Computational Study of Phase Transitions

Mainak Pal, Indian Association for the Cultivation of Science

May 13, 2019

Submitted as Phase-II Project Report for Masters in Physics

Abstract

Metropolis Monte Carlo Simulations of the 2 dimensinal Ising model on a square lattice has been implemented in three different paradigms, a standard CPU serial version, a CPU parallel version with OpenMP and a parallel GPGPU version with CUDA. Performance of these codes were studied by compairing runtimes. OpenMP code was run on an INTEL i5 processor and a XeonPhi machine. The CUDA code was run on an NVIDIA GEFORCE 940MX laptop card. We have found upto x6 speedup in CUDA when considering only metropolis configuration updates. We have studied quantities such as magnetization, susceptibility, specific heat, spin-spin correlation at different lattice sizes as a function of temperature which allow us to find the finite size scaling functions of these quantities and measure the transition temperature with accuracy. An exact transition temperature found from duality transformations of the 2 dimensional Ising model which has also been worked out explicitly.

Keywords: Ising model, Self Duality, Finite Size Scaling, Monte Carlo, OpenMP, CUDA

Contents

1 Introduction **3**

2 Theory and Methods **3**
2.1 Ising Model 3
2.2 Self Duality 4
2.3 Simulating Ising Model 6
2.4 Analysing Monte Carlo Data 9
2.4.1 Thermalization 9
2.4.2 Symmetry Breaking 9
2.4.3 Thermodynamic Averages and Errorbars 9
2.5 Finite Size Scaling 10
2.6 Parallelisation Schemes 12
2.6.1 CPU 12
2.6.2 GPU 12
2.6.3 Runtime Comparisons 12

3 Results **14**

4 Codes **20**

5 Acknowledgements **27**

1 Introduction

This is the last phase of a two phase project. In the *Phase-I* of the project we had studied different magnetic phases of the Hubbard model with Hartree-Fock self consistent mean field approximation on a 2 dimensional bipartite lattice. We obtained a qualitative phase digram of the model which had ferromagnetic, parmagnetic and antiferromagnetic phases. This is a continuation of the previous work where we continue to look at phase transitions but now in a more quantitative manner. Now we want to locate the transition points precisely rather than having only a qualitative phase diagram.

In order to deal with phase transitions in a more accurate computational manner we have decided to simulate the two dimensional Ising model on a square lattice. In doing so we have a simpler model with a second order phase transition at a finite temperature and this allows us to focus on techniques related to extracting information about the transition from the simulathetion. We have also studied finite size scaling effects of the system and how finite size influences results such as the transition temperature.

In the first section we give a very short introduction to the Ising model it's partition function and Hamiltonian and discuss the motivation for doing numerical simulation even though an exact solution exists.

Then we describe the method of duality transformation which allows us, by a series of variable transformations, to rewrite the partition function in terms of dual spin variables which gives us back again a two dimensional Ising model but at a different temperature. We can then use the fact that the model at hand has only one transition temperature and exploit the self duality of the model to completely determine this transition temperature.

Next we give a brief introduction the the Monte Carlo method and how it can be applied to simulate our system. We have used Metropolis local update algorithms. We perform a Monte Carlo experiment and describe how to analyze the data in order to extract information about physical observabes such as magnetization, susceptibility, energy, specific heat and correlation functions. Quoting error bars are important in numerical simualtions. For primary observables such as magnetization or energy this is straightforward but for secondary quantities more work has to be done. We have described how to do this using the Blocking Method and the Jackknife method.

After that we describe the theory of finite size scaling which describes how the infinite volume results need to be modified in the finite lattice size region. This is very helpful when performing numerical simulations because in any simulation the lattice size has to be finite however large it may be.

To efficiently simulate large lattice sizes we have parallelised the simulations on CPU using OpenMP and on GPGPU using CUDA. This itself is an interesting area and we have studied how the performance of parallelisation changes with respect to workload and available threads by verifying Amdahl's Law and Gustafson's Law.

2 Theory and Methods

2.1 Ising Model

In this model we have spin variables $\{S_i = \pm 1\}$ sitting at each lattice site $i \in L$ and they interact with their nearest neighbours. L is the entire lattice containing the spin variables. The model is described

by the partition function

$$\mathcal{Z} = \sum_{\{S\}} e^{-\beta \mathcal{H}} \tag{2.1}$$

With the Hamiltonian given by

$$\mathcal{H} = -J \sum_{\langle ij \rangle} S_i S_j \tag{2.2}$$

Here $\{S\}$ denotes sum over all spin configurations and $\langle ij \rangle$ denotes that i, j are nearest neighbours. When $J > 0$ we have a ferromagnetic (ordered) ground state and $J < 0$ means we have an paraagnetic (disordered) ground state below the critical temperature. The exact solution of this model is given by Lars Onsager but the solution is particular to the Ising model only and cannot be generalized to other models. We want to look at more general methods such as numerical simulations which can be applied to other models as well. We can then make use of the fact that there is an exact solution to eliminate any kind of errors or crosscheck the codes with the exact solution. Before discussing and implementing the Metropolis algorithm we will have a look at a nice analytical procedure called duality tranformation by which we can compute the transition temperature exactly without going into the exact solution of Onsager.

2.2 Self Duality

Duality transformation are powerful analytical methods which can transform one theory in the high temperature region to another theory in low temperature region. In field theory this often becomes transformation of a strong coupling theory to a weak coupling theory. It may happen that by doing such transformation we can reduce difficult problems in one regime to easier problems in other regimes. This technique can also be applied to Ising model and it will turns out that the 2 dimensional Ising model is dual to itself. Meaning that an Ising model problem at inverse temperature β becomes under this transformation another Ising model problem at a different inverse temperature $\tilde{\beta}$ related to β in some fashion. We will find the exact realtion as we proceed. In the next few steps we will demonstrate the duality transformation which is basically rewriting the partition function of the system in a new set of variables.[1]

Again the partition function of the 2 dimensional Ising model is given by

$$\mathcal{Z} = \sum_{\{S\}} \prod_{\langle ij \rangle} e^{\beta S_i S_j} \tag{2.3}$$

We have absorbed the coupling constant J into β. Now the product $S_i S_j$ in the exponential can only have values ± 1 this makes it possible to rewrite the partition function in the following format

$$\mathcal{Z} = \sum_{\{S\}} \prod_{\langle ij \rangle} \sum_{\{k\}=0}^{1} C_{k_{\mu;i}}(\beta) (S_i S_j)^{k_{\mu;i}} \tag{2.4}$$

Where

$$C_0(\beta) = \cosh \beta \quad \text{and} \quad C_1(\beta) = \sinh \beta \tag{2.5}$$

And the $k_{\mu;i}$ are so called *link variables* which live on the link or the bond between two nearest neighbour sites. We make this identification because the term in the Hamiltonian $S_i S_j$ comes from the interaction between the sites at position i and j and can be thought of like a bond or a link between them.

Now grouping all factors of S_i associated with a given site and rearranging the terms we get

$$\begin{aligned}\mathcal{Z} &= \sum_{\{S\}}\sum_{\{k\}}\prod_{l} C_{k_\mu}(\beta)\prod_i (S_i)^{\sum_i k_\mu} \\ &= \sum_{\{k\}}\prod_{l} C_{k_\mu}(\beta)\prod_i \sum_{s=\pm 1} (S_i)^{\sum_i k_\mu} \\ &= \sum_{\{k\}}\prod_{l} C_{k_\mu}(\beta)\prod_i 2\delta_2\Big(\sum_i k_\mu\Big)\end{aligned} \tag{2.6}$$

Here $\sum_i k_\mu$ means the sum of the k values of all the 4 links connected to the site i and $\delta_2(x) = \delta(x \bmod 2)$. Now we wish chose such a representation that the delta function in the last expression becomes manifest. For doing this we construct a *dual lattice* from the original one by placing a vertex of the new lattice in the center of each square of the original lattice. In this method we have constructed a new square lattice of same spacing as the original one but one which is shifted by half a lattice spacing in both directions. We define dual lattice variables $\{\sigma_i\}$ living on the dual lattice by the following relation which as required removes the delta function automatically.

$$k_{\mu,i} = \frac{1}{2}(1 - \sigma_i\sigma_{i-\hat{\nu}}) \qquad \mu \neq \nu \tag{2.7}$$

We now check our proposal by noticing that the argument of the delta function

$$\sum_i k_{\mu,i} = 2 - \frac{1}{2}(\sigma_1\sigma_2 + \sigma_2\sigma_3 + \sigma_3\sigma_4 + \sigma_4\sigma_1) \tag{2.8}$$

Would be even for any configuration of the variables $\{\sigma_i = \pm 1\}$ and we can drop the delta functions. Doing so and changing to dual lattice variables we can rewrite the partition function as

$$\mathcal{Z} = \frac{1}{2}2^N \sum_{\{\sigma\}}\prod_{l_d} C_{(1-\sigma_i\sigma_j)/2}(\beta) \tag{2.9}$$

Where N is the numnber of lattice sites and the product over l_d signifies product over all links existing in the dual lattice. Extra factor $1/2$ comes from the fact that our summation has changed from $\{k_\mu\}$ to $\{\sigma_i\}$ but $\{\sigma_i\}$, $\{-\sigma_i\}$ both give the same result. As the only allowed values of k_μ for the Ising model are 0,1 we can easily observe that

$$\begin{aligned}C_k(\beta) &= \cosh(\beta)\big[1 + k(\tanh(\beta) - 1)\big] \\ &= \cosh(\beta)\exp\Big[k\ln(\tanh(\beta))\Big] \\ &= \big[\cosh(\beta)\sinh(\beta)\big]^{1/2}\exp\Big[-\sigma_i\sigma_j\frac{1}{2}\ln(\tanh(\beta))\Big]\end{aligned} \tag{2.10}$$

This means we can further write the partition function as

$$\begin{aligned}\mathcal{Z} &= \frac{1}{2}\big[2\cosh(\beta)\sinh(\beta)\big]^N \sum_{\{\sigma\}}\exp\Big[-\frac{1}{2}\ln(\tanh(\beta))\sum_{\langle ij\rangle}\sigma_i\sigma_j\Big] \\ &= \frac{1}{2}\big[\sinh(2\tilde{\beta})\big]^{-N}\sum_{\{\sigma\}}\exp\Big[\tilde{\beta}\sum_{\langle ij\rangle}\sigma_i\sigma_j\Big]\end{aligned} \tag{2.11}$$

Where $\tilde{\beta} = -\frac{1}{2}\ln(\tanh(\beta))$ is the exact relation between inverse temperature and dual inverse temperature. Now except for an overall factor the above partition function is exactly similar to that

of the standard two dimensional Ising model. This implies that we get back the model we started with under duality transformation, this nice property is called self duality. Again we note that $\tilde{\beta}$ is a monotically decreasing function of β so that high temperature regimes are mapped to low temperature regimes and vice versa.

This self duality is very much similar to what happens when we do Fourier transformation of the solutions of the Quantum Harmonic Oscillator to go to momentum space representation. There if we have a highly localized wavefunction in position space then in momentum space it becomes highly spread out in momentum space and vice versa.

Let us now discuss the most important implication of this self duality property for our purpose. The free energy of the system is defined as

$$F(\beta) = \lim_{N\to\infty} \frac{1}{N} \ln \mathcal{Z}(\beta) \tag{2.12}$$

Again we have just shown that

$$\mathcal{Z} = \sum_{\{S\}} \prod_{\langle ij \rangle} e^{\beta S_i S_j} = \frac{1}{2} \big[\sinh (2\tilde{\beta}) \big]^{-N} \sum_{\{\sigma\}} \exp \Big[\tilde{\beta} \sum_{\langle ij \rangle} \sigma_i \sigma_j \Big] \tag{2.13}$$

Which then for free energy implies that

$$F(\beta) = -\sinh(2\tilde{\beta}) + F(\tilde{\beta}) \tag{2.14}$$

Now if we suppose to know that the 2 dimensional Ising model has only one transition point i.e. the free energy $F(\beta)$ has only point of nonanalyticity and $\tilde{\beta}(\beta)$ is a monotonic function of β then it *must* be the case that the point of nonanalyticity or the phase transition point is given by

$$\beta_c = -\frac{1}{2} \ln(\tanh(\beta_c)) = \tilde{\beta}_c \tag{2.15}$$

This transcedental equation completely determine the exact transition point of the 2 dimensional Ising model without going into the mathematical difficulties of the exact solution presented by Onsagar. Solving this transcendental equation gives us the transition temperature. Let us expand the above equation

$$e^{-2\beta_c} = \tanh(\beta_c) = \frac{1 - e^{-2\beta_c}}{1 + e^{-2\beta_c}} \tag{2.16}$$

From which we get

$$e^{-2\beta_c} = \sqrt{2} - 1 \implies \beta_c = -\frac{1}{2} \ln(\sqrt{2} - 1) \tag{2.17}$$

$$T_c = \frac{2}{\ln(\sqrt{2} + 1)} = 2.2691853.. \tag{2.18}$$

2.3 Simulating Ising Model

Let us discuss how to simulate the 2 dimensional Ising model on a square lattice using Metropolis monte calro algorithm. The procedure is as follows. [2]

- Define a lattice size n, temperature of the system T. Then create an integer valued matrix of order n by n whose values will represent the spin configuration of the Ising model.

- Initialize the configuration by assigning values to the spin. If all the values are taken same either all $+1$ or all -1 then it is called a *cold start*. If the initial values are assigned randomly then it is called a *hot start*.

  Both of these starts are equally valid. In fact we will show that starting from any initial configuration we obtain the same physical quantities after a sufficient amount of simulation time has passed. This happens because the system needs time to *forget* the history of its initial condition and this process is called *thermalization*.

- We then apply the Metropolis Algorithm to update the spin configuration a large number of times untill it has thermalized. After that we do another set of metropolis updates of the lattice to get sample spin configurations which follow the Boltzman distribution.

- From the set of configurations after thermalization we can extract information about any physical quantity we desire such as magnetization, energy, specific heat, susceptibility, correlation function, static structure factor and any other quantity we are interested in.

  Let us now discuss explain the metropolis algorithm

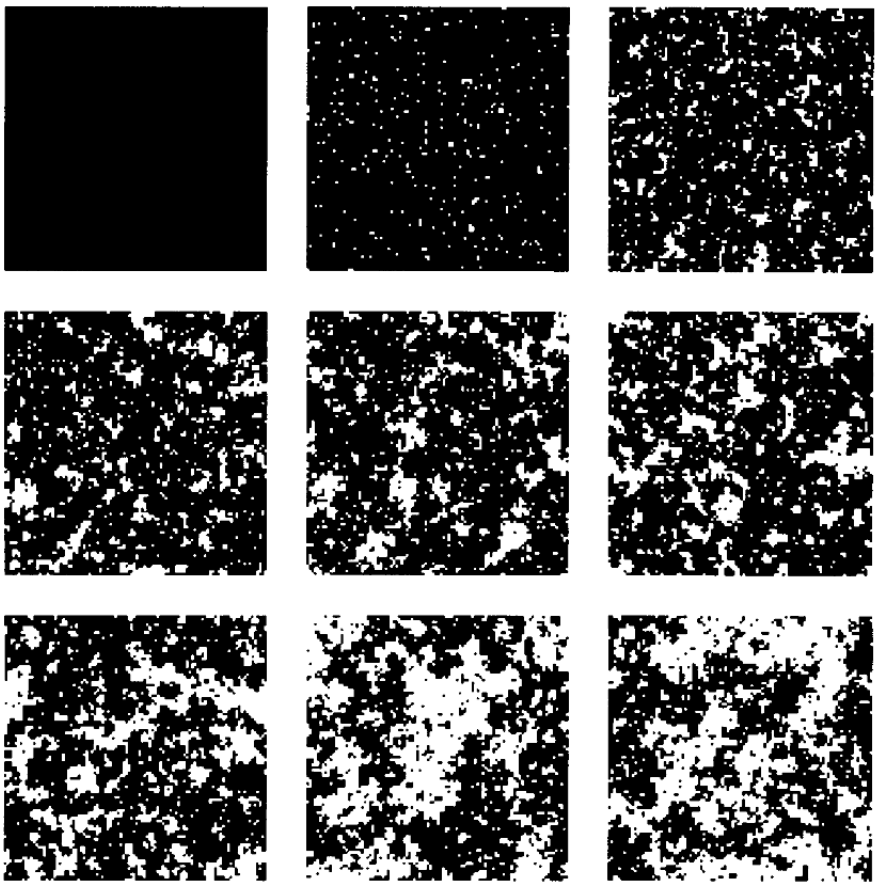
  1. Start with a particular initial condition either hot or cold
  2. Select a lattice site (x, y) either randomly or systematically
  3. Ask what is the energy cost ΔE if we were to flip this spin keeping all the other spins in the lattice untouched. This energy cost would be given by

$$\Delta E_{xy} = S(x,y)\Big[S(x+1,y) + S(x-1,y) + S(x,y+1) + S(x,y-1)\Big] \tag{2.19}$$

  4. If $\Delta E_{xy} \leq 0$ then we accept the flip and update the configuration.
  5. If $\Delta E_{xy} > 0$ then we accept the flip with a probability $e^{-\beta \Delta E_{xy}}$
  6. Repeat steps 3-5 for all the lattice sites. This is straightforward if we are looping over the lattice indices otherwise. If we are selecting sites randomly then we do this n^2 number of times for a $n \times n$ lattice, so that each site on average can be updated once.
  7. Steps 1-6 is called a single monte carlo sweep. We need this perform this entire procedure on the lattice several times to achieve thermalization after which we again perform large number of monte carlo sweeps to obtain the dataset for measuring physical observables.

- In order to avoid finite size effents we work with periodic boundary conditions which makes the 2D lattice equivalent to a torus. For periodic boundary conditions we need to give special care to find the neigbours of a given site (x, y) if it is at some edge of the lattice. They are found using the following simple functions for a $n \times n$ lattice.

  1. Position of neighbour on right $\big(mod(x, n) + 1, y\big)$
  2. Position of neighbour on left $\big(mod(x - 2 + n, n) + 1, y\big)$
  3. Position of neighbour upwards $\big(x, mod(y, n) + 1\big)$
  4. Position of neighbour downwards $\big(x, mod(y - 2 + n, n) + 1\big)$



2.4 Analysing Monte Carlo Data

2.4.1 Thermalization

We first need to make sure that the system has actually thermalized and the equilibrium properties are independent of the initial configuration it started from i.e. whether hot start or cold start it would give us the same thermodynamic averages. This is demonstrated later in the Results section.

2.4.2 Symmetry Breaking

One interesting thing to notice is the direction of magnetization below the critical temperature. In actual experiments due the very small magnetic fields (such as that of Earth) which exist everywhere and cannot be exactly cancelled out by experimental setup. So there is a preferred direction of magnetization in the system and below T_c the direction of magnetization is decided by this preffered direction. In simulations we don't have these external fields and the system has equal probability to land up in all up or all down states below T_c. If we have a small external magnetic field even if its is infinitesimal, the up/down symmertry is broken and the spins will allign the direction of this external field.

We also found that if we continue to do metropolis updates after the system has reached a state with $\langle m \rangle = 1$ it can evolve into a $\langle m \rangle = -1$ configuration in a finite amount of time. The reason for this is that there is the system has a finite size and there is a finite temperature and hence non-zero fluctuations in the system. So the time required in going from all up to all down state depends heavily on the temperature T and the system size n of the system. Increasing the temperature shortens this flip time and increasing the system size increases the flipping time exponentially. This implies that with a thermodynamically large system size this flipping would take infinite time.

2.4.3 Thermodynamic Averages and Errorbars

After the system has thermalized we can start measuring various physical quantities. Let us start with the simplest one - magnetization. If we want to observe the magnetization we should observe $\langle |m| \rangle$ as this is going to take care of the sign and just show how much magnetization the system has.

Thermodynamic average of any quantity A in statistical mechanics is given as

$$\langle A \rangle = \sum_{\{\sigma_i\} \in \Omega} \frac{1}{\mathcal{Z}} e^{-\beta E(\{\sigma_i\})} A(\{\sigma_i\}) \tag{2.20}$$

As we have used the metropolis algorithm to do importance sampling with the Boltzman factor $e^{-\beta E(\{\sigma_i\})}$ already the monte carlo time series data we have for any quantity is already distributed with the Boltzman weight. So we only need to take simple average over our time series dataset after thermalization to obtain the correct average value. So we have

$$\langle A \rangle = \frac{1}{t_M - t_0} \sum_{i=1}^{M} A(t_i) \tag{2.21}$$

Where t_0 is the time taken for the system to thermalize and $A(t_i)$ are the values of the quantity A at time step t_i. It is also important to know the accuracy of our simulation by getting an errorbar on the mean value $\langle A \rangle$. There are two types of sources here. One is statistical error which originates due the inherent randomness of the monte carlo method. If one does a histogram with the values $A(t_i)$ one would find that it always has some spread about the mean value $\langle A \rangle$. So what we say is that the error bar in measuring $\langle A \rangle$ is just the standard deviation of the dataset. In this sense taking Monte Carlo data is very much like performing actual experiments and obtaining data.

For a 2 dimensional Ising model having n^2 lattice sites we can define magnetization and energy per site respectively as

$$m = \frac{1}{n^2} \sum_{i \in L} S(i) \tag{2.22}$$

$$E = \frac{1}{n^2} \sum_{\langle ij \rangle} -JS(i)S(j) \tag{2.23}$$

These are primary observables in a typical Ising model monte carlo experiment. We measure these quantities after every monte carlo sweep and take their averages to find $\langle |m| \rangle$ or $\langle m^2 \rangle$.From these primary quantities we can construct secondary quantities. A single observation these quantities require an entire monte carlo dataset. Two of the most important quantities of this kind are magnetic susceptibility and specific heat. We can define them as

$$\chi = \frac{1}{T} \Big(\langle m^2 \rangle - \langle |m| \rangle^2 \Big) \tag{2.24}$$

$$C_v = \frac{1}{T^2} \Big(\langle E^2 \rangle - \langle E \rangle^2 \Big) \tag{2.25}$$

As a single observation of χ or C_v takes an entire dataset, finding their errorbars are slighty more complicated than those of primary quantities. We have used $\langle |m| \rangle$ instead of $\langle m \rangle$ so that flipping of the magnetization state would not render average value of an actual magnetized state zero. There are 3 main methods to obtain errorbars on secondary quantities.

- Blocking method

  In this method we divide the entire dataset into several sections (say B number of them) which we call bins. We then compute the secondary quantities such as susceptibility or specific heat for all the datapoints inside the individual bins. Doing this for all the bins gives us a mean and a distribution of the quanitity measured.

- Jackknife method

  Jackknife method is quite simillar to blocking method. Here also we divide the dataset into several bins. But when taking avreages for the i^{th} bin we take data from all the bins except the i^{th} one find secondary quantities. We then do the same for all bins and again we have a mean and a standard distribution which gives us an errorbar.

  Monte Carlo simulations depend highly on the quality of random number generators used. These generators are always pseudo random and hence there is always some small amount of correlation between the numbers generated. The methods discussed above help in reducing the correlations in the dataset. The measure for this correlation for a quantity is given by the corresponding autocorrelation function.

2.5 Finite Size Scaling

An obvious disadvantage of performing numerical simulations is that we always have to work with finite lattice sizes however large they may be. This has its effects in the results of the simulation. So if try to compare these with theoritical results which are usually derived for infinite system sizes there will be discrepancies small or large depending on the case. This discrepancies are more dominant in critical phenomenon. We can understand this with a simple example. Let us think of the correlation length of the system. At the transition point the correlation length diverges which is a measure of how many sites are correlated with one another. If we are working with a finite lattice size then our correlation length cannot be infinite but is limited by the size of the system we are simulating. This

effects the critical properties of the system. So there are two length scales at play here - the system size and the correlation length. Let us consider a quantity Q which exhibits a power-law divergent behaviour near T_c. [3]

$$Q \sim |t|^{-\kappa} \tag{2.26}$$

Whereas the correlation length of the system near criticality follows a similar divergent behaviour in the form

$$\xi \sim |t|^{-\nu} \tag{2.27}$$

These two relations together imply

$$Q \sim \xi^{\kappa/\nu} \tag{2.28}$$

This behaviour holds good in the regime $\xi \ll L$ where L is the lattice size. But when $\xi \sim L$ this behaviour can no longer continue as the lattice is finite. The maximum value this quantity can have is

$$Q_{\max} \sim L^{\kappa/\nu} \tag{2.29}$$

Similarly the reduced temperature scales as

$$\left|t_{\max}(L)\right| \sim L^{-1/\nu} \tag{2.30}$$

These scaling laws from a more general scaling hypothesis which says that any singular quantity should scale as

$$Q(t, L) = L^{\sigma} f(\xi/L) \tag{2.31}$$

Using the relation $\xi \sim \left|t\right|^{-1/\nu}$ we can express this as

$$Q(t, L) = L^{\sigma} g(tL^{1/\nu}) \tag{2.32}$$

This scaling law holds both above and below the transition temperature $t > 0$ and $t < 0$. Exactly at the transition temperature we recover

$$Q(0, L) = L^{\sigma} \tag{2.33}$$

To relate σ to standard critical exponents one can use the fact that for fixed t close to zero, as the system grows the behaviour for any $t \neq 0$ eventually has to be given by $Q(t, L \to \infty) \sim |t|^{-\kappa}$. To obtain this from the scaling function $g(x)$ it must have an assymptotic behaviour of the kind

$$g(x) \sim x^{-\kappa} \qquad \text{for } x \to \infty \tag{2.34}$$

And for the size dependence to cancel out one must have $\sigma = \kappa/\nu$. Therefore

$$Q(t, L) = L^{\kappa/\nu} g(tL^{1/\nu}) \tag{2.35}$$

To extract the scaling function from numerical data we define the following

$$y(L) = Q(t, L)L^{-\kappa/\nu} \quad \text{and} \quad x(L) = tL^{1/\nu} \tag{2.36}$$

And then plot $y(L)$ versus $x(L)$ for different system sizes. If this hypothesis is correct we will find that the data for different lattice sizes at different temperatures will fall onto a single curve. This is called *data collapse*.

2.6 Parallelisation Schemes

To simulate the Ising model we are using metropolis Monte Carlo algorithm. What we need to realise is that the process of performing Metropolis updates of the lattice is highly parallel. This is because of the nearest neighbour nature of the model. The process of updating a spin at a single site depends only on it's nearest neighbour spins. So we can construct an imaginary bipartite lattice having two sublattices $\mathcal{A}$ and $\mathcal{B}$. Spins in $\mathcal{A}$ can only interact with spins in $\mathcal{B}$ which are it's nearest neighbours. This means that we can update all the spins which are in the sublattice $\mathcal{A}$ without causing any conflict. Same holds for $\mathcal{B}$ as well. If we want to perform simultaneous updates of this kind we need to use parallel programing models rather than the serial ones. We have implemented two different kinds of parallelisation one on the CPU with OpenMP and one on the GPGPU with CUDA. They both have their advantages and disadvantages which we now list below.

2.6.1 CPU

Advantages

- The most important ingredient in performing monte carlo simulations are random number generators. If the numbers are correlated or biased then the simulations have systematic errors. Most random number generators are written for the CPU and give very high quality random numbers which facilitate the simulations. We have used RANLUX generators in our codes.
- For CPU parallelisation we have used OpenMP which is a directive based. This means that we do not need to consider any technical details of how to create threads and how to synchronize them and so on. The way OpenMP works is that we write a directive before and after the portion we want to be parallelised. We can specify cetain parameters such as number of threads and decide which variables will be shared and which will be private. The compiler automatically generates parallel code for that part of the program using multiple threads.

Disadvantages

- Normal laptop CPU's have high speed processors nowadays but the thread numbers available are quite low to get significant speedup using OpenMP. Specialized hardware such as many core processors with smaller clock speeds but with far more threads are suited for this purpose. These devices are not available to all OpenMP users when compared. So when running the codes from a laptop or a desktop the advantage is more if we use the CUDA platform as most systems have a respectable graphics card.

2.6.2 GPU

Advantages

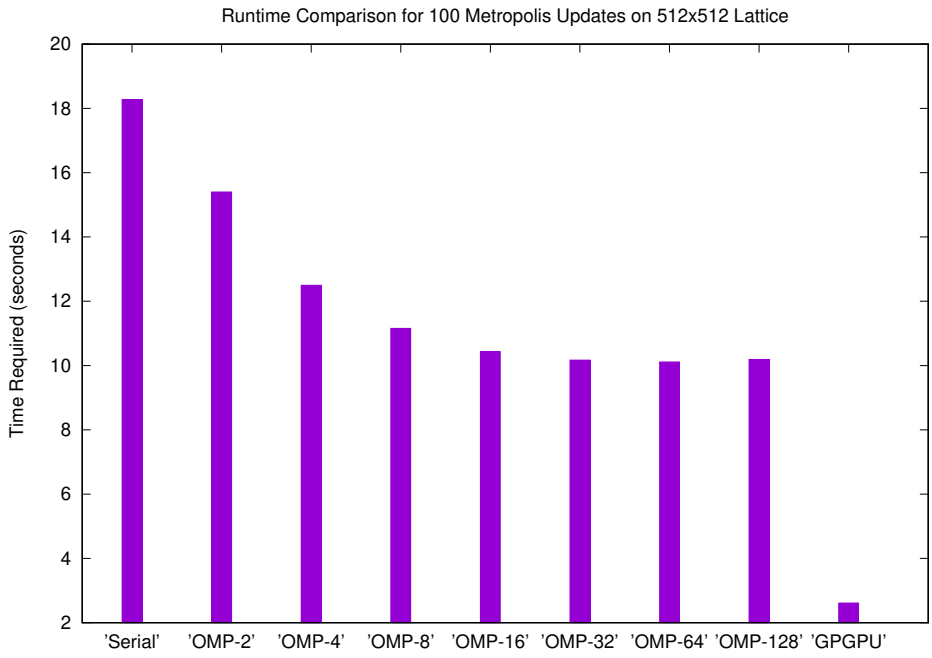
- They have large number of threads when compared to CPU it is better at doing large number of simple tasks such as flipping spins.

Disadvantages

- All the high quality random number generators available are sequential in nature. Although we found that the in built random number generator in CUDA called CURAND was good enough for Ising model it may not be the same situation for more complicated monte carlo simulaitons.
- Unlike OpenMP which generates the codes to be run on parallel automatically, for CUDA we have to write the kernels which are subroutines to be executed on the GPGPU parallaly.

2.6.3 Runtime Comparisons

In the following we will compare performance of the metropolis update algorithm in different paradigms.


Runtime Comparison for 100 Metropolis Updates on 512x512 Lattice
Time Required (seconds)
20
18
16
14
12
10
8
6
4
2
'Serial'
'OMP-2'
'OMP-4'
'OMP-8'
'OMP-16'
'OMP-32'
'OMP-64'
'OMP-128'
'GPGPU'

3 Results

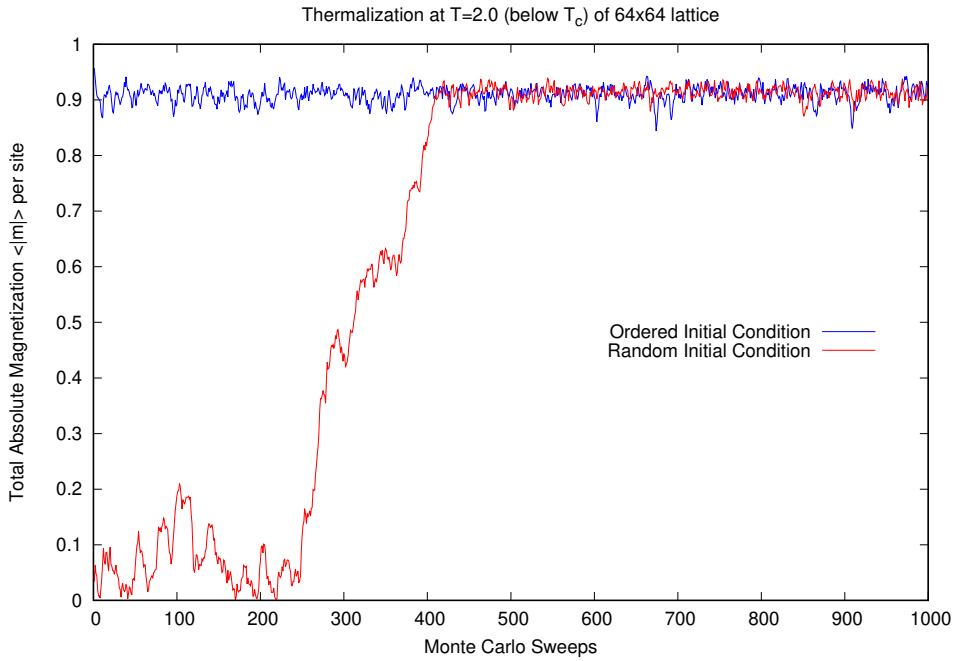


Figure 1: Thermalization of Monte Carlo time data, both initial conditions reach the average value

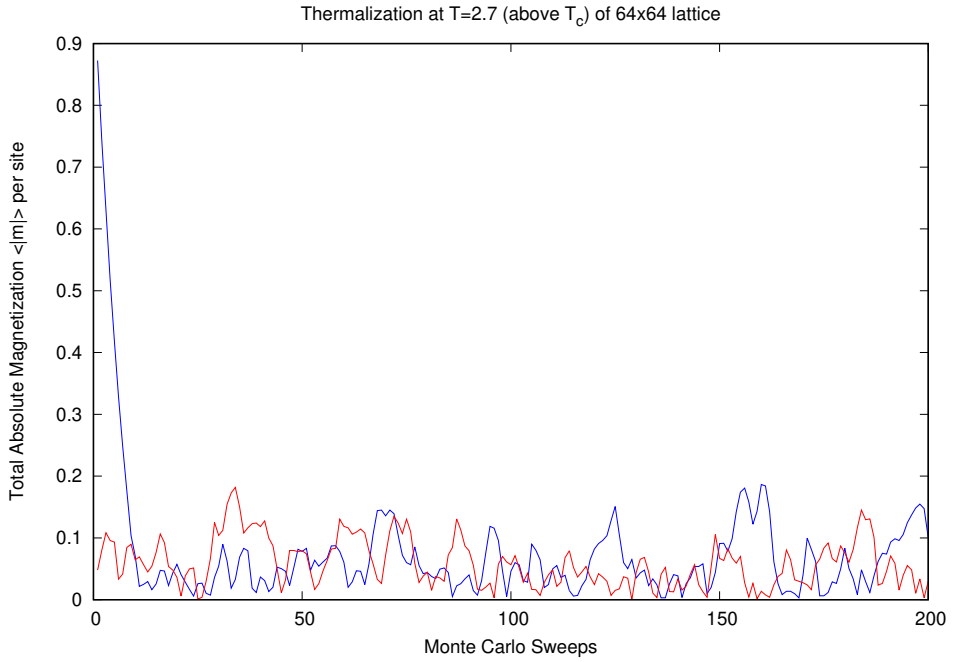
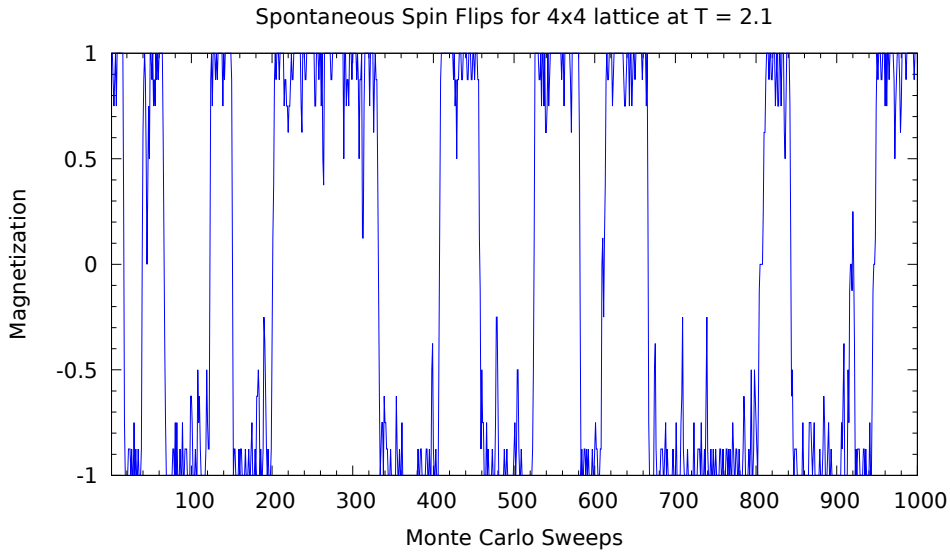
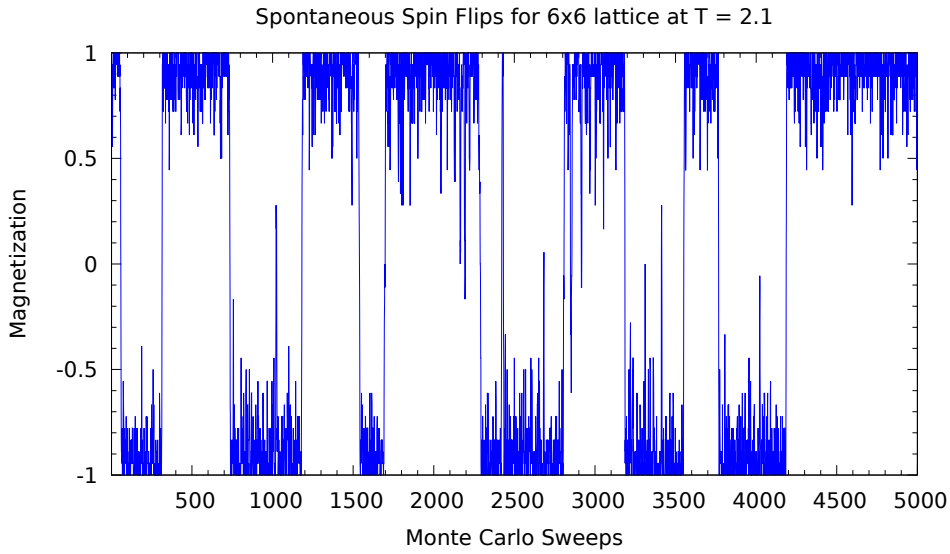
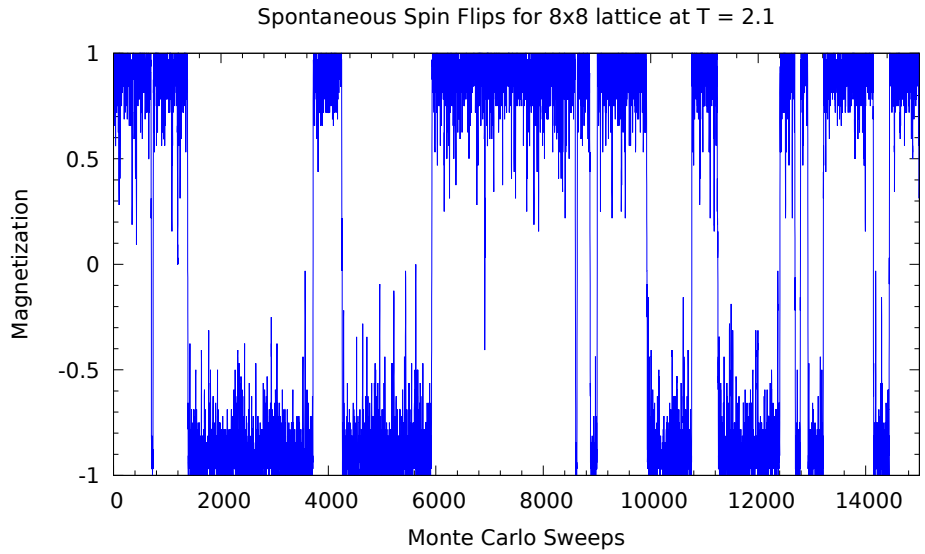
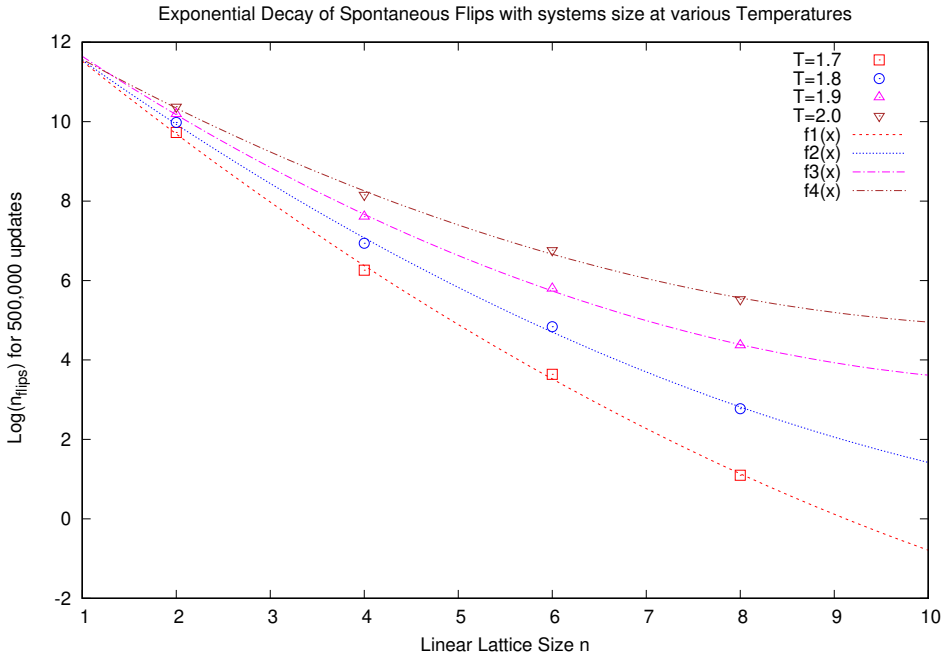


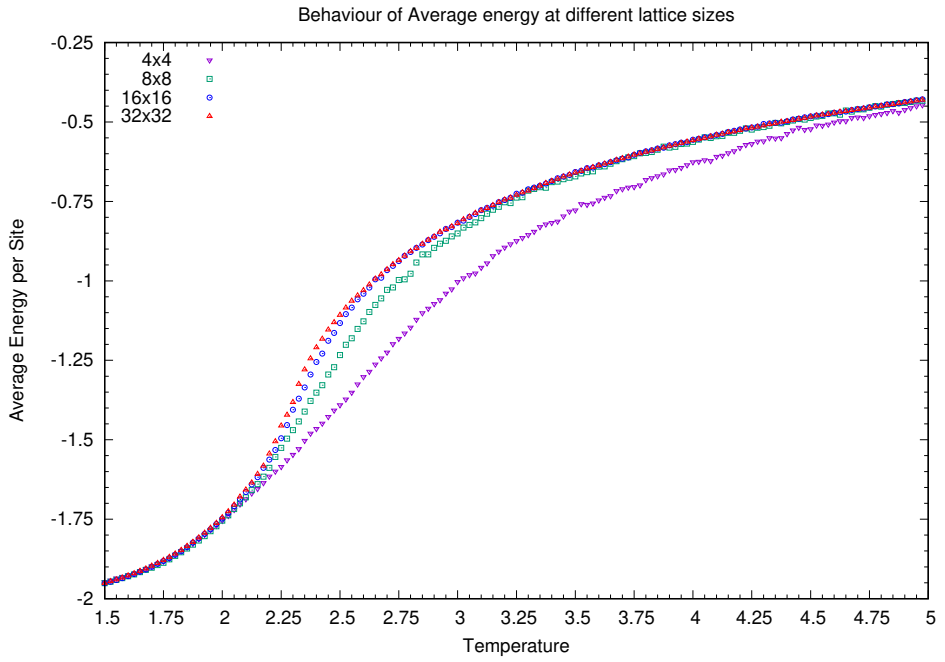
Figure 2: Thermalization of Monte Carlo time data, both initial conditions reach the average value

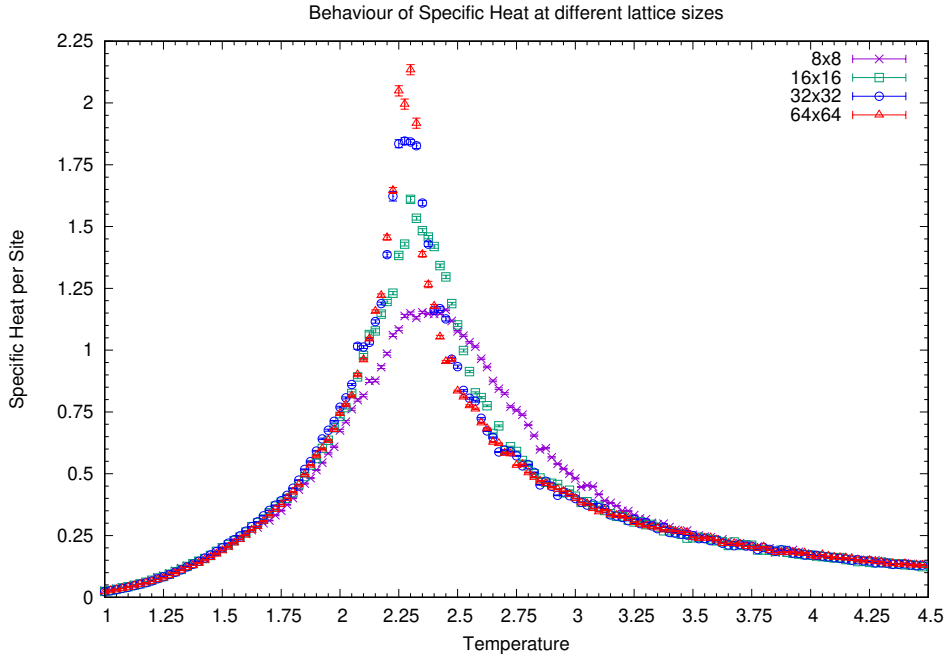

Spontaneous Spin Flips for 4x4 lattice at T = 2.1
Magnetization
1
0.5
0
-0.5
-1
100
200
300
400
500
600
700
800
900
1000
Monte Carlo Sweeps

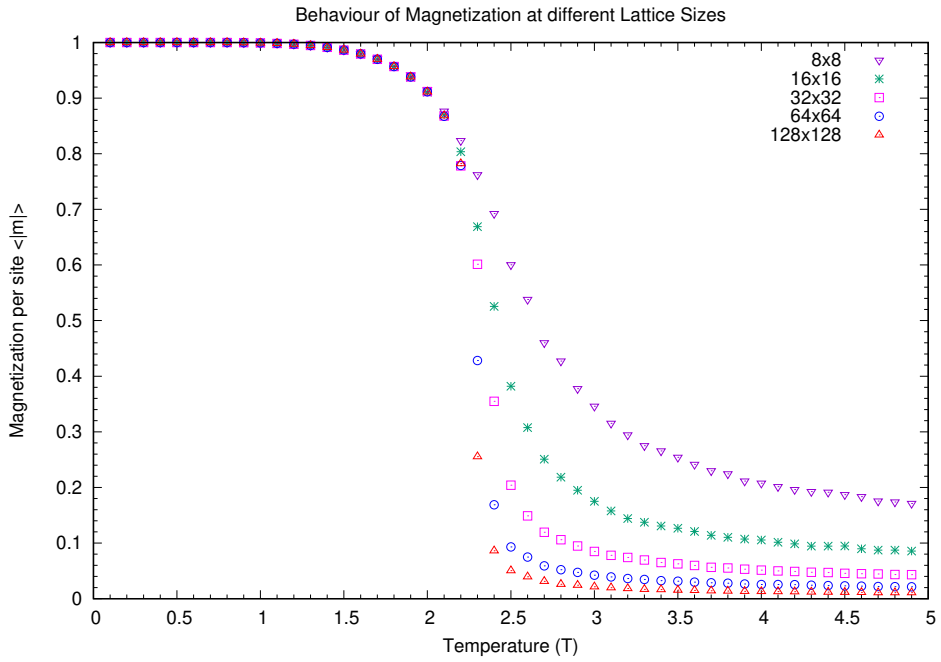

Spontaneous Spin Flips for 6x6 lattice at T = 2.1
Magnetization
1
0.5
0
-0.5
-1
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
Monte Carlo Sweeps

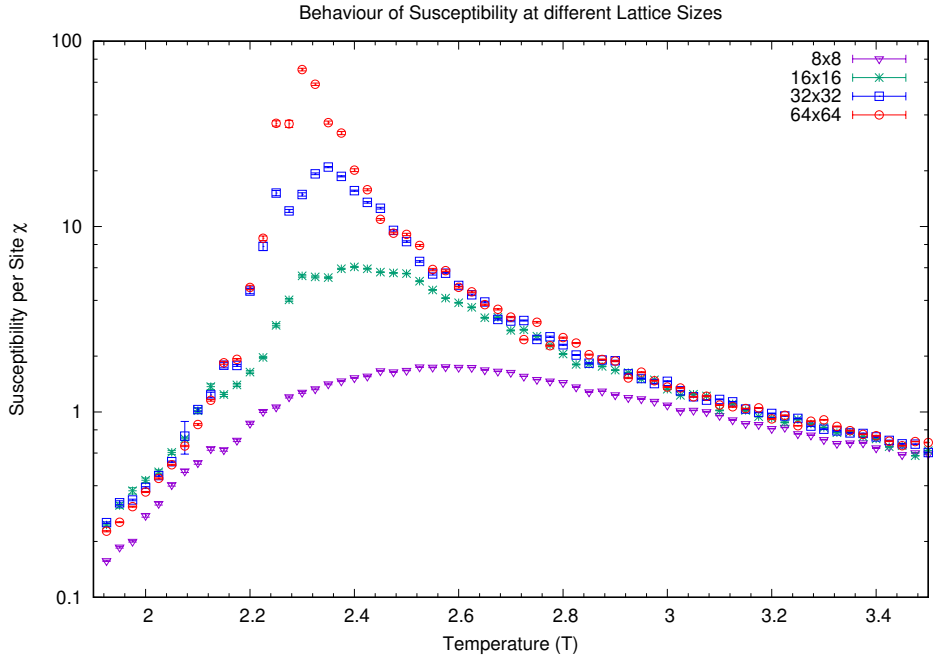

Spontaneous Spin Flips for 8x8 lattice at T = 2.1
Magnetization
Monte Carlo Sweeps
1
0.5
0
-0.5
-1
0
2000
4000
6000
8000
10000
12000
14000

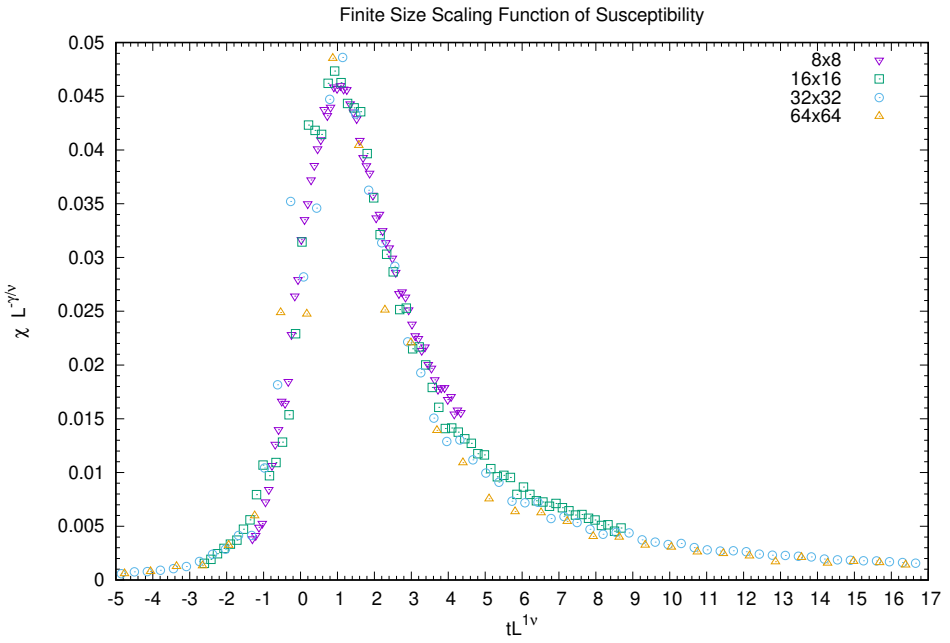

Exponential Decay of Spontaneous Flips with systems size at various Temperatures
Log(nflips) for 500,000 updates
Linear Lattice Size n
T=1.7
T=1.8
T=1.9
T=2.0
f1(x)
f2(x)
f3(x)
f4(x)
12
10
8
6
4
2
0
-2
1
2
3
4
5
6
7
8
9
10

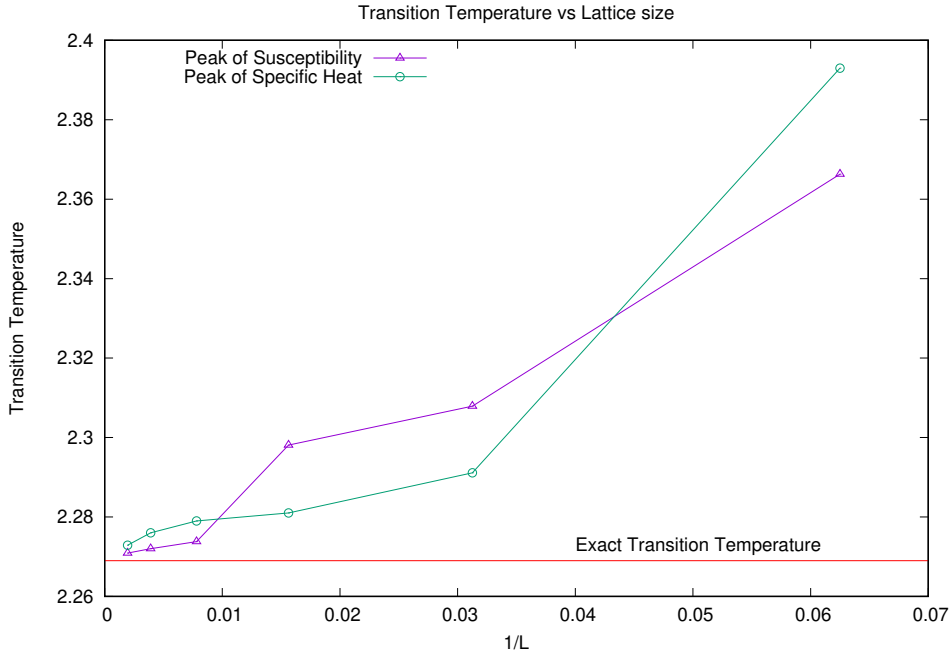

Behaviour of Average energy at different lattice sizes
4x4
8x8
16x16
32x32
Average Energy per Site
Temperature
-0.25
-0.5
-0.75
-1
-1.25
-1.5
-1.75
-2
1.5
1.75
2
2.25
2.5
2.75
3
3.25
3.5
3.75
4
4.25
4.5
4.75
5


Behaviour of Specific Heat at different lattice sizes
8x8
16x16
32x32
64x64
Specific Heat per Site
Temperature
2.25
2
1.75
1.5
1.25
1
0.75
0.5
0.25
0
1
1.25
1.5
1.75
2
2.25
2.5
2.75
3
3.25
3.5
3.75
4
4.25
4.5


Behaviour of Magnetization at different Lattice Sizes
Magnetization per site <|m|>
Temperature (T)
8x8
16x16
32x32
64x64
128x128
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5


Behaviour of Susceptibility at different Lattice Sizes
Susceptibility per Site χ
Temperature (T)
8x8
16x16
32x32
64x64
0.1
1
10
100
2
2.2
2.4
2.6
2.8
3
3.2
3.4


Finite Size Scaling Function of Susceptibility
8x8
16x16
32x32
64x64
χ L^-γ/ν
tL^1ν
0.05
0.045
0.04
0.035
0.03
0.025
0.02
0.015
0.01
0.005
0
-5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17


Transition Temperature vs Lattice size
Peak of Susceptibility
Peak of Specific Heat
Exact Transition Temperature
Transition Temperature
1/L
2.4
2.38
2.36
2.34
2.32
2.3
2.28
2.26
0 0.01 0.02 0.03 0.04 0.05 0.06 0.07

4 Codes

CPU OpenMP code for Ising model

```
module libs3
  use omp_lib
  use ranlxd_generator
  implicit none
  integer, parameter :: n = 256
  double precision, parameter :: N2 = float(n*n)

  contains

    subroutine mcsweep(s,beta)
      integer :: opt,s(n,n),i,j,x,y,xr,xl,yr,yl,delE
      double precision :: beta,u,v,r(n*n)


   !$call omp_set_num_threas(num_threads)

    call ranlxd(r)

    do opt=0,1

   !$omp parallel do default(private) shared(opt,s,beta,r)
      do i=1,n
       do j=1,n/2

         x = i
         y = 2*j-1 + mod(i+opt,2)

          xr = mod(x,n) + 1
          xl = mod(x-2+n,n) + 1
          yr = mod(y,n) + 1
          yl = mod(y-2+n,n) + 1

          delE = 2*s(x,y)*(s(xl,y)+s(xr,y)+s(x,yr)+s(x,yl))
          u = exp(-beta*delE) - r(x+(n-1)*y)
          u = abs(u)/u
          s(x,y) = s(x,y)*int(u)

       enddo
      enddo
   !$omp end parallel do

    enddo !opt

    end subroutine mcsweep




    subroutine findenergy(s,energy)
      integer :: s(n,n),x,y,i,j,xl,xr,yl,yr
      double precision :: energy,ener(n,n/2)
```

```
  !$call omp_set_num_threas(num_threads)
  !$omp parallel do default(private) shared(s,ener)
   do i=1,n
   do j=1,n/2

     x = i
     y = 2*j-1 + mod(i+1,2)

     ener(i,j) = 0.

      xr = mod(x,n) + 1
      xl = mod(x-2+n,n) + 1
      yr = mod(y,n) + 1
      yl = mod(y-2+n,n) + 1

     ener(i,j) = -s(x,y)*(s(xl,y)+s(xr,y)+s(x,yr)+s(x,yl))

    enddo
    enddo
  !$omp end parallel do

  energy = sum(ener)/N2

end subroutine findenergy

subroutine blocking(x,Nupdates,B,mean,errbar,temp)
  integer, intent(in) :: Nupdates,B
  double precision, intent(in) :: temp
  double precision, dimension(Nupdates), intent(in) :: x
  double precision, dimension(B) :: means,chi,means2
  double precision, intent(out) :: mean,errbar
  integer :: i,j,bsize
  double precision, dimension(:,:), allocatable :: dataset

  bsize = Nupdates/B

  allocate(dataset(B,bsize))

  do i=1,b
    dataset(i,:) = x((i-1)*bsize+1:i*bsize)
    means(i) = sum(dataset(i,:))/float(bsize)
    means2(i) = sum(dataset(i,:)*dataset(i,:))/float(bsize)
    chi(i) = n2*(means2(i)-means(i)*means(i))/(temp)
  enddo

  mean = sum(chi)/float(B)
  errbar = sqrt((sum(chi*chi)/float(B)-mean*mean)/float(B))

end subroutine blocking

subroutine jack(x,Nupdates,B,mean,errbar,temp)
  integer, intent(in) :: Nupdates,B
  double precision, dimension(Nupdates), intent(in) :: x
  double precision, intent(in) :: temp
  double precision, intent(out) :: errbar,mean
  double precision, dimension(B) :: m,m2,chi
```

```
      double precision, dimension(:,:), allocatable :: dataset
      integer :: i,j,bsize

      bsize = Nupdates/B

      allocate(dataset(B,bsize))

      do i=1,B
       dataset(i,:) = x((i-1)*bsize+1:i*bsize)
      enddo

      m = 0.d0
      m2 = 0.d0
      do i=1,B
        do j=1,B
         if (j.ne.i) then
           m(i) = m(i) + sum(dataset(j,:))
           m2(i) = m2(i) + sum(dataset(j,:)*dataset(j,:))
         endif
        enddo
        m(i) = m(i)/float(Nupdates-bsize)
        m2(i) = m2(i)/float(Nupdates-bsize)
        chi(i) = n2*(m2(i)-m(i)*m(i))/temp
      enddo


     mean = sum(chi)/float(B)
     errbar = sqrt(sum(chi*chi)/float(B)-mean*mean)



    end subroutine jack


end module libs3


program main
  use libs3
  use ranlxd_generator
  implicit none
  integer,parameter :: Nupdates = 1000
  integer :: s(n,n),i,nthreads
  double precision :: chiav,errbar
  double precision :: time1,time2
  double precision :: temp,beta,mag,tempi,tempf,dtemp
  double precision, dimension(Nupdates) :: x
  logical :: exists

  call rlxd_init(1,123456789)



 tempi = 1.9d0
 tempf = 2.d0
 dtemp = 0.025d0
```

```
 temp = tempi

 time1 = omp_get_wtime()

 nthreads = omp_get_num_threads()

 print*,n,nthreads,Nupdates

 !thremalize
 do i=1,500
   call mcsweep(s,beta)
   !print*,i
 enddo

 do i=1,Nupdates
   call mcsweep
 enddo


 time2 = omp_get_wtime()
 print*,time2-time1

end program main
```

GPGPU CUDA Code for Ising model

```
module libcuda
  use curand_device
  implicit none
  integer, parameter :: n = 64 !Lattice Size
  double precision, parameter :: n2 = float(n*n)

  contains
    attributes(global) subroutine rand_init(gen)
      integer :: i,j
      integer(8) :: seed,seq,offset
      type(curandStateXORWOW) :: gen(n,n)

      i = threadIdx%x ; j = blockIdx%x

      seed = 1234_8 + j*n*n + i*2 ; seq = 0_8 ; offset = 0_8

      call curand_init(seed,seq,offset,gen(i,j))

    end subroutine rand_init

    attributes(global) subroutine rand_update(rand,gen)
      type(curandStateXORWOW) :: gen(n,n)
      double precision :: rand(n,n)
      integer :: i,j

      i = threadIdx%x ; j = blockIdx%x

      rand(i,j) = curand_uniform_double(gen(i,j))

    end subroutine rand_update
```

```
attributes(global) subroutine mcupdate(s,beta,rand,opt)
  integer :: s(n,n),x,y,xr,xl,yr,yl,i,j,delE
  double precision :: rand(n,n),u
  integer, value :: opt
  double precision, value :: beta

  i = threadIdx%x ; j = blockIdx%x

  x = i ; y = 2*j-1+mod(i+opt,2)

  xr = mod(x,n) + 1
  xl = mod(x-2+n,n) + 1
  yr = mod(y,n) + 1
  yl = mod(y-2+n,n) + 1

  delE = 2*s(x,y)*(s(xr,y)+s(xl,y)+s(x,yl)+s(x,yr))

  u = rand(x,y) - exp(-beta*delE)
  u = u/abs(u)
  s(x,y) = int(u)*s(x,y)

end subroutine mcupdate

attributes(global) subroutine find_energy(s,ener)
  integer :: s(n,n),x,y,i,j,xr,xl,yr,yl
  integer, dimension(n,n/2) :: ener

  i = threadIdx%x ; j = blockIdx%x

  x = i ; y = 2*j-1 + mod(i+1,2)

  xr = mod(x,n) + 1
  xl = mod(x-2+n,n) + 1
  yr = mod(y,n) + 1
  yl = mod(y-2+n,n) + 1

  ener(i,j) = -s(x,y)*(s(xr,y)+s(xl,y)+s(x,yr)+s(x,yl))


end subroutine find_energy


subroutine blocking(x,Nupdates,B,mean,errbar,temp)
  integer, intent(in) :: Nupdates,B
  double precision, intent(in) :: temp
  double precision, dimension(Nupdates), intent(in) :: x
  double precision, dimension(B) :: means,chi,means2
  double precision, intent(out) :: mean,errbar
  integer :: i,j,bsize
  double precision, dimension(:,:), allocatable :: dataset

  bsize = Nupdates/B

  allocate(dataset(B,bsize))
```

```
      do i=1,B
        dataset(i,:) = x((i-1)*bsize+1:i*bsize)
        means(i) = sum(dataset(i,:))/float(bsize)
        means2(i) = sum(dataset(i,:)*dataset(i,:))/float(bsize)
        chi(i) = n2*(means2(i)-means(i)*means(i))/(temp)
      enddo


      mean = sum(chi)/float(B)
      errbar = sqrt((sum(chi*chi)/float(B)-mean*mean)/float(B))

    end subroutine blocking

    subroutine jack(x,Nupdates,B,mean,errbar,temp)
      integer, intent(in) :: Nupdates,B
      double precision, dimension(Nupdates), intent(in) :: x
      double precision, intent(in) :: temp
      double precision, intent(out) :: errbar,mean
      double precision, dimension(B) :: m,m2,chi
      double precision, dimension(:,:), allocatable :: dataset
      integer :: i,j,bsize

      bsize = Nupdates/B

      allocate(dataset(B,bsize))

      do i=1,B
       dataset(i,:) = x((i-1)*bsize+1:i*bsize)
      enddo

      m = 0.d0
      m2 = 0.d0
      do i=1,B
        do j=1,B
         if (j.ne.i) then
           m(i) = m(i) + sum(dataset(j,:))
           m2(i) = m2(i) + sum(dataset(j,:)*dataset(j,:))
         endif
        enddo
        m(i) = m(i)/float(Nupdates-bsize)
        m2(i) = m2(i)/float(Nupdates-bsize)
        chi(i) = n2*(m2(i)-m(i)*m(i))/temp
      enddo


     mean = sum(chi)/float(B)
     errbar = sqrt(sum(chi*chi)/float(B)-mean*mean)



    end subroutine jack


end module libcuda

program main
```

```
  use libcuda
  use cudafor
  use curand
  integer, parameter :: Nupdates = 5000
  integer, dimension(n,n) :: s
  integer, device :: s_d(n,n)
  double precision, device :: rand_d(n,n)
  integer, device :: ener_d(n,n/2)
  integer :: i,j,opt,B
  double precision :: temp,beta,rand(n,n),mmod,errbar,x(Nupdates),chiav
  integer :: ener(n,n/2)
  type(curandStateXORWOW), device :: gen_d(n,n)

  !Nupdates = 15000
  s = 1
  s_d = s

temp = 1.4d0
do
  call rand_init<<<n,n>>>(gen_d)

  beta = 1./temp

  !thermalize
  do j=1,Nupdates
   call rand_update<<<n,n>>>(rand_d,gen_d)
   do opt=0,1
    call mcupdate<<<n,n>>>(s_d,beta,rand_d,opt)
   enddo
  enddo

  do j=1,Nupdates
  !create uniform random numbers in gpu
    call rand_update<<<n,n>>>(rand_d,gen_d)
    do opt=0,1
      call mcupdate<<<n,n>>>(s_d,beta,rand_d,opt)
    enddo
    !call find_energy<<<n/2,n>>>(s_d,ener_d)
   !ener = ener_d
   !x(j) = sum(ener)/n2
   s = s_d
   x(j) = abs(sum(s))/n2
  enddo !j

  call jack(x,Nupdates,200,chiav,errbar,temp)

  print*,temp,chiav,errbar
  temp = temp + 0.025
  if (temp.gt.3.5) exit

enddo

end program main
```

5 Acknowledgements

I am really grateful to my project supervisor **Prof. Indra Dasgupta** for supporting me in pursuing the topics which I wanted to learn even though it was not his primary area of research. During this project I had support from a number people at IACS. I had some very helpful suggestions from **Prof. Arnab Sen**. I would also like to thank **Prof. Pushan Majumdar** for giving me accces to a XeonPhi machine to run and test OpenMP codes on and some very helpful advices on using GPUs.

References

[1] R. Savit, "Duality in field theory and statistical systems," *Rev. Mod. Phys.*, vol. 52, pp. 453–487, Apr 1980.

[2] J. M. Thijssen, *Computational Physics*. New York, NY, USA: Cambridge University Press, 2007.

[3] A. W. Sandvik, "Computational studies of quantum spin systems," *AIP Conference Proceedings*, vol. 1297, no. 1, pp. 135–338, 2010.