared memory concurrently. Ideally, each thread should execute exactly the same instructions, that is, branching points in the code should be reduced to a minimum. The blocks of a grid (up to $65\,536 \times 65\,536$) are scheduled independently of each other and can only communicate via global memory accesses. The threads within a block can make use of cheap synchronization barriers and communicate via the use of shared (or global) memory, avoiding race conditions via atomic operations implemented directly in hardware. On the contrary, the independent blocks of a grid

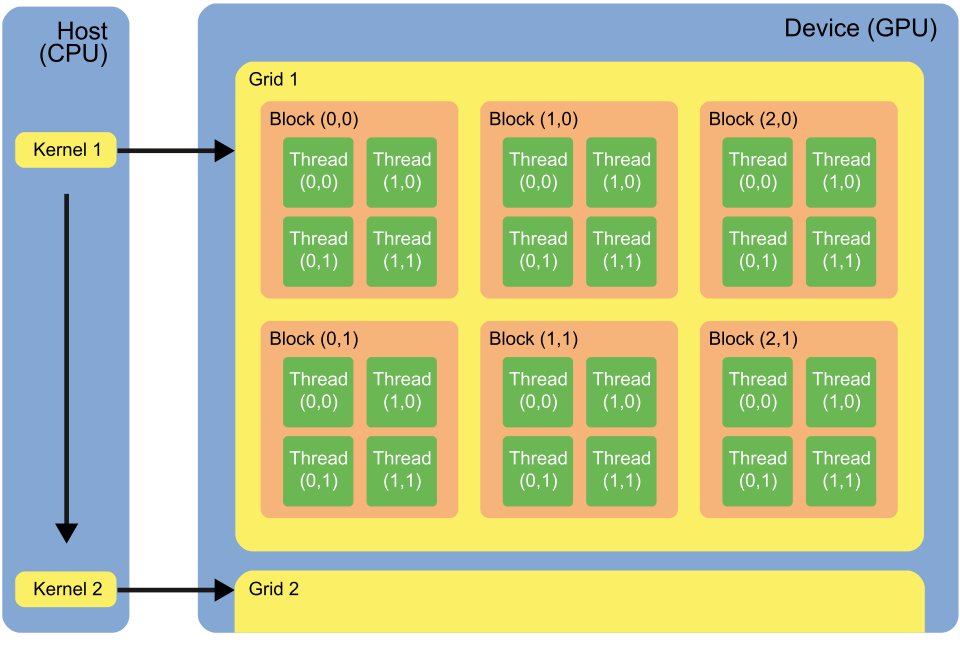

Fig. 2. (Color online) Parallel execution of a GPU program ("kernel") in a grid of thread blocks. Threads within a block work synchronously on the same dataset. Different blocks are scheduled for execution independent of each other.

cannot effectively communicate within a single kernel call. If synchronization between blocks is required, consecutive calls of the same kernel are required, since termination of a kernel call enforces all block computations and pending memory writes to complete.

The massively parallel architecture with its hierarchy of memories found on GPUs leads to a specific set of design goals for an algorithm to perform efficiently. In particular, programmers should strive to reach[10]

(1) a large degree of locality of the calculations, reducing the need for communication between threads,
(2) a large coherence of calculations with a minimum occurrence of divergence of the execution paths of different threads,
(3) a total number of threads significantly exceeding the number of available processing units,
(4) and a large overhead of arithmetic operations and shared memory accesses over global memory accesses.

The large number of individual threads is of particular importance as this allows the GPU scheduler to hide latencies: if a thread block issues an access, e.g. to global memory, the GPU's scheduler will suspend it for the number of cycles it takes to complete the memory accesses and, instead, execute another block of threads which has already finished reading or writing its data.

3. Simulations with Local Updates

As a rather general class of spin models, I considered the classical O(n) symmetric Hamiltonian

$$\mathcal{H} = -\sum_{\langle ij \rangle} J_{ij} \boldsymbol{s}_i \cdot \boldsymbol{s}_j, \tag{1}$$

where $n = 1$ corresponds to the discrete Ising model and $n = 2$ and $n = 3$ describe the continuous XY and Heisenberg models, respectively. The spins are located on square ($d = 2$) or simple cubic ($d = 3$) lattices and interact with nearest neighbors only. We first considered simulations using only single spin-flip moves accepted according to the Metropolis criterion,[11]

$$p_{\text{acc}}(s_i \mapsto s_i') = \min[1,\, e^{-\beta \Delta E}]. \tag{2}$$

In order to achieve an efficient implementation on GPU, one needs to allow for the parallel update of a large number of spins. This is most straightforwardly accomplished for the case of nearest-neighbor interactions by making use of a checkerboard decomposition of the lattice.[12] To make efficient use of *shared memory* that can be accessed concurrently with very low latencies from all threads on a multiprocessor, I use a two-level hierarchical decomposition, cf. Fig. 3 (see also Refs. 10 and 13). All spins of one of the big tiles plus some boundary layer are collaboratively loaded into

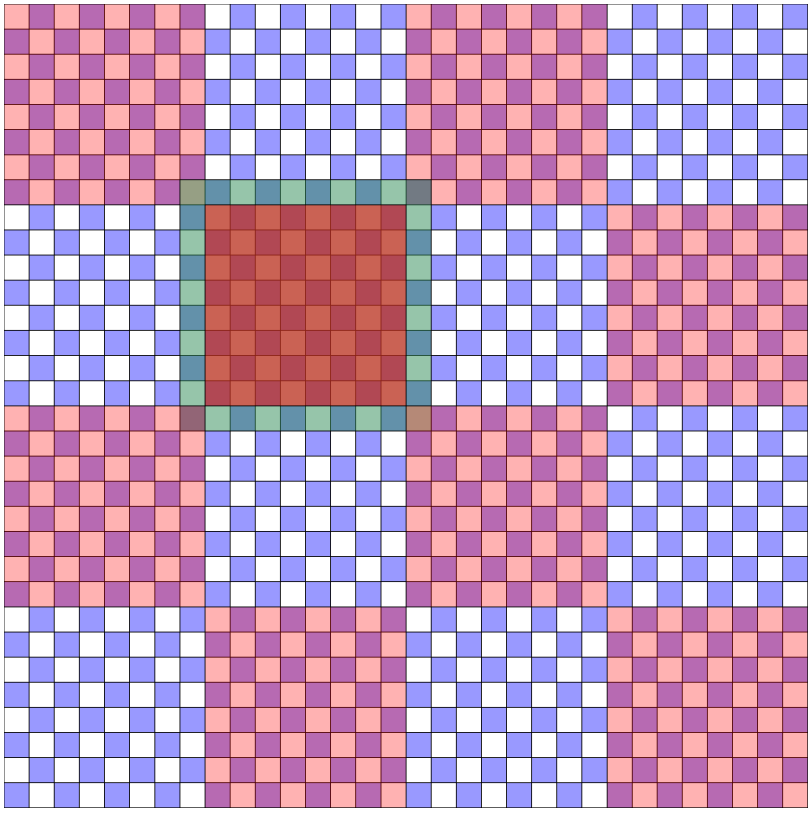

Fig. 3. (Color online) Double checkerboard decomposition of a square lattice of edge length $L = 32$ for performing a single spin-flip Metropolis simulation of spin models on GPU.

shared memory and subsequently updated by the individual threads of a thread block.[8] Inside of each tile, the checkerboard arrangement allows for all spins of one sub-lattice to be updated concurrently before a synchronization barrier occurs and the second sub-lattice is treated analogously. To amortize the effort of loading tiles into shared memory, a number k of updates of all spins of all big tiles of one color is performed before updating the second sub-lattice. Depending on the size of the tiles, this slows down the decorrelation of spin configurations. This effect, however, is more than counter-balanced by the performance increase, even close to criticality.[13] For good performance, a number of additional tricks are employed, including a pre-tabulation of the Boltzmann factors in Eq. (2) while storing this table as a texture,[8] and generation of random numbers even if they are not required to reduce thread divergence. The simulation code can be downloaded at the authors' web site.[14] For the comparisons discussed here, an array of simple 32-bit linear congruential pseudo-random number generators (one per thread) is used. Although these are known to have rather poor properties, for the purpose at hand they appear to be sufficient even

for high-precision results. The implementation of more generally appropriate generators is discussed in Ref. 10.

For the benchmarks, I compared the performance of the outlined GPU implementation on a Tesla C1060 as well as a more recent GTX 480 of the Fermi architecture series with the results of an optimized, single-threaded CPU code running on an Intel Core 2 Quad Q9650 at 3.0 GHz. For the Ising ferromagnet, $n = 1$ and $J_{ij} = 1$ in Eq. (1), a tile size of 16×16 spins is found to be optimal for sufficiently large systems.[13] The maximum performance reached on the GTX 480 is around 0.03 ns per spin flip (using $k = 100$), which is 235 times faster than the CPU implementation, cf. the data collected in Table 1 and in Fig. 4. The Tesla C1060, on the other hand, roughly performs at half of this speed. This speedup, however, is only reached for sufficiently large system sizes that allow to fully load the 240 and 480 cores of the C1060 and the GTX 480, respectively. Very similar relative performance is observed for the Ising model in three dimensions. For models with continuous spins, exemplified by the 2D Heisenberg model with $n = 3$ and $J_{ij} = 1$ in Eq. (1), issues of floating-point performance and precision become important. It is found that a mixed-precision calculation, where the spins are represented in single precision and only aggregate quantities such as the total energy are calculated in double precision (see "Metropolis single" in Table 1) yield high performance without problems with precision. If the hardware optimized implementations of the special functions (trigonometric, exponential, logarithmic etc.) provided in the CUDA framework are employed, total speedups beyond 1000 can be achieved as compared to CPU codes (see "Metropolis fast math" in Table 1).

Simulations of systems with quenched disorder allow for trivial parallelization over disorder realizations on top of the domain decomposition outlined above. For the Edwards-Anderson Ising spin glass with couplings $J_{ij} \in \{-J, J\}$ drawn from a

Table 1. Spin-flip times for simulations of various lattice spin models with different algorithms on an Intel Q9650, a Tesla C1060 and a GTX 480, respectively. Apart from the cluster, MUCA and WL simulations, multi-hit updates with $k = 100$ have been employed.

System	Algorithm	L	CPU ns/flip	C1060 ns/flip	GTX 480 ns/flip	Speedup
2D Ising	Metropolis	32	8.3	2.58	1.60	3/5
2D Ising	Metropolis	16 384	8.0	0.077	0.034	103/235
3D Ising	Metropolis	512	14.0	0.13	0.067	107/209
2D Heisenberg	Metro. double	4096	183.7	4.66	1.94	39/95
2D Heisenberg	Metro. single	4096	183.2	0.74	0.50	248/366
2D Heisenberg	Metro. fast math	4096	183.2	0.30	0.18	611/1018
2D spin glass	Metropolis	32	14.6	0.15	0.070	97/209
2D spin glass	Metro. multi-spin	32	0.18	0.0075	0.0023	24/78
2D Ising	Swendsen–Wang	10 240	77.4	—	2.97	–/26
2D Ising	MUCA	64	42.1	—	0.33	–/128
2D Ising	WL	64	43.6	—	0.94	–/46

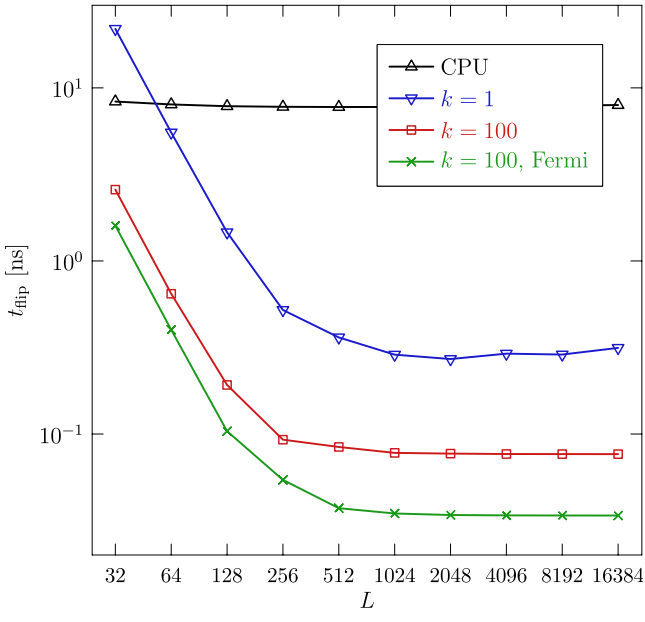

Fig. 4. (Color online) Time per spin-flip in the 2D Ising model on CPU and on GPU with different choices of k. GPU data are for the Tesla C1060 device apart from the lowest curve which is for a GTX 480 card.

bimodal distribution, we again find speedups of around 100 and 200 for the Tesla C1060 and the GTX 480, respectively. Further improvements can be achieved on using 64-bit multi-spin coding which allows for spin-flip times down to 2 pico-seconds on the GTX 480, cf. Table 1. Additional algorithmic components commonly used for the simulation of disordered systems, in particular the parallel tempering method, are also easily and efficiently implemented on GPU. For details see Ref. 10.

4. Cluster-Update Simulations

While single spin-flip simulations on a fixed lattice appear to be near optimal problems for the parallel compute model of GPUs, highly nonlocal updates such as the cluster algorithms used to beat critical slowing down in ferromagnetic models close to a continuous phase transition are significantly harder to efficiently implement in parallel. To test this, I considered different implementations of the Swendsen−Wang cluster algorithm[15] for the Ising model. An update consists of the following steps:

(1) Activate bonds between like spins with probability $p = 1 - e^{-2\beta J}$.
(2) Construct (Swendsen–Wang) spin clusters from domains connected by active bonds.
(3) Flip-independent clusters with probability 1/2.

While Steps 1 and 3 are completely local and hence can be easily performed in a highly parallel fashion using a single thread for updating a few bonds or sites, the

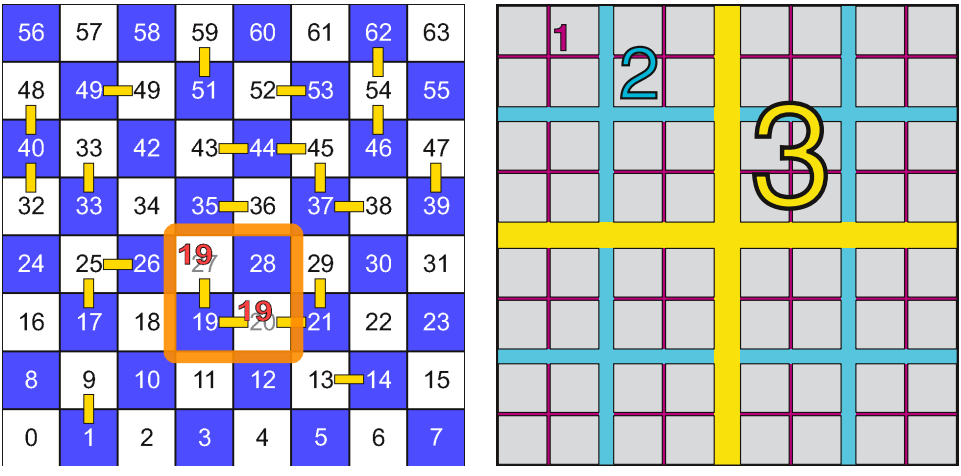

Fig. 5. (Color online) (a) Cluster identification on a 64×64 tile using the self-labeling algorithm with one thread per 2×2 spins. In every pass, each site examines its northward and eastward neighbors and, if they are connected by an active bonds, for each pair sets both labels to the minimum of the two current labels. (b) Hierarchical sewing of 64 tiles for label consolidation. On level 1, 2×2 tiles are sewn together to form 16 larger tiles. In levels 2 and 3, the tile numbers are reduced to 4 and 1, respectively.

cluster identification step is intrinsically nonlocal, in particular close to the critical point where the Swendsen–Wang clusters undergo a percolation transition. A number of different possibilities for performing the cluster identification on tiles and consolidating cluster labels consecutively have been analyzed in detail in Ref. 16. It is found that a procedure dubbed self-labeling, which is rather inefficient in terms of serial computation times but very easily parallelized, is optimal for labeling inside of tiles, cf. the left panel of Fig. 5. In a second step, cluster labels need to be consolidated between tiles. For large overall system sizes and simulations of spin models close to criticality, a hierarchical sewing scheme working on a forest of union-and-find trees has been developed for this purpose.[16] An illustration of this approach is depicted in the right panel of Fig. 5. The overall parallel performance of this combination of algorithmic components is significantly lower than that found for the local-update algorithms. Nevertheless, speedups close to 30 can be observed as compared to optimized serial code running on a single high-end CPU core, cf. the data collected in Table 1. More details can be found in Ref. 16.

5. Generalized Ensembles

Another class of algorithms which is local in terms of the update rule but, as it turns out, requires constant information about the status of a global quantity, are generalized-ensemble simulations such as MUCA[17] or WL[18] methods. Considering

the internal energy E as reaction coordinate, the canonical distribution $p_\beta(E) = Z_\beta^{-1}\Omega(E)\exp(-\beta E)$ is generalized to read $p_\beta(E) = Z_\beta^{-1}\Omega(E)/W(E)$. A flat histogram is reached if the weights $W(E)$ equal the density of states $\Omega(E)$ or, equivalently, $\omega(E) \equiv \ln W(E) = S(E)$, where $S(E)$ is the microcanonical entropy. While MUCA uses a series of fixed-ensemble, equilibrium simulations to estimate $W(E) = S(E)$, an analogous estimate is calculated online in a nonequilibrium simulation in the WL approach. These algorithms are difficult to parallelize since they require knowledge of the current value of a global reaction coordinate (such as energy or magnetization) prior to each update. This effectively serializes all updates performed on a single instance of the system. To still benefit from the parallel GPU architecture, one can use "windowing," i.e. the idea of applying algorithms separately in small, fixed energy windows[18] and gluing together the resulting estimates to reconstruct the overall $S(E)$. Another option is trivial parallelization to improve statistics and estimate statistical errors.[19]

For the 2D Ising model, it is found that "windowing" does not cause systematic deviations from the exact result for $S(E)$[20] as long as enough statistics is collected in each window. For the WL algorithm this means imposing a strict criterion as to when the energy histogram is considered flat; for the MUCA algorithm a sufficient number of tunneling events should be demanded. This allows, for instance, to construct $S(E)$ for a 64×64 system from windows as small as $\Delta E = 16$. The speedups of the GPU implementations using a sufficiently large number of windows and independent runs to fully load the GPU are summarized in Table 1. For MUCA, we arrive at a speedup of 128, similar to the results found for the local algorithms, whereas the WL approach, in its current implementation, allows a 46 times performance increase only. This difference results from the dynamical nature of the WL algorithm, where run times are random variables, which leads to thread divergence and idle cores on the GPU. A more sophisticated implementation using some load balancing scheme is, however, easily possible and is expected to results in an overall performance of the WL approach comparable to that of the MUCA simulations.

6. Conclusions

In this paper, I summarized results regarding the performance potential for the simulation of spin models on GPUs. Discussing a wide range of algorithms, ranging from single spin-flip Metropolis updates, over cluster algorithms to generalized-ensemble simulations, two main questions have been addressed: (a) Are spin models with short-range interactions suitable for simulations in the massively parallel environment provided by current GPUs? and (b) Are such GPUs versatile enough for the efficient implementation of the broad range of different algorithms used for the simulation of spin models in different situations? As the rather substantial speedups summarized in Table 1 illustrate, the considered class of models is almost ideally suited for exploiting the inherent parallelism of GPUs. While local updates can be parallelized rather straightforwardly, the nonlocal operations encountered in

cluster-update or generalized-ensemble simulations are considerably harder to adapt to a massively parallel environment. Although the gains compared to serial CPU code are somewhat smaller for these codes, they remain significant — also in terms of performance per Dollar and performance per Watt — even for the most difficult case of the connected component identification. While the implementations discussed here are based on the NVIDIA CUDA framework, very similar conclusions hold for the more general OpenCL API, or for calculations performed on ATI GPUs. It thus appears that general purpose GPU computing, although less versatile than traditional CPU computing, is sufficiently general to warrant the construction and use of dedicated GPU clusters for the purposes considered here.

Acknowledgments

The author acknowledges computer time provided by NIC Jülich under Grant No. hmz18, support by the "Center for Computational Sciences in Mainz" (SRFN) and funding by the DFG under contract No. WE4425/1-1 (Emmy Noether programme).

References

1. K. Binder and D. P. Landau, *A Guide to Monte Carlo Simulations in Statistical Physics*, 3rd edn. (Cambridge University Press, Cambridge, 2009).
2. J. Zinn-Justin, *Quantum Field Theory and Critical Phenomena*, 4th edn. (Oxford University Press, Oxford, 2002).
3. A. P. Young (ed.), *Spin Glasses and Random Fields* (World Scientific, Singapore, 1997).
4. Y. Holovatch (ed.), *Order, Disorder and Criticality*, Vol. 1 (World Scientific, Singapore, 2004).
5. Y. Holovatch (ed.), *Order, Disorder and Criticality*, Vol. 2 (World Scientific, Singapore, 2007).
6. F. Belletti *et al.*, *Comput. Sci. Engrg.* **11**, 48 (2009).
7. J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone and J. C. Phillips, *Proc. IEEE* **96**, 879 (2008).
8. D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors* (Elsevier, Amsterdam, 2010).
9. http://www.nvidia.com/object/cuda_home_new.html.
10. M. Weigel, *J. Comput. Phys.* **231**, 3064 (2012).
11. N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller and E. Teller, *J. Chem. Phys.* **21**, 1087 (1953).
12. D. Heermann and A. N. Burkitt, *Parallel Comput.* **13**, 345 (1990).
13. M. Weigel, *Comput. Phys. Commun.* **182**, 1833 (2011).
14. http://www.cond-mat.physik.uni-mainz.de/∼weigel/GPU.
15. R. H. Swendsen and J. S. Wang, *Phys. Rev. Lett.* **58**, 86 (1987).
16. M. Weigel, *Phys. Rev. E* **84**, 036709 (2011).
17. B. A. Berg and T. Neuhaus, *Phys. Rev. Lett.* **68**, 9 (1992).
18. F. Wang and D. P. Landau, *Phys. Rev. Lett.* **86**, 2050 (2001).
19. M. Weigel and T. Yavors'kii, *Phys. Procedia* **15**, 92 (2011).
20. P. D. Beale, *Phys. Rev. Lett.* **76**, 78 (1996).