Computer Physics Communications 181 (2010) 1549–1556

Contents lists available at ScienceDirect

Computer Physics Communications

www.elsevier.com/locate/cpc

# Multi-GPU accelerated multi-spin Monte Carlo simulations of the 2D Ising model ☆

Benjamin Block *, Peter Virnau, Tobias Preis

*Department of Physics, Mathematics and Computer Science, Johannes Gutenberg University Mainz, Staudingerweg 7, D-55128 Mainz, Germany*

ARTICLE INFO

*Article history:*
Received 17 March 2010
Received in revised form 10 May 2010
Accepted 17 May 2010
Available online 24 May 2010

*Keywords:*
Monte Carlo simulation
GPU computing
Ising model
Phase transition
Finite size scaling

ABSTRACT

A Modern Graphics Processing unit (GPU) is able to perform massively parallel scientific computations at low cost. We extend our implementation of the checkerboard algorithm for the two-dimensional Ising model [T. Preis et al., Journal of Chemical Physics 228 (2009) 4468–4477] in order to overcome the memory limitations of a single GPU which enables us to simulate significantly larger systems. Using multi-spin coding techniques, we are able to accelerate simulations on a single GPU by factors up to 35 compared to an optimized single Central Processor Unit (CPU) core implementation which employs multi-spin coding. By combining the Compute Unified Device Architecture (CUDA) with the Message Parsing Interface (MPI) on the CPU level, a single Ising lattice can be updated by a cluster of GPUs in parallel. For large systems, the computation time scales nearly linearly with the number of GPUs used. As proof of concept we reproduce the critical temperature of the 2D Ising model using finite size scaling techniques.


## 1. Introduction

Various scientific disciplines profited by GPU computing in recent years and are reporting impressive speedup factors in comparison to single Central Processor Unit (CPU) core implementations. GPU stands for Graphics Processing Units which are high-performance many-core processors that can be used to accelerate a wide range of applications. In the meantime, significant savings of computing time have been reported by a huge variety of fields: GPU acceleration can be used in astronomy [1] and radio astronomy [2]. Soft tissue simulation [3], algorithms for image registration [4], dose calculation [5], volume reconstruction from X-ray images [6], and the optimization of intensity-modulated radiation therapy plans [7] are examples for the numerous applications in medicine. Furthermore, DNA sequence alignment [8], molecular dynamics simulations [9–11], quantum chemistry [12], multipole calculations [13], density functional calculations [14,15], air pollution modeling [16], time series analysis focused on financial markets [17,18], and Monte Carlo simulations [19–22] benefited from GPU computing. For many applications, the accuracy can be comparable to that of a double-precision CPU implementation, such as in [23]—the latest generation of GPUs support not only single precision but also double precision floating point operations. The adaption of many computational methods is still in progress, e.g. the analysis of switching processes in financial markets [24,25]. Unfortunately, not all algorithms can be ported efficiently onto a GPU architecture. Particularly, serial algorithms are not suited for GPU computing (for an example see e.g. [26]).

Another crucial limitation is the lack of scalability as current programs typically utilize only single GPUs. As graphics processing hardware is targeted at a broad consumer market—the games industry—, graphic cards can be produced at low cost. On the other hand, to keep production costs low, the global memory is not upgradable and typically limited to 1 GB for consumer cards and 4 GB for Tesla GPUs. Using a recent consumer graphics card, we accelerated Monte Carlo simulations of the Ising model [22]. In [22], a 2D square spin lattice of dimension up to $1024^2$ spins could be processed on a consumer GPU. The Ising model as a standard model of statistical physics provides a simple microscopic description of ferromagnetism [27]. It was introduced to explain the ferromagnetic phase transition from the paramagnetic phase at high temperatures to the ferromagnetic phase below the Curie temperature $T_C$. A large variety of techniques and methods in statistical physics have originally been formulated for the Ising model and were generalized and adapted to related models and problems [28]. Due to its simplicity, which can be embodied by the possibility to use trivial parallelization approaches [29], the two-dimensional Ising model is well suited as a benchmark model since its properties are well studied [30–32] and many physical systems belong to the same universality class. The Ising model on a two-dimensional square lattice with no magnetic field was analytically solved

☆ Source code of our implementations for GPU clusters will be published on http://www.tobiaspreis.de after acceptance. In addition, the code can be downloaded from the Google Code project *multigpu-ising*.

* Corresponding author.
*E-mail addresses:* lhyanor@gmail.com (B. Block), virnau@uni-mainz.de (P. Virnau), mail@tobiaspreis.de (T. Preis).

0010-4655/$ – see front matter 
doi:10.1016/j.cpc.2010.05.005

**Table 1**
Key facts and properties of the GPU [42].

| | Tesla C1060 |
|---|---|
| Global video memory | 4096 MB |
| Streaming processor cores | 240 |
| Shared memory per block | 16 kB |
| Processor clock | 1.30 GHz |
| Memory clock | 800 MHz |
| Maximal power consumption | 187.8 W |

by Lars Onsager in 1944 [33]. The critical temperature at which a second order phase transition between an ordered and a disordered phase occurs can be determined analytically for the two-dimensional model ($T_C \approx 2.269185$ [33]).

Here we show how lattice sizes can be extended up to $100{,}000^2$ spins on one GPU device with 4 GB of global memory using a memory optimized encoding of the spins—one bit per spin. This number of spins turns out to be a hard limitation on a single device, since for larger system sizes, spin data would have to be transferred between device and host memory. Such a memory transfer would effectively rule out all performance benefits of a GPU implementation. Using a multi-spin coding scheme [34–37], computation to memory access ratio can be improved, resulting in a dramatically faster GPU performance.

We show that an extension of this approach can be used successfully to handle Monte Carlo simulations of the Ising model in a multi-GPU environment—GPU clusters. The scalability of this implementation is ensured by splitting the lattice into quadratic sublattices, and by placing them into the memory of different GPUs. Thus, each GPU can perform the calculation of one sublattice in its memory and pass the information about its borders on to its neighboring GPUs. Similar approaches have been used, e.g., for the calculation of density functionals [14].

This paper is organized as follows. In a brief overview in Section 2, key facts of the GPU architecture are provided in order to clarify implementation constraints for the following sections. Section 3 provides a survey of model definition and finite size scaling techniques used as proof of concept. In Sections 4 and 5, we describe details of the reference CPU implementation and our single GPU approach based on multi-spin coding. The multi-GPU accelerated Monte Carlo simulation of the 2D Ising model is covered in Section 6. To overcome the memory limitations of a single GPU with such a multi-GPU approach is of crucial importance as GPU clusters are currently set up in supercomputing facilities. Our conclusions are summarized in Section 7.

## 2. GPU device architecture

Simulating the Ising model at large system sizes requires a lot of processing performance. Physical and engineering obstacles in microprocessor design have resulted in flat performance growth for traditional single-core microprocessors. On the other hand, graphics hardware has become highly programmable, the fixed function pipelines have been replaced by programmable shader units that can perform the same operation on many sets of data in parallel. For a comprehensive overview of recent developments in computer graphics, especially programmable shader techniques, see [38]. With new, more flexible programming interfaces, these units can be utilized to perform general purpose computing in fields other than computer graphics.

For our GPU implementation, we use the Compute Unified Device Architecture (CUDA) released by NVIDIA for their recent graphics accelerator boards. The latest stable release at the time of writing is CUDA 2.3 [39]. Recently, other Application Programming Interfaces (APIs) for General Purpose computing on GPUs (GPGPU) became available, see e.g. OpenCL [40]. Additionally, efforts have been made to establish high-level programming environments [41] as well as the integration into existing compilers.

We use an NVIDIA Tesla C1060 as our CUDA enabled device, which offers 4 GB of GDDR3 global memory, see Table 1. This memory can store a multi-spin coded spin field of $100{,}000^2$ spins on one GPU. The reference CPU used in our tests is the Intel Xeon X5560 at a clock rate of 2.80 GHz and 8192 kB cache. The purpose of the CPU implementation is to have a fast and fair non-parallel reference implementation, not to benchmark the Intel CPU. Therefore, only one core of the CPU is used (without Hyper-Threading Technology).

CUDA implements a Single Instruction Multiple Thread (SIMT) approach. It is capable of running the same code in parallel, processed in a "grid". A grid is a number of blocks which in turn contain a defined number of threads. It extends the C language by the invocation of "kernels" that run in parallel in such a grid on the GPU:

```
cuda_kernel<<<gridDim, blockDim>>>(data);
```

The variable *gridDim* defines the number of blocks that run in parallel, and *blockDim* specifies the number of threads that run in each block. Threads in each block share a certain amount of "shared memory" which can be accessed roughly one order of magnitude faster than data in the global GPU memory. The Tesla C1060 is capable of processing a maximum of 512 threads per block. Kernels are executed on the actual hardware in units of "warps", where each warp executes one common instruction at a time.

With a larger number of threads in a block, memory latencies can be hidden more effectively. However, the hiding of memory latencies only results in better performance, if the number of registers used by a single thread is sufficiently small. To optimize execution time for a kernel, a grid size should be used that allows for a maximum "occupancy". The occupancy is the ratio of active warps to the maximum number of warps supported on a multiprocessor of the GPU. A multiprocessor contains amongst others eight scalar processor cores, a multi-threaded instruction unit, and shared memory. This ratio is a helpful number to determine how efficient the kernel will be on the GPU.

For the multi-GPU implementation, each CPU core runs a separate process and controls one of the available GPUs. Communication is established via the Message Passing Interface (MPI). Communication is needed frequently (see Section 6) which leads to a bottleneck for small systems, but is ruled out by the benefit of more available GPU cores for larger systems.

## 3. The two-dimensional Ising model

The Ising model is formulated on a two-dimensional square lattice, where on each lattice site a spin $S_i$ with a value of either $-1$ or 1 is located. The interaction of the spins is given by the Hamiltonian

$$\mathscr{H} = -J \sum_{\langle i,j \rangle} S_i S_j - H \sum_i S_i \tag{1}$$

where $H$ denotes an external magnetic field, which we will set to zero here. The lattice is updated according to the Metropolis criterion [43]. For each step, the energy difference $\Delta\mathscr{H} = \mathscr{H}_a - \mathscr{H}_b$ between two subsequent states $a$ and $b$ is calculated. The probability for the step to be accepted is given by $W_{a\to b} = \exp(-\Delta\mathscr{H}/k_B T)$ if $\Delta\mathscr{H} > 0$ and $W_{a\to b} = 1$ if $\Delta\mathscr{H} \leqslant 0$. Since only discrete values for this factor are possible, they should be pre-calculated on the CPU for each temperature and transferred to the GPU when the kernels are invoked.

To make efficient use of the GPU device structure, a parallelizable spin-update scheme has to be utilized. The ratio between memory latency and processing time on graphics cards is very large [39]. Thus, GPU cores can perform hundreds of instructions in the time of a single access to the global memory. By highly parallel processing, memory access latencies can be hidden effectively, and large acceleration factors achieved.

Parallel spin updates of the Ising model can only be done for non-interacting domains. The approach that each spin only interacts with its four nearest neighbors makes a checkerboard update feasible [22]. The lattice update is divided into two update steps $A$ and $B$. In step $A$, only the spins residing on a black site are updated since they are not interacting with each other. In step $B$, the spins on white lattice sites are updated. It is essential that update step $B$ is started after all updates of step $A$ are finished. Please note, that other methods for the spin updating process are also available, e.g. diverse cluster algorithms [44,45], perform particularly well close to the critical point. However, the systematic scheme of the checkerboard algorithm is most suitable for the GPU architecture realizing non-interacting domains where the Monte Carlo moves are performed in parallel.

In order to test the correctness of the implementation, we determine the critical temperature of the Ising spin system. We use finite size scaling and calculate the Binder cumulant [46,30]

$$U_4(T) = 1 - \frac{\langle M(T)^4 \rangle}{3\langle M(T)^2 \rangle^2} \tag{2}$$

with $M$ denoting the magnetization of a configuration at temperature $T$ and $\langle \ldots \rangle$ denoting the thermal average. Near a critical point, finite size scaling theory predicts the free energy and derived quantities like the magnetization to be a function of linear dimension $L$ over correlation length $\xi \simeq (T - T_C)^{-\nu}$. Therefore, moment ratios of the magnetization like, e.g. the Binder cumulant $U_4$, become independent of system size $N = n^2$ at the critical temperature $T_C$. To test our implementation, we perform several simulations close to the critical point for various linear dimensions $n$ of the simulation box and determine $U_4$.

## 4. Optimized reference CPU implementation

For our optimized CPU reference implementation, we focus on a single spin-flip approach which performs well for large lattice sizes. Multi-spin coding refers to all techniques that store and process multiple spins in one unit of computer memory. In CPU implementations, update schemes have been developed that allow to process more than one spin with a single operation [34–37]. We use a scheme which encodes 32 spins into one 32-bit integer in a linear fashion. The 32-bit type is chosen since register operations of current hardware perform fastest on this data type. The key ingredient for an efficient update algorithm of these 32-bit patterns is to use precomputed bit patterns that encode the evaluations of the flip condition expression

$$r < \exp(-\Delta\mathscr{H}/k_B T) \tag{3}$$

for every single spin bit—the variate $r$ is an independent and identically-distributed random number in $[0, 1)$. Since there are only two possible energy differences $\Delta\mathscr{H}$ with $\Delta\mathscr{H} > 0$, two Boolean arrays can encode the information of an evaluation of the flip condition. For reasonable results, N. Ito [36] suggested to use a pool of $2^{22}$ to $2^{24}$ Boltzmann patterns.

We call the two Boolean arrays $\exp 4$ and $\exp 8$. Our encoding is chosen to store a 1 into $\exp 4$ if $\exp(-8J/k_B T) < r < \exp(-4J/k_B T)$ and a 0 if not, and a 1 into $\exp 8$ if $r < \exp(-8J/k_B T)$ and a 0 if not.

For every spin update, the Monte Carlo simulation will draw a random address in this pool, and return the bit patterns at this address. To process a 32-bit spin pattern $s_0$, neighbor patterns $s_1, s_2, s_3, s_4$ have to be prepared, that contain the neighbors of the $i$th spin at their $i$th digit.

Since the calculation is the same for every bit, it is convenient to look at just one bit. The first step is to transform the spin variables into *energy variables* $i_n$ to get rid of the dependence of the initial state of $s_0$.

$$i_n = s_0 \wedge s_n, \quad \forall n \in \{1, 2, 3, 4\} \tag{4}$$

where $\wedge$ denotes an exclusive-OR operation. Because of the special encoding of the Boltzmann patterns, the acceptance condition for each spin can be expressed in a simple way:

$$i_1 + i_2 + i_3 + i_4 + 2 \cdot \exp 8_s + \exp 4_s \geqslant 2 \tag{5}$$

where $\exp 8_s$ and $\exp 4_s$ denote the $s$th Boltzmann patterns that encode the spin flip condition, and $s$ is a random position in the pool. It is possible to evaluate this expression for all 32 bits at once by applying a sequence of bitwise Boolean operations [36].

Since parallel updates are only allowed on non-interacting domains, an additional bit mask has to be applied to the update pattern, that only allows to flip every second spin at once. Details of the implementation can be found in [49].

**Table 2**
Comparison at lattice size $4096 \times 4096$: *CPU simple* encodes one spin in one integer, *CPU multi-spin coding* uses the efficient "multi-spin" update scheme presented in Section 4, *multi-spin unmodified* is a straightforward porting of this update to the GPU, *multi-spin coding on the fly* uses the same scheme but calculates the update patterns at each update step on the fly, and *multi-spin coding linear* determines one starting position in the random number pattern in the pool per block, and lets the threads read the random numbers linearly from that position on. The *shared memory* implementation (see Section 5.2) provides best quality random numbers. A preliminary test run on a Fermi GPU shows a factor of 1.82 compared to a Tesla C1060.

| | Spinflips per µs | Relative speed |
|---|---|---|
| *CPU simple* | 26.6 | 0.11 |
| *CPU multi-spin coding* | 226.7 | 1.00 |
| *shared memory* | 4415.8 | 19.50 |
| *shared memory (Fermi)* | 8038.2 | 35.46 |
| *multi-spin unmodified* | 3307.2 | 14.60 |
| *multi-spin coding on the fly* | 5175.8 | 22.80 |
| *multi-spin coding linear* | 7977.4 | 35.20 |

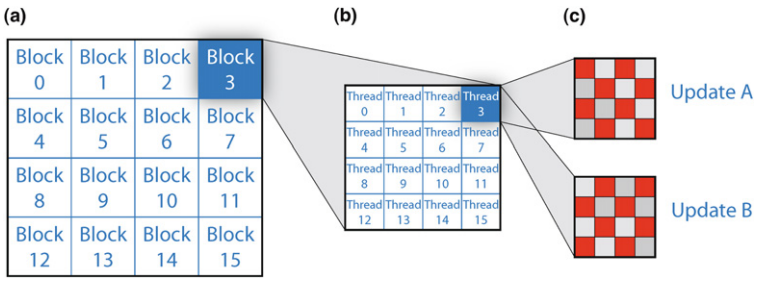


**Fig. 1.** (Color online.) The spin lattice is processed by a variable number of blocks (a), where each block runs a variable number of threads (b). The threads update the spin lattice in two steps, *A* and *B*, using two kernel invocations (c).

## 5. Single-GPU implementation

### 5.1. Problems arising from a straightforward porting

On graphics cards, memory access is very costly compared to operations on registers. The great advantage of multi-spin coding is that only one memory access is needed to obtain several spins at once. The CPU implementation could be ported to GPU with a kernel that uses less than 16 registers. This allows an optimal occupancy of the GPU up to a maximum block size of 512 threads. Even though the update scheme presented in Section 4 performs vastly faster on the CPU compared to an implementation with integer representations of each spin, a straightforward GPU port of this scheme is not optimal.

The reason for the poor performance is that parallel threads in one warp have to access global memory in a random fashion which is very costly. The execution speed can be improved by drawing only one random position per block, and let all the threads in this block read the patterns linearly, starting from the drawn starting position. This however, reduces the quality of the flip patterns—this could in principle be compensated by using a significantly larger pool of random numbers. Another option is to calculate the spin flip patterns on the fly using a random number generator on the GPU (see Section 5.3) instead of looking them up from the global memory. It turns out however, that the sophisticated update scheme has no benefit here, anymore. The performance of these implementations is compared in Table 2. It should be emphasized that the quality of random numbers differs between the implementations. In the next section, we present another update scheme that works well on the GPU, which prevents pitfalls with the quality of the random numbers.

### 5.2. Extraction into shared memory

The main goal of the following implementation is to reduce access to the global memory of the GPU, which is extremely costly. The best performance without pre-calculating flip patterns can be achieved by extracting the single spins into shared memory and performing the calculations on integer registers. The spin field on the graphics card is encoded in quadratic blocks of $4 \times 4$ spins (hereafter referred to as "meta-spins") which can be stored as binary digits of one unsigned short integer (2 bytes), which can be accessed by a single memory lookup. Single spin values can be extracted from one meta-spin for example using the expression

```
s[x,y] = ((meta-spin & (1 << (y * 4 + x)) != 0) * 2 - 1)
```

which returns a value of either $-1$ or 1. Here, "<<" denotes a bitwise left-shift operation. This is a slightly more complicated expression than for a linear layout, but it makes sense for a multi-GPU implementation, where border information has to be transferred between various GPUs (see Section 6). This approach realizes that each spin uses exactly one bit of memory. The spin field is stored in global memory, which is expensive to access.

To process the spin field on the GPU, the spin field is subdivided into quadratic subfields which can be processed by threads grouped into one block (see Fig. 1). Each thread of this block processes a "meta-spin" of $4 \times 4$ spins. At the beginning of a kernel, it retrieves 5 meta-spins from the global memory, namely its own and its four neighboring meta-spins (Fig. 2a). This information is used to extract the information of the $4 \times 4$ spins. Each thread will store the spin field of $4 \times 4$ spins as well as the neighboring spins in a $6 \times 6$ integer array in shared memory, which allows for fast computation of the spin flips. The spin update is performed in two steps as described before.

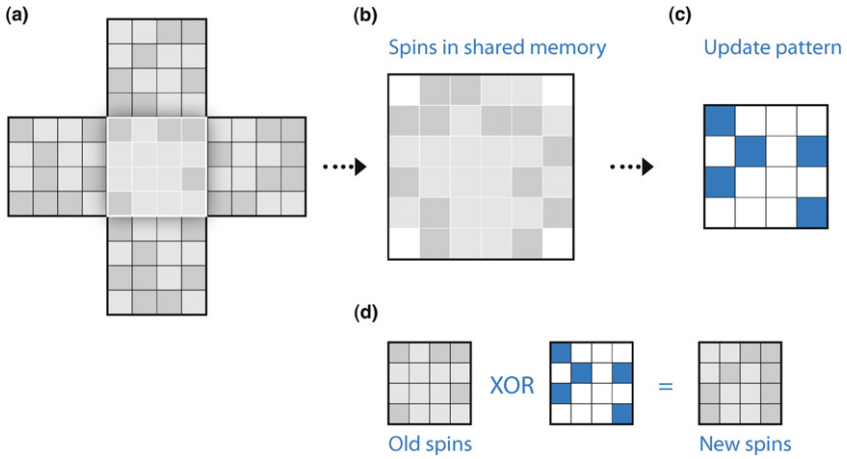


**Fig. 2.** (Color online.) (a) The way a kernel processes a $4 \times 4$ meta-spin. (b) Spins are extracted into shared memory and an update pattern is created (c). (d) Afterwards, the new spins are obtained using the update pattern (spins on blue sites will be flipped, spins on white sites will not be flipped), and written back to global memory.

A first kernel is needed to update the "black sites" on a checkerboard pattern, and a second processes the "white sites". The update kernel for the white sites has to wait until all black sites have been updated. Thus, two separate kernels are needed. There is no other way to achieve global synchronization between the threads.

Each kernel creates an update pattern, where each binary digit indicates if the associated spin has to be flipped or not. At the end of the kernel execution, the $4 \times 4$ meta-spins are updated with one single global memory write.

In summary, each update thread executes the following steps:

1. Look up meta-spins from global memory (see Fig. 2a).
2. Extract meta-spins into $6 \times 6$ integer array in shared memory which then contains the $4 \times 4$ meta-spins and the neighbors (see Fig. 2b).
3. For all 8 white/black sites $s_i$ in the $4 \times 4$ field, draw a random number and evaluate the Metropolis criterion.
4. Generate the update pattern (set the $i$th bit to 1, if the flip of the $i$th was accepted) (Fig. 2c).
5. Update the meta-spin by an XOR operation with the update pattern to obtain the spins at the next timestep (Fig. 2d).

Although the update scheme sounds hardly efficient, it dramatically reduces global memory access compared to the previous implementation, which results in faster computation times on GPU hardware.

After the update is completed, the magnetization per spin $m(T)$ has to be extracted from the lattice. In a first step, the magnetization of each block can be summed using the shared memory of each block by employing a binary tree reduction and writing out the total magnetization of the slice back to the main memory. The final summation of the magnetizations of the blocks can be done either on the CPU or on the GPU at about equal speeds.

### 5.3. Random number generation

For every update thread a random number is needed, either to decide if the spin is flipped or not, or to look up an update pattern in global memory. This is why an efficient method to create random numbers is needed.

In our implementation, we use an array of linear congruential random number generator (LCRNG) which is one of the oldest and most studied algorithm to generate pseudo random numbers [47]. A single random number generator provides the random numbers for every update thread $j$. A sequence of random numbers for the $j$th thread $x_{i,j}$ (where $i \in \mathbb{N}$) is generated by the recurrence relation

$$x_{i+1,j} = (a \cdot x_{i,j} + c) \bmod m \tag{6}$$

where $a, c$ and $m$ are integer coefficients. An appropriate choice of these coefficients is responsible for the quality of the produced random numbers. We use $a = 1{,}664{,}525$ and $c = 1{,}013{,}904{,}223$ as suggested, e.g., in [48]. Since by construction, results on a 32-bit register are truncated to the endmost 32 bits, the modulo operation $m$ is set to $2^{32}$. By normalizing ($y_{i,j} = \mathrm{abs}(x_{i,j}/2^{31})$) the LCRNG can be used to generate random numbers $y_{i,j}$ in the interval $[0; 1)$. For the GPU, an array of random numbers that provides a single random number seed for every spin update thread can be generated by the iteration

$$x_{0,j+1} = (16807 \cdot x_{0,j}) \bmod m \tag{7}$$

with $x_{0,0} = 1$.

### 5.4. Performance comparison

The multi-spin implementations are compared to simple implementations on both CPU and GPU (see Fig. 3). As a measurement for the performance of an implementation, we use the number of single spin flips per second, which also allows to compare results for different lattice sizes. The temperature is set to $0.99 T_C$.

The GPUs perform most efficient for lattice sizes of a linear dimension beyond $4096 \times 4096$. For this lattice size, a GPU is faster by a factor of about 15–35, depending on the implementation and the resulting quality of random numbers. For the simple implementation

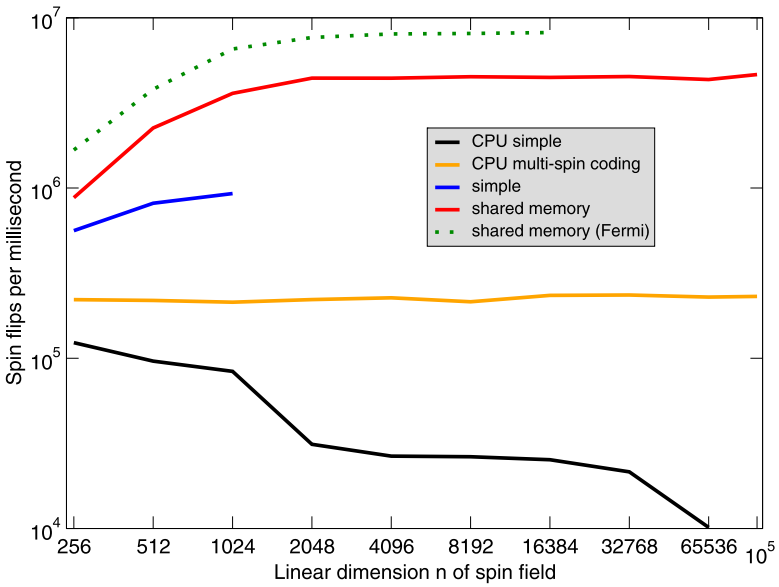


**Fig. 3.** (Color online.) Benchmarking the implementations: The system is simulated at a constant temperature of $T = 0.99T_C$. The performance of the straight port varies strongly with lattice size because of the large block size of 512 threads, while the shared memory implementation offers stable performance over a wide range of sizes and offers better quality random numbers (comparable to the simple CPU implementation). The dotted line shows a preliminary benchmark of a Fermi GPU which became available in April 2010. A GeForce GTX 480 provides the following features: 1536 MB global memory, 480 streaming processor cores, 1.40 GHz processor clock, 1848 MHz memory clock, and a maximal power consumption of 250 W.

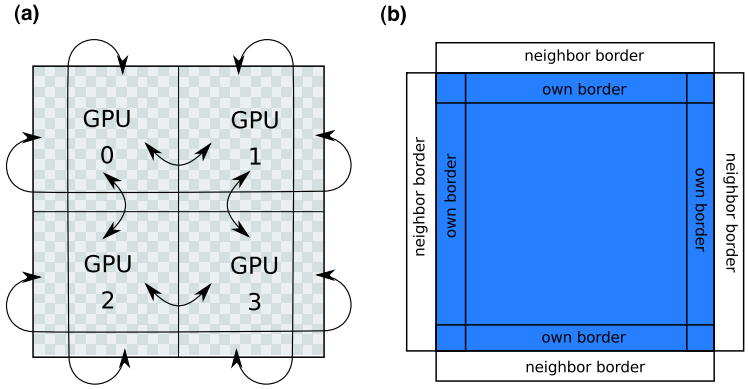


**Fig. 4.** (Color online.) (a) Each GPU processes a "meta-spin" lattice of size $N = n^2$. The lattices are aligned on a super-lattice, and the outer borders are connected via periodic boundary conditions. In this example, 4 GPUs process a system of $2^2 \cdot N$ spins. (b) A meta-spin update needs the 4 nearest neighbor meta-spins. On the borders of a lattice, each GPU needs the spin information of the neighboring lattices. The border information has to be passed between the GPUs. In our implementation this is done by using 8 neighbor arrays.

used in [22], between $1024 \times 1024$ and $2048 \times 2048$ spins the spin field size becomes comparable to the CPU L3 cache size, which leads to a higher rate of costly L3 cache misses. This is the point at which the simple implementation becomes inefficient.

## 6. Multi-GPU approach

### 6.1. Implementation

The general idea is to extend the quadratic lattice by putting multiple quadratic "meta-spin" lattices next to each other in a super-lattice (see Fig. 4a for a $2 \times 2$ super-lattice) and let each lattice be handled by one of the installed GPUs. On the border of each lattice, at least one of the neighboring sites is located in the memory of another GPU (see Fig. 4b). For this reason, the spins at the borders of each lattice have to be transferred from one GPU to the GPU handling the adjacent lattice. This can be realized by introducing four neighbor arrays containing the spins of the lattices' own borders, and four arrays for storing the spins of its adjacent neighbors (Fig. 4c).

At the beginning of the execution, each MPI process initializes its own spin lattice, writes out its border spins into its own border arrays and sends them to its neighbors. In return it receives the adjacent borders from the according MPI processes. After this initialization phase, spins and random seeds are transferred to the GPU.

Then, a single lattice update has to be performed in the following way:

1. Copy neighbor borders to GPU memory
2. Call kernel to perform update (A)
3. Call kernel to extract borders from the spin array to own borders array

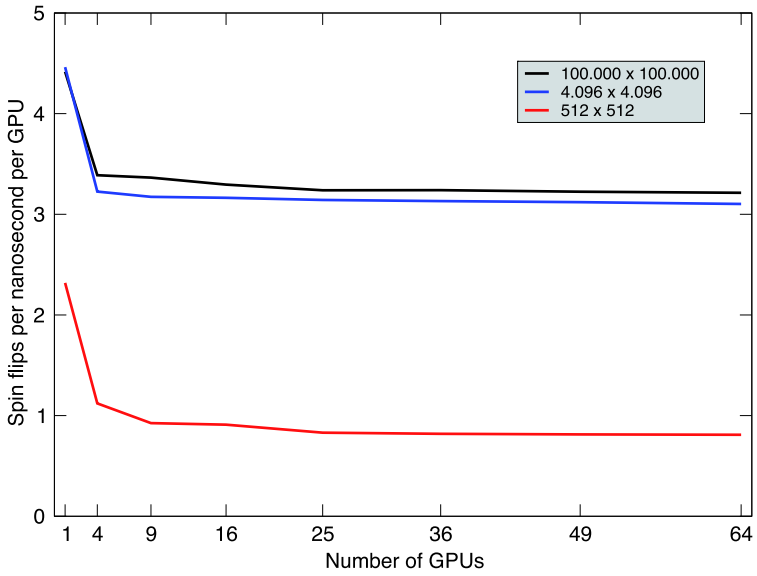


**Fig. 5.** (Color online.) Cluster performance for various system sizes (per GPU). For more than one GPU, spin flip performance scales nearly linearly with the amount of GPUs. Again, optimal performance is reached at a lattice size of about $4096 \times 4096$ per GPU. Using 64 GPUs, a performance of 206 spinflips per nanosecond can be achieved on a $800{,}000 \times 800{,}000$ lattice.

4. Copy own borders to host memory
5. Exchange borders with the other MPI processes
6. Copy neighbor borders to GPU memory again
7. Call kernel to perform update (B)
8. Call kernel to extract borders from spin array again
9. Transfer own borders to host memory
10. Exchange borders with other MPI processes
11. Retrieve processed data from GPU

It turns out that the transfer time was not the limiting factor for our purposes but rather the latency of the memory accessed.

### 6.2. GPU cluster performance

For performance measurements on the GPU cluster, the shared memory implementation (see Section 5.2) was used, since it provided stable performance for various lattice sizes and because the memory layout is symmetric in the $x$- and $y$-direction, resulting in symmetric communication data. The tests were run on a GPU cluster with two Tesla C1060 GPUs in each node. Communication is established via Double Data Rate InfiniBand. The performance for various system sizes (see Fig. 5) provides evidence that for more than one GPU, spin flip performance scales nearly linearly with the amount of GPUs. The drop from one GPU to four GPUs is due to the communication overhead produced by exchanging borders. For larger system sizes, the communication overhead per CPU/GPU remains constant. An optimal performance is reached for lattice sizes beyond $4096 \times 4096$ per GPU. For 64 GPUs—the NEC Nehalem Cluster maintained by the High Performance Computing Center Stuttgart (HLRS) provides 128 GPUs—, a performance of 206 spinflips per nanosecond can be achieved on a $800{,}000^2$ 2D Ising lattice, i.e. we can update the whole lattice in about three seconds.

## 7. Conclusion

We presented two major improvements over our previous work. By using multi-spin coding techniques, we improved the computation to memory access ratio of our calculations dramatically, resulting in better overall performance. On a single GPU, up to 7.9 spinflips per nanosecond are possible, 15 to 35 times faster than our highly optimized CPU version, depending on the implementation and the quality of random numbers. The other improvement targets the utilization of GPU clusters, where the 2D Ising lattice is distributed over many GPUs. We show that our implementation scales nearly linearly with the number of GPUs, which allows us to process huge Ising lattices on GPU clusters. Preliminary tests on an NVIDIA GPU of the latest generation—the Fermi architecture which offers twice the amount of streaming processor cores—indicate an additional speedup of roughly 1.8 compared to a Tesla C1060.

## Acknowledgements

We thank K. Binder, D. Stauffer, and M. Tacke for fruitful discussions. GPU time was provided on the NEC Nehalem Cluster by the High Performance Computing Center Stuttgart (HLRS). We are grateful to T. Bönisch, B. Krischok, and H. Pöhlmann for their support at the HLRS. This work was financially supported by the Deutsche Forschungsgemeinschaft (DFG) and benefited from the Gutenberg-Akademie and the Schwerpunkt für rechnergestützte Forschungsmethoden in den Naturwissenschaften and the Materialwissenschaftliches Forschungszentrum of the Johannes Gutenberg University Mainz.

## References


[1] E.B. Ford, Parallel algorithm for solving Kepler's equation on graphics processing units: Application to analysis of Doppler exoplanet searches, New Astronomy 14 (2009) 406–412, doi:10.1016/j.newast.2008.12.001.
[2] C. Harris, K. Haines, L. Staveley-Smith, GPU accelerated radio astronomy signal convolution, Experimental Astronomy 22 (2008) 129–141, doi:10.1007/s10686-008-9114-9.
[3] Z.A. Taylor, O. Comas, M. Cheng, J. Passenger, D.J. Hawkes, D. Atkinson, S. Ourselin, On modelling of anisotropic viscoelasticity for soft tissue simulation: Numerical solution and GPU execution, Medical Image Analysis 13 (2009) 234–244.
[4] X. Gu, H. Pan, Y. Liang, R. Castillo, D. Yang, D. Choi, Implementation and evaluation of various demons deformable image registration algorithms on a GPU, Physics in Medicine and Biology 55 (2009) 207–219, doi:10.1088/0031-9155/55/1/012.
[5] X. Gu, D. Choi, C. Men, H. Pan, A. Majumdar, S. Jiang, GPU-based ultra-fast dose calculation using a finite size pencil beam model, Physics in Medicine and Biology 54 (2009) 6287–6297, doi:10.1088/0031-9155/54/20/017.
[6] D. Gross, U. Heil, R. Schulze, E. Schömer, U. Schwanecke, GPU-based volume reconstruction from very few arbitrarily aligned X-ray images, SIAM Journal on Scientific Computing 31 (2009) 4204–4221, doi:10.1137/080736739.
[7] C.H. Men, X.J. Gu, D.J. Choi, A. Majumdar, Z.Y. Zheng, K. Mueller, S.B. Jiang, GPU-based ultrafast IMRT plan optimization, Physics in Medicine and Biology 54 (2009) 6565–6573, doi:10.1088/0031-9155/54/21/008.
[8] C. Trapnell, M.C. Schatz, Optimizing data intensive GPGPU computations for DNA sequence alignment, Parallel Computing 35 (2009) 429–440, doi:10.1016/j.parco.2009.05.002.
[9] J.A. Anderson, C.D. Lorenz, A. Travesset, General purpose molecular dynamics simulations fully implemented on graphics processing units, Journal of Computational Physics 227 (2008) 5342–5359, doi:10.1016/j.jcp.2008.01.047.
[10] M.S. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. Legrand, A.L. Beberg, D.L. Ensign, C.M. Bruns, V.S. Pande, Accelerating molecular dynamic simulation on graphics processing units, Journal of Computational Chemistry 30 (2009) 864–872, doi:10.1002/jcc.21209.
[11] J.A. van Meel, A. Arnold, D. Frenkel, S.P. Zwart, R.G. Belleman, Harvesting graphics power for MD simulations, Molecular Simulation 34 (2008) 259–266, doi:10.1080/08927020701744295.
[12] I.S. Ufimtsev, T.J. Martinez, Quantum chemistry on graphical processing units. 1. Strategies for two-electron integral evaluation, Journal of Chemical Theory and Computation 4 (2009) 222–231, doi:10.1021/ct700268q.
[13] N.A. Gumerov, R. Duraiswami, Fast multipole methods on graphics processors, Journal of Computational Physics 227 (2008) 8290–8313, doi:10.1016/j.jcp.2008.05.023.
[14] L. Genovese, M. Ospici, T. Deutsch, J.F. Mehaut, A. Neelov, S. Goedecker, Density functional theory calculation on many-cores hybrid central processing unit–graphic processing unit architectures, Journal of Chemical Physics 131 (2009) 034103, doi:10.1063/1.3166140.
[15] K. Yasuda, Accelerating density functional calculations with graphics processing unit, Journal of Chemical Theory and Computation 4 (2008) 1230–1236, doi:10.1021/ct8001046.
[16] F. Molnar, T. Szakaly, R. Meszaros, I. Lagzi, Air pollution modelling using a graphics processing unit with CUDA, Computer Physics Communications 181 (2010) 105–112, doi:10.1016/j.cpc.2009.09.008.
[17] T. Preis, P. Virnau, W. Paul, J.J. Schneider, Accelerated fluctuation analysis by graphic cards and complex pattern formation in financial markets, New Journal of Physics 11 (2009) 093024, doi:10.1088/1367-2630/11/9/093024.
[18] T. Preis, W. Paul, J.J. Schneider, Fluctuation patterns in high-frequency financial asset returns, Europhysics Letters 82 (2008) 68005, doi:10.1209/0295-5075/82/68005.
[19] J. Yin, D.P. Landau, Phase diagram and critical behavior of the square-lattice Ising model with competing nearest-neighbor and next-nearest-neighbor interactions, Physical Review E 80 (2009) 051117, doi:10.1103/PhysRevE.80.051117.
[20] A. Badal, A. Badano, Accelerating Monte Carlo simulations of photon transport in a voxelized geometry using a massively parallel graphics processing unit, Medical Physics 36 (2009) 4878–4880, doi:10.1118/1.3231824.
[21] J.S. Meredith, G. Alvarez, T.A. Maier, T.C. Schulthess, J.S. Vetter, Accuracy and performance of graphics processors: A quantum Monte Carlo application case study, Parallel Computing 35 (2009) 151–163, doi:10.1016/j.parco.2008.12.004.
[22] T. Preis, P. Virnau, W. Paul, J.J. Schneider, GPU accelerated Monte Carlo simulation of the 2d and 3d Ising model, Journal of Computational Physics 228 (2009) 4468–4477, doi:10.1016/j.jcp.2009.03.018.
[23] M.J. Harvey, G.D. Fabritiis, An implementation of the smooth particle mesh Ewald method on GPU hardware, Journal of Chemical Theory and Computation 5 (2009) 2371–2377, doi:10.1021/ct900275y.
[24] T. Preis, H.E. Stanley, Switching phenomena in a system with no switches, Journal of Statistical Physics 138 (2010) 431–446, doi:10.1007/s10955-009-9914-y.
[25] H.E. Stanley, S.V. Buldyrev, G. Franzese, S. Havlin, F. Mallamace, P. Kumar, V. Plerou, T. Preis, Correlated randomness and switching phenomena, Physica A 389 (2010) 2875–2888.
[26] D. Reith, P. Virnau, Implementation and performance analysis of bridging Monte Carlo moves for off-lattice single chain polymers in globular states, Computer Physics Communications 181 (2010) 800–805, doi:10.1016/j.cpc.2009.12.012.
[27] E. Ising, Beitrag zur Theorie des Ferromagnetismus, Z. Phys. 31 (1925) 253–258.
[28] K. Binder, E. Luijten, Monte Carlo tests of renormalization-group predictions for critical phenomena in Ising models, Physics Reports 344 (2001) 179–253, doi:10.1016/S0370-1573(00)00127-7.
[29] N. Ito, Parallelization of the Ising simulation, International Journal of Modern Physics C 4 (1993) 1131–1135.
[30] D. Landau, K. Binder, A Guide to Monte Carlo Simulations in Statistical Physics, 2nd edition, Cambridge University Press, Cambridge, 2005.
[31] H.E. Stanley, D. Stauffer, J. Kertesz, H.J. Herrmann, Dynamics of spreading phenomena in two-dimensional Ising models, Physical Review Letters 59 (1987) 2326–2328.
[32] D. Stauffer, Kinetics of clusters in Ising-models, Physica A 186 (1992) 197–209.
[33] L. Onsager, Crystal statistics. I. A two-dimensional model with an order-disorder transition, Physical Review 65 (3–4) (1944) 117–149, doi:10.1103/PhysRev.65.117.
[34] S. Wansleben, J.G. Zabolitzky, C. Kalle, Monte Carlo simulation of Ising models by multispin coding on a vector computer, Journal of Statistical Physics 37 (1984) 271–282.
[35] R. Zorn, H.J. Herrmann, C. Rebbi, Parallelization of the Ising simulation, Computer Physics Communications 23 (1981) 337–342.
[36] N. Ito, Y. Kanada, An effective algorithm for the Monte-Carlo simulation of the Ising-model on a vector processor, Supercomputer 25 (1988) 31.
[37] N. Ito, Y. Kanada, Monte Carlo simulation of the Ising model and random number generation on the vector processor, Proceedings of Supercomputing 90 (1990) 753–763, doi:10.1109/SUPERC.1990.130097.
[38] T. Akenine-Möller, E. Haines, N. Hoffman, Real-Time Rendering, A.K. Peters, Ltd., 2008.
[39] NVIDIA Corporation, NVIDIA CUDA—Programming Guide, version 2.3.1, 2009.
[40] NVIDIA Corporation, OpenCL Programming Guide for the CUDA Architecture, version 2.3, 2009.
[41] P. Messmer, P.J. Mullowney, B.E. Granger, GPULIB: GPU computing in high-level languages, Computing in Science and Engineering 10 (2008) 70–73.
[42] NVIDIA Corporation, NVIDIA Tesla C1060 Specifications, 2009.
[43] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, E. Teller, Equation of state calculations by fast computing machines, Journal of Chemical Physics 21 (6) (1953) 1087–1092, doi:10.1063/1.1699114.
[44] R.H. Swendsen, J.-S. Wang, Nonuniversal critical dynamics in Monte Carlo simulations, Physical Review Letters 58 (2) (1987) 86–88, doi:10.1103/PhysRevLett.58.86.
[45] U. Wolff, Collective Monte Carlo updating for spin systems, Physical Review Letters 62 (4) (1989) 361–364, doi:10.1103/PhysRevLett.62.361.
[46] K. Binder, Finite size scaling analysis of Ising model block distribution functions, Z. Phys. B 43 (1981) 119–140, doi:10.1007/BF01293604.
[47] J.J. Schneider, S. Kirkpatrick, Stochastic Optimization, Springer, Berlin, 2006.
[48] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, Numerical Recipes: The Art of Scientific Computing, Cambridge University Press, Cambridge, 2007.
[49] B. Block, Diploma Thesis, Johannes Gutenberg University Mainz, 2010.