CMST 21(2) 69-98 (2015) DOI:10.12921/cmst.2015.21.02.003

Monte Carlo Simulations of the Ising Model on GPU

Jacek Wojtkiewicz[1*], **Krzysztof Kalinowski**[2]

[1] *Department for Mathematical Methods in Physics*
Faculty of Physics, University of Warsaw
ul. Pasteura 5, 02-093 Warszawa, Poland
[*] *E-mail: wjacek@fuw.edu.pl*

[2] *R.D.Labs, Sp. z o.o.*
ul. Hoża 43/49 m. 11, 00-681 Warszawa, Poland

Received: 18 June 2014; revised: 17 March 2015; accepted: 21 April 2015; published online: 15 June 2015

Abstract: Monte Carlo simulations of two- and three-dimensional Ising model on graphic cards (GPU) are described. The standard Metropolis algorithm has been employed. In the framework of the implementation developed by us, simulations were up to 100 times faster than their sequential CPU analogons. It is possible to perform simulations for systems containing up to 10^9 spins on Tesla C2050 GPU. As a physical application, higher cumulants for the 3d Ising model have been calculated.
Key words: Monte Carlo simulation, Ising model, CUDA, GPU programming

I. INTRODUCTION

Graphic cards (Graphic Processor Unit – GPU) are now a very powerful tool in computational physics. Calculations on them are performed by multitudes of relatively slow units, each able only to perform a handful of operations. However, those small workers were designed to work on large data sets. Current games require millions of vectors to be transformed and millions of pixels to be placed on the screen in a fraction of a second. That kind of operation seems close to what we really need in e.g. Monte Carlo simulations. Moreover, with frameworks like CUDA (Computing Unified Device Architecture) developed in nVidia, anyone with a good knowledge about C/C++ and spare time can obtain significant results as programming GPUs is now relatively easy [1, 2].

However, in order to effectively utilize its potential one must be aware of its architecture details and rethink the used algorithms. First of all, problems of parallel computation have to be taken into account when working on CUDA kernels. Second, the optimal use of limited memory must also be taken into account. Moreover, copying memory between host and device is very expensive in terms of time – memory transfers should be reduced to a minimum as they are often the slowest part of a CUDA program.

In our paper, we describe some programming aspects of the Monte Carlo simulations of two- and three-dimensional Ising model on GPU's. Similar efforts have been undertaken so far [3, 4, 5] and our work is a certain extension to the mentioned authors. We can improve their results in aspects of more optimal usage of memory (so simulations of larger systems are possible) and in speed of calculations. As an application of our approach, we calculate higher cumulants for the three-dimensional Ising model.

The outline of the paper is as follows. In Sec. II, we describe the Monte Carlo simulation with the use of GPU. Sec. III contains our results, and Sec. IV a summary as well as remarks on some open problems. In Sec. V we list the program for MC simulation of the three-dimensional Ising model on GPUs with the use of CUDA software.

II. DESCRIPTION OF THE SIMULATION

II. 1. Implementation of the simulation on GPU

In the paper, we consider the ferromagnetic Ising model. It is the spin model defined on the simple cubic lattice in d dimensions $\mathbb{Z}^d$ by the Hamiltonian: $H = J\sum_{\langle ij\rangle} s_i s_j$. Here, s_i denote the value of the spin on the $i-$th site; every spin s_i can take values ± 1. The sum is taken over the nearest neighbours (symbol $\langle\rangle$). A more detailed discussion of the motivations coming to this model follows in Sec. III.1.

We have used the standard Metropolis algorithm [6]. Its implementation on CPU is also standard, so we describe below only some important aspects of implementation on graphic cards.

The main difference between CPU and GPU is that we can do many operations simultaneously on GPU. We could assign one thread to each spin on the lattice and use them to perform large updates in a single iteration. However, this comes with a drawback: spins that were used for determining the flip cannot be flipped in the same cycle. If they were, the result of such operation would be undetermined, as it would not be possible to tell in what order those spins will be updated. To tackle this problem there is a method called *Checkerboard decomposition* [3], [4], [5]. The idea is as follows: The whole lattice of the nearest neighbours Ising model is split into two parts (we call them "black" and "white" – like the checkerboard). They are updated one after another. When we update the "white" part of the lattice, 'black' spins are not changed, but only used to compute flip probability. One drawback of this method is that only even sizes of the lattice can be used, but since we want to work with sizes of around 10^6 spins, that should not make a significant difference. An idea of this method is shown in Fig. 1.

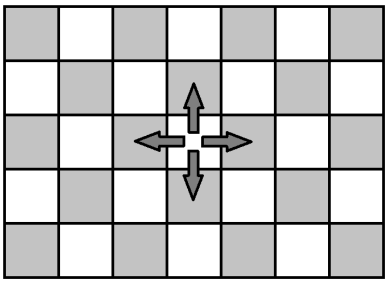

Fig. 1. Example of checkerboard decomposition for the nearest neighbours Ising model. Arrows indicate spins required for update on the given position. Black spins depend only on the white ones, and accordingly white spins depend only on the black ones.

The second problem we might encounter is assigning spins to threads on GPU. In the case of nVidia CUDA, we obtained threading system composed of three elements: threads, blocks and a grid, and we need to fit our solution into that system. The basic difference between them is explained in Fig. 2. We have to take into account several new problems, e.g. if the total number of threads that can be run in parallel is 1536, while the maximum block size is 1024 threads. It is best to create blocks of size 512 threads, as this allows scheduler to run three blocks at once in parallel. We also have to decide how we want to assign spin to our threads. There are two common ways of doing this: each block represents a rectangle or a line. In our simulations, we split our systems into rectangles for 2d, and into lines for 3d. However, it seems that properly used caching techniques, that is fetching large memory chunks for whole block into the shared memory, could yield better performance in case of rectangles, as most of the spins needed for an update would be available in an instant.

It is important to note that this particular decomposition works only for some special interactions, like nearest neighbours or chess knight. After understanding how this model works, it is easy to create similar decompositions for other interactions. Some interactions may split lattice into more than two independent parts, but still provide a solution.

In the nearest neighbours problem, only the closest spins are needed in calculation of energy. This is, however, not the only split that has to be done when working on GPU. Once we split the spins into those read-only and updated, we need to divide the entire lattice for multiprocessors to handle. There are multiple solutions to that problem. One is favouring long lines of spins, the other one favours rectangles. In our solution we used splitting into rectangles based on F. Wende presentation and explanation [5] for two dimensions, and into lines for three dimensions.

Another difference between CPU and GPU is the usage of memory. While it is rather easy to use one bit per spin on CPU, it is harder on GPU. Multiple threads would like to access the same byte of memory only to flip one bit. The solution we have chosen was to use one $int8\_t$ for each spin, which is an 8–bit, platform independent, signed integer for C/C++ and is defined in *stdint.h*. This is not an optimal, but the easiest one from the programming point of view. The optimal solution would make each GPU thread work not on a single spin, but on groups of spins. This way there would be no race condition to memory, and thus we would still be able to keep the advantage of using the least possible amount of space. However, usually the problem is not with memory taken by lattice. The largest lattice we considered was of size 500^3. This grid would be using only around 600 MB of device memory, which is less than available memory on a standard graphical card (1 GB).

Another problem is with using conditional variables on GPU. It is worth noting that some conditions can be replaced with mathematical operations or completely omitted. We have checked that in some cases replacing conditionals can yield better performance (12% gain in speed by replacing two and removing four conditions), yet are harder to understand and thus to debug. Each of these operation requires the knowledge of the structure of data used for storing numbers. An example

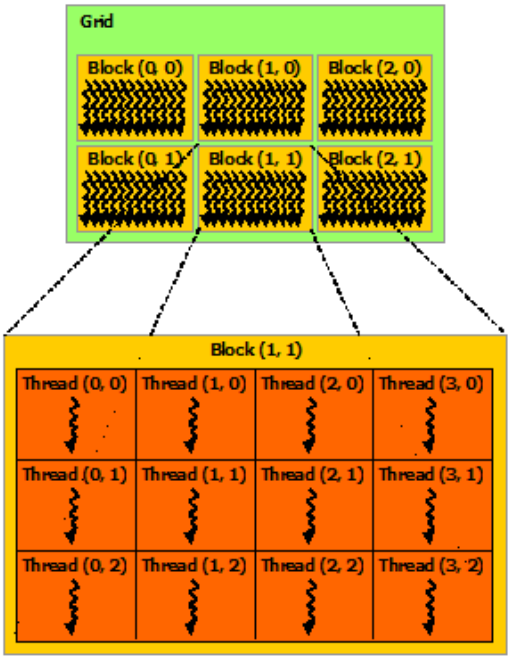

Fig. 2. Comparison of nVidia CUDA thread, block and grid. Grid is composed from blocks, which in turn are made from threads. Image taken from docs.nvidia.com

of replacing conditionals with mathematical calculations can be used to obtain periodic boundary conditions. In our simulations we are iterating over an entire lattice of spins and for each spin we have to obtain values of spins of its neighbours. However, inside computer memory there is no such thing as periodic table and accessing memory of table index -1 would result in a segmentation fault. To solve this problem we can either use everyday conditional like:

```
if (tableIndex < 0) tableIndex =
    tableSize - 1;
```

or use simple mathematical calculations. With the code below we map range [-N; 2N] into [0; N] with multiplications and additions. Here we use here knowledge about how signed numbers are stored in memory. In 32-bit signed integer the most significant bit is understood as -2^{31}, so it means that when it is set, the entire number must be below zero. So, if we can extract this value, it can be used to remap indexes. The solution is presented below:

```
tableIndex += 1;
tableIndex +=
  (((-tableIndex) >> 31) -
    ((tableIndex - tableSize) >> 31))
    * tableSize;
return tableIndex - 1;
```

It might seem as a lot of operation for a simple wrapping index, but it is easier to perform for GPU than branching that is used by CUDA when resolving a conditional.

Another example is much more sophisticated. Here we are looking for a solution to the following problem: when a floating point number is greater than 0, add some value to another; otherwise do nothing. It looks like a common problem with programming, so a natural solution would be to use conditionals. However, if we know how floating point numbers are encoded on our architecture, we can perform this with only mathematics. This solution assumes that numbers are encoded using the IEEE 754 convention in which the most significant bit of a number is the sign. So, if a number is greater than 0, it is set to 1, otherwise to 0. Going further, if we can extract it then a simple multiplication would be enough to solve our problem. The code below assumes that we are using 32-bit floating point numbers, but it can also be easily adapted to 64-bit floating point numbers. Again, we use the *stdint.h* library with its $int32\_t$ (32-bit signed integer, platform independent). As we cannot perform bitwise operations on floating point numbers, we are using a simple cast to integer, so that we can access single bits of our number. The entire solution is presented below:

```
int8_t bit = (*((int32_t*) & testValue)
  >> 31);
resultValue += bit
  * addOnlyIfTestValueGreaterThan0;
```

As presented, with some additional knowledge about low level number representations, we are able to replace / remove the conditional code from our multi-threaded CUDA computations.

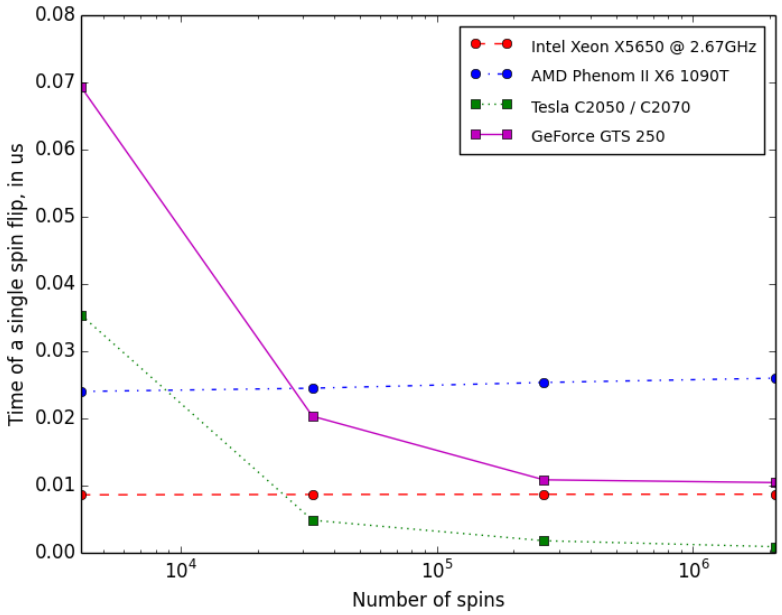

Fig. 3. Comparison between CPU and GPU. In the figure we see effective time of flipping one spin compared for different lattice sizes for different computational units. Results from a single core of a CPU are compared to a single GPU unit.

In our simulations we were using a single CUDA thread for each spin. We also tried to perform an entire Ising simulation in a single CUDA thread. In this test each thread was maintaining its own lattice, and all of them were completely independent. Not only this test has yielded poor performance, it would also be impractical for large scale calculations because of the amount of used memory.

The second aim was to create as big lattice as possible to count in finite time, and compare results across CPU and GPU with exact solutions for critical temperature and various cumulats.

II. 2. Monte Carlo simulations

For the generation of random numbers we used *rand()* from *stdlib.h* on CPU and *CURAND_RNG_PSEUDO_DEFAULT* generator from *curand.h* library provided by nVidia for GPU. It is worth noting that values returned from *curand.h* library are from 0 excluding to 1 including, so a single subtraction was needed before we could use the numbers.

Each of those simulations was started with a warm-up consisting of 2000 sweeps followed by 10000 steps, each consisting of 500 sweeps and data gathering.

A single sweep on GPU was trying to perform a flip on half of the spins from the whole system. It was possible due to using Checkerboard Decomposition connected with high level of parallelisation.

It is also important to note that below critical temperature simulation was started as cold (all spins in one direction), and above – as hot (all spins in randomly generated directions). Each of those simulations was repeated 10 times and counted averages and errors. A single simulation for system sizes 512×512 took around 20 hours on a single CPU core of Intel Xeon X5650 and around 15 minutes on a Tesla 2050 GPU unit (this value refers to the total run, i.e. computation on GPU and transfer of results to CPU).

Comparison of speed of several different computational units were performed. A sample of results can be found in Fig. 3. The hardware of the whole computer with graphic cards for which we present results were: GeForce GTS 250 cooperating with Intel QuadCore 9550 CPU, and Tesla 2050 cooperating with AMD Phenom II X6 1090 T CPU. However, in both cases only one core of CPU was involved in communication between GPU and CPU.

III. RESULTS

III. 1. Definition of the model

We consider the *ferromagnetic Ising model*. It is defined on the simple cubic lattice in d dimensions $\mathbb{Z}^d$ by the Hamiltonian:

$$H = J \sum_{\langle ij \rangle} s_i s_j \tag{1}$$

Here s_i denote the value of the spin on the $i-$th site; s_i can take values ± 1. The sum is taken over the nearest neighbours; it is denoted by the symbol $\langle\rangle$. We consider the ferromagnetic model, for which $J > 0$. Without the lack of generality,

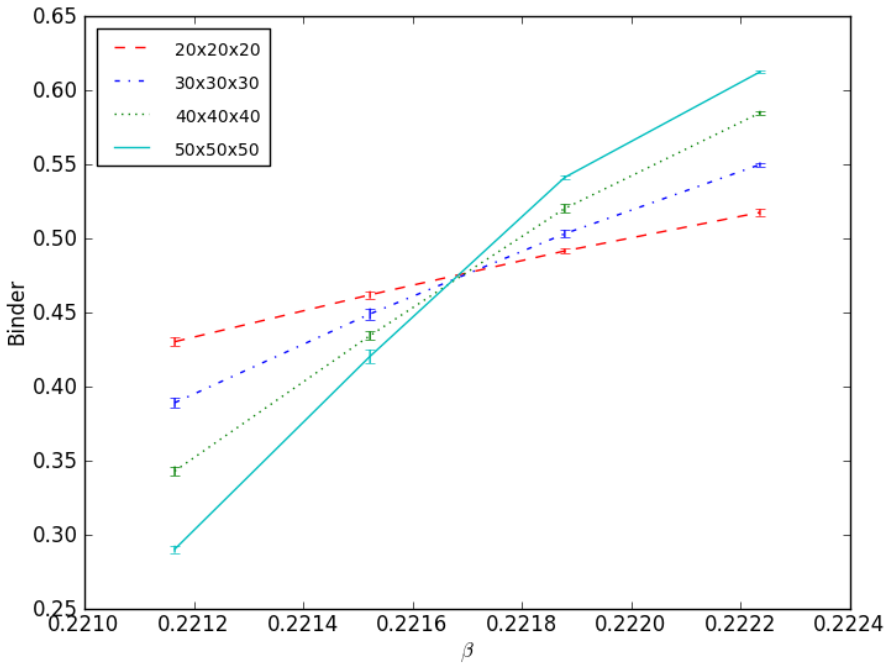

Fig. 4. Intersection of $V_4(L)$ for CPU. An analogous situation takes place in the case of GPU. There is no difference in the used scale; differences are on the fourth decimal places

we take the coupling constant $J = 1$; eventual change of J would correspond to rescaling of the (inverse) temperature.

The Ising model has been invented by Lenz in 1920 during his attempts to understand the phenomenon of ferromagnetism. It was solved in dimension one by Lenz' Ph.D. student E. Ising in 1925. It turned out that in one dimension it is *no* phase transition, i.e. spontaneous magnetization in positive temperature. The milestone in understanding the situation in higher dimensions was an exact solution of the 2d model (in zero magnetic field) by Onsager in 1944. It turned out that the 2d model exhibits the phase transition, i.e. spontaneous magnetization appears below certain positive critical temperature. However, there are two important problems not solved so far: the exact solution of the 2d model in the presence of magnetic field, and the exact solution of the 3d model. But it is possible to understand properties of the model by other means, both rigorous and numerical. At present time, the Ising model is employed in diverse areas of theoretical physics: the solid state physics as model of magnetism, lattice gas or binary fluid, in quantum field theory, in econophysics [6].

III. 2. Benchmarks: critical temperature and heat capacity

We have begun our simulations with reproduction of known results for the Ising model.

The first quantity was the *critical temperature*. In statistical mechanics it is defined as a value of temperature where physical quantities (magnetization, specific heat, susceptibility...) are *non-analytic* in the thermodynamic limit [7, 8].

We have obtained it in a standard manner by looking at the point where Binder cumulants for various lattice sizes are crossing [6]. Instead of temperature itself, we consider (more natural in statistical mechanics) the *inverse temperature* β, defined as

$$\beta = \frac{1}{k_B T},$$

k_B is the Boltzmann constant.

Our results are:

- Two-dimensional model:

$$\beta_c \approx 0.4408(20) \quad \text{CPU}, \qquad \beta_c \approx 0.4408(20) \quad \text{GPU} \tag{2}$$

The exact value $\beta_c = \ln(1+\sqrt{2})/2 \approx 0.440687...$

- Three-dimensional model:

$$\beta_c \approx 0.2217(2) \quad \text{CPU}, \qquad \beta_c \approx 0.2217(2) \quad \text{GPU} \tag{3}$$

The exact value is not known. Significant digits coming from a whole variety of methods are: $\beta_c \approx 0.22165$ (see [9-21]).

We illustrate the method for the 3d model in Fig. 4.

We have also reproduced plots of specific heat for various lattice sizes. Results of plots are consistent with numerous ones existing in literature.

III. 3. Cumulants – motivation

In the simulations described above (calculating critical temperature and specific heat) we also obtained values of *higher cumulants at the critical point*.

The analysis of cumulants is an interesting theoretical problem for many reasons. First, it touches the status of the *universality principle* that has been postulated in the 1960s in the aspect of *critical exponents* [6], and further extended to *amplitude ratios* and *scaling functions* [22].

In the first case, the universality principle is well examined. In general, there is a consensus that statistical-mechanical models can be divided into a small number of *universality classes*. Every universality class is completely characterized by such features as dimension of the system, symmetry of the order parameter and the range of interactions. For every model within the given universality class, values of the critical exponents are the same and independent of model details such as particular form of interactions [6, 8]. The situation is less clear for Universal Amplitude Ratios. In general, one assumes that within every universality class amplitude ratios are constant and independent of details of interactions: however, they can depend on boundary conditions [22]. This has been confirmed for a whole variety of models [9, 10, 22], but quite a few counter-examples have been found [23] and still there are controversies on this subject [24]. One can suspect that breaking universality for amplitude ratios is more transparent for higher cumulants. So we decided to calculate them for the 3d Ising model as the first step towards examination of (non?)universality.

Moreover, the cumulants also measure deviation of magnetization fluctuations at criticality from a Gaussian distribution, and to describe precisely such non-Gaussianness, one should know their values.

One more motivation comes from the quantum field theory. Known is certain conjecture about *non-positivity of untruncated six-point vertex function* in certain quantum field theories [7]. They are expressible by sixth order cumulants. So we hope that our results can be useful in testing this conjecture.

Finally, let us look at the two-dimensional Ising model for which the values of cumulants in periodic boundary conditions at critical temperature are predicted by the conformal field theory. These predictions have been confirmed with high precision by Monte Carlo simulations [25]. But for the 3d Ising model, to our best knowledge, there are almost no such predictions (the only exception we have found is a recent paper [26]; however, the cumulants are defined there in a different manner than we do, and it is hardly possible to make comparisons). So we decided to calculate higher cumulants by Monte Carlo simulations. Here we have no such predictions as in two dimensions, where values of all cumulants are exactly (although not rigorously) known. We consider development of such predictions as a challenge for theory.

III. 4. Cumulants – results

We take the following definitions of cumulants:

$$V_{2n} = \frac{\langle M^{2n} \rangle}{\langle M^2 \rangle^n}. \tag{4}$$

We consider the following values of n: $n = 2, 3, 4, 5$.

In the literature, the *Binder cumulant*:

$$V_B = 1 - \frac{1}{3} V_4 \tag{5}$$

is often used instead of V_4.

III. 4. 1. Two-dimensional Ising model

Here, the benchmarks are results due to Salas and Sokal [25] based on the conformal-field theory, high-precision Monte Carlo simulations using Swendsen-Wang algorithm and theory of finite-size scaling.

During our simulations, we encountered the problem of *large fluctuations* in critical temperature. For smaller systems (squares 64^2 and 128^2) fluctuations of cumulants were relatively small, while for squares 256^2 and 512^2 they were large. We illustrate this situation in Table 1 (we reproduce the result for the cumulant V_8 only, but situation is similar for other cumulants, too).

Tab. 1. Uncertainties in determination of V_8 for 2d Ising model

size (L^2)	CPU	GPU
64^2	1.888 (18)	1.879 (11)
128^2	1.894(28)	1.883(27)
256^2	1.926(63)	1.933(46)
512^2	1.922(98)	1.923(89)

For this reason, our determination was not precise. We have obtained it by extrapolation of finite-size data, described in more details in the next Subsubsection. Within the error bar, our results are consistent with those given in [25] (see Table 2).

Tab. 2. Values of cumulants for 2d Ising model (extrapolated)

Cumulant	Salas&Sokal [25]	CPU	GPU
V_4	1.167923(2)	1.169(4)	1.170(5)
V_6	1.455649 (7)	1.47(4)	1.47 (4)
V_8	1.8925 (2)	1.91 (4)	1.92 (5)
V_{10}	2.5396 (3)	2.58 (5)	2.59 (6)

III. 4. 2. Three-dimensional Ising model

In simulation of the 3d model we observed much less fluctuations than in two dimensions. The precision of determination of cumulants was roughly independent of the lattice size. This opportunity allowed us to apply the *extrapolation procedure* to obtain values of cumulants in the limit of lattice size tending to infinity.

Tab. 3. Values of cumulants for 3d Ising model: CPU results

size (L^3)	V_4	V_6	V_8	V_{10}
20^3	1.580 (5)	2.995 (24)	6.38 (9)	14.8 (3)
30^3	1.594 (8)	3.064 (35)	6.63 (13)	15.7 (4)
40^3	1.589 (8)	3.035 (35)	6.52 (12)	15.3 (4)
50^3	1.596 (7)	3.064 (30)	6.62 (10)	15.6 (4)
∞ (extrapolation)	1.608 (7)	3.118 (31)	6.82(11)	16.2(4)

In order to obtain the value of given quantity in the limit of infinite lattice, one must perform some kind of extrapolation. It is done with the aid of the *finite-size scaling* (FSS) theory [27]. One assumes that in the neighborhood of critical point, physical quantities have certain forms of expansion in small parameters: $T - T_c$, $h - h_c$, $\frac{1}{L}$ (T – temperature, h – magnetic field, L – length of the system). A precise form of such FSS expansion, as a rule, is not known rigorously, but there are hints coming from the Renormalization Group which allow to write down reliable forms of such expressions.

For cumulants at critical temperature there are some indications that the FSS expansion takes the form [9, 10, 12]

$$V_{2n}(L) = V_{2n}(\infty) + AL^{-\Delta} + A_1 L^{-\Delta_1} + \dots \qquad (6)$$

However, there exist quite a few different expressions with different exponents Δ_i and amplitudes A_i, coming from various scenarios of the Renormalization Group action near the critical point [25]. There are also various methods of fitting experimental data to obtain values of V_{2n}, A_i, Δ_i. For this reason we decided to take the simplest expression

$$V_{2n}(L) = V_{2n}(\infty) + AL^{-\Delta} \qquad (7)$$

where we took $\Delta = -0.83$ [9, 12]. We fitted our data to expression 7 in order to obtain values of $V_{2n}(\infty)$ and A. We present our results in Tables 3 and 4.

Here we have (almost) no reference to comparison. The only exception are results for the Binder cumulant V_4: 1.604 [9], [10] (after conversion to the definition we used).

IV. SUMMARY, CONCLUSIONS, PERSPECTIVE FOR FUTURE WORK

Our results show that GPU is as precise as CPU and much faster than CPU for the Ising model when system sizes are sufficiently large, namely 10^5 spins and bigger.

It takes time and effort to learn all important parts of CUDA programming and be able to use it well. Graphical cards are not the remedy to all of our computational problems. If not used with proper planning, those methods would not lead to performance increase. It should be understood as a special tool for parallel computation, not as a magical device that will speed up our program. Before trying to use it, it is important to consider dividing the procedure into many small, independent functions. Only if that can be done, and the current speed using CPU is not satisfying, we should start using CUDA for the problem. The results can be astonishing as it is a powerful technology.

In the presented form the Ising model is not the most interesting option, but it can still lead to some interesting models. The following models are only examples of a whole family of remarkable and interesting ones: the Falicov-Kimball model [28], [29] in perturbative regime and ANNNI (Axial Next Nearest Neighbour Ising) [30]. As it has been shown, results can be obtained much faster for bigger lattices, so it would be advisable to use for analysis of currently skipped system sizes because of the time of execution of simulations. In particular ANNNI models seems to need the power of graphical cards. We encounter here very large periodic structures (elementary cells up to few hundreds of sites). However, the higher the period is, the bigger size the lattice must be, because the size of the elementary cell grows. With GPUs we could simulate systems of size 10^6 spins in matter of minutes, and 10^8 spins in matter of days.

Tab. 4. Values of cumulants for 3d Ising model: GPU results

size (L^3)	V_4	V_6	V_8	V_{10}
20^3	1.582 (7)	3.000 (33)	6.38 (12)	14.8 (4)
30^3	1.591 (11)	3.045 (47)	6.55 (17)	15.4 (6)
40^3	1.592 (7)	3.051 (28)	6.58 (10)	15.4 (3)
50^3	1.594 (10)	3.061 (46)	6.61 (17)	15.6 (6)
∞ (extrapolation)	1.605 (9)	3.119 (36)	6.84 (14)	16.4(6)

The big challenge is development of *clustering algorithms* on graphic cards. As it is well known, the Metropolis algorithm is not the best tool to simulate the Ising model for temperatures close to the critical one due to critical slowing down. Much better results can be obtained by using clustering algorithms like Wolff or Swendsen-Wang. A single step, near the critical point, is equivalent to hundreds of Metropolis steps.

We were thinking about such problem. However, we have found out that it is much harder than we expected. It remains a big challenge for graphic card programmers. If successful, development of such algorithms would give a deep insight into the nature of critical point for a big family of lattice models.

Acknowledgments

We would like to thank Prof. A. Drzewiński (Zielona Góra University) and dr G. Banach (INTiBS PAN, Wrocław) for giving us access to Tesla machine, and for Prof. G. Musiał (UAM Poznań) for giving us access to a multi-core server. We are also indebted to dr G. Banach for his valuable remarks and suggestions.

References

[1] nVidia, NVIDIA CUDA C Programming Guide by nVidia Corporation http://developer.download.nvidia.com/compute/-DevZone/docs/html/C/doc/CUDA C Programming Guide.pdf
[2] nVidia, CUDA C Best Practices Guide by nVidia Corporation http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA C Best Practices Guide.pdf
[3] T. Preis, P. Virnau, W. Paul, J. J. Schneider: *J. Comp. Phys.* 228 (2009)
[4] M. Weigel, *Comp. Phys. Comm.* **182**, 1833 (2011)
[5] F. Wende Implementation of the 2D Ising Model using CUDA http://www.physik.uni- leipzig.de/bordag/GCC/Talks/Wende.pdf
[6] J. J. Binney, N. J. Dowrick, A. J. Fisher, M. E. J. Newman: *The Theory of Critical Phenomena. An Introduction to the Renormalization Group*. Clarendon Press, Oxford 1992.
[7] J. Glimm, A. Jaffe: Quantum Physics – A Functional Integral Point Of View. Springer-Verlag, New York - Heidelberg - Berlin 1981.
[8] J. R. Baxter, Exactly Solved Models in Statistical Mechanics, Academic Press, 1989
[9] E. Luijten, Ph. D. Thesis, Delft University 1997.
[10] H. W. J. Blöte, E. Luijten and J. R. Heringa, *J. Phys.* **A28**, 6289 (1995).
[11] P. Butera and M. Comi, *Phys. Rev.* **B 56**, 8212 (1997).
[12] A. L. Talapov and H. W. J. Blöte, *J. Phys.* **A 29**, 5727 (1996).
[13] R. Gupta and P. Tamayo, *Int. J. Mod. Phys.* **C 7**, 305 (1996).
[14] D. P. Landau, *Physica* **A 205**, 41 (1994).
[15] H. W. J. Blöte and G. Kamieniarz, *Physica* **A 196**, 455 (1993).
[16] C. F. Baillie, R. Gupta, K. A. Hawick and G. S. Pawley, *Phys. Rev.* **B 45**, 10438 (1992).
[17] F. Livet, *Europhys. Lett.* **16**, 139 (1991).
[18] A. M. Ferrenberg and D. P. Landau, *Phys. Rev.* **B 44**, 5081 (1991).
[19] N. Ito and M. Suzuki, *J. Phys. Soc. Jpn.* **60**, 1978 (1991).
[20] H. W. J. Blöte, J. A. de Bruin, A. Compagner, J. H. Croockewit, Y. T. J. C. Fonk, J. R. Heringa, A. Hoogland and A. L. van Willigen, *Europhys. Lett.* **10,** 105 (1989).
[21] A. Rosengren, *J. Phys.* **A 19**, 1709 (1986).
[22] V. Privman, P. C. Hohenberg and A.Aharony: *Universal Critical-Point Amplitude Relations*. In: *Phase Transitions and Critical Phenomena* vol. 14, C. Domb and J. Lebowitz, Eds., Academic Press 1991.
[23] W. Selke and L. S. Shchur, *J. Phys.* **A 38**, L739 (2005).
[24] W. Selke, *Europhys. J.* **B 51**, 223 (2006).
[25] J. Salas, A. D. Sokal, *J. Stat. Phys.* **98**, 551 (2000).
[26] Chen Li Zhu, Pan Xue, Chen Xiao Song, Wu Yuan Fang: *Chinese Physics* **C36**, 727 (2012)
[27] M. N. Barber: *Finite Size Scaling*. In: *Phase Transitions and Critical Phenomena* vol. 8, C. Domb and J. Lebowitz, Eds., Academic Press 1983.
[28] L. M. Falicov, J. C. Kimball: *Phys. Rev. Lett.* **22**, 997 (1969).
[29] Ch. Gruber and N. Macris: *Helv. Phys. Acta* **69**, 850 (1996).
[30] W. Selke: *Phys. Reports* **170**, 213 (1988).

Jacek Wojtkiewicz (dr hab.) is an adiunct in the Department for Mathematical Methods in Physics, Faculty of Physics, Warsaw University. He is an author of more than 30 papers, devoted to cosmochemistry, general relativity, theory of phase transitions, statistical mechanics, numerical methods in statistical physics (Monte Carlo, exact diagonalization, DMRG). His present scientific interests cover statistical mechanics, theory of phase transitions (both rigorously and numerically), strongly correlated electron systems (Hubbard model, Falicov-Kimball model), theoretical bases of photovoltaics. He is also active on the area of popularization of physics. Some of his actions are: presentations, lectures, popular-scientific articles, book on the history of physics (co-author: Andrzej Drzewiński). His hobby is mountaineering and music.

Krzysztof Kalinowski is an absolvent of the Faculty of Physics, Warsaw University, with specialization in 'Mathematical and computer modeling of physical processes'. Actually he is working as Research and Development specialist in the consortium R.D. Labs. He has a profound knowledge in programming in many languages as C, C++, Java, CUDA, as well as numerous programming skills: network programming, numerical and graph algorithms, multiple software design patterns. His hobby are travelling and computer games, both as player and as programmer.

 DOI:10.12921/cmst.2015.21.02.003

V. APPENDIX – CODE FOR ISING 3D CUDA

Listing 1. config.h

```
1  #ifndef CONFIG_H
2  #define CONFIG_H
3
4  #include <stdint.h>
5
6  typedef float FLOAT;
7
8  extern double beta;
9
10 extern double J;
11
12 extern double B;
13
14 extern uint32_t X_N;
15 extern uint32_t Y_N;
16 extern uint32_t Z_N;
17
18 extern uint64_t MC_N;
19 extern uint64_t WARMUP;
20 extern uint64_t SKIP;
21
22 /** \brief Parse command line arguments
23  *
24  *  This function un-parses all needed arguments from command line, and, if needed
25  *  fills the~default values.
26  *  If no parameters are passed, this function just prints list of available
27  *  options and returns -1.
28  *
29  *  Note: at the~very least, 2 parameters must be passed to program - vertical
30  *        and horizontal size of the~lattice.
31  *
32  *  \param[in] argc - amount of command line arguments, passed from main
33  *  \param[in] argv - char table filled with arguments, passed from main
34  *
35  *  \returns
36  *  0 on success
37  *  != 0 on failure (unable to parse some particular
38  *                   parameter, proper output is printed on stderr)
39  */
40 int parse_args(int argc, char **argv);
41
42 /** \brief Print current status
43  *
44  *  This function prints all current values of configuration parameters
45  *  that will be used in upcoming simulation.
46  */
47 void hello_message();
48
49 #endif
```

Listing 2. config.cpp

```
1  #include "config.h"
2
3  #include <stdio.h>
4  #include <stdlib.h>
```

```
5  #include <string.h>
6
7  double beta = 1.0;
8  double J = 1.0;
9  double B = 0.0;
10 uint32_t X_N = 0;
11 uint32_t Y_N = 0;
12 uint32_t Z_N = 0;
13 uint64_t MC_N = 0;
14 uint64_t WARMUP = 0;
15 uint64_t SKIP = 8;
16
17 void print_usage(char *my_name)
18 {
19   fprintf(stderr, "Usage: %s [options] X_N Y_N Z_N\n"
20                   "\tX_N - spin amount, X axis. X_N >= 3\n"
21                   "\tY_N - spin amount, Y axis. Y_N >= 3\n"
22                   "\tZ_N - spin amount, Z axis. Z_N >= 3\n"
23                   "\t-J - value of interaction coupling, default - 1.0\n"
24                   "\t-beta - value of inverse temperature, default - 1.0\n"
25                   "\t-B - value of magnetic field, default - 0.0\n"
26                   "\t-N - sweep count for Monte Carlo, default - 10000\n"
27                   "\t-W - sweep count for warm-up, default - 100\n"
28                   "\t-S - steps between data gathering, default - 8\n",
29                   my_name);
30 }
31
32 int read_argv_double(double *out, int current, int argc, char **argv)
33 {
34   if (current >= argc) return 1;
35   *out = atof(argv[current]);
36   return 0;
37 }
38
39 int read_argv_uint64_t(uint64_t *out, int current, int argc, char **argv)
40 {
41   if (current >= argc) return 1;
42   int i = atoi(argv[current]);
43   if (i < 0) return 2;
44   *out = (uint64_t) i;
45   return 0;
46 }
47
48 int parse_args(int argc, char **argv)
49 {
50   if (argc == 1) {
51     print_usage(argv[0]);
52     return -1;
53   }
54
55   /*
56       This all is done to obtain windows and linux compatibility.
57       If it was to be used with linux only, getopt library should
58       be used.
59    */
60
61   int current_arg;
62   for (current_arg = 1; current_arg < argc; ++current_arg) {
63     if (argv[current_arg][0] == '-') {
```

```
64
65        if (strcmp(argv[current_arg], "-J") == 0) {
66          if (read_argv_double(& J, current_arg + 1, argc, argv)) {
67
68            fprintf(stderr, "Failed to parse value for -J\n");
69            return -2;
70          }
71          current_arg++;
72        } else if (strcmp(argv[current_arg], "-beta") == 0) {
73          if (read_argv_double(& beta, current_arg + 1, argc, argv)) {
74
75            fprintf(stderr, "Failed to parse value for -beta\n");
76            return -3;
77          }
78          current_arg++;
79        } else if (strcmp(argv[current_arg], "-B") == 0) {
80          if (read_argv_double(& B, current_arg + 1, argc, argv)) {
81
82            fprintf(stderr, "Failed to parse value for -B\n");
83            return -4;
84          }
85          current_arg++;
86        } else if (strcmp(argv[current_arg], "-N") == 0) {
87          if (read_argv_uint64_t(& MC_N, current_arg + 1, argc, argv)) {
88
89            fprintf(stderr, "Failed to parse value for -N\n");
90            return -5;
91          }
92          current_arg++;
93        } else if (strcmp(argv[current_arg], "-W") == 0) {
94          if (read_argv_uint64_t(& WARMUP, current_arg + 1, argc, argv)) {
95
96            fprintf(stderr, "Failed to parse value for -N\n");
97            return -5;
98          }
99          current_arg++;
100       } else if (strcmp(argv[current_arg], "-S") == 0) {
101         if (read_argv_uint64_t(& SKIP, current_arg + 1, argc, argv)) {
102
103           fprintf(stderr, "Failed to parse value for -S\n");
104           return -5;
105         }
106         current_arg++;
107       } else {
108         fprintf(stderr, "Unknown command '%s'\n", argv[current_arg]);
109       }
110     } else {
111
112       if (X_N == 0) {
113         X_N = (uint32_t) atoi(argv[current_arg]);
114       } else if (Y_N == 0) {
115         Y_N = (uint32_t) atoi(argv[current_arg]);
116       } else if (Z_N == 0) {
117         Z_N = (uint32_t) atoi(argv[current_arg]);
118       }
119     }
120   }
121
122   if (X_N < 3) {
```

```
123       fprintf(stderr, "Parameter X_N less than 3\n");
124       return -5;
125     }
126
127     if (Y_N < 3) {
128       fprintf(stderr, "Parameter Y_N less than 3\n");
129       return -5;
130     }
131
132     if (Z_N < 3) {
133       fprintf(stderr, "Parameter Z_N less than 3\n");
134       return -5;
135     }
136
137     if (MC_N == 0) {
138       MC_N = 10000;
139     }
140
141     if (WARMUP == 0) {
142       WARMUP = 100;
143     }
144
145     return 0;
146   }
147
148   void hello_message()
149   {
150     fprintf(stderr, "Welcome!\n"
151             "Current parameters:\n"
152             "* X_N = %u\n"
153             "* Y_N = %u\n"
154             "* Z_N = %u\n"
155             "* J = %f\n"
156             "* beta = %f\n"
157             "* B = %f\n"
158             "* MC_N = %lu\n"
159             "* WARMUP = %lu\n"
160             "* SKIP = %lu\n"
161             "Performing calculations. Please wait...\n",
162             X_N, Y_N, Z_N,
163             J, beta, B,
164             (long unsigned int) MC_N,
165             (long unsigned int) WARMUP,
166             (long unsigned int) SKIP);
167   }
```

Listing 3. SpinTable.cuh

```
1  #ifndef SPIN_TABLE_CUDA_H
2  #define SPIN_TABLE_CUDA_H
3
4  #include <cstdlib>
5
6  /** "Interface" for table of spins
7   *
8   *  This header presents a set of implementation-independent
9   *  functions, that can be used to manipulate three-dimensional
10  *  spin lattices. Thanks to this, we can easily replace
11  *  internal representation in a way transparent for the
12  *  rest of code.
```

```
13  */
14 struct SpinTable
15 {
16   unsigned int X_N, Y_N, Z_N;
17   char *table;
18
19   /** \brief Initializes spin table
20    *
21    *  This function uses values from config and allocates
22    *  memory for a new spin table.
23    *  Allocated table CANNOT be used in calculations,
24    *  unless it is further prepared with either
25    *  randomize_init or clear_init.
26    *
27    *  \param[in] X - horizontal size of the lattice
28    *  \param[in] Y - vertical size of the lattice
29    *  \param[in] Z - depth of the lattice
30    */
31   void host_init(int X, int Y, int Z)
32   {
33     X_N = X;
34     Y_N = Y;
35     Z_N = Z;
36
37     cudaMalloc((void **) & table, sizeof(char) * X_N * Y_N * Z_N);
38     clear_table();
39   }
40
41   /** \brief Destroys spin table
42    *
43    *  This function frees all resources used by given
44    *  spin table. After this call, table becomes
45    *  unusable, unless it is initialized again.
46    */
47   void host_destroy()
48   {
49     cudaFree(table);
50   }
51
52   /** \brief Retrieves value of a single spin from spin table
53    *
54    *  DEVICE FUNCTION
55    *
56    *  Returns value of a spin under given position, assuming
57    *  periodic boundary conditions (hyper-torus).
58    *
59    *  \param[in] x - horizontal index of desired spin
60    *  \param[in] y - vertical index of desired spin
61    *  \param[in] z - depth of desired spin
62    *
63    *  \returns
64    *  -1 or 1, depending on pointed spin
65    */
66   __device__ int get(int x, int y, int z)
67   {
68     if (x < 0) x += X_N;
69     else if (x >= X_N) x -= X_N;
70     if (y < 0) y += Y_N;
71     else if (y >= Y_N) y -= Y_N;
```

```
72      if (z < 0) z += Z_N;
73      else if (z >= Z_N) z -= Z_N;
74
75      return table[x + y * X_N + z * X_N * Y_N];
76    }
77
78    /** \brief Change value of a single spin from spin table
79     *
80     *  DEVICE FUNCTION
81     *
82     *  "Flips" single spin under given position, assuming
83     *  periodic boundary conditions (hyper-torus). By "flip" we
84     *  understand operation that changes spin table, so
85     *  the~following calls would yield given results:
86     *  get  1 -> flip -> get -1
87     *  get -1 -> flip -> get  1
88     *
89     *  \param[in] x - horizontal index of desired spin
90     *  \param[in] y - vertical index of desired spin
91     *  \param[in] z - depth of desired spin
92     */
93    __device__ void flip(int x, int y, int z)
94    {
95      if (x < 0) x += X_N;
96      else if (x >= X_N) x -= X_N;
97      if (y < 0) y += Y_N;
98      else if (y >= Y_N) y -= Y_N;
99      if (z < 0) z += Z_N;
100     else if (z >= Z_N) z -= Z_N;
101
102     table[x + y * X_N + z * X_N * Y_N] *= -1;
103   }
104
105   /** \brief Change value of a single spin from spin table
106    *
107    *  DEVICE FUNCTION
108    *
109    *  \param[in] x - horizontal index of desired spin
110    *  \param[in] y - vertical index of desired spin
111    *  \param[in] z - depth of desired spin
112    *  \param[in] bit - new value for spin, should be either 1 or -1
113    */
114   __device__ void set(int x, int y, int z, int bit)
115   {
116     if (x < 0) x += X_N;
117     else if (x >= X_N) x -= X_N;
118     if (y < 0) y += Y_N;
119     else if (y >= Y_N) y -= Y_N;
120     if (z < 0) z += Z_N;
121     else if (z >= Z_N) z -= Z_N;
122
123     table[x + y * X_N + z * X_N * Y_N] = bit;
124   }
125
126   /** \brief Fills entire table with spins randomly
127    *
128    *  Each spin on the~table is assigned value either 1 or -1,
129    *  with equal probability.
130    *  After this operation, spin table is ready to use.
```

```
131     *
132     *  \param[in] seed - unused
133     */
134    void generate_random(unsigned seed)
135    {
136      char *tmp_table = (char *) malloc(sizeof(char) * X_N * Y_N * Z_N);
137
138      for (unsigned int i = 0; i < (X_N * Y_N * Z_N); ++i) {
139        tmp_table[i] = 2 * (rand() % 2) - 1;
140      }
141
142      cudaMemcpy(table, tmp_table, sizeof(char) * X_N * Y_N * Z_N, cudaMemcpyHostToDevice);
143
144      free(tmp_table);
145    }
146
147    /** DEPRECATED, use clear_table instead
148     */
149    void generate_clear()
150    {
151      char *tmp_table = (char *) malloc(sizeof(char) * X_N * Y_N * Z_N);
152
153      for (unsigned int i = 0; i < (X_N * Y_N * Z_N); ++i) {
154        tmp_table[i] = 1;
155      }
156
157      cudaMemcpy(table, tmp_table, sizeof(char) * X_N * Y_N * Z_N, cudaMemcpyHostToDevice);
158
159      free(tmp_table);
160    }
161
162    /** \brief Fills entire table with spins of value 1
163     *
164     *  Given table is set, so that get on any spin will return 1.
165     *  After this operation, spin table is ready to use.
166     */
167    void clear_table()
168    {
169      cudaMemset(table, 1, sizeof(char) * X_N * Y_N * Z_N);
170    }
171  };
172
173  #endif
```

Listing 4. termodyna_struct.h

```
1  #ifndef TERMODYNA_STRUCT_H
2  #define TERMODYNA_STRUCT_H
3
4  /**
5   *  structure used to gather all physical quantities
6   *  from simulation.
7   */
8  typedef struct termodyna_struct_s
9  {
10   double U;
11   double U_S;
12
13   double M;
14   double M_S;
```

```
15    double M_S2;
16    double M_6;
17    double M_8;
18    double M_10;
19
20    double V4, V6, V8, V10;
21  } termodyna_struct_t;
22
23  /** \brief Prints thermodynamical data from simulation
24   *
25   *  This method calculates thermodynamic quantities and prints
26   *  them on the~screen. Calculated values are:
27   *  - specific heat
28   *  - magnetic susceptibility
29   *  - Binder cumulant
30   *  - higher order cumulants (V4, V6, V8, V10)
31   *  Each value is presented "per spin" (value for entire lattice
32   *  is divided by number of spins in the~system).
33   *
34   *  \param[in] data - structure filled with information to be printed
35   */
36  void print_structure_data(termodyna_struct_t *data);
37
38  #endif
```

Listing 5. termodyna_struct.cpp

```
1   #include "termodyna_struct.h"
2
3   #include "config.h"
4
5   #include <stdio.h>
6   #include <math.h>
7
8   void print_structure_data(termodyna_struct_t *data)
9   {
10    fprintf(stderr, "\nValues per spin\n");
11    double num_spin = X_N * Y_N * Z_N;
12
13    fprintf(stderr, "M = %f\n", data->M / num_spin);
14    fprintf(stderr, "U = %f\n", data->U / num_spin);
15    fprintf(stderr, "X = %f\n", beta * (data->M_S - data->M * data->M) / num_spin);
16    fprintf(stderr, "c = %f\n", beta * beta * (data->U_S - data->U * data->U) / num_spin);
17    fprintf(stderr, "binder = %f\n", (1.0 - data->M_S2 / (3.0 * data->M_S * data->M_S)));
18    fprintf(stderr, "V4 = %f\n", data->V4);
19    fprintf(stderr, "V6 = %f\n", data->V6);
20    fprintf(stderr, "V8 = %f\n", data->V8);
21    fprintf(stderr, "V10 = %f\n", data->V10);
22  }
```

Listing 6. time_update.h

```
1   #ifndef TIME_UPDATE_H
2   #define TIME_UPDATE_H
3
4   #include <stdio.h>
5   #include <time.h>
6   #include <sys/time.h>
7   #include <stdint.h>
8
```

```
 9 /** "Interface" for time measurement
10  *
11  *  This header presents a set of implementation-independent
12  *  functions, that can be used to measure time of execution
13  *  of particular part of code.
14  *
15  *  Those timers are not only used to measure time, but also
16  *  to determine maximal, minimal and average time spent
17  *  on given operation.
18  */
19
20 /**< Timer data **/
21 typedef struct time_data_s
22 {
23   struct timeval min, max, avg, current;
24   uint32_t count;
25 } TimeData;
26
27 /** \brief Initializes time data
28  *
29  *  After this operation, timer is ready to use.
30  *
31  *  \param[in/out] td - TimeData to be initialized
32  */
33 void init_timedata(TimeData *td);
34
35 /** \brief Start measurement
36  *
37  *  Starts given timer. Calling this function
38  *  while timer is running will "restart" it,
39  *  that is drop the~measurement and start it again.
40  *
41  *  \param[in/out] td - timer to be started
42  */
43 void start_measure(TimeData *td);
44
45 /** \brief Stops measurement
46  *
47  *  Stops given timer and calculates time spent.
48  *
49  *  \param[in/out] td - timer to be stopped
50  */
51 void end_measure(TimeData *td);
52
53 /** \brief Prints measurements
54  *
55  *  Prints minimal, maximal and average measurement
56  *  performed by given timer.
57  *  Note: stdout and stderr are valid FILE objects,
58  *        and thus can be used to print output to the~screen.
59  *
60  *  \param[in] td - timer to be printed
61  *  \param[out] f - FILE object to write results to
62  */
63 void print_results(TimeData *td, FILE *f);
64
65 #endif
```

Listing 7. time_update.cpp

```
#include "time_update.h"

#include <string.h>

void init_timedata(TimeData *td)
{
  memset(td, 0, sizeof(TimeData));

  td->min.tv_sec = (uint32_t) -1;
  td->min.tv_usec = (uint32_t) -1;
}

void start_measure(TimeData *td)
{
  gettimeofday(&(td->current), NULL);
}

void end_measure(TimeData *td)
{
  struct timeval end, diff;
  gettimeofday(&end, NULL);
  timersub(&end, &(td->current), &diff);

  if (timercmp(&diff, &(td->min), <)) {
    td->min = diff;
  }
  if (timercmp(&diff, &(td->max), >)) {
    td->max = diff;
  }

  timeradd(&(td->avg), &diff, &end);
  td->avg = end;
  td->count += 1;
}

void print_results(TimeData *td, FILE *f)
{
  struct timeval tmp;
  tmp.tv_sec = td->avg.tv_sec / td->count;
  tmp.tv_usec = ((td->avg.tv_sec % td->count) * 1000000 + td->avg.tv_usec) / td->count;

  fprintf(f, "min: %us %uus\n", (uint32_t) td->min.tv_sec, (uint32_t) td->min.tv_usec);
  fprintf(f, "avg: %us %uus\n", (uint32_t) tmp.tv_sec, (uint32_t) tmp.tv_usec);
  fprintf(f, "max: %us %uus\n", (uint32_t) td->max.tv_sec, (uint32_t) td->max.tv_usec);
}
```

Listing 8. main.cu

```
/*****************************************************
* Compilation:
* make
*
* Print usage:
* ./Ising3d
*
* Example of invocation:
* ./Ising3d 100 100 100 -N 10000 -W 2000 -S 16 -beta 0.2201
*
* this line will start Ising simulation on single core
* with the following parameters:
```

```
13  * size:                    100x100x100
14  * inverse temperature:        0.2201
15  * magnetic field:                  0
16  * interaction coupling value:      1
17  * warm up sweeps:               2000
18  * Monte Carlo sweeps:          10000
19  * between measurements sweeps:    16
20  ****************************************************/
21
22  #include <stdio.h>
23  #include <stdlib.h>
24  #include <string.h>
25  #include <math.h>
26  #include <time.h>
27  #include <sys/time.h>
28  #include <unistd.h>
29
30  #include "config.h"
31  #include "termodyna_struct.h"
32
33  #include <cuda.h>
34  #include <curand.h>
35  #include "SpinTable.cuh"
36  #include "time_update.h"
37
38  /** \brief Checking for errors from CUDA
39   *
40   *  CUDA Helper function.
41   *  Prints given message on screen, whenever
42   *  it uncovers Error in CUDA processing.
43   *
44   *  \param[in] title - message to be printed,
45   *                     along with CUDA error
46   *
47   *  \returns
48   *  true on success
49   *  false on CUDA error
50   */
51  bool cuda_check_error(const char *title)
52  {
53    cudaError_t ce = cudaGetLastError();
54    if(ce != cudaSuccess) {
55      printf("%s: %s\n", title, cudaGetErrorString(ce));
56      return false;
57    }
58
59    return true;
60  }
61
62  /** \brief Wait until the~end of CUDA kernel
63   *
64   *  CUDA Helper function.
65   *  This function will wait for CUDA kernel to end,
66   *  without using 100% of CPU.
67   *
68   *  \param[in/out] evt - reusable CUDA Event
69   */
70  void sleep_till_end(cudaEvent_t &evt)
71  {
```

```
72    cudaEventRecord(evt, 0);
73
74    while (cudaEventQuery(evt) == cudaErrorNotReady) {
75      usleep(1);
76    }
77  }
78
79  /** simple macro that replaces variable name
80   *  with exactly the~same string
81   *  Example:
82   *  TO_STR(test_variable) = "test_variable"
83   */
84  #define TO_STR(param) #param
85
86  /** Configuration structure
87   *
88   *  Will be copied to CUDA constant memory
89   */
90  struct Config
91  {
92    FLOAT J, B, beta;
93
94    uint32_t X_N, Y_N, Z_N;
95
96    uint32_t seed;
97
98    FLOAT Udelta[14];
99    FLOAT testers[14];
100 };
101
102 /**< data for single CUDA thread (for device) **/
103 struct ThreadData
104 {
105   FLOAT U;
106   FLOAT M;
107 };
108
109 /**< data for single CUDA thread (for host) **/
110 struct ThreadDataH
111 {
112   double U;
113   double M;
114 };
115
116 /**< Structure for holding device and host tables **/
117 struct Tables
118 {
119   SpinTable host1, *dev;
120
121   /** \brief Initializer
122    *
123    *  Creates spin table and device pointer
124    *  to it. Also initializes it into either
125    *  hot or cold state, depending on beta
126    *  read from configuration.
127    */
128   void init()
129   {
130     host1.host_init(X_N, Y_N, Z_N);
```

```
131
132       if (beta < 0.225) {
133         fprintf(stderr, "Hot start\n");
134         host1.generate_random(time(0) + getpid());
135       } else {
136         fprintf(stderr, "Cold start\n");
137         host1.generate_clear();
138       }
139
140       cudaMalloc((void **) & dev, sizeof(SpinTable));
141
142       cudaMemcpy(dev, & host1, sizeof(SpinTable), cudaMemcpyHostToDevice);
143     }
144
145     /** \brief Destructor
146      *
147      *  Frees allocated resources.
148      */
149     void destroy()
150     {
151       cudaFree(dev);
152
153       host1.host_destroy();
154     }
155   };
156
157   /**< Holder for data for all thread and for random number generator **/
158   struct Actors
159   {
160     ThreadData *h_td, *d_td;
161     FLOAT *d_random;
162     uint32_t N;
163     curandGenerator_t prng;
164     curandStatus_t rand_res;
165
166     /** \brief Initializer
167      *
168      *  Allocates host and device memory for all threads.
169      *  Initializes random number generator.
170      */
171     void init()
172     {
173       N = X_N * Y_N * Z_N / 2;
174
175       cudaMalloc((void **) & d_td, sizeof(ThreadData) * N);
176       cudaMalloc((void **) & d_random, sizeof(FLOAT) * N);
177       h_td = (ThreadData *) malloc(sizeof(ThreadData) * N);
178
179       clear_device();
180
181       curandCreateGenerator(&prng, CURAND_RNG_PSEUDO_DEFAULT);
182
183       curandSetPseudoRandomGeneratorSeed(prng, time(0) + getpid());
184     }
185
186     /** \brief Generates random numbers
187      *
188      *  Generates ALL random number needed in the~next half-sweep
189      *  (single pass of checkerboard decomposition)
```

```
190     */
191    void generate_random()
192    {
193      rand_res = curandGenerateUniform(prng, (FLOAT *) d_random, N);
194
195              (double *) d_random, sizeof(FLOAT) * N);
196      if (rand_res != CURAND_STATUS_SUCCESS) {
197        fprintf(stderr, "Unable to generate all numbers: %u\n",
198            (uint32_t) rand_res);
199        abort();
200      }
201    }
202
203    /** \brief Resets threads data on device
204     */
205    void clear_device()
206    {
207      cudaMemset(d_td, 0, sizeof(ThreadData) * N);
208    }
209
210    /** \brief Copies data from threads from device to host
211     *
212     *  Note: This operation is slow, should be used only
213     *        when really needed.
214     */
215    void copy_to_host()
216    {
217      cudaMemcpy(h_td, d_td, sizeof(ThreadData) * N,
218              cudaMemcpyDeviceToHost);
219    }
220
221    /** \brief Destructor
222     *
223     *  Frees allocated resources
224     */
225    void destroy()
226    {
227      cudaFree(d_random);
228      cudaFree(d_td);
229      free(h_td);
230      curandDestroyGenerator(prng);
231    }
232  };
233
234  /**< CUDA Device configuration (constant memory) **/
235  __constant__ Config cfg;
236
237  /** \brief Count interaction for a single spin (global version)
238   *
239   *  DEVICE FUNCTION
240   *
241   *  This assumes nearest neighbour interaction.
242   *  Note: given spin position will be understood with
243   *        periodic boundary conditions.
244   *
245   *  \param[out] data - structure to write result to
246   *  \param[in] spin_t1 - spin table to get spins from
247   *  \param[in] x - horizontal position of desired spin
248   *  \param[in] y - vertical position of desired spin
```

```
249  *  \param[in] z - depth of desired spin
250  */
251 __device__ void single_step_full(ThreadData *data,
252                  SpinTable *spin_t1,
253                  int x, int y, int z)
254 {
255   if ((x >= cfg.X_N) || (y >= cfg.Y_N) || (z >= cfg.Z_N)) return;
256
257   int center, down, right, in;
258   center = spin_t1->get(x, y, z);
259   down = spin_t1->get(x, y + 1, z);
260   right = spin_t1->get(x + 1, y, z);
261   in = spin_t1->get(x, y, z + 1);
262
263   data->M += center;
264   data->U += - cfg.J * (center * down +
265               center * right +
266               center * in)
267         - cfg.B * center;
268 }
269
270 /** \brief Counts magnetization and energy for entire lattice
271  *
272  *  KERNEL
273  *
274  *  Hamiltonian inside, nearest neighbour interaction and
275  *  periodic boundary conditions are assumed.
276  *
277  *  \param[out] data_t - structure to be filled with information
278  *                       about magnetization and energy of the~lattice
279  *  \param[in] spin_t1 - spin table to count values from
280  */
281 __global__ void count_full_energy(ThreadData *data_t,
282                   SpinTable *spin_t1)
283 {
284   uint32_t x, y, z;
285
286   x = blockIdx.x;
287   y = blockIdx.y;
288   z = threadIdx.x * 2;
289
290   int add = (x + y) % 2;
291   z += add;
292
293   int idx = (x + y * cfg.X_N + z * cfg.X_N * cfg.Y_N) / 2;
294   ThreadData *data = & data_t[idx];
295
296   single_step_full(data, spin_t1, x, y, z);
297
298   __syncthreads();
299
300   if (add) {
301     z -= 1;
302   } else {
303     z += 1;
304   }
305
306   single_step_full(data, spin_t1, x, y, z);
307 }
```

```
308
309 /** \brief Single spin update
310  *
311  *  DEVICE FUNCTION
312  *
313  *  This is the~realization of Metropolis algorithm.
314  *  - delta energy for current configuration is calculated
315  *  - using pre-calculated tables, we determine whether
316  *    this configuration should change or not
317  *  - if needed, flip is performed
318  *
319  *  \param[out] data - structure to write result to
320  *  \param[in] random - random number associated with this thread,
321  *                      used for determining spin flip.
322  *  \param[in/out] spin_t1 - spin table to get spins from
323  *  \param[in] x - horizontal position of desired spin
324  *  \param[in] y - vertical position of desired spin
325  *  \param[in] z - depth of desired spin
326  */
327 __device__ void single_step(ThreadData *data,
328                             FLOAT *random,
329                             SpinTable *spin_t1,
330                             int x, int y, int z)
331 {
332   if ((x >= cfg.X_N) ||
333     (y >= cfg.Y_N) ||
334     (z >= cfg.Z_N)) return;
335
336   int center, down, right, up, left, cross, in, out;
337   center = spin_t1->get(x, y, z);
338   up = spin_t1->get(x, y - 1, z);
339   down = spin_t1->get(x, y + 1, z);
340   left = spin_t1->get(x - 1, y, z);
341   right = spin_t1->get(x + 1, y, z);
342   in = spin_t1->get(x, y, z + 1);
343   out = spin_t1->get(x, y, z - 1);
344
345   cross = center * down +
346       center * up +
347       center * right +
348       center * left +
349       center * in +
350       center * out;
351
352   int index = (cross + 6) / 2 + 7 * ((center + 1) / 2);
353   FLOAT Udelta = cfg.Udelta[index];
354
355   FLOAT tester = cfg.testers[index];
356
357   FLOAT randed = (FLOAT) 1.0 - *random;
358
359   if ((Udelta < 0) || (randed < tester)) {
360
361     spin_t1->flip(x, y, z);
362
363     data->M += -2.0f * center;
364     data->U += Udelta;
365   }
366 }
```

```
367
368 /** \brief Checkerboard decomposition step
369  *
370  *  KERNEL
371  *
372  *  This function updates half of the lattice using
373  *  Metropolis algorithm.
374  *  This one is very simple, thanks to splitting
375  *  cube into lines.
376  *
377  *  \param[out] data_t - data table for all threads
378  *  \param[in] random_t - table with random numbers for all threads
379  *  \param[in/out] spin_t1 - spin table to be modified
380  *  \param[in] adder - determines, whether we're updating "white"
381  *                     or "black" part of the~checkerboard.
382  */
383 __global__ void count_MC_step(ThreadData *data_t,
384                               FLOAT *random_t,
385                               SpinTable *spin_t1,
386                               int adder)
387 {
388   uint32_t x, y, z;
389
390   x = blockIdx.x;
391   y = blockIdx.y;
392   z = threadIdx.x * 2 + (x + y + adder) % 2;
393   int idx = (x + y * cfg.X_N + z * cfg.X_N * cfg.Y_N) / 2;
394
395   ThreadData *data = & data_t[idx];
396   FLOAT *random = & random_t[idx];
397
398   single_step(data, random, spin_t1, x, y, z);
399 }
400
401 /** \brief Preparation of global tables for device
402  *
403  *  Main values from configuration are copied.
404  *
405  *  Energy delta and probability tables are filled:
406  *  There are 14 possibilities (7 for center spin with
407  *  value 1 and 7 with value -1). We assign to each of
408  *  that configuration an integer from 0 (including) to
409  *  13 (including). Then we calculate energy difference
410  *  for virtual spin flip performed on that particular
411  *  configuration, and probability of real spin flip.
412  */
413 void fill_and_export_cfg()
414 {
415   cfg.J = J;
416   cfg.beta = beta;
417   cfg.B = B;
418
419   cfg.X_N = X_N;
420   cfg.Y_N = Y_N;
421   cfg.Z_N = Z_N;
422
423   cfg.seed = time(0) + getpid();
424
425   for (uint8_t i = 0; i < 14; ++i) {
```

```
426       int c = (i / 7) * 2 - 1;
427       int e = (i % 7) * 2 - 6;
428
429       FLOAT delta = 2.0 * (J * e + B * c);
430
431       cfg.Udelta[i] = delta;
432       cfg.testers[i] = exp(-beta * delta);
433     }
434
435     cudaMemcpyToSymbol(TO_STR(cfg), &cfg, sizeof(cfg));
436   }
437
438   /** \brief Copies all data obtained on GPU
439    *
440    *  Helper function.
441    *  Copies all actors and sums up their magnetizations
442    *  and energies obtained in simulation.
443    *
444    *  \param[in] act - Actor structure, to take data from
445    *  \param[in/out] ret - host data structure, to
446    *                       accumulate results in
447    */
448   void sumup_actors(Actors &act, ThreadDataH &ret)
449   {
450     act.copy_to_host();
451
452     for (uint32_t i = 0; i < act.N; ++i) {
453       ret.M += act.h_td[i].M;
454       ret.U += act.h_td[i].U;
455     }
456   }
457
458   /** \brief Final set of calculations
459    *
460    *  This method divides given values by amount of
461    *  Monte Carlo steps taken. Also calculates
462    *  higher order cumulants.
463    *
464    *  \param[in/out] final - set of values to be recalculated
465    */
466   void divide_final(termodyna_struct_t *final)
467   {
468     final->M /= MC_N;
469     final->M_S /= MC_N;
470     final->M_S2 /= MC_N;
471     final->M_6 /= MC_N;
472     final->M_8 /= MC_N;
473     final->M_10 /= MC_N;
474     final->U /= MC_N;
475     final->U_S /= MC_N;
476
477     final->V4 = final->M_S2 /
478       (final->M_S * final->M_S);
479     final->V6 = final->M_6 /
480       (final->M_S * final->M_S * final->M_S);
481     final->V8 = final->M_8 /
482       (final->M_S * final->M_S * final->M_S * final->M_S);
483     final->V10 = final->M_10 /
484       (final->M_S * final->M_S * final->M_S * final->M_S * final->M_S);
```

```
485 }
486
487 /** \brief Program entry
488  *
489  *  \param[in] argc - amount of command line arguments passed to program
490  *  \param[in] argv - arguments passed to program
491  */
492 int main(int argc, char **argv)
493 {
494   srand(time(0) + getpid());
495
496   struct timeval tstart, tend, tdiff;
497   // start measuring total time used by algorithm
498   gettimeofday(& tstart, NULL);
499
500   // read configuration from command line
501   if (parse_args(argc, argv)) {
502     return 1;
503   }
504
505   TimeData time_data;
506   init_timedata(& time_data);
507
508   // print current configuration
509   hello_message();
510   // prepare spin configuration tables and copy them to device
511   fill_and_export_cfg();
512
513   // prepare spin table
514   Tables tables;
515   tables.init();
516
517   // prepare actor data
518   Actors actors;
519   actors.init();
520
521   termodyna_struct_t full;
522   // prepare structure to keep data in
523   ThreadDataH base;
524   memset(& full, 0, sizeof(full));
525   memset(& base, 0, sizeof(base));
526
527   // pick block and grid sizes
528   dim3 block_size = dim3(Z_N / 2, 1, 1);
529   dim3 grid_size = dim3(X_N, Y_N, 1);
530
531   fprintf(stderr, "grid: (%u, %u, %u), block: (%u x2, %u, %u)\n",
532       grid_size.x, grid_size.y, grid_size.z,
533       block_size.x, block_size.y, block_size.z);
534   fprintf(stderr, "effective: grid: %u, block: %u\n",
535       grid_size.x * grid_size.y * grid_size.z,
536       block_size.x * block_size.y * block_size.z);
537
538   cudaEvent_t evt;
539   cudaEventCreate(& evt);
540
541   // count energy of the~system we start with
542   count_full_energy<<<grid_size, block_size>>>(actors.d_td, tables.dev);
543   sleep_till_end(evt);
```

```
544   if (! cuda_check_error("count_full_energy")) return 1;
545   sumup_actors(actors, base);
546   actors.clear_device();
547
548   // perform warm-up
549   for (uint32_t i = 0; i < WARMUP; ++i) {
550     start_measure(& time_data);
551
552     actors.generate_random();
553     count_MC_step<<<grid_size, block_size>>>(actors.d_td,
554             actors.d_random, tables.dev, 0);
555     sleep_till_end(evt);
556     if (! cuda_check_error("warmup")) return 1;
557
558     actors.generate_random();
559     count_MC_step<<<grid_size, block_size>>>(actors.d_td,
560             actors.d_random, tables.dev, 1);
561     sleep_till_end(evt);
562     if (! cuda_check_error("warmup")) return 1;
563
564     end_measure(& time_data);
565   }
566
567   // the simulation
568   for (uint32_t i = 0; i < MC_N * SKIP; ++i) {
569     start_measure(& time_data);
570
571     actors.generate_random();
572     count_MC_step<<<grid_size, block_size>>>(actors.d_td,
573             actors.d_random, tables.dev, 0);
574     sleep_till_end(evt);
575     if (! cuda_check_error("main sweep 0")) return 1;
576
577     actors.generate_random();
578     count_MC_step<<<grid_size, block_size>>>(actors.d_td,
579             actors.d_random, tables.dev, 1);
580     sleep_till_end(evt);
581     if (! cuda_check_error("main sweep 1")) return 1;
582
583     end_measure(& time_data);
584
585     if ((i % SKIP) == 0) {
586       sumup_actors(actors, base);
587       actors.clear_device();
588
589       full.M += base.M;
590       full.M_S += pow(base.M, 2.0);
591       full.M_S2 += pow(base.M, 4.0);
592       full.M_6 += pow(base.M, 6.0);
593       full.M_8 += pow(base.M, 8.0);
594       full.M_10 += pow(base.M, 10.0);
595       full.U += base.U;
596       full.U_S += pow(base.U, 2.0);
597     }
598   }
599
600   print_results(& time_data, stderr);
601
602   divide_final(& full);
```

```
603     // print output data
604     print_structure_data(& full);
605
606     // free memory and cleanup
607     cudaEventDestroy(evt);
608     actors.destroy();
609     tables.destroy();
610     cudaThreadExit();
611
612     gettimeofday(& tend, NULL);
613     // check and print how long it took
614     timersub(& tend, & tstart, & tdiff);
615     fprintf(stderr, "Obliczenia zajely: %lu.%06lus\n",
616         (unsigned long int) tdiff.tv_sec,
617         (unsigned long int) tdiff.tv_usec);
618
619     return 0;
620 }
```

Listing 9. Makefile

```
1  CXXFLAGS=-Wall -Wextra -O2
2  NVCCFLAGS=-O2 --ptxas-options=-v -arch=sm_21
3
4  all: time_update.o config.o termodyna_struct.o main.o
5    g++ ${CXXFLAGS} -o Ising3d *.o -L/usr/local/cuda/lib64 -lcuda -lcudart -lcurand
6
7  debug:
8    nvcc --cubin --pxtas-options=-v main.cu
9
10 %.o: %.cu
11   nvcc ${NVCCFLAGS} -c $<
12
13 clean:
14   rm -f *.o Ising3d
```