SIMULATION OF ISING SPIN MODEL USING CUDA

MIRO JURIŠIĆ

Supervisor: dr.sc. Dejan Vinković

Split, November 2011

Master Thesis in Physics

Department of Physics
Faculty of Natural Sciences and Mathematics
University of Split

Abstract

This thesis is about the utilization of Graphics Prossing Units (GPUs) for simulating the Ising spin model. The simulated problem is a simple one, but it can be used as a basis for more complex problems. This subject was chosen in order to explore the possibilities of the next generation of supercomputers that are utilizing numerical accelerators such as GPUs. Namely, GPUs are built for graphics, but they can also be used for general computation in science simulations. CUDA architecture enables a simple implementation of C/C++ codes on both CPUs and GPUs, and their intercommunication. In this thesis we calculated energy, magnetization, specific heat and magnetic susceptibility for various lattice sizes in the Ising model. Typical speed ups achieved by GPUs in comparison to CPUs in our simulations are factors between 50 and 200.

1 Introduction

Ising model was proposed in 1920 by Wilhelm Lens as a problem to his student Ernest Ising for modeling the (anti-) ferromagnetic systems[1]. Today, the Ising model is a widely used standard model of statistical physics. About 800 papers are being published every year that use this model to address problem in such diverse fields as neural networks, protein folding, biological membranes, social behavior and economics[2]. In introduction we show the significance of this model through basics of statistical mechanics and through description of the exact solution. In a further section, some computational aspects relevant for this thesis will be mentioned.

1.1 Statistical mechanics

The essential purpose of statistical mechanics is to understand the behavior of macroscopic physical systems by looking at its microscopic structure. In addition to macroscopic states introduced by thermodynamics, statistical mechanics introduces microstates, or configurations. Macroscopic states are defined by a few thermodynamical quantities, such as pressure and density, while microscopic states are specified by dynamical variables of all its constituents. Generally this leads to an extremely large number of degrees of freedom, which in almost all cases makes it impossible to find an exact solution. Nevertheless, statistical mechanics is able to give predictions of the systems behavior by means of a probability theory and statistics. To be able to use apparatus of probability theory, new concepts were introduced. Partition function is one of them, the general form for a classical system is defined as

$$Z = \sum_{all\ \ states} e^{-H/k_b T} \tag{1}$$

where H is the Hamiltonian for the system, T is the temperature, and k_b is the Boltzmann constant. Partition function contains essential information about the system under consideration and it is directly connected to other thermodynamical quantities. Form of partition function (1) shows dependences upon the size of the system and the number of degrees of freedom for each particle. Only in a few cases, where the interactions between particles are simple, evaluation of the partition function is possible. The probability of any particular microstate or configuration is also connected to the partition function. The

probability that the system is in particular state μ is given by

$$P_\mu = e^{-H_\mu/k_bT}/Z \tag{2}$$

Probability of a particular state is very important for the evolution of the system and for the Monte Carlo method, i.e. numerical method that uses random number, which is explained in more detail in the next chapter. The connection between statistical mechanics and thermodynamics is illustrated by

$$F = -k_bT \ln Z \tag{3}$$

where F is the free energy of the system and all other thermodynamic quantities can be calculated by appropriate differentiation of Eqn.(3). For an example, internal energy U can be obtained from the free energy via

$$U = -T^2\partial(F/T)/\partial T. \tag{4}$$

Since the number of different microstates is usually huge, we are not only interested in probabilities of individual microstates, but also in probabilities of macroscopic variables that can be measured in an experiment. For example, relation for the specific heat C_V can be found by looking at the fluctuation of an internal energy. The average energy is denoted $\bar{U}$ and U is a fluctuating quantity that corresponds to the energy of particular microstate. They are connected trough moments

$$\bar{U}(\beta) = \langle H(\mu)\rangle \equiv \sum_\mu P_\mu H(\mu) \tag{5}$$

$$\langle H^2\rangle = \sum_\mu H^2 P_\mu \tag{6}$$

and note the relation $-(\partial U(\beta)/\partial\beta)_V = \langle H^2\rangle - \langle H\rangle^2$ where $\beta = 1/k_bT$. The specific heat thus yields a fluctuation relation $C_V = (\partial U/\partial T)_V$ and

$$k_bT^2C_V = \langle H^2\rangle - \langle H\rangle^2 = \langle(\triangle U)\rangle_{NVT} \qquad \triangle U \equiv H - \langle H\rangle \tag{7}$$

1.2 Order parameter

Order parameter is some property of the system that is non-zero in the ordered phase and identically zero in the disordered phase. The order parameter is defined differently in different kinds of physical systems. Depending on the physical system, an order parameter may be measured by a variety of experimental methods. For example, in liquid crystals where order parameter is an orientation order it can be measured by scattering methods. An order parameter may be a scalar quantity or may be a multicomponent or even complex quantity. Phase transitions of a system can be described as a transition between disordered state and one which was non-zero order parameter at some finite temperature, the transition temperature T_C. If the first derivatives of the free energy are discontinuous at the transition temperature Tc , the transition is termed first order. If the first derivatives of the free energy are continuous, but the second derivatives of the free energy are discontinuous at the transition temperature, the transition is termed second order.

1.3 Ising model

Ising model is a simple model which is used to describe magnetic system. It postulates a periodic d-dimensional lattice, that is made up of magnetic dipoles, each of them associated with a spin s_i, where i represent lattice site. It is assumed that the spins can either be up or down with the value +1 associated with up and the value -1 associated with down. Hamiltonian of a system is

$$E = H = -J \sum_{<kl>}^{N} s_k s_l - B \sum_{k}^{N} s_k \tag{8}$$

where J is a coupling constant, $\langle kl \rangle$ is a sum over nearest neighbors, B is an external magnetic field and N is the number of magnetic dipoles in a system. Coupling constant J denotes the strength of the coupling between adjacent spins and depending of its sign, system can prefer ferromagnetism ($J > 0$) or anti - ferromagnetism ($J < 0$). The first solution for Ising model was given by Ernst Ising itself, but only for one dimensional problem(d=1). In the first dimension Ising model does not exhibit a phase transition, while in the higher dimensions there exists the transition temperature T_C, above which

order parameter is zero. Order parameter for ferromagnetic coupling is magnetization M

$$M = \sum_{all\ i} s_i \tag{9}$$

and for anti - ferromagnetic coupling is a sum over every second magnetic dipole. Ising model has a second order phase transition, so quantities like the heat capacity C_V and the susceptibility χ are discontinuous or diverge at the critical point in the thermodynamic limit, i.e., with an infinitely large lattice. For a finite lattice however, C_V and χ will not exhibit a diverging behavior, but they will however show a broad maximum near T_C. Similar to connection of variance of internal energy to heat capacity(7), relation for magnetic susceptibility is

$$\chi = \frac{1}{k_b T}\langle(\triangle M)\rangle = \frac{1}{k_b T}(< M^2 > - < M >^2). \tag{10}$$

A probability distribution used to evaluate expressions for heat capacity and magnetic susceptibility is given by the Boltzmann distribution

$$P_i(\beta) = e^{(-\beta E_i)}/Z \tag{11}$$

with $\beta = 1/k_B T$ being the inverse temperature, E_i is the energy of a microstate i while Z is the partition function(see equation 1). Looking at probability distribution it is easy to notice that temperature is used as a parameter and that this is the canonical ensemble.

1.3.1 Exact solution of the 2D Ising model on finite squared lattices

The first solution of two dimension Ising model in zero external magnetic field was proposed by L.Onsager [3] in 1944. This solution is valid only in the thermodynamic limit and because in simulation we always have finite sized lattices more appropriate solution is given by A. E. Ferdinand and M. E. Fisher [4] that concerns problem with finite size effects. Canonical partition function Z_{mn} of a finite m x n square lattice with periodic boundary condition in a zero external magnetic field is

$$Z_{mn}(T) = \frac{1}{2}(2sinh(2K))^{\frac{1}{2}mn}\sum_{i=1}^{4} Z_i K, \tag{12}$$

where K is defined as $K = J/k_b T$. The partial partition functions Z_i are defined as

$$Z_1 = \prod_{r=0}^{n-1} 2cosh\frac{1}{2}m\gamma_{2r+1};\ Z_2 = \prod_{r=0}^{n-1} 2sinh\frac{1}{2}m\gamma_{2r+1};\ Z_3 = \prod_{r=0}^{n-1} 2cosh\frac{1}{2}m\gamma_{2r};\ Z_4 = \prod_{r=0}^{n-1} 2sinh\frac{1}{2}m\gamma_{2r} \tag{13}$$

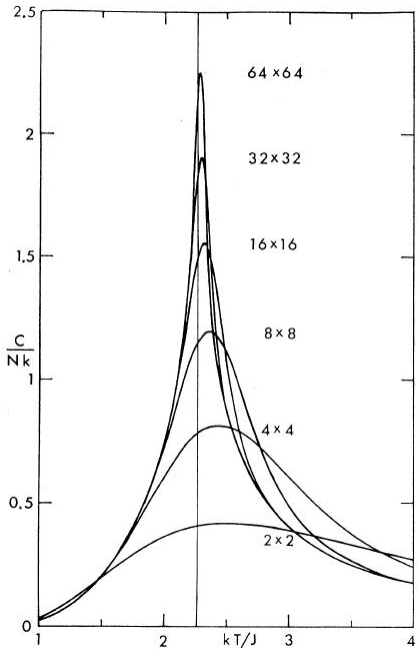


Figure 1: The specific heat per spin C(T) for small Ising lattices with boundary condition in a zero magnetic field according to A.E.Ferdinand and M.E.Fisher[4] for mxn square lattice with $m = n = 2, 4, 8, 16, 32$ and $64 (N = mn)$.

where

$$cosh\gamma_l = cosh2Kcoth2K - cos(l\pi/n). \tag{14}$$

The internal energy per spin is given by

$$\begin{aligned} \frac{U_{mn}}{mn} &= -(mn)^{-1} J \frac{d}{dK} lnZ_{mn} \\ &= -Jcoth2K - \frac{J}{mn}[\sum_{i=1}^{4} Z_i'][\sum_{i=1}^{4} Z_i]^{-1} \end{aligned} \tag{15}$$

while the specific heat per spin is

$$\begin{aligned} \frac{C_{mn}}{k_B mn} &= (mn)^{-1} K^2 \frac{d^2}{dK^2} lnZ_{mn} \\ &= = -2Kcosch^2(2K) + \frac{K^2}{mn}[\frac{\sum_{i=1}^{4} Z_i'}{\sum_{i=1}^{4} Z_i''} - (\frac{\sum_{i=1}^{4} Z_i'}{\sum_{i=1}^{4} Z_i})^2] \end{aligned} \tag{16}$$

where the primes denote differentiation with respect to K.

1.4 HPC - High Performance Computing

Term high performance computing is used for computing on supercomputers and large clusters and objective of this section is to show increasingly important role of massively

parallel computing. By definition, parallel computing is a simultaneous use of multiple compute resources to solve a computational problem. At this moment, supercomputers are in petascale range (10^{15} FLOPS), with attendance to reach exascale range(10^{18} FLOPS) by the year 2018[5]. Most of the software that are in use today are programmed for single CPU machines. These programs are said to be serial, because instructions are executed one after another.

1.4.1 Flynn's taxonomy

Flynn's taxonomy[6] classifies computer architecture depending on a number of concurrent instructions and data streams. For single instruction architecture there is only one instruction stream running on CPU during any clock cycle. Likewise, for single data there is only one data stream as input data during any clock cycle. For multiple instruction computer architectures have multiple processors and every processor may execute a different instruction stream independently of each other. For multiple data there are also multiple processors and every processor may work with a different data stream with same or different instructions. Every computer architecture belongs into one of these four classes.

SISD Single Instruction, Single Data	MISD Multiple Instruction, Single Data
SIMD Single Instruction, Multiple Data	MIMD Multiple Instruction, Multiple Data

1.4.2 Parallel Computer Memory Architectures

Parallel computer architectures have more than one core on a single CPU, where each have some small amount of memory so it can function properly. Next to these small memory parts there is usually a much larger memory card that can accommodate huge amount of data needed for simulations. Depending on memory resources per single CPU there are shared, distributed or hybrid memory architectures[7].
Shared memory architecture means that multiple processors share the same memory resources as global address space and that changes in a memory are visible to all processors. Advantage of this architecture is fast data sharing between processors, but sharing of memory space means that programmer needs to synchronize usage of this memory space

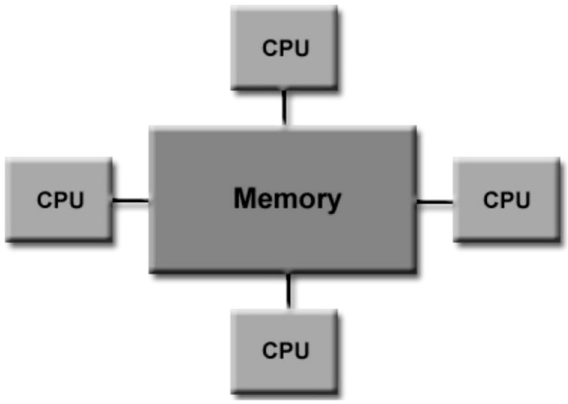


Figure 2: Schematic view of shared memory architecture

to ensure that processors are using correct data. Main disadvantage is the lack of scalability between memory and CPUs. By increasing number of CPUs, traffic on the shared memory-CPU path increases and this can slow down system because of memory management.

Unlike shared memory systems, distributed memory systems have processors with their own inter-processor memory. These systems require a communication network to connect different processors local memories. There is no concept of global address space across all processors and each processor operates independently. Changes in local memory have no effect on the memory of other processors so there is no worry about accessing correct data like in shared memory. To access data of another processor, programmer needs to explicitly define how, when and which data is communicated. Main advantage of this system is disadvantage of shared system, namely scalability with number of processors, because of local memory increasing number of processors will also increase size of memory. Other advantage is a fast access to local memory in comparison to communication with memory using some interface. Main disadvantage are communication mechanisms that are needed to share data among different processors.

The largest and fastest computers in the world today employ hybrid memory architecture. They have advantages and disadvantages same as shared and distributed memory architectures. The shared memory component is usually machine that can be addressed as global memory, while the distributed memory component is the networking of multiple processors with their own memory.

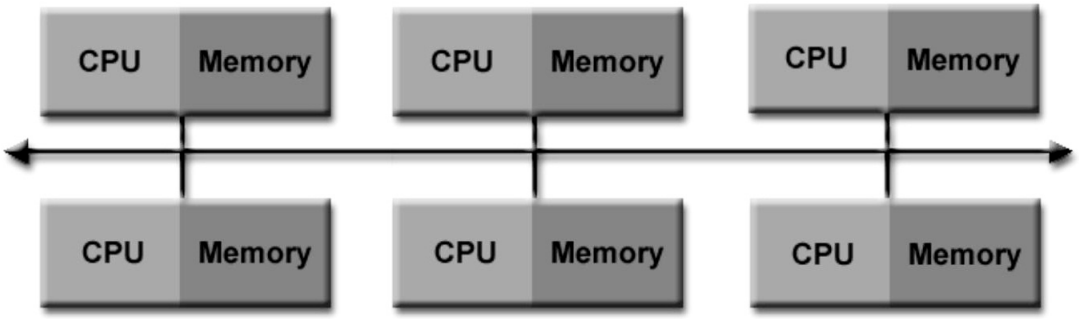


Figure 3: Schematic view of distributed memory architecture

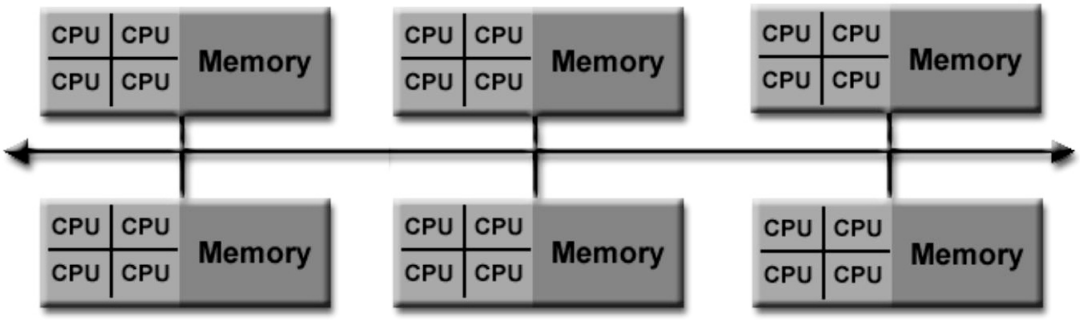


Figure 4: Schematic view of hybrid memory architecture.

1.4.3 Parallel programming models

Parallel programming models are abstraction above hardware and memory architectures. There are several models that are used often like shared-memory (OpenMP), message passing (MPI), threads. These models are not specific to a particular type of machine or memory architecture and any model can theoretically be implemented on any hardware. All these models have some mechanisms to allow communication among processors. For example, in the shared-memory programming model all processors have the same global address space, which they read and write asynchronously and there is a need for locks to control access to the shared memory. In a message passing programing model there is a mechanism of messages and processors that communicate by sending and receiving messages.

1.4.4 CUDA and TOP500

CUDA is not only a parallel programming model, rather it is a new programming architecture. It is new hardware and software design connected to boost effectiveness of a graphics cards manufactured by NVIDIA, explained in more details in section 3. In 1993, because of the competition between manufacturers and users of supercomputers, high-performance computing community decided to assemble and maintain a list of the 500 most powerful computer systems. This list has been updated twice a year since June 1993 with the assistance of computational scientists, manufacturers and the Internet community. In the present list, which is called the TOP500[8], computers are ranked by their performance on the LINPACK Benchmark[9]. The LINPACK software solves a dense system of linear equations. Used benchmark allows the user to scale the size of the problem and to optimize the software in order to achieve the best performance for a given machine. This performance does not reflect the overall performance of given system, as no single number ever could. It does, however, reflect the performance of dedicated system for solving a dense system of linear equations. Since the problem is very regular, the performance achieved is quite high, and the performance numbers give a good correction of peak performance.

Rank	Site	Computer/Year Vendor	Cores	R_{max}	R_{peak}	Power
1	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect / 2011 Fujitsu	548352	8162.00	8773.63	9898.56
2	National Supercomputing Center in Tianjin China	Tianhe-1A - NUDT TH MPP, X5670 2.93Ghz 6C, NVIDIA GPU, FT-1000 8C / 2010 NUDT	186368	2566.00	4701.00	4040.00
3	DOE/SC/Oak Ridge National Laboratory United States	Jaguar - Cray XT5-HE Opteron 6-core 2.6 GHz / 2009 Cray Inc.	224162	1759.00	2331.00	6950.60
4	National Supercomputing Centre in Shenzhen (NSCS) China	Nebulae - Dawning TC3600 Blade, Intel X5650, NVidia Tesla C2050 GPU / 2010 Dawning	120640	1271.00	2984.30	2580.00
5	GSIC Center, Tokyo Institute of Technology Japan	TSUBAME 2.0 - HP ProLiant SL390s G7 Xeon 6C X5670, Nvidia GPU, Linux/Windows / 2010 NEC/HP	73278	1192.00	2287.63	1398.61

Figure 5: Top500 list from June 2011. shows that the second place holds supercomputer accelerated by NVIDIA GPU.

2 Monte Carlo methods in Statistical mechanics

Since Statistical mechanics deals with systems with large number of degrees of freedom, the basic idea is to use computer simulation to explicitly follow trajectory of a system. For physically meaningful boundary conditions and particle interaction, trajectory will simulate the behavior of a real assembly of particles and statistical analysis of the trajectory will determine meaningful predictions of properties of the assembly. There are two classes of methods in Statistical mechanics. One is called Molecular dynamics, it considers classsical dynamical model of atoms and trajectory is formed by intergrating Newton's equations of motion. This method provides dynamical properties as well as statistical properties in equilibrium. The other, Monte Carlo method, cannot give dynamical proterties of a system, but it is applicable to wider range of problems like quantum systems and lattice model (Ising model).

2.1 A Monte Carlo trajectory

A trajectory is a chronological sequence of configurations for a system or path in a configuration space. For Ising model, configuration is the list of spin variables, $\psi = (s_1, s_2, ..., s_N)$, and trajectory is $\psi(t)$, where t represents number of steps. Configuration properties change as trajectory progress. An average value of a property $G_\psi = G(s_1, s_2, ..., s_N)$ over configurations visited during a trajectory with T steps is

$$\langle G \rangle = \frac{1}{T} \sum_{t=1}^{T} G_{\psi(t)}. \tag{17}$$

In Monte Carlo method, trajectory is produced in such manner that thermal equilibrium averages are given by

$$\langle G \rangle = \lim_{T->\inf} \langle G \rangle_T \tag{18}$$

Because trajectories are performed only for a finite time, an average over configurations will provide only estimate of real average. For Eqn.(18) to be true, the trajectories need to be ergodic. It means that if we simulate system for long enough all configurations will be visited, no matter of probability of particular configuration. The Monte Carlo method is a method for performing random walk through configuration space. Size of a configuration

space is in the most cases astronomically large. For two-dimensional Ising model, with lattice size 10x10, number of configurations is $2^{100} > 10^{25}$. Straightforward sampling of all configurations would be impractical, even for lattice sizes 10x10, so there are methods that would sample representative fraction of configuration space. Improved Monte Carlo trajectory is a random walk trough the states that are statistically most important, so called importance sampling method. Statistically unimportant configurations are the ones with negligible weight in Boltzmann distribution.

2.2 Markov process

Since Monte Carlo steps are not truly dynamical steps in time, but rather artificial steps, there is a great deal of flexibility in choosing particular algorithm to perform random walk, but there are also some constrains that can be explained trough Markov process. All Monte Carlo algorithms are based on Markov processes in order to move in configuration space. A Markov process is a random walk with a selected probability for making a move and with independence on the previous steps, with independence of the history of the system[10]. Repeated use of the Markov process produces a so-called Markov chain. The Markov process needs to obey two important conditions, that of already mentioned ergodicity and detailed balance. These conditions are only constrains on our algorithms for accepting or rejecting random moves. To check these conditions, probabilities $P(j->i)$ for the system to be in state i after previously having been in state j is introduced. At each step the system must go somewhere, so condition

$$\sum_i P(j->i) = 1 \tag{19}$$

is satisfied. Detail balance condition ensures that the correct equilibrium distribution will be simulated. From definition of being in equilibrium comes expression

$$\sum_j W(j->i)w_j = \sum_i W(i->j)w_i \tag{20}$$

where $W(j->i)$ is a transition rate for crossing from configuration j into configuration i while w_i is a probability of configuration i. This condition, that the rates are equal going from and to a given state i are in general not sufficient to guarantee that after finite number of simulation steps we will reach the correct distribution. So there is stronger condition,

the condition of detailed balance

$$W(j->i)w_j = W(i->j)w_i. \tag{21}$$

For the purpose of later calculation, more appropriate form of detailed balance condition is

$$\frac{W(j->i)}{W(i->j)} = \frac{w_i}{w_j} \tag{22}$$

2.3 Metropolis algorithm for Ising model

The Metropolis algorithm is particular Markov chain Monte Carlo method for obtaining a sequence of random samples from the Boltzmann probability distribution[10]

$$w_i = \frac{exp(-\beta(E_i))}{Z}. \tag{23}$$

Including expression for Boltzmann distribution(23) in to detailed balance(22), we get

$$\frac{w_i}{w_j} = exp(-\beta(E_i - E_j)). \tag{24}$$

Since we do not know the analytic form of the transition rate between configuration, we are free to model it as

$$W(i->j) = g(i->j)A(i->j) \tag{25}$$

where g is a selection probability while A is the probability for accepting a move. Since all possible spin orientations are equally likely to appear, the selection probability is proportional to inverse number of spins N, namely

$$g(i->j) = \frac{1}{N} \tag{26}$$

Since the selection ratio is the same for both transitions, the detailed balance gives

$$\frac{A(j->i)}{A(i->j)} = exp(-\beta(E_i - E_j)) \tag{27}$$

To perform random walk through configuration space, rule for performing a move and for accepting or rejecting it is needed. There are various algorithms that satisfy the detailed balance conditions and ergodicity. The Metropolis algorithm has a

$$A(j->i) = \begin{cases} exp(-\beta(E_i - E_j)) & \text{if } E_i - E_J > 0 \\ 1 & \text{else} \end{cases} \tag{28}$$

rule for accepting or rejecting a move. It says, if the move is energetically favorable it is accepted 100%, and if it is not, probabilities of accepting a move is $exp(-\beta(E_i - E_j))$. Simple Metropolis algorithm is shown in **Algorithm 1**, where selection of a spin s[x][y] at location (x,y) that changes its orientation is randomly chosen. Since energy contribution of a single spin depends only of its nearest neighbours, there is no need to perform loop over all spins for calculating new energy and magnetization of a new configuration after accepting spin change. To deal with this problem, only contribution of a chosen spin for energy and magnetization prior and after spin change are needed, $E_{new} = E_{old} + E_{[x][y]\ new} - E_{[x][y]\ old}$ and $M_{new} = M_{old} + M_{[x][y]\ new} - M_{[x][y]\ old}$.

Algorithm 1 Metropolis algorithm for two-dimensional lattice of size N. s[x][y] represents spin at location (x,y). $\triangle E$ is energy difference between new state and old state.

$x \Leftarrow random[0, N-1]$
$y \Leftarrow random[0, N-1]$
$s[x][y]* = -1$
calculate $\triangle E$
if $exp(-\triangle E/T) >= random[0, 1]$ **then**
  $E_{old} \Leftarrow E_{new}$
  $M_{old} \Leftarrow M_{new}$
else
  $s[x][y]* = -1$
end if

3 GPGPU - General Purpose Computation on Graphics Processing Units

Graphics processing unit (GPU) was originally developed for fast output of images onto computer display. Thanks to market demand for high-quality and real time graphics in computer applications, there is a huge advancement in graphics performance over the last few decades. First graphics hardware were expensive large systems, which evolved into small inexpensive PC accelerators. Performance increased from 50 million pixels per second to 1 billion pixels per second and from 100,000 vertices per second to 10 million vertices per second[11]. Performance increase is not only due to shrinking of semiconductor devices, they also have resulted from innovations in graphics algorithms and hardware design.

3.1 Graphics cards architecture

Early stage hardware design was a fixed-pipeline hardware that was configurable but not programmable. Configurable means that application from CPU could send different data and different commands, but they would follow always the same processing steps. The commands are given by calling an API functions. An API is a standardized layer of software, that allows application to use software or hardware services. The most used graphics APIs are DirectX(from Microsoft) and OpenGL(open source). Each next generation introduced additional hardware resources that would add additional reconfigurability to the device and new series of APIs, but built-in functions were not enough for ever more sophisticated application demands. The evolution path of graphics processors to an array of unified processors was natural, while at certain stages, graphics processors do a great deal of floating-point arithmetic on a completely independent data that can be exploited by hardware parallelism. Hardware parallelism is a reason for such a difference in design between GPU and a CPU(figure 6) and also in performance difference(figure 7). Even though programmability and performance increased, usability of GPU for general purpose computation was low because of time needed to be spend on writing codes. To write a code in DirectX or OpenGL, programmer needed to know a lot about graphics libraries and also programmer needed to convert data into pixels, so that GPU can process them. It was time consuming, until programming architecture like CUDA appeared.

Figure 6: GPU is specialized for computer-intensive, high parallel computation and therefore designed such that more transistors are devoted to data processing rather that data cache and flow control as in CPUs.

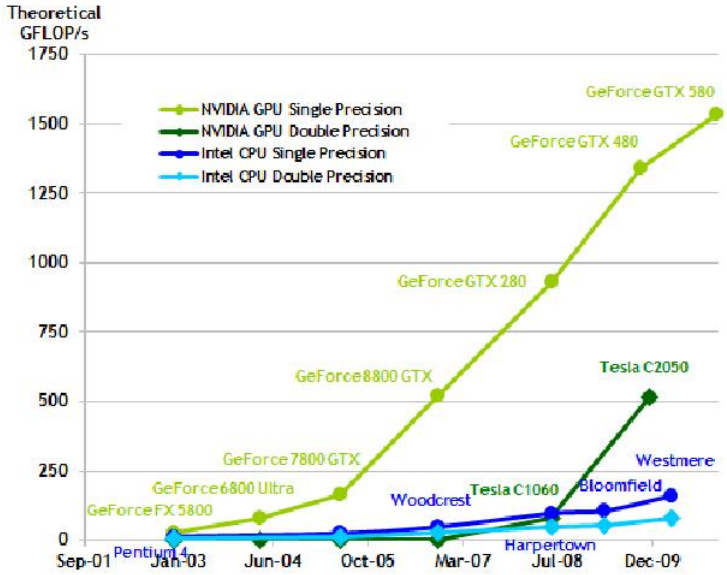


Figure 7: Floating point operations per second theoretical peak performances. GPU outperforms CPU almost 1:6

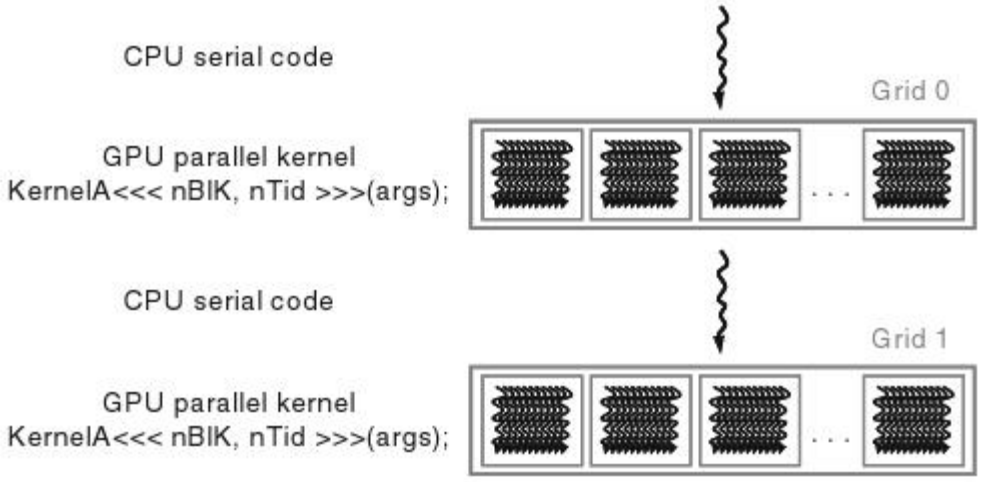


Figure 8: An example of a program structure that calls two kernel functions. Sequential part has only one thread line, while kernel has many threads distributed on device multiprocessors.

3.2 CUDA - Compute Unified Device Architecture

CUDA is a programing computer architecture developed by NVIDIA to simplify usage of GPU for developers with no prior experience with graphics. NVIDIA has built a new hardware and software architecture which can perform both traditional graphics-rendering tasks and general-purpose tasks. CUDA was first built into the card GeForce 8800, in November 2006. CUDA comes with a software environment that allows developers to use several high-level programming languages and or to use application programming interfaces. CUDA supports CUDA C, CUDA FORTRAN, OpenCL, and DirectCompute. CUDA C is a minimum extension of a standard C and at its core are three key abstractions: memory hierarchy, thread hierarchy and barrier synchronization.

3.2.1 Program structure

Even though figure 7 shows that GPUs can do more floating-point operations per second, it does not mean that CPUs will not outperform GPUs in some tasks. These are theoretical peak performances tested with highly regular and parallel problems. By looking at figure 6, one could guess that CPUs will perform better at problems that do not exhibit parallelism because of bigger and highly sophisticated control unit as well as larger cache, compared

to GPU. The main idea of GPGPU is to use both CPU and GPU together as a hybrid system. Problem is divided into parts that exhibit high parallelism and run it on GPU, while rest of the program would run on CPU. A CUDA program is a unified source code that is a mix of both host and device code. The NVIDIA C compiler *nvcc* separates host from device code during the compilation process. The host code is further compiled with the hosts standard C compilers and executed on CPU. The device code is further compiled by the nvcc and executed on a GPU device. Schematic view of example program that runs on both host and device is shown in figure 8. At first, serial part of program runs on the host, after which function that runs on device is called and, when it finishes execution, it is back on the host. Then another function is called that runs on the device. To distinguish which a function is going to run on device and which on host, CUDA has a special function type qualifiers.

- without qualifier, a function is by default host function
- *__host__* a function that will run only on host and that is callable only from host function
- *__global__* a function that will run on the device and is callable from the host only. Such function is also called kernel.
- *__device__* a function that will run on the device and it is callable only from the device function

Functions that run on the device have more restriction on their definitions than standard C functions. For instance *__global__*functions must have void return type and do not support recursion. *__device__*and *__global__* functions cannot have a variable number of arguments and cannot declare static variables inside their body. One can check all restriction in CUDA C programming guide[12].

3.2.2 Memory transfer and memory hierarchy

Host and the device are separate devices, each having its own memory spaces. Host cannot view data on the device and vice versa. To use device processing resources, one needs first

to allocate memory space on the device, then send data from host memory space to the device memory space, then resulting data needs to be copied from the device memory space to the host memory space and the last step is to deallocate device memory space. For these data management tasks, CUDA programmer uses runtime API functions $cudaMalloc()$ for allocation, $cudaMemcpy()$ for copying data from and to host and from and to device, and $cudaFree()$ for deallocating memory. Description of these API functions and all others are in NVIDIA CUDA Library Documentation[13]. Focusing just on device memory, one finds several levels of memory that a programmer can use. To optimize performance of device code programmer must know several facts about each memory level. Programmer needs to know the size of memory level, speed of access, scope of data residing on specific memory level, permissions to write and read data and lifetime of data. Some of these necessary informations depend on a graphics card[13] while others are mentioned here:

Global memory - variable qualifier $\_\_device\_\_$

Global memory is the largest memory space on the device and every streaming multiprocessor(SM), defined in Section 3.2.3., have exclusive right to read and write data, but also host has right to read and write data. This means that most communications and data gathering between host and the device will be through global memory. It takes 400 to 600 clock cycles of memory latency to access data on global memory by multiprocessor. Scope of data residing in global memory is entire grid(explained in Section 3.2.3) and lifetime is application execution time.

Local memory - variable qualifier $\_\_local\_\_$ **or array without qualifier**

Local memory also resides on device and can be accessed by only specific multiprocessor. This memory is usually generated automatically by compiler when data that are set for registers is larger that available memory space. This can lead to major slowdown of a simulation because local memory is just a part of global memory and likewise has the same memory latency. Scope of data is particular thread and lifetime is the same as lifetime of that particular thread.

Shared memory - variable qualifier $\_\_$ $\_\_shared\_\_$

Shared memory is on-chip and that makes it much faster than the local and global memory spaces. Since each multiprocessor has its own shared memory only scalar processors that reside on that specific multiprocessor have acess to this memory space.

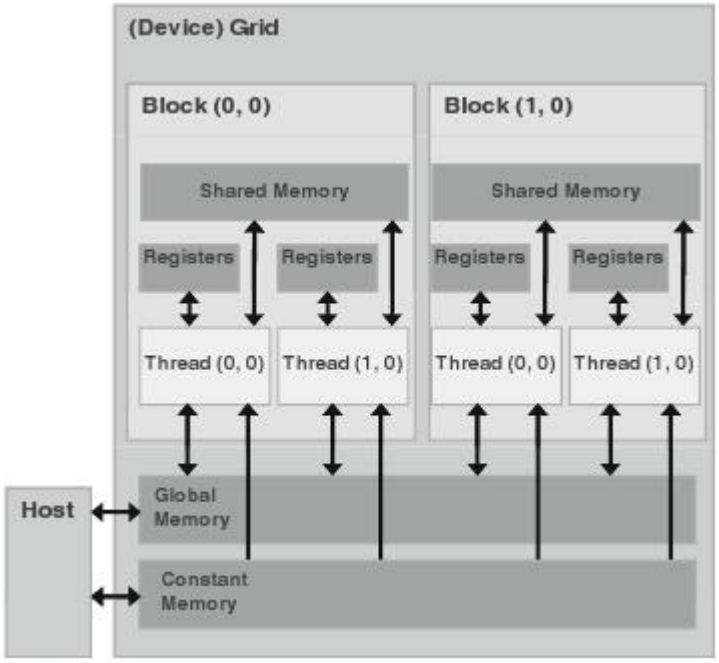


Figure 9: Memory hierarchy of hybrid system. Arrows show writing and reading permissions between memory levels.

Scope shared memory is inside a block(Section 3.2.3) and lifetime is kernel execution time.

Constant memory - variable qualifier *__constant__*

Constant memory is also on-chip memory space. Its memory latency has the same number of clock cycles as shared memory. Unlike shared memory accessed for writing has exclusively only host while device can only read this data. This memory space is usually used for often used constants needed for simulation. Scope of constant memory is entire grid and lifetime is application execution time.

Registers - variable without qualifiers(accept arrays)

Registers are memory space that are specific for each scalar processor so each processor has right to write and read from only its own 32 bit registers. Variable type qualifiers specify the memory location on the device of a variable. Scope is particular thread and lifetime is kernel execution time.

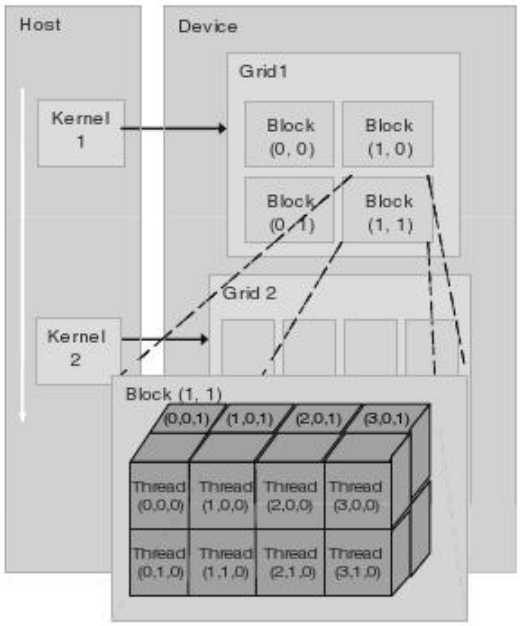


Figure 10: Grid launch is a multidimensional

3.2.3 CUDA threads and execution configuration <<< .... >>>

The CUDA hardware architecture is built around a scalable array of multi-threaded Streaming Multiprocessors (SMs), designed to execute hundreds of threads concurrently. To manage such a large amount of threads, NVIDIA designed a unique architecture called SIMT (Single-Instruction, Multiple-Thread), not included in Flynn's taxonomy for computer architecture (Section 1.3.1). It leverages thread-level parallelism by using hardware multi-threading. The SIMT architecture is akin to SIMD computer architecture. The main difference is that SIMD vector organizations expose the SIMD width to the software, whereas SIMT instructions specify the execution and branching behavior of a single thread.

When a CUDA program on the host invokes a kernel function $kernel\_func <<< Dg, Db >>> (parameter)$, it is launched as a grid of independent, scalar threads. Grid is a two level hierarchy of threads. At first level there are 1D or 2D thread blocks, and at the second level, each block is a 1D, 2D or 3D group of threads. These levels exist so that threads can distinguish themselves and to maximize hardware performance. Blocks of the grid are enumerated and distributed concurrently to multi-

processors with available execution capacity. As thread running blocks terminate, waiting blocks are launched on the vacated multiprocessors. The threads of a thread block are also enumerated and executed in groups of 32 parallel threads called warps. Enumeration of threads at both levels is done by built-in struct variables $threadIDx$ and $blockIDx$, called thread ID. At the call of a kernel, programmer must explicitly describe hierarchy of threads, meaning that programmers need to specify dimensionality and size of particular dimensions for each level of a grid. This is called execution configuration,
$<<< Dg, Db >>>$, where Dg and Db are built-in variable type $dim3$ with Dg representing dimensions of a grid at first level, number of blocks, and Db represents dimensions of each block. Dg and Db have restriction on size, while there is a maximum number of resident blocks and a maximum number of resident warps per multiprocessor for a given kernel that depends on the amount of registers and shared memory used by the kernel and the amount of registers and shared memory available on the multiprocessor. To assist programmers in choosing thread block size based on register and shared memory requirements, the CUDA SDK (Software Development Kit) provides a spreadsheet called the CUDA Occupancy Calculator, where occupancy is defined as the ratio of the number of resident warps to the maximum number of resident warps for various graphics cards.

3.2.4 Barrier synchronization and atomic functions

Threads in a block can specify barrier synchronization points. $void\ \_\_syncthreads()$ is a function that synchronizes all threads in a block, execution resumes normally only when all threads in a block have reached this point. It is important to notice that synchronization function does not work between threads from different blocks. Synchronization is usually used to avoid read-after-write, write-after-read, or write-after-write hazards, so synchronization points stand in-between accessing memory. Even though synchronization is a very useful tool, it can however lead to slow down of execution time or produce some other unwanted effects if some threads in a block evaluate branch conditions differently.
An atomic function performs a read-modify-write operation on data in global or shared memory and it guarantees performance without interference from other threads. It means that no other threads can access the same address until operation is complete. In other words each read, modify, write to that location are serialized, but the order in which they occur is undefined. There are several atomic functions. For example,

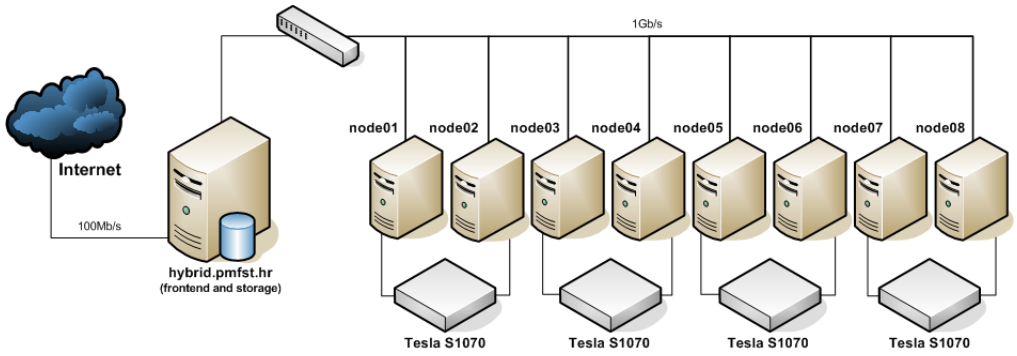
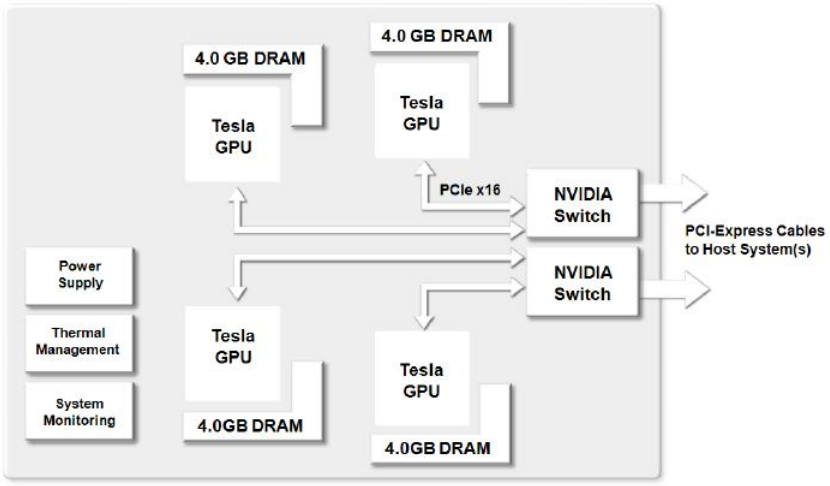


Figure 11: Schematic representation of hybrid cluster at University of Split, Faculty of Natural sciences and Mathematics [13]

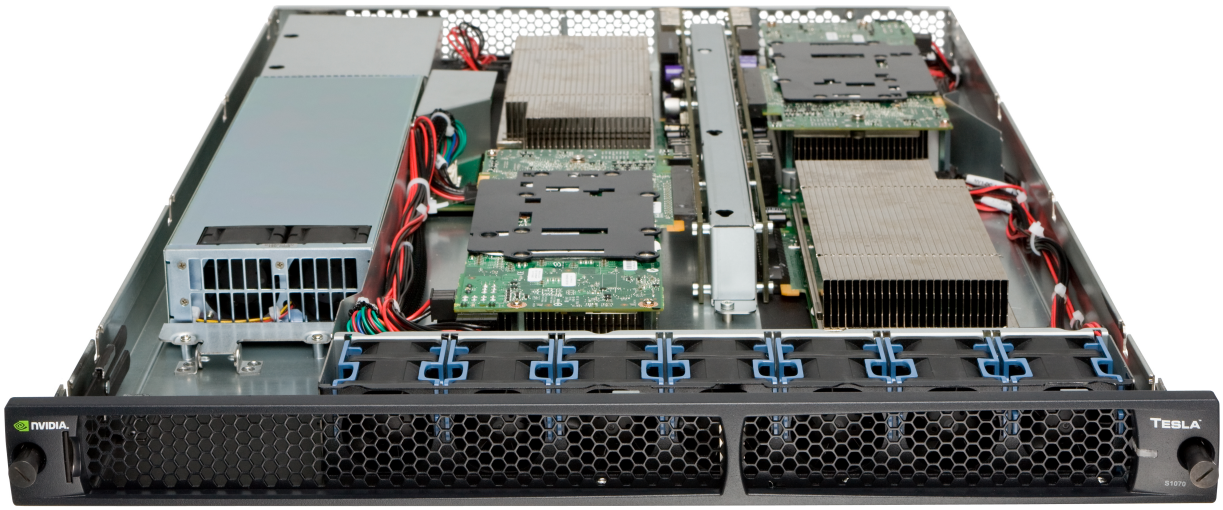
$int\ atomicAdd(int*\ address,\ int\ val)$ used in our simulation. It reads a value from *address*, locks that *address*, adds value *val* to the value in *address* and stores the result back to the same address in memory after which this *address* becomes available to other threads.

3.3 HYBRID cluster

For the purpose of this thesis, HYBRID cluster has been used. The cluster is managed by the Department of Physics, University of Split, Croatia[13]. HYBRID is an experimental cluster that explores High Performance Computing technology. The project consists of an interdisciplinary consortium that uses the cluster for science research, assess the applicability of GPGPU techniques for scientific applications, provide feedback on practical problems with the cluster utilization and demonstrate the overall proof of concept needed for further investment into the technology. HYBRID cluster combines two quad-core processors and NVIDIA TESLA 10 series GPUs as showed in figure 11. The NVIDIA Tesla S1070 Computing System is a 1U rack-mount system with four Tesla T10 computing processors[14]. This system connects to one or two host systems via one or two PCI Express cables. A Host Interface Card (HIC) is used to connect each PCI Express cable to a host. The host interface cards are compatible with both PCI Express 1x and PCI Express 2x systems figure 12.



(a) Tesla S1070 System Architecture, schematic view



(b) Tesla S1070

Figure 12: Tesla S1070 System Architecture

3.4 Parallel Metropolis algorithm using CUDA

Looking at acceptance rule for metropolis algorithm in equation 28, updating decision for spin s_i depends only on its four (in 2D) neighbors while energy contribution of a spin depends only on the nearest neighbors. It allows calculations to be made local and highly parallel by using lattice decompositions of the checkerboard type, figure 13. **Algorithm 1** needs to be modified in such a way that spins are not chosen at random for changing their orientation, but to alter them in a predefined order and to update a large set of spins at the same time. The authors of Ref. [15] worked on strips or columns of the lattice for spin update, similar to a checkerboard decomposition, but referenced setup does not take the hierarchical memory organization into account, so their spin field resides in global memory at all time. The authors of Ref.[16] have done the same lattice decomposition that is used in this thesis with difference in memory mapping and threads execution. This thesis shows trend in HPC, where one has a sequential program, usually thousand of lines of code, that at first needs to be analyzed by looking for a data parallelism parts and then these parts have to be rewritten in parallel.

3.4.1 Single checkboard decomposition

Single checkboard decomposition algorithm,(described in table s**Algorithm 2**) is summarized in Figure 13. The hole grid of spins from global memory are first decomposed in to blocks and mapped to shared memory of each block. Also boundary spins from neighboring block are copied to particular block. Then spin flip condition is posed on, at first even that at odd spins to avoid memory access interference. After this step, after all spin sites are tested for spin flip, values of spins are returned to global memory. This algorithm uses values of spins in shared memory only once while it has to return values to global memory because boundary values of each block belong to neighboring block and they need to be updated. This algorithm updates half of spins at once, while others half are neighboring spins that are used to calculate energy.

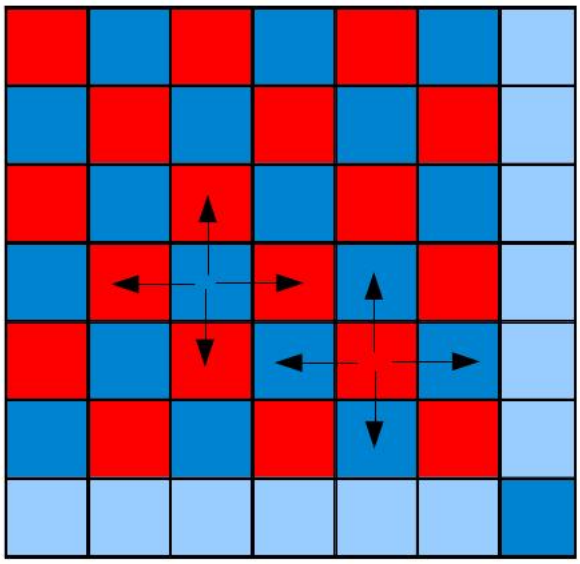


Figure 13: Checkboard decomposition, energy contribution of a single spin depends only on the nearest neighbors, so when uploading "red" spins we can have "blue" spins in shared memory while they do not change and vice versa.

3.4.2 Double checkboard decomposition

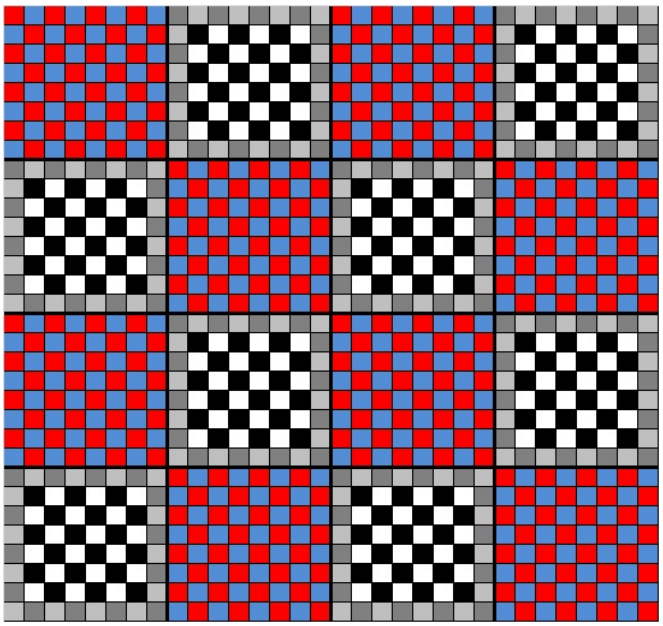
The second algorithm, **Algorithm 3** and figure 14, so called double checkboard decomposition algorithm, uploads half of blocks, even or odd, to shared memory and by doing that it can use values of spins stored in shared memory many times while value of boundary spins of a block do not change because they belong to neighboring block that is not set for execution. Inside particular block execution is same as in single checkboard decomposition. Since much time is consumed for data transfer between global and shared memory, and since access to shared memory is faster than access to global memory, by using shared memory more times in between transferring data, double checkboard decomposition is expected to be faster than single checkboard decomposition even though this algorithm updates only quarter of all soins at once.

Algorithm 2 Single checkboard parallel Metropolis algorithm

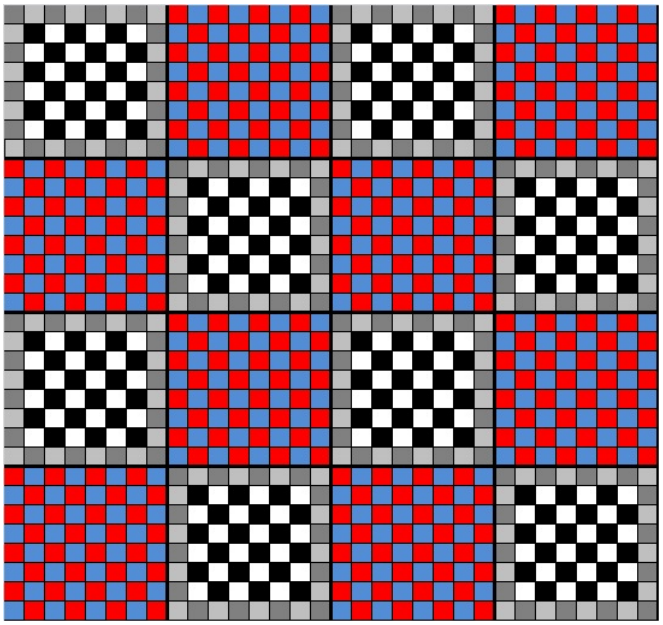
```
shared memory ⇐ global memory
if threadID == even then
  Metropolis update of each thread with appropriate ID
  synchronization point
else
  Metropolis update of each thread with appropriate ID
  synchronization point
end if
global memory ⇐ shared memory
```

Algorithm 3 Double checkboard parallel Metropolis algorithm

```
shared memory ⇐ global memory
if blockID == even then
  for i = 1 → LOCALFLIP do
    if threadID == even then
      Metropolis update of each thread with appropriate ID
      synchronization point
    else
      Metropolis update of each thread with appropriate ID
      synchronization point
    end if
  end for
  global memory ⇐ shared memory
  shared memory ⇐ global memory
else
  for i = 1 → LOCALFLIP do
    if thredID == even then
      Metropolis update of each thread with appropriate ID
      synchronization point
    else
      Metropolis update of each thread with appropriate ID
      synchronization point
    end if
  end for
end if
global memory ⇐ shared memory
```



(a) The first step - odd block



(b) The second step - even block

Figure 14: Double checkboard decomposition

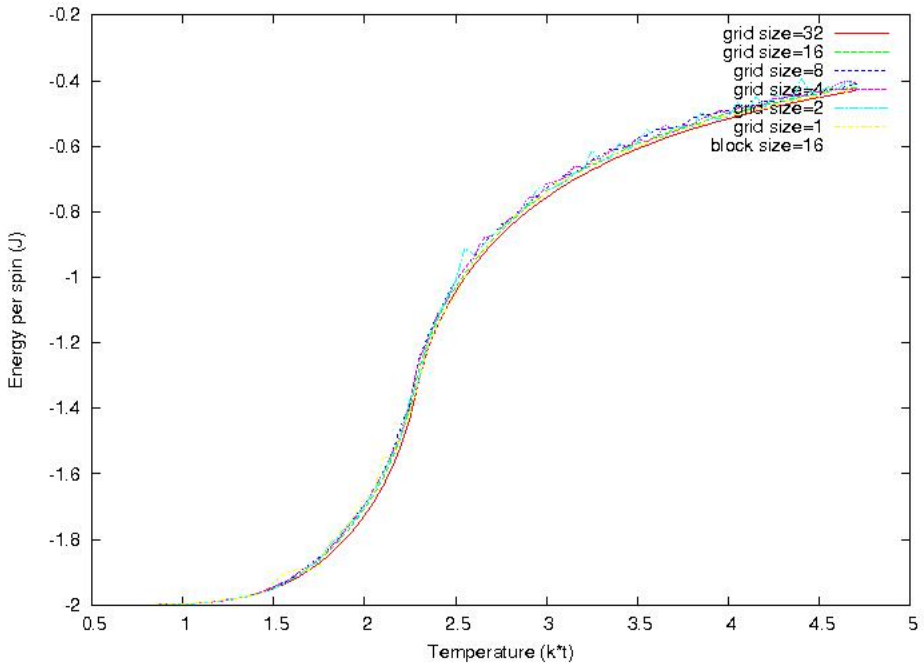


Figure 15: Energy per spin for deferent lattice sizes. Lattice size is $block\_size * grid\_size$ and $block\_size$ is in all cases 16.

4 Results

4.1 Physical properties of Ising Spin model

Figures 15-19 show results for the significant data of an Ising model. Figure 15 shows energy per spin in a temperature range, where Ising 2D model exhibits phase transition. Results show that at low temperatures, lower that transition temperature, energy per spin is minimum and it gets closer to zero above transition temperature. Figure 16 shows specific heat that is connected to energy fluctuation(7), at same temperature range. At temperature of phase transition fluctuations of energy are high as shown by large values of specific heat. It is also important to notice that results show that specific heat is highly dependent on lattice size as mentioned in introduction. Figure 17 shows magnetization per spin which also show phase transition in the same temperature range. At temperatures below transition temperature magnetization per spin is 1, meaning that all spins are align.

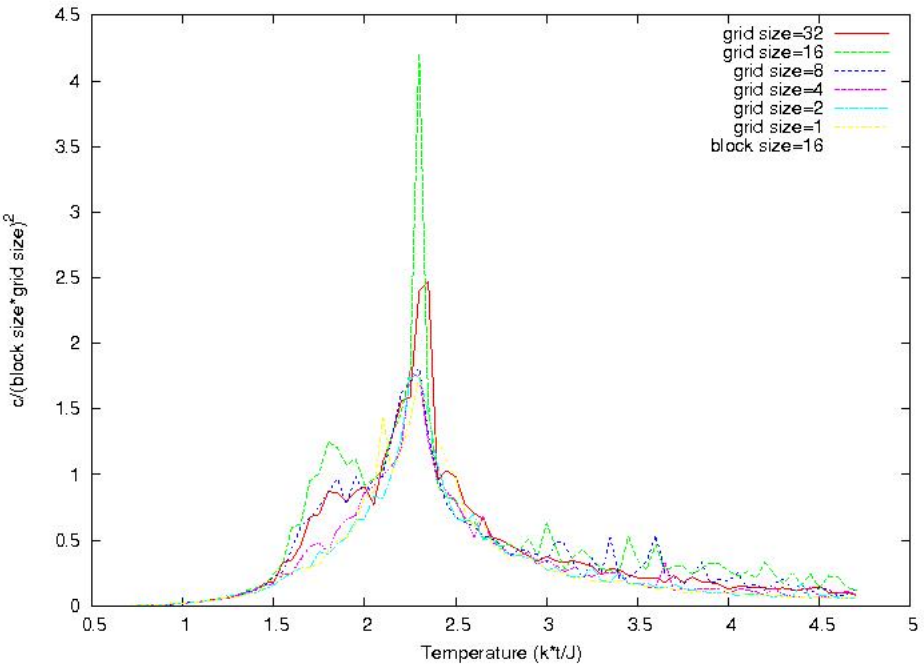


Figure 16: Specific heat per spin for different lattice sizes. Lattice size is $block\_size * grid\_size$ and $block\_size$ is in all cases 16.

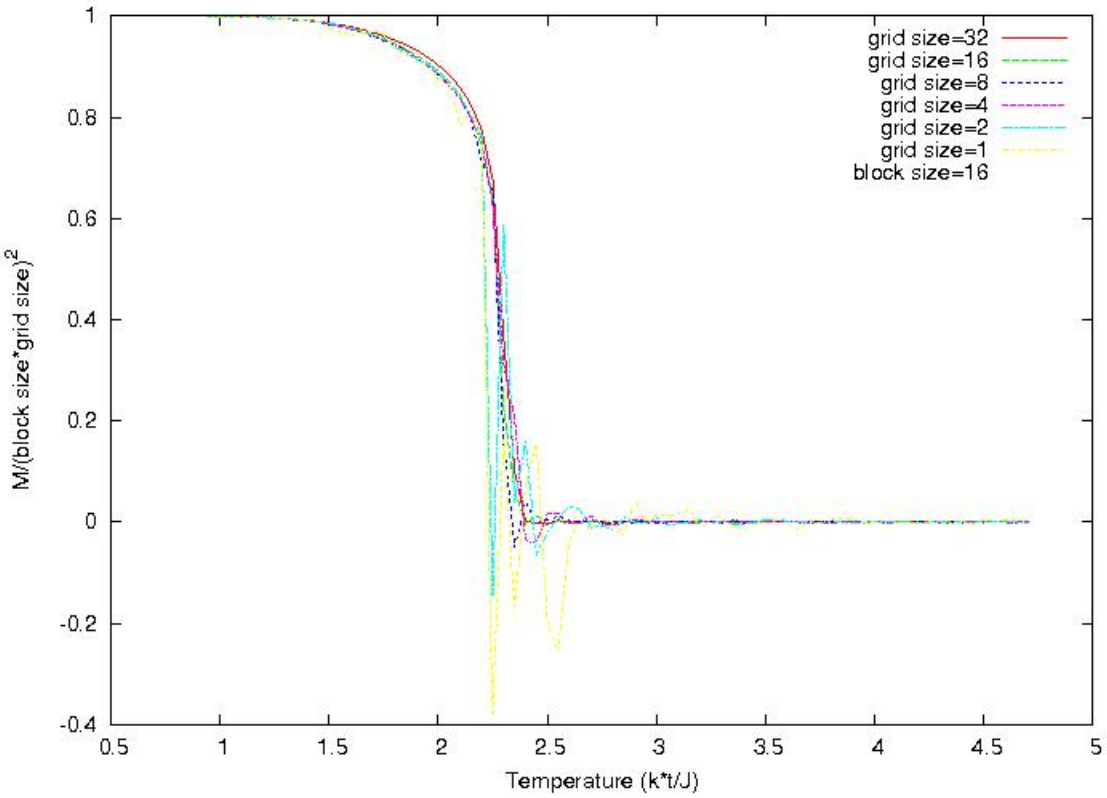


Figure 17: Magnetization per spin for deferent lattice sizes. Lattice size is $block\_size * grid\_size$ and $block\_size$ is in all cases 16.

Above transition temperature magnetization per spin drops to zero, which is typical for random orientation of spins. Figure 18 shows magnetic susceptibility that is connected to magnetization fluctuations(10). Similar to specific heat, magnetic susceptibility is high at transition temperature due to high fluctuations in magnetization. For larger lattice sizes, fluctuations are large. The same holds for values of specific heat. Figure 19 shows absolute value of magnetization per spin and it shows results similar to figure 17, but with the deference of not having negative values for magnetization.

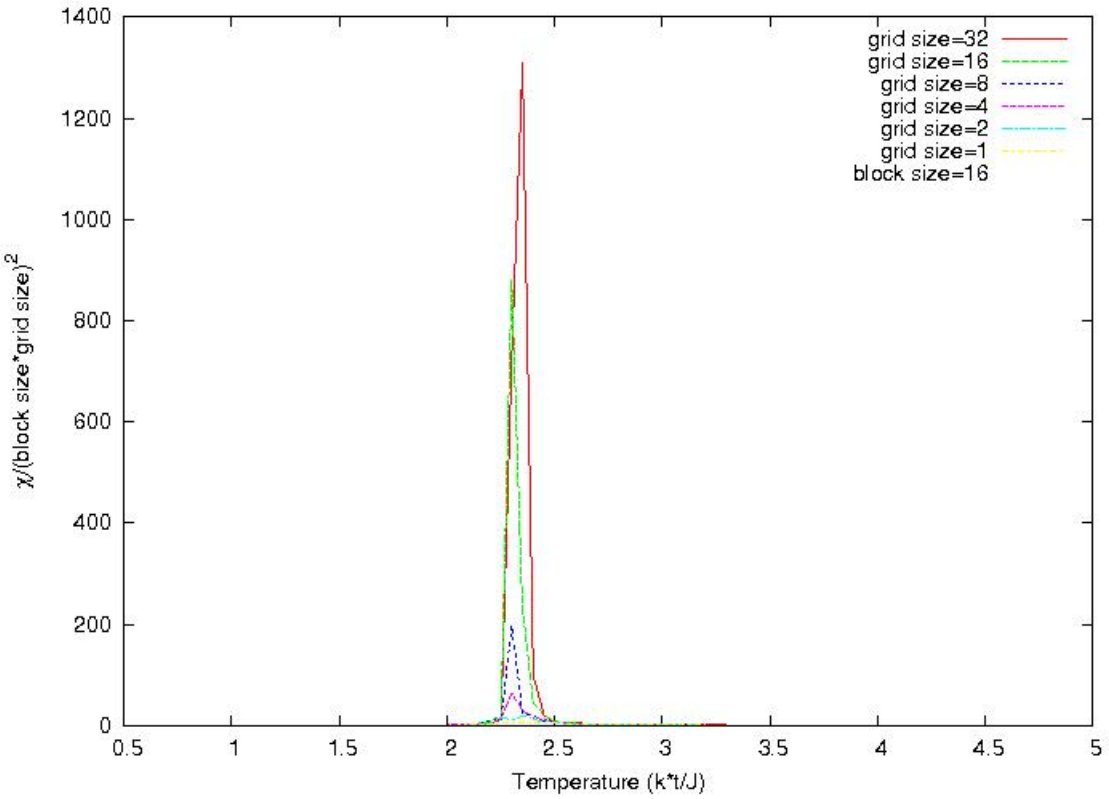


Figure 18: Magnetic susceptibility per spin for deferent lattice sizes. Lattice size is $block\_size * grid\_size$ and $block\_size$ is in all cases 16.

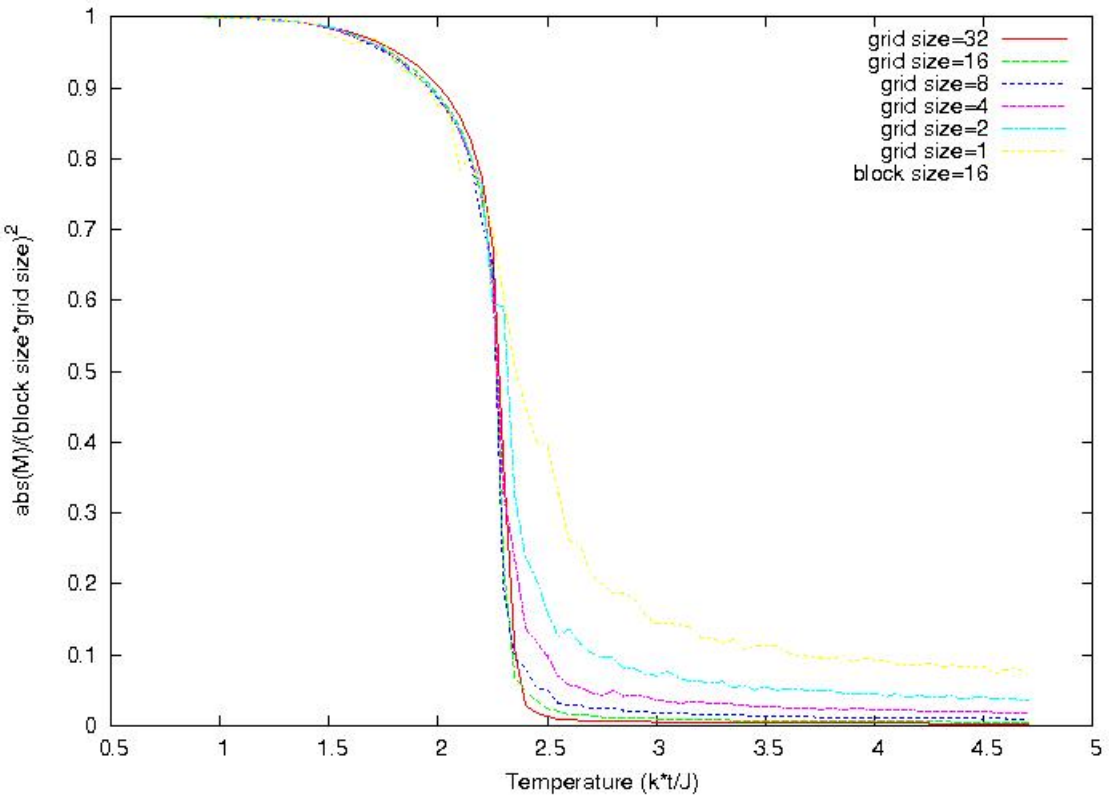


Figure 19: Absolute magnetization per spin for deferent lattice sizes. Lattice size is $block\_size * grid\_size$ and $block\_size$ is in all cases 16.

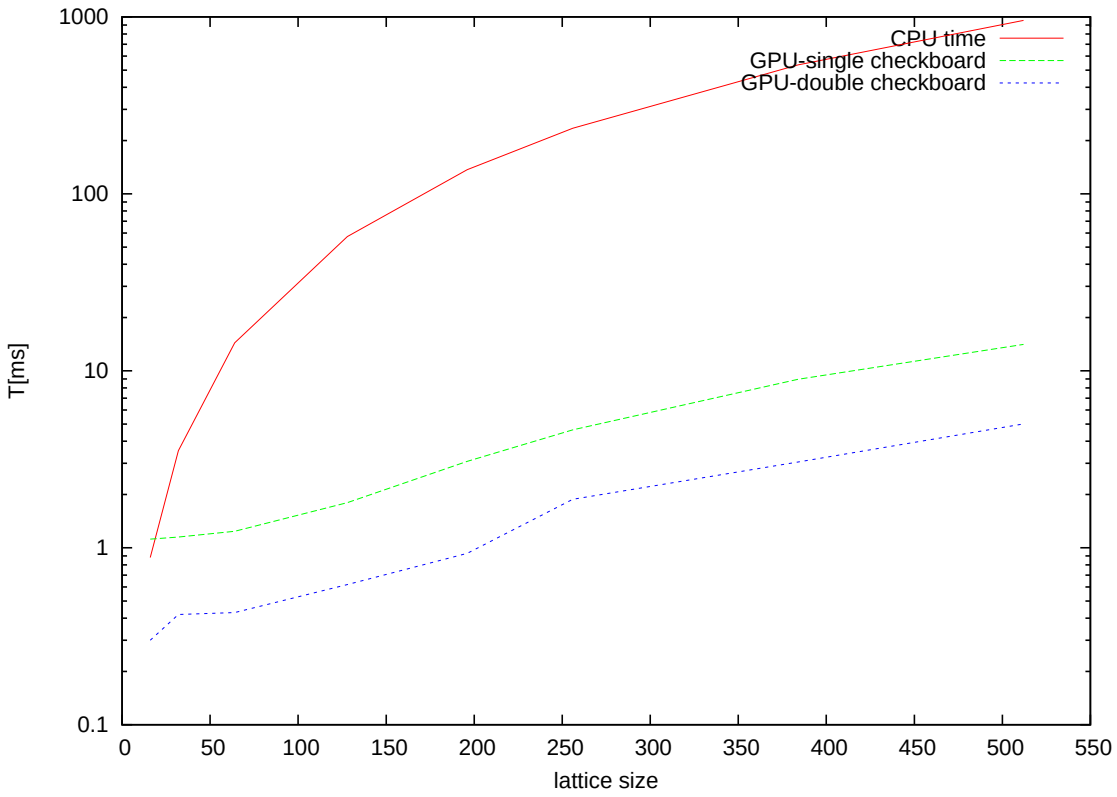


Figure 20: Execution time for CPU and for two GPU codes (block size is 16).

4.2 Advantages of GPU over CPU

Figure 20 is the most important graph for this thesis. It shows execution times for three different programs. Comparison is made between the CPU code and two different GPU codes. CPU execution time is much longer than that of GPU and difference rises with size of the lattice. Figure 21 shows the same data as figure 20, but this time GPU execution time is divided by CPU execution time to get speedup. One of the specific properties of GPU cards is existence of "magic" numbers. For some specific size of grid and block size, performance changes suddenly. The same is shown in Figure 21. As noted in gugure captions, block size is always 16, but grid size changes from 1 to 32. At grid size 16, same as block size, performance drops and thet it rises again.

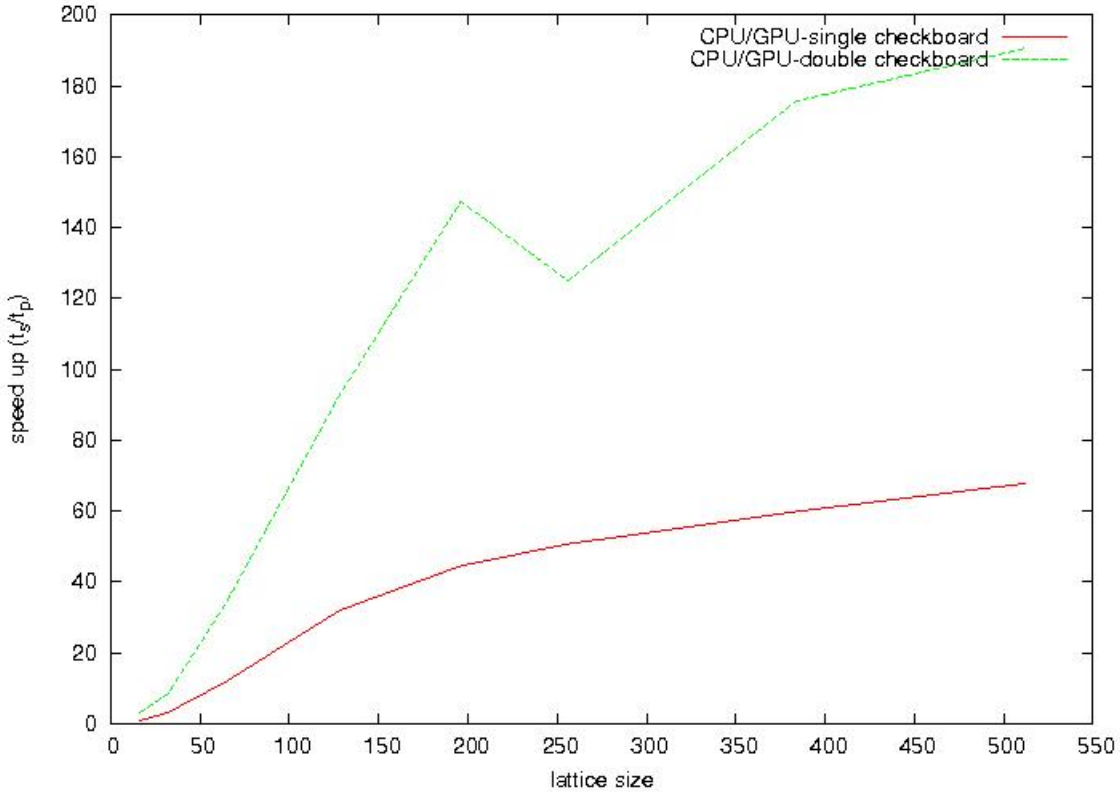


Figure 21: Speedup of GPU codes (block size is 16).

5 Discussion and conclusion

GPU outperforms CPU in simulation of Ising model by factor of about 50 to 200, depending on the lattice size. For larger lattice sizes(more than about 50x50) performance difference grows in favor of GPU. It is in agreement with previous work but it is also a future trend that will use graphic cards for general computation rather than just for graphics. One could easily transform program written for this thesis to perform simulation in other problem where interaction is bounded to nearest neighbors only. Or one could continue to work on the same program by optimizing it even more. For instance, on-chip texture memory has not been used in our cases. More sophisticated mapping of memory from global to shared could be implemented. Another significant conclusion of our work is that we have to become familiar with the structure of CUDA programming architecture. It is minimal extension to standard high level programs like C. Time spent on getting familiar with CUDA eviroment pays back while there is a huge speedup in execution time. It would be interesting to test this program on newer generations of NVIDIA graphics cards, Fermi architecture cards. CUDA architecture is only usable on NVIDIA graphics cards, while another programming model called Open CL is being developed, and it also works on heterogeneous computer architectures, but it is used across different manufactures of graphic cards.

A CUDA code - kernel function

```
__global__ void deviceMetropolis(int *spins, int  *EMD,float
   temp_GPU,int *ranD)
{
    __shared__ int block_spin[block_size+2][block_size+2];
    __shared__ int E[block_size][block_size];

   int it = threadIdx.x;//dimenzija unutar bloka
   int kt = threadIdx.y;
   int i = block_size * blockIdx.x + it;
   int k = block_size * blockIdx.y + kt;
   int Ni = block_size*grid_size;
   int Nk = block_size*grid_size;
   int ik = Ni*k + i;
  float temp = temp_GPU;
  unsigned int ran=ranD[ik];
  __syncthreads();
   block_spin[it+1][kt+1]=spins[ik];
   if (i==0) block_spin[it][kt+1]=spins[ik+(Ni-1)];
   else if (it==0) block_spin[it][kt+1]=spins[ik-1];
   if (i==Ni-1) block_spin[it+2][kt+1]=spins[ik-(Ni-1)];
   else if (it==block_size-1) block_spin[it+2][kt+1]=spins[ik+1];
   if (k==0) block_spin[it+1][kt]=spins[ik+(Nk-1)*Ni];
   else if (kt==0) block_spin[it+1][kt]=spins[ik-Ni];
   if (k==Nk-1) block_spin[it+1][kt+2]=spins[ik-(Nk-1)*Ni];
   else if (kt==block_size-1) block_spin[it+1][kt+2]=spins[ik+Ni
      ];
     float dE;
   __syncthreads();

   if((it+kt+2)\%2==0)
       {  E[it][kt]=-1*block_spin[it+1][kt+1]*
                                (block_spin[it][kt+1]+block_spin[it
                                   +2][kt+1]+
                                block_spin[it+1][kt]+block_spin[it
                                   +1][kt+2]);
        dE=-2.*(float)E[it][kt];
             __syncthreads();
               if (MULT*(*(unsigned int *)(&RAN(ran)))<=__expf((-
                  dE)/temp) )
                 {
                        block_spin[it+1][kt+1]*=-1;
                        E[it][kt]*=-1;
                        }
```

```
        }
__syncthreads();
        if((it+kt+2)\%2!=0)
       {            E[it][kt]=-1*block_spin[it+1][kt+1]*
                              (block_spin[it][kt+1]+block_spin[it
                                 +2][kt+1]+
                               block_spin[it+1][kt]+block_spin[it
                                  +1][kt+2]);
   dE=-2.*(float)E[it][kt];
        __syncthreads();
               if (MULT*(*(unsigned int *)(&RAN(ran)))<=__expf((-
                  dE)/temp) )
             {
             block_spin[it+1][kt+1]*=-1;
            E[it][kt]*=-1;
             }
       }
spins[ik]=block_spin[it+1][kt+1];
  ranD[ik]=ran;

 __syncthreads();
 int nTotalThreads = block_size;

   while(nTotalThreads > 1)
        {
                int halfPoint = (nTotalThreads >> 1);
                if (kt < halfPoint)
                {
                block_spin[it+1][kt+1]+=block_spin[it+1][kt+
                   halfPoint+1];
                E[it][kt] += E[it][kt+halfPoint];
                }
                __syncthreads();
                nTotalThreads = (nTotalThreads >> 1);
        }
__syncthreads();
 int mTotalThreads = block_size;
        while(mTotalThreads > 1)
        {
                int half = (mTotalThreads >> 1);
                if ((it < half)&&(kt==0))
                {
                block_spin[it+1][1]+=block_spin[it+half+1][1];
                E[it][0] += E[it + half][0];
                }
                __syncthreads();
```

```
                mTotalThreads = (mTotalThreads >> 1);
        }
__syncthreads();
if((it==0) & (kt==0)){
 atomicAdd( &EMD[0],E[it][kt]);
  atomicAdd(&EMD[1],block_spin[it+1][kt+1]);
}
 __syncthreads();
}
```

References

[1] E.Ising, *Beitrag zur Theorie des Ferro- und Paramagnetismus* (Thesis, Hamburg) (1924).

[2] C. Stutz, B. Williams, Physics Today **52**, 106 (1999).

[3] L. Osnager, Phys. Rev.,**65**, 117 (1944).

[4] A. E. Ferdinand, M. E. Fisher, Bounded and inhomogeneous Ising models. I. Specific heat anomaly of a finite lattice, Phys. Rev. **185** 832 (1969).

[5] Supercomputing, Portland, OR, Nov., 2009.

[6] M. J. Flynn, *Some computer organizations and their effectiveness* IEEE TRANSACTIONS ON COMPUTERS, **9**, c-21 Sep. (1972)

[7] Lawrence Livermore National Laboratory web page: `https://computing.llnl.gov/tutorials/parallel_comp/`

[8] TOP500 project: `http://www.top500.org`

[9] High performance Linpack Benchmark: `http://www.netlib.org/benchmark/hpl/`

[10] M. Hjorth-Jensen, *Computational Physics*, University of Oslo, (2007).

[11] David B. Kirk, Wen-mei W. Hwu, *Programming Massively Parallel Processors*, Morgan Kaufmann Publishers, (2010).

[12] NVIDIA, *NVIDIA CUDA C Programming Guide*, (Version 3.2).

[13] Department of Physics, University of Split, Croatia, cluster homepage: `http://www.gpuhybrid.org/`

[14] NVIDIA, *Tesla S1070 GPU Computing System* , (SP-04154-001_v02), (2008).

[15] T. Preis, P. Virnau, W. Paul, J. J. Schneider, GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model, J. Comput. Phys. **228** 4468, (2009).

[16] M. Weigel, *Simulating spin models on GPU*, `arXiv:1006.3865v1`

[17] M. Hjorth-Jensen homepage: `http://folk.uio.no/mhjensen/`