Parallel Implementation of the Heisenberg Model using Monte Carlo on GPGPU

Alessandra M. Campos, João Paulo Peçanha, Patrícia Pampanelli, Rafael B. de Almeida, Marcelo Lobosco, Marcelo B. Vieira, and Sócrates de O. Dantas

Universidade Federal de Juiz de Fora, DCC/ICE and DF/ICE,
Cidade Universitária, CEP: 36036-330, Juiz de Fora, MG, Brazil
`{amcampos,joaopaulo,patricia.pampanelli,rafaelbarra,marcelo.lobosco, marcelo.bernardes}@ice.ufjf.br,dantas@fisica.ufjf.br`

Abstract. *The study of magnetic phenomena in nanometer scale is essential for development of new technologies and materials. It also leads to a better understanding of magnetic properties of matter. An approach to the study of magnetic phenomena is the use of a physical model and its computational simulation. For this purpose, in previous works we have developed a program that simulates the interaction of spins in three-dimensional structures formed by atoms with magnetic properties using the Heisenberg model with long range interaction. However, there is inherent high complexity in implementing the numerical solution of this physical model, mainly due to the number of elements present in the simulated structure. This complexity leads us to develop a parallel version of our simulator using General-purpose GPUs (GPGPUs). This work describes the techniques used in the parallel implementation of our simulator as well as evaluates its performance. Our experimental results showed that the parallelization was very effective in improving the simulator performance, yielding speedups up to 166.*

Key words: Computational Physics, Heisenberg Model, High Performance Computing, Many-core Programming, Performance Evaluation.

1 Introduction

The magnetic phenomena are widely used in the development of new technologies, such as electric power systems, electronic devices and telecommunications systems, among many others. To a better understanding of magnetism, it is essential the study of materials in nanoscale. The research at atomic scale has taken the physicists Albert Fert from France and Peter Grünberg from Germany to discover, independently, a novel physical effect called giant magnetoresistance or GMR. By such important discovery, they won the 2007 Nobel Prize in Physics. The GMR effect is used in almost every hard disk drives, since it allows the storage of highly densely-packed information.

The magnetic phenomenon is associated to certain electrons properties: a) the angular momentum, related to electrons rotation around the atomic nucleus; and b) spins, a quantum mechanics property essential to the magnetic behavior. When magnetic atoms are brought together they interact magnetically, even without an external magnetic field, and thus may form structures at the nanoscale. Computer-aided simulations can be used to study such interactions; these simulators contribute to the understanding of magnetism in nanometer scale providing numerical information about the phenomenon. Physicists and engineers may use these simulators as virtual labs, creating and modifying features of magnetic systems in a natural way. Moreover, visual and numerical data are useful in the comprehension of highly complex magnetic systems.

Particularly, we are interested in simulating the behavior of ferromagnetic materials. The interaction among ferromagnetic atoms is well-defined by the Heisenberg model. This model was introduced by Heisenberg in 1928 [1] and represents mathematically the strong alignment presented by spins in a local neighborhood. The Heisenberg model is a statistical mechanical model used in the study of ferromagnetism.

In a previous work [2], we have presented and implemented a computational model used to simulate the spins interaction in three-dimensional magnetic structures using the Heisenberg model with long range interaction. The main goal of our simulator is to provide a tool to analyze volumetric magnetic objects under an uniform external magnetic field. The spins interaction occurs in a similar way to the classical n-body problem [3]. Nevertheless, our work focuses on the solution of interaction among particles that assemble in crystalline arrangements. The complexity of this problems is $O(N^2)$, where N is the number of spins in the system.

In order to reduce the costs associated with the computational complexity, we have developed a parallel version of our simulator using General-purpose Graphics Processing Units (GPGPUs). GPGPUs were chosen due to their ability to process many streams simultaneously. The present work describes the techniques used in our parallel implementation as well as evaluates its performance. These techniques can be easily extended to other problems with similar features. Our experimental results showed that the parallelization was very effective in improving the simulator performance, yielding speedups up to 166.

For the best of our knowledge, the main contributions of our paper are the following. First, we observed that the energy of each atom in the Heisenberg model can be computed independently. This is the main factor that contributed to the speedups we achieved. A previous work [4], which performed simulations on 3D systems using a simpler spin model (the Ising model [5]), obtained a speedup of 35. Second, this is the first work in literature that uses Compute Unified Device Architecture (CUDA) to successfully implement the Heisenberg model with long range interaction. Finally, we are the first to propose an automatic generation, at run-time, of the execution configuration of a CUDA kernel.

The remainder of this paper is organized as follows. Section 2 gives an overview of the physical model used in the simulations. Section 3 gives a brief description of CUDA. Section 4 presents the computational models. In Section 5, we evaluate the impact of the techniques used in the parallelization on the simulator performance. Section 6 presents related work and we state our conclusions in Section 7.

2 Physical Model

All materials can be classified in terms of their magnetic behavior falling into one of five categories depending on their bulk magnetic susceptibility $\chi = |\boldsymbol{M}|/|\boldsymbol{H}|$ ($\boldsymbol{H}$ is the external magnetic field vector and $\boldsymbol{M}$ the magnetization vector). The five categories are: a) ferromagnetic and b) ferrimagnetic ($\chi \gg 1$); c) diamagnetism ($\chi < 1$); d) paramagnetic ($\chi > 0$); and e) antiferromagnetic (χ small). In particular, this work focuses on modeling elements with ferromagnetic properties, whose magnetic moments tend to align to the same direction. In ferromagnetic materials, the local magnetic field is caused by their spins.

Loosely speaking, the spin of an atom in quantum mechanics refers to the possible orientations that their subatomic particles have when they are, or are not, under the influence of an external magnetic field. The spin representation using an arbitrary 3D oriented vector is known as Heisenberg model [6].

Suppose you have a closed system composed by a single crystalline or molecular structure, which could take the form of any basic geometric shape, such as spheres, cubes and cylinders. The atoms of this structure are modeled as points in the space (grid) and are associated with a $\mathbf{S}_i \in \mathbb{R}^3$ vector that represents their spins. The atoms are equally spaced in a regular grid of variable size. The unique external influence is an uniform magnetic field $\boldsymbol{H}$.

Such magnetic structures, when influenced by $\boldsymbol{H}$, tends to orient their spins in the direction of the external field, but this effect can be influenced by temperature. As the system evolves, the spins rotate, trying to adjust their direction to the applied external magnetic field. Because each atom has an unique energy and a well-defined position in space, it is possible to calculate the total system energy E_t, given by the interaction of N dipoles, as follows:

$$E_t = \frac{A}{2} \sum_{\substack{i,j=1 \\ i \neq j}}^{N} \omega_{ij} - J \sum_{\substack{i,k=1 \\ i \neq k}}^{N} \mathbf{S}_i \cdot \mathbf{S}_k - \sum_{i=1}^{N} D(\mathbf{S}_i \cdot \boldsymbol{H}), \tag{1}$$

where ω_{ij} represents the dipole-dipole interaction between the i-th and j-th spin, and is given by:

$$\omega_{ij} = \frac{\mathbf{S}_i \cdot \mathbf{S}_j}{|\mathbf{r}_{ij}|^3} - 3\frac{(\mathbf{S}_i \cdot \mathbf{r}_{ij})\,(\mathbf{S}_j \cdot \mathbf{r}_{ij})}{|\mathbf{r}_{ij}|^5}, \tag{2}$$

where $\mathbf{S}_i$ is the spin of the i-th particle, $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$ is the position vector that separates the particles i and j.

In Equation 1, A is the intensity of the dipole-dipole interaction, J is the ferromagnetic factor characteristic of the object in question, $\boldsymbol{H}$ is the uniform external field and D is related to its amplitude. The first term of Equation 1 is called long-range dipole-dipole term. This is the most expensive term to compute since our main goal is to access the energy of every individual atom. The computational complexity of this term is $O(N^2)$, where N is the number of spins.

The second term of Equation 1, called ferromagnetic interaction, is a short-range interaction where, in a regular cubic grid, we have only six nearest neighbors elements ($k = 1, 2, 3, ..., 6$) influencing the final energy of a given spin. The last term of the same equation refers to the influence of the external field vector $\boldsymbol{H}$ on each particle.

A detailed analysis of the energy along a Monte Carlo simulation can reveal very important information: the moment that occurs a phase transition from ferromagnetic to a paramagnetic behavior. The main goal of computing this equation is to find, for a given spin configuration, the E_t corresponding to the critical temperature value that causes a phase transition on the system.

3 General-Purpose Computing on Graphics Processing Units - GPGPU

NVIDIA's CUDA (Compute Unified Device Architecture)[7] is a massively parallel high-performance computing platform on General-Purpose Graphics Processing Unit or GPGPUs. CUDA includes C software development tools and libraries to hide the GPGPU hardware from programmers.

In GPGPU, a parallel function is called kernel. A kernel is a function callable from the CPU and executed on the GPU simultaneously by many threads. Each thread is run by a *stream processor*. They are grouped into blocks of threads or just blocks. A set of blocks of threads form a grid. When the CPU calls the kernel, it must specify how many threads will be created at runtime. The syntax that specifies the number of threads that will be created to execute a kernel is formally known as the execution configuration, and is flexible to support CUDA's hierarchy of threads, blocks of threads, and grids of blocks.

Since all threads in a grid execute the same code, a unique set of identification numbers is used to distinguish threads and to define the appropriate portion of the data they must process. These threads are organized into a two-level hierarchy composed by blocks and grids and two unique values, called *blockId* and *threadId*, are assigned to them by the CUDA runtime system. These two build-in variables can be accessed within the kernel functions and they return the appropriate values that identify a thread.

Some steps must be followed to use the GPU: first, the device must be initialized. Then, memory must be allocated in the GPU and data transferred to it. The kernel is then called. After the kernel have finished, results must be copied back to the CPU.

4 Computational Model

The physical problem is mapped onto a statistical one and solved using a Monte Carlo method called Metropolis algorithm [8]. The Metropolis dynamics is based on a single spin-flip procedure. Briefly, at each iteration a random spin is selected, its value is also changed randomly and the new total system energy is computed using Equation 1. Therefore, the algorithm decides whether the spin should take the new orientation or return to the old state. If the new system energy is lower than the previous one, it is accepted. Otherwise, the spin arrangement temperature may be interfering in the system. In this case, the new value is accepted only if the following condition applies: $e^{(-\Delta E/Kt)} > R$, where R is a random value in the range $[0, 1]$. Otherwise, the new value is rejected and the system returns to its previous state.

An important step in the Metropolis algorithm is the choice of random values. It is important to ensure that the probability of choosing a particle has uniform distribution. To achieve this condition, we use the Mersenne Twister algorithm [9], which has a period of $2^{19937} - 1$, in all random steps.

The parallelization process focused on the computation of the total system energy due to its huge computational time cost. It is based on the observation that the energy of each atom can be computed independently. This fact can be easily checked in Equation 1: the atom energy depends only on the spin-orientation of other atoms. Thus, the energy of distinct regions of the space can also be computed independently. So, in order to increase the performance, the main process can issue multiple threads to compute the energy for each part of the space containing the ferromagnetic object. A detailed discussion on the CUDA implementation is presented in this section. A CPU-based multithreaded version of the code was also implemented for comparison purposes.

Both algorithms depicted in this section use an implicit representation of the magnetic object. The simulation area consists of a three-dimensional matrix. Each spin position can be obtained implicitly, by the triple $(x, y, z) \in \mathbb{N}^3$ which corresponds to its own matrix coordinates. Thus, it is not necessary to store individual spin positions, assuring a faster and simpler data manipulation. Another advantage of the matrix representation is that it is trivial to model complex geometries using implicit equations.

4.1 Multithreaded Version

The multithreaded version of our simulator uses a dynamic spatial partition scheme. The scheme is straightforward: the space is divided into plans, which are stolen by threads following a work stealing algorithm. The thread computes the energy of the plan, which is calculated as the sum of the energies of all spins located in the plan. The total system energy is obtained as the sum of the energies of all plans. The division of space into plans follows the directions given by the axis XY, YZ or ZX.

During the execution, the user can choose the number of threads that will be created. This number is usually equal to the number of processors and/or cores available in the machine. Basically, these are the steps followed during the work stealing:

- Each thread picks a different plan from the space. The access to the plan is synchronized;

– After finishing its job, the thread adds its result into a specific variable. The access to this shared variable is synchronized;
– If one or more plans are still available, i.e., if their energies were not calculated, the algorithm continues, as Figure 1 illustrates. Otherwise, the algorithm returns the total energy value.

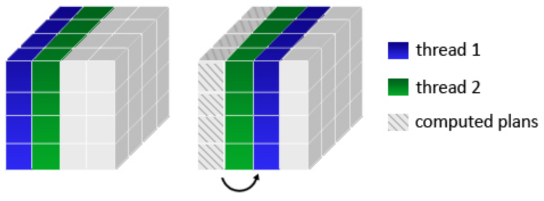

Fig. 1. Thread load balance using aligned plans.

4.2 CUDA Version

In our first approach, we organized the matrix contiguously in global memory as an unidimensional vector. The access to the matrix was done linearly. Figure 2 illustrates this initial mapping attempt. The GPU was only used to calculate the dipole-dipole energy. Using this approach, the kernel is called by the CPU to calculate the dipole-dipole interaction for each atom in the system. Following the physical model, the current energy for each atom is obtained accessing in global memory all other atoms in the system. After the dipole-dipole energy is computed, the result is copied back to the CPU. The CPU then calculates the other terms of Equation 1 and completes the execution of the code, including the execution of the Mersenne Twister algorithm.

However, the performance of our first approach was lower than expected. Two distinct factors contributed to the poor performance: a) the memory accesses by threads were not organized to exhibit favorable access patterns, so memory accesses could not be coalesced by GPU, and b) the initialization and data transfers to and from GPU were executed at each Monte Carlo step, which represented a large amount of overhead. So, in order to improve the performance, we restructured our code.

The first modification that has been implemented is related to the computation of the system energy. While in our first approach the energy of a single spin was calculated at each Monte Carlo step, in our second approach the energy of each particle and its interactions with all others is calculated in parallel. In this second approach, all the energies presented in Equation 1 are computed in GPU, differently from the first approach, where only the dipole-dipole energy was calculated in GPU. After computing the energies of all spins, we update

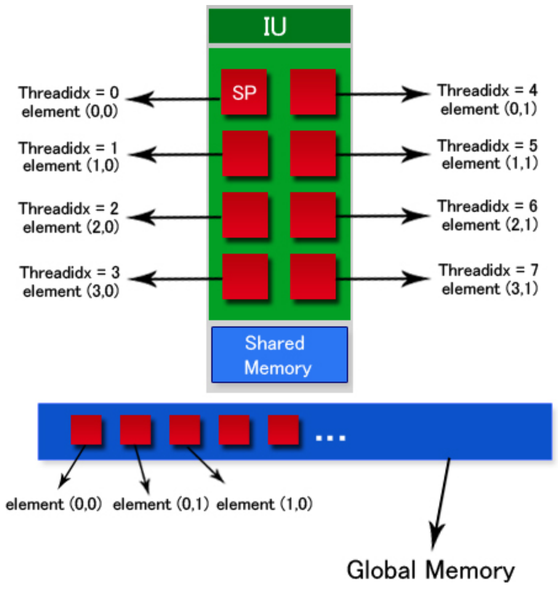

Fig. 2. Mapping of 4×2 matrix assuming the use of only one subset of eight stream processors

these energies in global memory. At the end of the computation, taking advantage of the location of computed energies in the global memory, we reduce the vector of energies on the GPU. Then the CPU gets the total energy from GPU and tests it. If it is not accepted, the system state is restored to its previous configuration.

Another important difference between both approaches is the way data is mapped into GPU memory. In our first approach, the data was completely stored in global memory. Although global memory is large, it is slower than other memories available in GPU devices, such as the shared memory, a type of memory allocated to thread blocks. However, the shared memory is smaller than our data structure. So we use both global and shared memory to store the data using the well-known tiling technique. Using tiles, the data is partitioned into subsets so that each tile fits into the shared memory. However, it is important to mention that the kernel computation on these tiles must be done independently of each other.

The third modification is related to the way the GPU hardware is used. If the number of threads to be created is smaller than a given threshold value, we modify the way the computation is done. In this case, two or more threads are associated to each spin and collaborate to calculate its dipole iteration. Threads collaborate in a simple way: the tile is split up among threads, so each thread will be responsible for calculating the dipole interaction of its spin with part of the spins that composes a tile. For example, if our algorithm decides to create two threads per spin, than one thread will be responsible for calculating the iterations of that spin with the spins that composes the first half of the tile while the second one will be responsible for calculating its iterations with the spins of the second half of the tile. This approach improves the GPU usage because more threads are created, while reducing, at the same time, the total computation done by a single thread.

A final modification in our first approach was the decomposition of our matrix in three distinct float vectors containing respectively: a) the atom position in 3D space, b) its spin orientation, and c) its energy. This modification was inspired by the work described in [10] with the objective of optimizing memory requests and transfers and avoid memory conflicts. The idea is to reduce the chance of accessing the same memory position concurrently, and to better align the data structures. The first vector was declared as *float4*, the second one as *float3* and the last one as *float*. In the case of the first vector, three values form the coordinates, and the fourth value is a uniquely defined index. This index is used to avoid the computation of the long range energy of the particle with itself.

Automatic Generation of the Execution Configuration. When the host invokes a kernel, it must specify an execution configuration. In short, the execution configuration just means defining the number of parallel threads in a group and the number of groups to use when running the kernel for the CUDA device. The programmer is responsible for providing such information. The choice of the execution configuration values plays an important role in the performance of the application.

In our implementation, the execution configuration of a kernel is automatically generated at run-time: the number of threads per block and the number of blocks are calculated based on the number of spins present in the system. In order to improve performance, our goal is to obtain the maximum number of threads per block. To obtain this number, some aspects must be taken into account, such as hardware characteristics and the amount of resources available per thread.

We start our algorithm querying the device to obtain its characteristics. Some information are then extracted, such as the number of multiprocessors available. Then, some values are computed, such as mnt, the minimum number of threads that must be created to guarantee the use of all processors available in the GPGPU architecture. This value is equal to the maximum number of threads per block times the number of multiprocessors. We use 256 as the maximum number of threads per block because this was the maximum value that allows a kernel launch. Then, we compute the number of threads that will be used during computation. This value is calculated in two steps. The first step considers that one thread will be used per spin, while the second step takes into account the use of multiples threads per spin. In the first step, we start setting $number\_of\_threads$ per block equal to one. Then we verify if the number of spins is a prime number: the algorithm tries to find the Greatest Common Divisor (GCD) between 1 and the radix of the number of spins, since this value is enough to determine if the number of spin is prime or not. If the number is prime, we have our worst case which keeps the *number_of_threads* per block equal to one. During the computation of the *GCD*, we store the quotient of the division between the number of spins and the divisor found. We verify if the quotient is into the interval between 1 and 256. If so, it is considered as a candidate to be the *number_of_threads* per block, otherwise the divisor is considered as a candidate. The second step evaluates if the use of multiple threads per spin is viable. To do so, the algorithm compares the number of spins with *mnt*. If the number of spins is equal to or greater than *mnt*, the value obtained in the first step is maintained as the *number_of_threads* per block. Otherwise the algorithm tries to arrange the spins in a bi-dimensional matrix, where x represents the number of spins per block while y represents the number of threads per spin. We try to arrange threads in such a way that the two dimensions of the grid, x and y, reflects the warp size and the number of stream processors available in the machine. The third dimension, z, will be equal to one. If no arrange of x and y can be found, the block and grid dimensions are, respectively, equal to ($number\_of\_threads$, 1, 1) and ($number\_of\_spins/number\_of\_threads$, 1,1). If an arrange was found, the block and grid dimensions are, respectively, equal to (x, y, 1) and ($number\_of\_spins/x$,1,1).

The CUDA algorithm. Figure 3 summarizes the complete CUDA algorithm as well as the techniques employed to achieve better performance. The first step of the algorithm is to decompose the data matrix in three distinct vectors: a) direction, b) position and c) energy. These vectors are copied into the GPU's global memory. Then, we calculate the execution configuration of the kernel using the algorithm described in the previous subsection. After this, the CPU calls the kernel. Each thread accesses a particular spin according to its unique identification and copies its direction and position values into the local memory. When all threads of the grid finish this step, they begin to calculate the dipole interaction: each thread calculates the dipole energy between the spin kept in its local memory and the subset of spins (or part of it, in the case of multiple threads per spin) stored in the tile, that was brought from the shared memory to the local memory. Due to the way data is organized, all memory transfers are done without blocking the threads. The result is then added to the partial result stored in a local variable. This step is repeated until all threads have calculated the interaction of its local spin and all other spins of the system. Then, each thread calculates the ferromagnetic factor and the interaction with the external field. These values are then added to the dipole value just found and the final result is written back in an unique position of a vector, called energy vector, that is stored into the global memory. The CPU then calls another kernel that computes the reduction of the energy vector. The sum of all energy vector positions represents the new energy of the system.

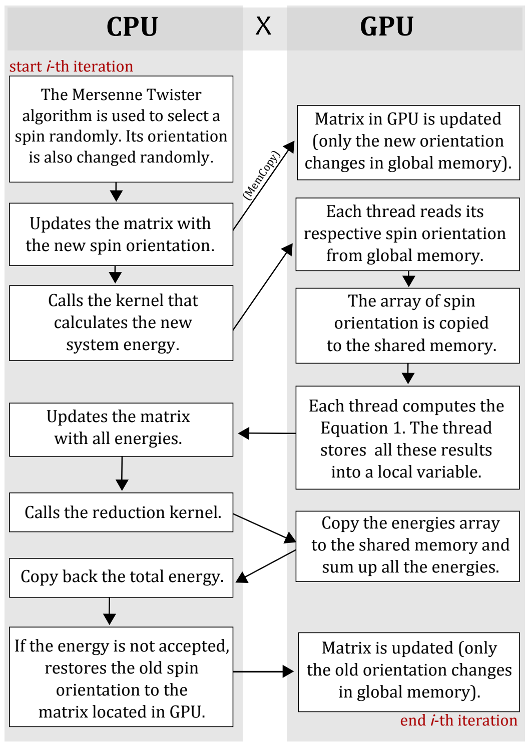
CPU
X
GPU
start *i*-th iteration
The Mersenne Twister algorithm is used to select a spin randomly. Its orientation is also changed randomly.
Updates the matrix with the new spin orientation.
Calls the kernel that calculates the new system energy.
Updates the matrix with all energies.
Calls the reduction kernel.
Copy back the total energy.
If the energy is not accepted, restores the old spin orientation to the matrix located in GPU.
(MemCopy)
Matrix in GPU is updated (only the new orientation changes in global memory).
Each thread reads its respective spin orientation from global memory.
The array of spin orientation is copied to the shared memory.
Each thread computes the Equation 1. The thread stores all these results into a local variable.
Copy the energies array to the shared memory and sum up all the energies.
Matrix is updated (only the old orientation changes in global memory).
end *i*-th iteration

Fig. 3. Algorithm to calculate the total energy of the system.

5 Experimental Evaluation

In this section, we present experimental results obtained with three distinct versions of our simulator: a) the sequential version, b) the multithread version and c) the CUDA version. Both the sequential and multithread implementations have been tested on a Dual Intel Xeon E5410 2.33Ghz with 4 GB of memory. The CUDA implementation has been tested on a Intel Core 2 Quad Q6600 2.4Ghz with a NVIDIA GeForce 9800 GTX+ and on a Intel Core 2 Quad Q9550 2.83Ghz with a NVIDIA GeForce 295 GTX. All machines run Ubuntu 9.04 64-bits. The NVIDIA GeForce 9800 graphic card has 128 stream processors, 16 multiprocessors, each one with 16KB of shared memory, and 512MB of global memory. The NVIDIA GeForce 295 GTX has two GPUs containing 240 stream processors, 30 multiprocessors, each one with 16KB of shared memory, and 862MB of global memory. However, only one of the two 295 GTX GPUs was used during the experiments.

The performance figures of our simulator were collected by calculating the energy of a cube completely filled with spins. Six different cube sizes were used as benchmarks. To guarantee the stability of the system energy, we executed 5000 iterations for each benchmark. We submitted the benchmarks 10 times to

Table 1. Grid size, serial execution time, and parallel execution times for the multi-threaded and CUDA versions of our simulator. All times are in seconds.

Grid size	Sequential	Multithread (8 threads)	GeForce 9800 GTX+	GeForce 295 GTX
8x8x8	85.0s	15.0s	5.3s	2.6s
10x10x10	325.0s	64.0s	20.0s	4.9s
12x12x12	967.5s	159.0s	53.4s	18.0s
14x14x14	2,438.0s	346.0s	139.8s	24.0s
16x16x16	5,425.8s	664.0s	298.1s	52.1s
20x20x20	20,760.3s	3,035.0s	1,140.9s	82.7s

all versions of our simulator, and reported the average execution time for each benchmark in Table 1. The standard deviation obtained was negligible.

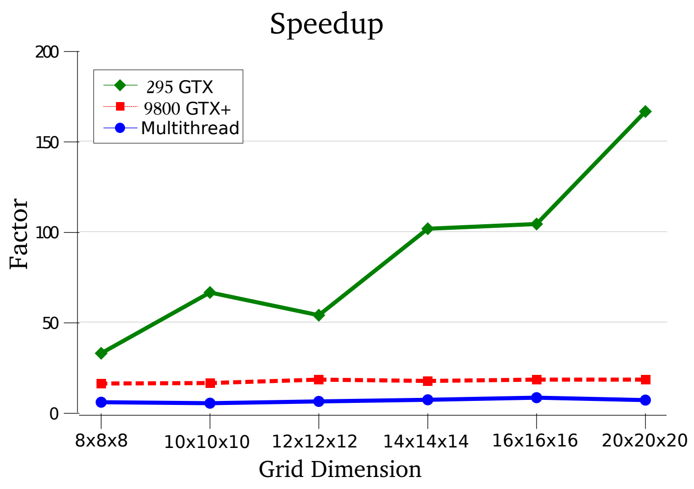

Fig. 4. Speedups over sequential version

In Figure 4, we present speedups for each of the simulator versions. The speedup figures were obtained by dividing the sequential execution time of the simulator by its parallel version. Figure 4 shows that our parallel versions were very effective in improving the simulator performance, yielding speedups between 5.1 to 166. Despite the multithreaded version speedups were respectable, ranging from 5.1 to 8.1 for those six benchmarks on an 8 core machine, its performance was below that of CUDA version. CUDA speedups range from 16 to 166. We can observe a small reduction in the speedup of CUDA version for both 9800 GTX+ and the 295 GTX cards when running the 12x12x12 and the 16x16x16 cube sizes. We suspect that this happens due to a problem in the grid mapping.

Table 2. Energy average after 5000 interactions.

Grid size	Sequential	GeForce 295 GTX	Standard Deviation
8x8x8	-29.21122	-29.777496	0.400418
10x10x10	-13.729978	-13.735782	0.004104
12x12x12	-6.444052	-6.444051	0.000001
14x14x14	-2.876331	-3.172576	0.209477
16x16x16	-1.510165	-1.492171	0.012724
20x20x20	-0.532878	-0.576437	0.030801

The differences between GeForce 9800 GTX+ and GeForce 295 GTX are more evident for larger systems. This occurs because GeForce 295 GTX has 240 stream processors (per GPU), 862MB of memory capacity (per GPU) and 223.8 GB/s of memory bandwidth, while GeForce 9800 GTX+ has 128 stream processors, 512MB of memory capacity and 70.4 GB/s of memory bandwidth. Recall that the 295 GTX has 2 GPUs, but only one is used in the experiments. For small configurations, with 512 spins (8x8x8), 1,000 spins (10x10x10) and 1,728 spins (12x12x12), the 295 GTX outperforms the 9800 GTX+, in average, by a factor of 3.0. For medium configurations, with 2,744 (14x14x14) and 4,096 (16x16x16), the 295 GTX outperforms the 9800 GTX+, in average, by a factor of 5.7. For big configurations, with more than 8,000 spins, 295 GTX outperforms the 9800 GTX+ by a factor of 14. With more processors available, the 295 GTX can execute more blocks simultaneously and thus reduce the computation time.

Table 2 shows the values of energy obtained by both CPU and GPU at the end of all simulations. A small difference, around 5%, in average, can be observed between values computed by GPU and CPU. This difference can be caused by the use of single-precision values by the GPU code. We believe the use of double-precision arithmetic can reduce this error.

6 Related Works

Several proposals to reduce the total time in Monte Carlo simulations can be found in the literature. An approach based in spin clusters flip was introduced by Swendsen and Wang [11]. Recently, Fukui and Todo [12] achieved an interesting result developing an $O(N)$ algorithm using this idea. A parallel version of the Monte Carlo method with Ising spin model was proposed by [13]. A theoretical study of magnetic nanotube properties using a model similar to the described in this paper can be found in [6].

The intrinsic parallelism presented by GPUs contributes positively for modeling systems with high computational complexity. The GPU cards are widely used to perform physical simulations like: fluid simulation [14], particle simulation [15], molecular dynamics [16], interactive deformable bodies [17], and so on.

Tomov *et al.* [18] developed a GPU based version of Monte Carlo method using the Ising model for ferromagnetic simulations. In the Ising spin model [5], the spin can adopt only two directions: ± 1. The Heisenberg model used in this work is much less restrictive and provides more realistic numerical results. Tomov *et al.* did not use a long range interaction, while our work uses this interaction. Although Tomov *et al.* have implemented a simpler model, they have not obtained an good speedup: a speedup of three times was achieved when using their parallel GPU version.

Another interesting GPU version for the Ising model was proposed by Preis *et al.* [4]. They performed simulations using 2D and 3D systems and obtained results 60 and 35 times faster, respectively, when compared to the CPU version. Also, in this work, the long range factor was not used. We include the long range dipole-dipole term (Eq. 1) because of its effects on the phase of the system. As our experiments have shown, even using a complex model, our implementation was very effective in improving performance.

7 Conclusions and Future Works

In this work, we presented an algorithm to simulate the Heisenberg model with long range interaction using GPGPUs.

In order to evaluate our simulator, we compared the new implementation using CUDA with both multithreaded and sequential versions. The results reveal that our CUDA version was responsible for a significant improvement in performance. The gains due to the optimizations presented along this paper were very expressive, yielding speedups up to 166 when using a 240 stream processors GPU. Although the speedup obtained was respectable, we believe that we could achieve better speedups if larger system configurations were used.

One important factor that have contributed to the expressive results we have obtained was our observation that the energy of each atom could be computed independently. Thus, the energy of distinct regions of the space could also be computed independently. So, in order to increase the performance, multiple threads could be issued to compute the energy for each part of the space containing the ferromagnetic object.

Finally, for the best of our knowledge, we are the first to propose an automatic generation, at run-time, of the execution configuration of a GPGPU kernel. For this purpose, the number of spins in the system, as well as the total amount of memory each thread uses, are taken into account to calculate the execution configuration.

The techniques presented in this paper are not restrict to simulate the Heisenberg model with long range interaction. We believe that they can be applied to improve the performance of any GPGPU-based application. In the future, the ideas behind the algorithm that performs the automatic generation of the execution configuration can be part of the CUDA compiler and/or its run-time system. Some additional research must be done to verify whether the configuration obtained is the optimum one or not. We plan to investigate this too.

As future works, we also intend to study the impact of different geometries, such as sphere, cylinder and spherical shell, in performance. We also plan to extend our work to use a cluster of GPGPUs. A cluster of GPGPUs is necessary because the physicists are interested in analyzing systems composed by a huge number of spins. At the moment, we can deal with almost 50,000 spins, but we plan to deal with millions of them.

Acknowledgment

The authors thank to CAPES (*Coordenação de Aperfeiçoamento de Pessoal de Ensino Superior*), FAPEMIG (*Fundação de Amparo à Pesquisa do Estado de Minas Gerais*), CNPq (*Conselho Nacional de Desenvolvimento Científico e Tecnológico*) and UFJF for funding this research.

References

1. Heisenberg, W. J. Phys **49** (1928) 619
2. Peçanha, J., Campos, A., Pampanelli, P., Lobosco, M., Vieira, M., Dantas, S.: Um modelo computacional para simulação de interação de spins em elementos e compostos magnéticos. XI Encontro de Modelagem Computacional (2008)
3. Blelloch, G., Narlikar, G.: A practical comparison of n-body algorithms. In: Parallel Algorithms. Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society (1997)
4. Preis, T., Virnau, P., Paul, W., Schneider, J.J.: Gpu accelerated monte carlo simulation of the 2d and 3d ising model. Journal of Computational Physics **228**(12) (2009) 4468 – 4477
5. Ising, E.: Beitrag zur Theorie der Ferromagnetismus. Z. Physik **31** (1925) 253–258
6. Konstantinova, E.: Theoretical simulations of magnetic nanotubes using monte carlo method. Journal of Magnetism and Magnetic Materials **320**(21) (2008) 2721 – 2729
7. NVIDIA: Nvidia cuda programming guide. Technical report, NVIDIA Corporation (2007)
8. Metropolis, N., Rosenbluth, A.W., Rosenbluth, M.N., Teller, A.H., Teller, E.: Equation of state calculations by fast computing machines. Journal of Chemical Physics **21** (1953) 1087–1092
9. Matsumoto, M., Nishimura, T.: Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. Model. Comput. Simul. **8**(1) (1998) 3–30
10. Nyland, L., Harris, M., Prins, J.: Fast n-body simulation with cuda. In Nguyen, H., ed.: GPU Gems 3. Addison Wesley Professional (August 2007)
11. Swendsen, R.H., Wang, J.S.: Nonuniversal critical dynamics in monte carlo simulations. Physical Review Letters **58**(2) (January 1987) 86+
12. Fukui, K., Todo, S.: Order-n cluster monte carlo method for spin systems with long-range interactions. Journal of Computational Physics **228**(7) (2009) 2629 – 2642
13. Santos, E.E., Rickman, J.M., Muthukrishnan, G., Feng, S.: Efficient algorithms for parallelizing monte carlo simulations for 2d ising spin models. J. Supercomput. **44**(3) (2008) 274–290

14. Harada, T., Tanaka, M., Koshizuka, S., Kawaguchi, Y.: Real-time particle-based simulation on gpus. In: SIGGRAPH '07: ACM SIGGRAPH 2007 posters, New York, NY, USA, ACM (2007) 52
15. Kipfer, P., Segal, M., Westermann, R.: Uberflow: a gpu-based particle engine. In: HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, New York, NY, USA, ACM Press (2004) 115–122
16. Yang, J., Wang, Y., Chen, Y.: Gpu accelerated molecular dynamics simulation of thermal conductivities. J. Comput. Phys. **221**(2) (2007) 799–804
17. Georgii, J., Echtler, F., Westermann, R.: Interactive simulation of deformable bodies on gpus. In: Proceedings of Simulation and Visualisation 2005. (2005) 247–258
18. Tomov, S., McGuigan, M., Bennett, R., Smith, G., Spiletic, J.: Benchmarking and implementation of probability-based simulations on programmable graphics cards. Computers and Graphics **29**(1) (2005) 71 – 80