Project 9a: XY-Model

Computational Physics (physics760)

Project Report

Daniel Weller

HISKP, University of Bonn

Bonn, March 2011

Abstract

This document summarizes my work regarding the implementation of the xy-Model using cuda. First, a short overview of the project is given. Then theoretical as well as technical aspects will be discussed. The report closes after presenting measured properties of the model with an analysis on the shift of the critical temperature due to finite size effects with a lineup of calculation times on different hardware.

Contents

1 Introduction

In the following, the properties of the xy-model descending from a simulation with the Metropolis algorithm will be discussed. The xy-model in this context is treated as a two-dimensional lattice described by the Hamiltonian

$$\mathcal{H}(\mathbf{s}) = -\sum_{\langle i,j \rangle} s_i \cdot s_j = -\sum_{\langle i,j \rangle} \cos(\phi_i - \phi_j) \quad , \tag{1.1}$$

where $\langle i, j \rangle$ denotes all next neighbouring pairs of spins s_i. The state of each two-dimensional spin is given by its angle $\phi_i \in \{0..2\pi\}$, the state of the complete lattice is given by $\boldsymbol{s} = (s_1, s_2, \ldots, s_N)$.

The aim of the simulation is to measure the temperature and size dependence of different observable parameters of the model. Also the occuring phase transition shall be demonstrated. The so-called *Kosterlitz-Thouless*-transition yields a critical temperature T_c, its value can be extracted within this project.

My motivation to implement the algorithm with cuda is based on my employment at *Institut für Numerische Simulation*, University of Bonn, which is listed as one of the first "cuda Research Centers" in Germany in [2]. The xy-model seems to be a decent start into the subject of parallelization.

2 Theoretical and Technical Aspects

2.1 XY-Model and Monte Carlo: Metropolis Algorithm

In addition to the Hamiltonian given in equation 1.1, the partition function of the xy-model is given by

$$\mathcal{Z} = \int \mathrm{d}s \exp(-\mathcal{H}/(k_B T)) \quad . \tag{2.1}$$

According to the Metropolis algorithm, a sequence of random samples resp. observables at a given temperature[1] T is generated. Prior to measurement, the system has to reach thermal equilibrium. Hence, a certain quantity of monte-carlo-steps is necessary before the start of recording. From each recorded sample, the mean value and its statistical error is obtained.

To execute lattice updates in a preferably parallel fashion, a *checker-board*-like segmentation of the spins as denoted in figure 2.1 seems promising. Since the Hamiltonian only depends on nearest neighbour interaction, the energy difference Δh as well

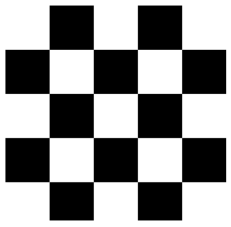


Fig. 2.1: *Checker board: all white (black) fields are updated at once.*

[1] in the following, k_B will be considered as equal to 1

has the same dependence. The slightly modified procedure looks as follows:

1. Start with initial configuration $\boldsymbol{s}$ (random or defined values)
2. Perform local update of each spin on "white" lattice site simultaneously:
   (a) uniformly generate new state, i.e. randomly choose angle from $\{0..2\pi\}$
   (b) accept update with probability $p = \min\left(1, \exp\left(-\Delta h/T\right)\right)$
3. Repeat step 2 for all "black" sites
4. if applicable, take measurement
5. continue from step 2.

To simplify things, only squared arrangements with side length L were considered. The implementation with cuda gives further limitations, which will be mentioned in Section 2.3.

2.2 Kosterlitz-Thouless Transition

In contrast to the previous work of several other authors[2], Kosterlitz and Thouless 1973 showed that the spin model treated in this project indeed posesses a phase transition [1]. However, the transition is not "of the usual type". Kosterlitz and Thouless argue that below some critical temperature T_c, so called *vortices* occure in metastable states. Above T_c, the vortices become free. The phase transition is then given by a "sudden change in the response to an applied magnetiv field."[3]

Vortices describe topological formations of multiple spins. They are parted into a *clockwise* and *counterclockwise* group, whereas occurrence is only pairwise. Different configurations contain spins aligned circle-wise or diametric around a center point. Also a somehow "repelling" appearence is possible. Nevertheless, in terms of this project, only indirect implications of the Kosterlitz-Thouless-transition will be observed. Employing measurements on the magnetic susceptibility and other observables (see sec. 3), the critical temperature will be determined.

2.3 CUDA

The Compute Unified Device Architecture (acronym: cuda) is a technique developed by Nvidia. It allows for comparatively straightforward implementation of parallel algorithm

[2] Mermin and Wagner 1966, Wegner 1967. Berezinskii 1970, as mentioned in [1]
[3] [1], p. 1190

to be run on graphics devices. The advantage of this technique is based on the nature of graphics cards: the typical purpose is to perform a huge amount of rather simple calculations in parallel, for instance to build up each pixel of an image. With steady progression, the capability of the devices rises. More complex arithmetical instructions became feasible, and with it, computations for scientific tasks can benefit from the speedup through parallelization with cuda [3].

One common task is to determine the average value of an array of numbers. The idea is to reduce the amount of data by a factor (commonly 256), and finish the calculation on the CPU. Implementation with cuda may look like this:

Listing 1: value reduction

```
1  __global__ void
2  getAverage( real* field, real* result ) {
3  __shared__ real avg[blockDim.x];
4  avg[threadIdx.x] =
5          field[ blockDim.x*blockIdx.x + threadIdx.x ];
6
7  for (uint stride = blockDim.x/2; stride > 1; stride/=2)
8  {
9          __syncthreads();
10         if (threadIdx.x < stride) {
11                 avg[threadIdx.x] =
12                         (avg[threadIdx.x]+avg[threadIdx.x+stride])/2.0;
13         }
14 }
15 __syncthreads();
16
17 if (threadIdx.x == 0) result[blockIdx.x] = avg[0];
18 }
```

The so-called kernel is executed via `getAverage<<<blocks,threads>>>(input,output);`, which stems from the organisation of threads in cuda. As included in the example code, the kernel operates on a grid, subdivided into blocks. Each block consists of several threads, and each thread performs the same procedure defined by the kernel. To distinguish between different threads and blocks, coordinates are attached to every element.

2.4 Parallel Random Number Generation

When simulating the xy-model in parallel, one has to consider the availability of random numbers[4]. Each iteration requires $2 \cdot L^2$ independent random values in the graphics device,

[4] in terms of this report: random numbers actually means pseudo random numbers.

which cannot be drawn simply from common functions like `random()`. Different approaches will be discussed in this section.

2.4.1 Generation with CPU

The naive or simple approach to feed random numbers to the algorithm is to create the required data on the host CPU and copy it to the device memory. The kernel executing spin updates then pulls the appropriate value from memory. Creation and update alternate throughout the simulation. The quality of the random numbers depends on the generator. Well established methods (gsl_rng with various generators, e.g. mt19937 [5]) exist. However, the performance upgrade gained by parallelization suffers from the massive overhead of transferring memory from host to device.

2.4.2 Generation with GPU

To avoid memory transfer, random numbers should be produced on the device. Again, an alternating pattern can be implemented, but also a device function to pickup a number *on the fly* is possible. The following paragraphs give a short outline about different approaches.

curand Curand is a library that "provides facilities that focus on the simple and efficient generation of high-quality pseudorandom numbers" [6]. The library available only in the newest version of the cuda toolkit – 3.2 – contains a host API as well as a device API. The first instance can be used for the alternating fashion, the second is useful for a call within each thread. According to [6], the rng used for a uniform distribution has a period "greater than 2^{190}" [6], which is sufficiently large for lattices of reasonable sizes. As of first tests with curand, the documentation was unclear about the period of the generator. The document stated a period of "$2^{192} - 2^{31}$", whereas the papers cited in the document describe the algorithm to have a period of "2^{32} up to 2^{192}", depending on the configuration. Also, the cluster of the HISKP does not have the according version of cuda installed at this time. Hence, further progress was done without the curand library.

ranlux There exist at least some implementations for parallel random number generation. One example I have examined is an implementation of the *ranlux* algorithm. Although the code found in [13] provides seemingly random numbers (not tested), the implementation itself seems to fail in stability: Somehow, only a first fraction of the array got filled, the rest kept its value (mostly 0). Even intensive study did not help to clarify this sporadic misbehaviour [5].

[5] Most likely, i failed using the code correctly.

Mersenne Twister The developers from Nvidia provide a sample package to the cuda toolkit. Amongst examples for the proper use of curand, a parallel implementation of the Mersenne Twister [14] is included. With small adjustments, the code from the SDK has been taken to satisfy the need for random numbers.

The discussed implementation uses 4096 parallel instances of the MersenneTwister algorithm. Each twister is initialized with the same configuration parameters, and subsequently seeded with the same integer value. Thus, for small amounts of numbers, each instance gives highly correlated values. To overcome this property, the code was changed to provide each instance with an initial seed via `random()`.[6] Hence, for small numbers, the returned values are distributed by `random()`, for larger amounts, the independent instances of the twister provide "good randomness".

3 Discussion of Measured Observables

To characterize the temperature dependent behaviour of the xy-model, the observables described in the following sections have been measured. Each measurement was taken after an appropriate count of monte carlo steps to reach thermal equilibrium. The number of datapoints has been chosen depending on the size of the lattice. Since the mentioned occurence of vortices at higher temperatures, the system can fall into semi-stable states when lowering the temperature. Therefore the measurement was taken once with "hot" initial values (random angles) starting with high temperature, and once with a "cold start" (all angles have the same value) starting at low temperature. The system was then given time to reach thermal equilibrium, and subsequently measured over several steps. Then, the temperature was adjusted and after reaching balance, the next measurement was taken.

3.1 Energy per Spin

The total energy of spins is given by eq. 1.1. For low temperature, the energy divided by the number of lattice sites reaches its minimum at -2, as shown in fig. 3.1. Its slope changes at about the critical temperature from linearly rising to asymptotically aproaching zero. For all lattice sizes measured, the energy dependence was the same.

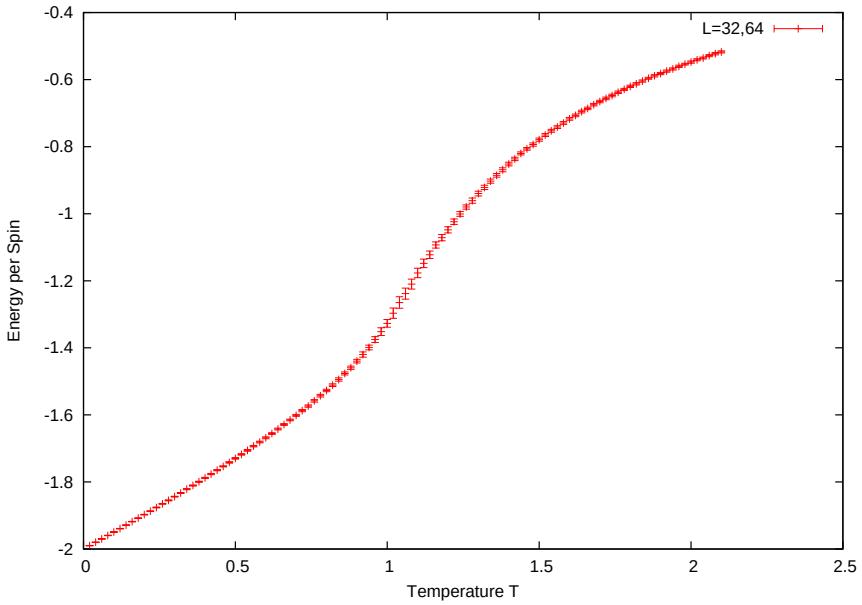


Fig. 3.1: *Temperature dependence of the energy per spin.*

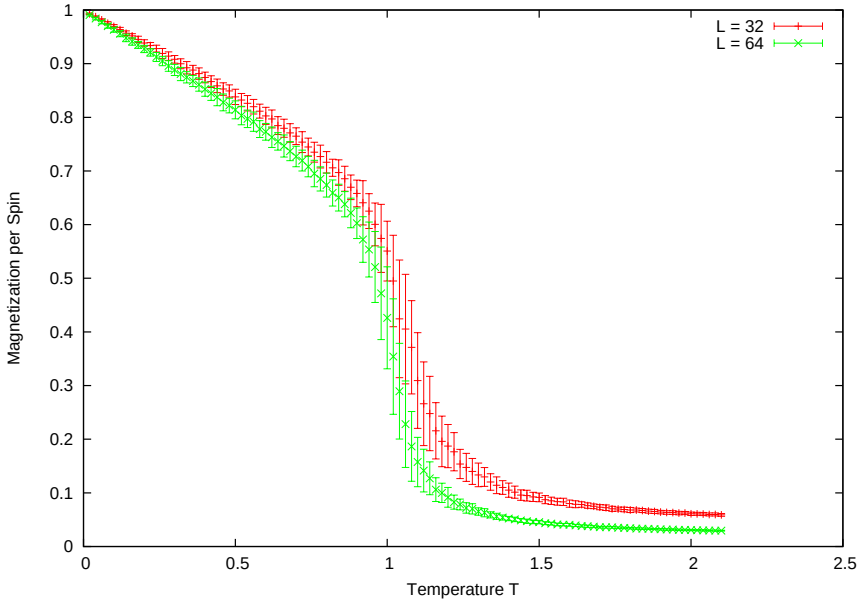


Fig. 3.2: *Norm of magnetization per spin.*

3.2 Magnetization

The magnetization per Spin (cf. fig. 3.2) – or to be more precise: the absolute value of the total magnetization – is given by

$$|\boldsymbol{M}|^2 = \left(\sum_i \cos\phi_i\right)^2 + \left(\sum_i \sin\phi_i\right)^2 \quad . \tag{3.1}$$

The magnetization starts at about 1. With rising temperature, the alignment of spins breaks up. Depending on the size of the lattice, the slope of the curve varies: the bigger L, the steeper the drop at the critical temperature. The minimum distance to the zero-axis gets smaller for bigger side lengths.

3.3 Specific Heat

By using the variance of the energy, one obtains the specific heat by evaluation of the following expression:

$$C_V = \frac{1}{T^2}\left(\langle E^2\rangle - \langle E\rangle^2\right) \tag{3.2}$$

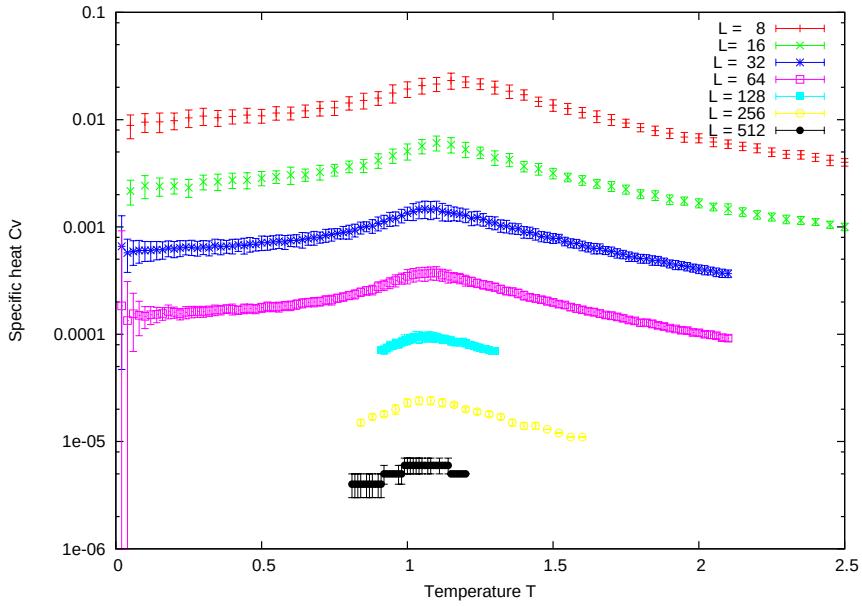


Fig. 3.3: *Logarithmic plot of the specific heat for different lattice sizes.*

This observable is very sensitive to the lattice size. As shown in fig. 3.3, the temperature at which the maximum of the curve is settled varies. This is because of finite size effects:

[6] After each call of the appropriate kernel, the twisters have to be reseeded, otherwise an identical stream of random numbers will be returned.

The critical temperature is shifted, depending on L. Finite size effects will be discussed in section 4.

3.4 Magnetic Susceptibility

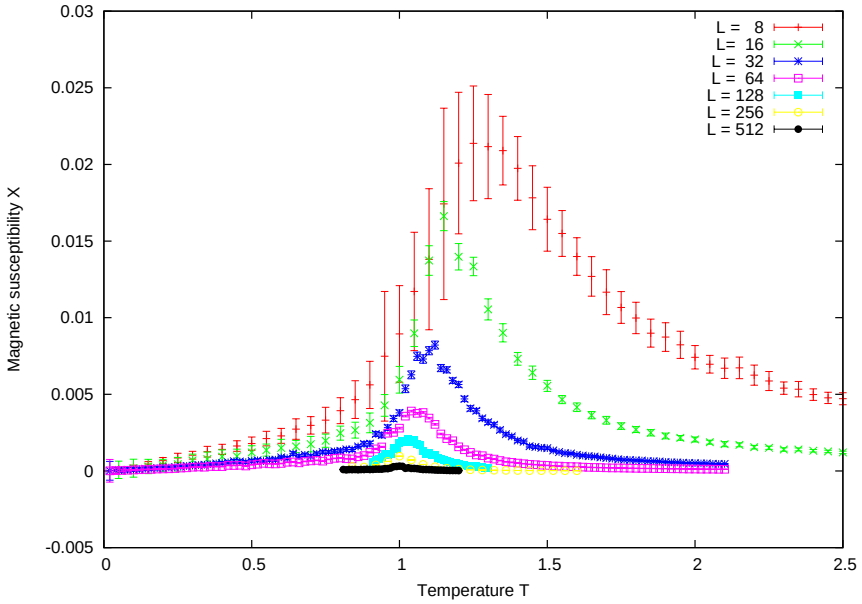


Fig. 3.4: *Magnetic susceptibility.*

Alike, the magnetic susceptibility is calculated, and the results are plotted in fig. 3.4:

$$\chi = \frac{1}{T}\left(\langle M^2\rangle - \langle M\rangle^2\right) \quad . \tag{3.3}$$

The location of the maximum again depends on the finite size of the lattice, as well as the height of the peak[7].

3.5 Helicity Modulus

From [9], eq. 3.1, a formula for the so-called *spin stiffness* or *helicity modulus* can be drawn:

$$\Upsilon = -\frac{1}{2}\langle E\rangle - \frac{1}{T}\left\langle \left(\sum_{\langle i,j\rangle} \sin\left(\phi_i - \phi_j\right)\vec{r}_{i,j}\cdot\vec{e}\right)^2 \right\rangle \quad , \tag{3.4}$$

with $\vec{r}_{i,j}$ being the vector from site i to j, and $\vec{e}$ is an arbitraty unit vector. The spin stiffness is a measure for the energy change caused by a rotation on "the order parameter of

[7]Note that the maximum of the total susceptibility rises with increasing L. However, plot 3.4 showing the value per spin illustrates the size dependence more clearly.

a magnetically long-range-order system along a given direction by a small angle" [10]. For small temperatures, there is a long range order, so Υ is close to 1. For temperatures above T_c, the long range order is destroyed. Hence, Υ dropps to zero. An interesting property of

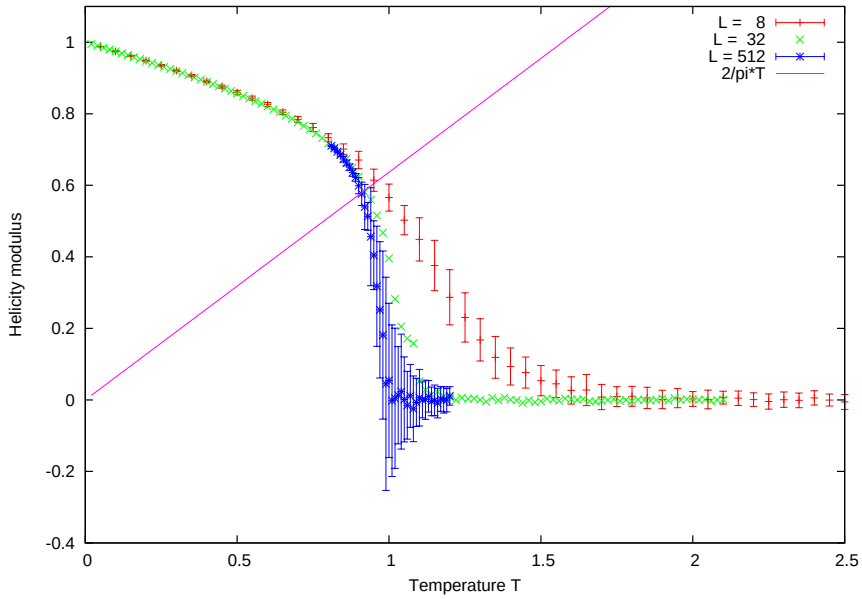


Fig. 3.5: *Intersection of the spin stiffness with* $^2/_\pi T$.

the spin stiffnes is shown in fig. 3.5: the intersection of the graph with a line of slope $^2/_\pi$ gives an approximation for T_c [9]. This will be discussed in the following section.

4 Dealing with Finite Size Effects

According to [8], the shifted temperature is given by

$$T^*(L) \approx T_c + \frac{\pi^2}{4c(\ln L)^2} \quad . \tag{4.1}$$

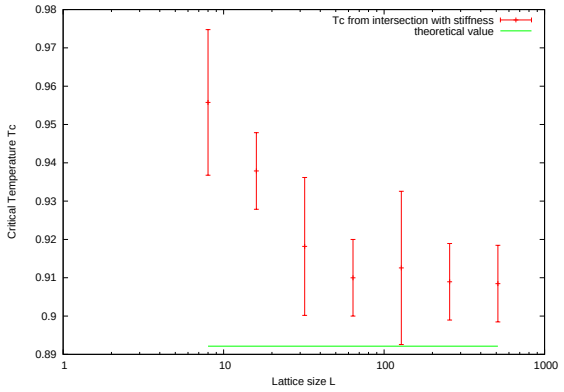
The value of $T^*(L)$ for an infinitely large lattice should be the same as the critical temperature $T_c \approx 0.89213(10)$ determined in by Olsson in [11]. Figure 4.1a shows the values extracted from the measurements of the spin stiffness. Figure 4.1b is a plot from values obtained by examining the maximum of the susceptibility.

The value extracted with a linear fit in fig. 4.1b is

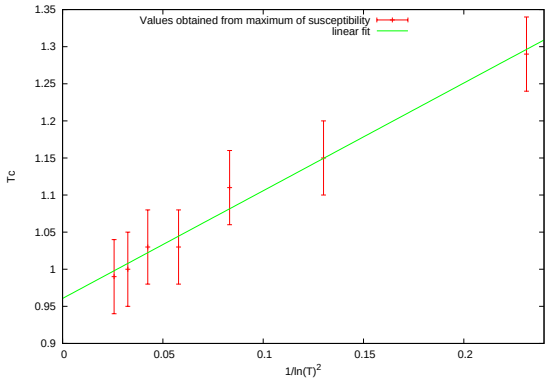
$$T_c = 0.96 \pm 0.1 \quad , \tag{4.2}$$

and overestimates T_c. Nevertheless, in [12] Palma et al. found similar deviations from the theoretical value. The paper yields $T_c \approx 0.94$[8]

[8]extracted from [12], figure 2



(a) *T_c obtained by intersection (cf. fig. 3.5).*



(b) *Estimating T_c with a linear fit.*

Fig. 4.1: *Estimates for the critical temperature.*

5 Conclusion

The results presented in the previous chapters agree with the expectations based on references (both papers and comparison with results from implementation solely on the CPU). The temperature dependence of different observables in the xy-model have been measured, using both "state of the art" hardware such as `Tesla C2050` as well as "consumer electronics", namely the `ION`-chip resp. `GeForce 9400M`. To finalize this work, the following figures show the average computation time for different tasks on different hardware: Fig. 5.1 depicts the time necessary for the update algorithm. The time consumption of one complete iteration (update + measurement) is plotted in fig. 5.2. Calculations have been cut to about 1 percent of the initial time. Even considering latest generation processors will still leave a multiplier of about 30.

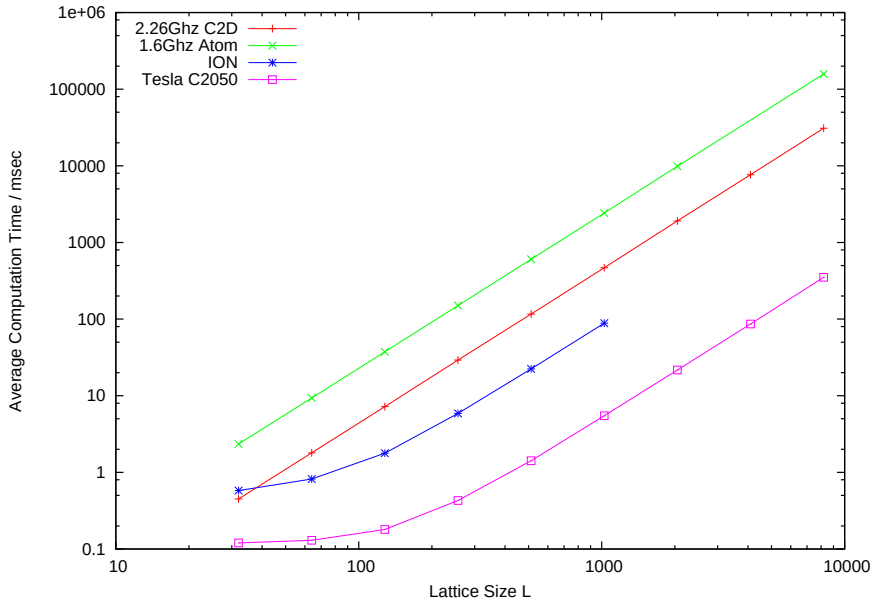


Fig. 5.1: *Time consumption of one monte carlo step on different hardware.*

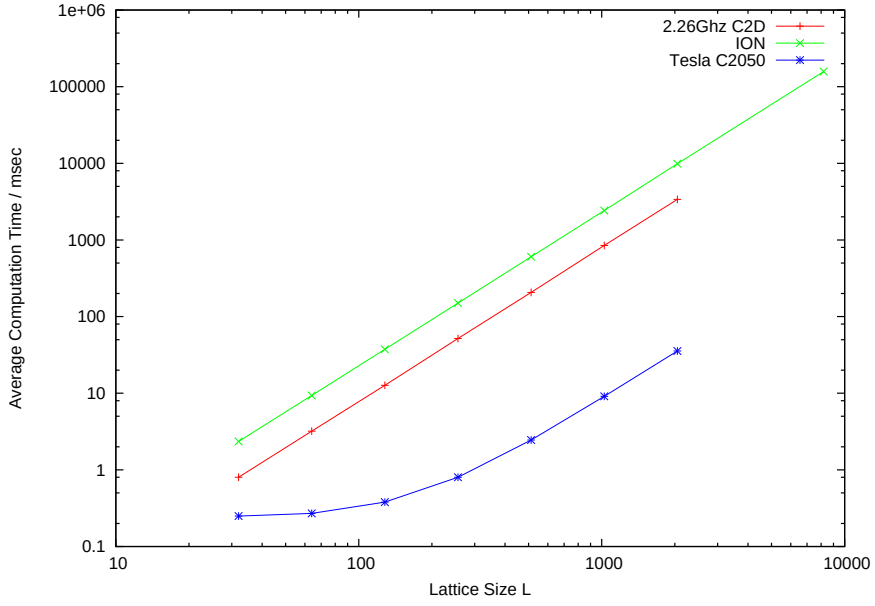


Fig. 5.2: *Spin update and measuring procedure.*

References

[1] J. M. Kosterlitz and D. J. Thouless, J. Phys. C: Solid State Phys. 6 (1973), 1181

[2] http://research.nvidia.com/content/fraunhofer-unibonn-crc-summary, as of 13.3.2011

[3] http://developer.download.nvidia.com/compute/cuda/3_2/docs/CUDA_3.2_Math_Libraries_Performance.pdf, as of 14.3.2011

[4] http://www.beyond3d.com/images/articles/cuda-intro/BGT.png, as of 14.3.2011

[5] http://www.gnu.org/software/gsl/manual/html_node/Random-Number-Generation.html, as of 14.3.2011

[6] http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CURAND_Library.pdf, as of 14.3.2011

[7] http://www.phy.duke.edu/∼rgb/General/dieharder.php, as of 14.3.2011

[8] S. G. Chung, Phys. Rev. B 60, (1999), 11761

[9] S. Teitel and C. Jayaprakash, Phase transtions in frustrated two-dimensional xy models, Phys. Rev. B 27, 598 - 601 (1983)

[10] S. E. Krüger *et al.* , Phys. Rev. B 73, 094404 (2006) ADS

[11] P. Olsson, Phys. Rev. B 52, 4526 - 4535 (1995)

[12] G. Palma *et al.* , Phys. Rev. E 66, 026108 (2002)

[13] F. Wende, diploma thesis, http://edoc.hu-berlin.de/docviews/abstract.php?id=37361 (15.3.2011)

[14] http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html#MersenneTwister, as of 15.3.2011