Computer Physics Communications 182 (2011) 1265–1271

Contents lists available at ScienceDirect

Computer Physics Communications

www.elsevier.com/locate/cpc

Benchmarking GPU and CPU codes for Heisenberg spin glass over-relaxation

M. Bernaschi [a,*], G. Parisi [b], L. Parisi [a]

[a] *Istituto Applicazioni Calcolo, CNR, Viale Manzoni, 30 - 00185 Rome, Italy*
[b] *Physics Department, University of Rome "La Sapienza", P.le A. Moro, 2 - 00185 Rome, Italy*

ARTICLE INFO

Article history:
Received 29 August 2010
Received in revised form 23 December 2010
Accepted 27 February 2011
Available online 3 March 2011

Keywords:
Spin systems
GPU
Vector processing

ABSTRACT

We present a set of possible implementations for Graphics Processing Units (GPU) of the Over-relaxation technique applied to the 3D Heisenberg spin glass model. The results show that a carefully tuned code can achieve more than 100 GFlops/s of sustained performance and update a single spin in about 0.6 nanoseconds. A multi-hit technique that exploits the GPU shared memory further reduces this time. Such results are compared with those obtained by means of a highly-tuned vector-parallel code on latest generation multi-core CPUs.

© 2011 Elsevier B.V. All rights reserved.

1. Introduction

In spite of the availability of high performance multi-core systems based on traditional architectures, there is recently a renewed interest in floating point *accelerators* and *co-processors* that can be defined as devices that carry out arithmetic operations concurrently with or in place of the CPU. Among the solutions that have received more attention from the high performance computing community there are the NVIDIA Graphics Processing Units (GPU), originally developed for video cards and graphics, since they are able to support very demanding computational tasks. As a matter of fact, astonishing results have been reported by using them for a number of applications covering, among others, atomistic simulations, fluid-dynamic solvers and option pricing. Simulations of statistical mechanics systems based on Monte Carlo techniques are another example of applications that may benefit of the GPU computing capabilities. In the present work we report the results obtained by following different approaches for the implementation on GPU of the Over-relaxation technique [1] for a typical statistical mechanics system: the classic Heisenberg spin glass model. For a fair comparison, we developed also a highly tuned vector-parallel implementation for standard multi-core architectures that we tested on three high-end CPUs. All source codes we developed are available for inspection and further tests. Our results confirm that GPUs may provide a sizeable advantage with respect to traditional architectures for this kind of simulations.

The paper is organized as follows: Section 2 contains a short introduction to the features of spin systems that are of interest from the computational viewpoint; Section 3 summarizes the main features of the GPUs used for the experiments; Section 4 describes the different GPU and CPU implementations of the 3D Heisenberg spin glass model; Section 5 presents the performances obtained. Section 6 concludes with a summary of the main results and a perspective about possible future activities in this field.

2. Spin systems

In statistical mechanics "spin system" indicates a broad class of models used for the description of a number of physical phenomena. Although apparently quite simple, the study of spin systems is by no means trivial and most of the times numerical simulations (very often based on Monte Carlo methods) are the only way to understand their behavior. A spin system is usually described by its Hamiltonian which has the following general form

$$H = -\sum_{i \neq j} J_{ij}\sigma_i\sigma_j. \tag{1}$$

The spins are defined on a *lattice* which may have one, two, three or even a higher number of dimensions. The sum in Eq. (1) runs usually on the first neighbors of each spin (2 in 1D, 4 in 2D and 6 in 3D). The spins σ and the couplings J may be either discrete or continuous and their values determine the specific model. In the present work we focus on the Heisenberg spin glass model where σ_i is a 3-component vector such that $\vec{\sigma_i} \in \mathbb{R}^3$, $|\sigma_i| = 1$ and J_{ij} is Gaussian distributed with average value equal to 0 and variance equal to 1.

In a 3-dimensional system of size L^3, the contribution to the total energy of the spin $\vec{\sigma_i}$ with coordinates x, y, z such that $i = x + y \times L + z \times L^2$ is

* Corresponding author.
E-mail address: m.bernaschi@iac.cnr.it (M. Bernaschi).

0010-4655/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.cpc.2011.02.012

Table 1
Main features of the NVIDIA GPUs used for the experiments.

GPU model	Tesla C1060	Tesla C2050	GTX 480
Number of multiprocessors	30	14	15
Number of cores	240	448	480
Shared memory per multiprocessor (in bytes)	16 384	49 152	49 152
L1 cache (in bytes)	N/A	16 384	16 384
L2 cache (in Kbytes)	N/A	768	768
Number of registers per multiprocessor	16 384	32 768	32 768
Max number of thread per block	512	1024	1024
Clock rate	1.3 GHz	1.15 GHz	1.4 GHz
Memory bandwidth	102 GB/s	144 GB/s	177 GB/s
Error Checking and Correction (ECC)	No	Yes	No

$$J_{x+1,y,z}\vec{\sigma}_{x,y,z}\cdot\vec{\sigma}_{x+1,y,z} + J_{x-1,y,z}\vec{\sigma}_{x,y,z}\cdot\vec{\sigma}_{x-1,y,z} + J_{x,y+1,z}\vec{\sigma}_{x,y,z}\cdot\vec{\sigma}_{x,y+1,z} + J_{x,y-1,z}\vec{\sigma}_{x,y,z}\cdot\vec{\sigma}_{x,y-1,z} + J_{x,y,z+1}\vec{\sigma}_{x,y,z}\cdot\vec{\sigma}_{x,y,z+1} + J_{x,y,z-1}\vec{\sigma}_{x,y,z}\cdot\vec{\sigma}_{x,y,z-1} \quad (2)$$

where $\cdot$ indicates the scalar product of two $\vec{\sigma}$ vectors. In most Monte Carlo techniques used for the simulation of the Heisenberg spin glass model (Metropolis, *Heat Bath*, *etc.*) it is necessary to evaluate the expression in Eq. (2) for each spin. The main goal of the present work is to describe several possible approaches and to assess what is the most effective scheme for computing this expression on a GPU. As a consequence we are not going to address other issues, like the generation of random numbers, even if we understand their importance for an efficient GPU based simulation of spin systems, because they are already faced in other studies [2]. Actually, other authors already described efficient techniques for the simulation, on GPU, of spin systems (*e.g.*, [3,4] for the Ising model in 2D and 3D and [5] for the three-dimensional Heisenberg anisotropic model). However their analysis appears somehow limited since they present results basically for a single implementation whereas a GPU offers several alternatives for an effective implementation that deserve to be considered and analyzed.

3. Graphic processing unit and CUDA

In Table 1 we report the key aspects of the three GPUs we used for our numerical experiments: a Tesla C1060, a Tesla C2050 and a GTX 480. The C2050 and the GTX 480 are based on the latest architecture ("Fermi") recently introduced by NVIDIA.

The memory hierarchy is one of the most distinguish features of the NVIDIA GPUs. The different levels of memory can be briefly described as follows:

- *global* memory (DRAM): this is the main memory of the GPU and any location of it is visible by any thread. The bandwidth between the *global* memory and the multiprocessors is more than 100 GB/s but the latency for the access is also large (approximately 300–400 cycles);
- *shared* memory: access to data stored in the shared memory has a latency comparable to the latency of the registers (if there are not *bank conflicts*). However, shared memory variables are *local* to the threads running within a single multiprocessor and the size of the shared memory is tiny compared to the global memory that is, usually, in the range of GBytes;
- *registers*: on a GPU there are thousands of 32 bits registers. It is worth noting that, for each multiprocessor, there is more space for data in the registers than in the shared memory.
- *cache*: L1 and L2 caches have been included in the Fermi architecture. Currently, on each multiprocessor, there are 64 KB of private L1 cache that can be split, at run time, in a 48 KB shared memory and a 16 KB L1 cache or in a 16 KB shared memory and a 48 KB L1 cache. The L2 cache is shared among all multiprocessors and its size is 768 KB.

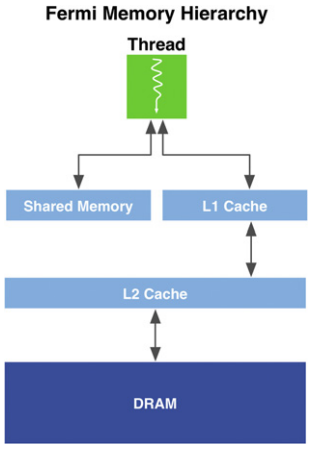

Fig. 1. Memory hierarchy in the Fermi architecture.

- *constant* and *texture*: these are special memories used respectively to store constant values and to cache global memory (separate from register and shared memory) offering dedicated interpolation hardware separate from the thread processors.

Fig. 1 summarizes the memory hierarchy of a GPU that implements the Fermi architecture.

Data placement in the global or shared memory can be controlled explicitly and this, as shown in Section 5, makes a significant difference from the performance viewpoint.

For the GPU programming, we employed the version 3.0 of the CUDA Software Development Toolkit that offers an *extended* C compiler and is available for all major platforms (Windows, Linux, Mac OS). The extensions to the C language supported by the compiler allow starting computational kernels on the GPU, copying data back and forth from the CPU memory to the GPU memory and explicitly managing the different types of memory available on the GPU (with the notable exception of the caches). The programming model is defined by NVIDIA as Single Instruction Multiple Threads (SIMT). Each multiprocessor is able to perform the same operation on different data 32 times, so the basic computing unit (called *warp*) consists of 32 threads. To ease the mapping of data to threads, the threads identifiers may be multidimensional and, since a very high number of threads run in parallel, CUDA groups threads in *blocks* and *grids*.

One of the crucial requirements to achieve a good performance on the NVIDIA GPU is to hide the high latency of the *global* memory accesses (both read and write) by following a set of rules that depend on the specific level of the architecture. The respect of such rules enables to obtain what is called, in CUDA jargon, "coalesced" accesses. Also important is to avoid running out of registers since registers-spilling, although supported, has a very high cost.

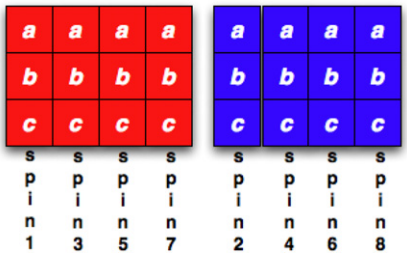

Fig. 2. Memory data layout of the spins. Memory ordering is row-major.

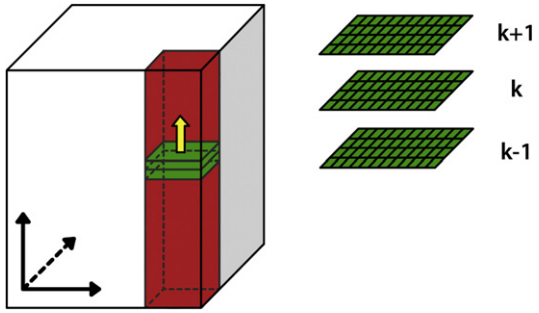

Fig. 3. The "three-planes" mechanism.

Further information about the features of the NVIDIA GPU and the CUDA programming technology can be found in [6].

4. GPU and CPU implementations

Our performance metrics is the time required for the update of a single spin by using the so-called Over-relaxation method [1] in which:

$$\overrightarrow{\sigma_{new}} = 2(\overrightarrow{H_\sigma} \cdot \vec{\sigma}_{old}/\overrightarrow{H_\sigma} \cdot \overrightarrow{H_\sigma})\overrightarrow{H_\sigma} - \overrightarrow{\sigma_{old}}$$

is the maximal move that leaves the energy invariant, so that the change is always accepted. For $\vec{\sigma}$ defined in $[x, y, z]$,

$$\begin{aligned}\overrightarrow{H_\sigma} = {} & J_{[x+1,y,z]}\vec{\sigma}_{[x+1,y,z]} + J_{[x-1,y,z]}\vec{\sigma}_{[x-1,y,z]} \\ & + J_{[x,y+1,z]}\vec{\sigma}_{[x,y+1,z]} + J_{[x,y-1,z]}\vec{\sigma}_{[x,y-1,z]} \\ & + J_{[x,y,z+1]}\vec{\sigma}_{[x,y,z+1]} + J_{[x,y,z-1]}\vec{\sigma}_{[x,y,z-1]}. \end{aligned} \quad (3)$$

Albeit the dynamics is micro-canonical, the Over-relaxation is very effective when used in combination with an irreducible dynamics such as the *Heat Bath*. The main reason for choosing the Over-relaxation update as a benchmark is that it does not require random numbers. Moreover, it requires only simple floating point arithmetic operations (20 sums, 3 differences and 28 products) plus a single division. Since there are no random numbers involved, the Over-relaxation update can be hardly considered a Monte Carlo method but nevertheless it provides very useful indications because its *core* is the evaluation of the expression in formula (3) that is used by most Monte Carlo methods as well. Likewise Monte Carlo methods, the Over-relaxation update can be carried out in parallel only on non-interacting spins. To this purpose a check-board decomposition can be applied similar to that used for vector processors [7]. This technique has been already implemented also for GPUs in [3].

4.1. GPU implementations

As already mentioned in Section 3, a careful data organization is fundamental to achieve good performances from a GPU. According to the *best practices* in CUDA programming, it is better to organize data as *structures of arrays*. In the present case this indication leads to split the spins in two non-interacting subsets (let us call them *red* and *blue* spins), and then store the spins in three distinct arrays, one for each spin component (see Fig. 2). The same data layout can be used for the couplings. Besides that, since the shared memory of the GPUs is very fast, it would look reasonable to use it for storing spins and the couplings among them. Unfortunately, the total size of the shared memory (corresponding to the sum of the shared memory of all GPU multiprocessors) is limited (well below 1 MB even on the latest generation GPUs) and only very small systems could fit completely in it. For the three-dimensional Heisenberg model, six floating point numbers per lattice point are required (three for the components of the spin and three for the couplings). In single precision, six floating point numbers occupy 24 bytes. With a total size of the shared memory in the range of 0.7 MB, only ∼22 000 lattice points could be stored corresponding to a linear size $L < 30$. As a consequence, the simulation of systems with $L \geqslant 32$ requires a "swapping" mechanism. A similar problem arises trying to use a GPU to solve a Laplace equation (the Laplace equation may be solved by using the Jacobi scheme that requires the evaluation of an expression very similar to that shown in formula (2)). In this context a quite elegant solution has been proposed in [8] and [9] where only three planes are stored in shared memory. The three planes serve as a cyclic buffer. At each step along the Z-direction a new XY-plane enters in the buffer replacing the plane that does not serve to update the current plane. Such a scheme is shown in Fig. 3. However, this scheme when applied in such a simple way to a spin system has a major drawback since on each plane only half of the spins can be updated concurrently (due to the constraint of updating only non-interacting spins), so, half of the threads are idle waiting their turn after the loading of the spins. If a thread loads both a red and a blue spin, it is never idle (the same thread updates a red and then a blue spin) but the total number of active threads remains the same with the additional drawback that a single thread needs to access the global memory twice. To overcome this limitation, we developed a new scheme which makes use of four planes instead of three. In this scheme two consecutive planes (say planes k and $k+1$) are updated concurrently in two substeps:

- *substep 1*: the red spins of plane k and plane $k+1$;
- *substep 2*: the blue spins of plane k and plane $k+1$.

Fig. 4 shows a simplified case with only 16 threads. With this scheme two planes are replaced at the end of each step along the Z-direction and each step increases Z of 2 units. A multi-hit variant of the four-planes scheme has been developed as well. The multi-hit version allows an indirect measure of the overhead due to the initial loading of spins and couplings (see Section 5).

In both shared memory versions, there is an additional issue that is worth to be discussed. To update the spins on the boundaries among the subdomains, spins from neighboring subdomains are needed. Since no synchronization is possible among thread blocks, there is no way to guarantee that, within the same iteration, the spins of the other subdomains have been already updated or not (both conditions, updated or not, would be acceptable, it is the mix of updated and not updated spins that breaks the algorithm). To solve the problem, for the spins on the boundaries among the subdomains, the update is carried out by alternating between red and blue spins every two iterations. That is, on even iterations, only red spins on the boundaries are updated by freezing the value of the blue spins. Then, on odd iterations, only blue spins are updated (by freezing the value of the red spins). The global synchronization in the end of each iteration guarantees the consistency of this procedure.

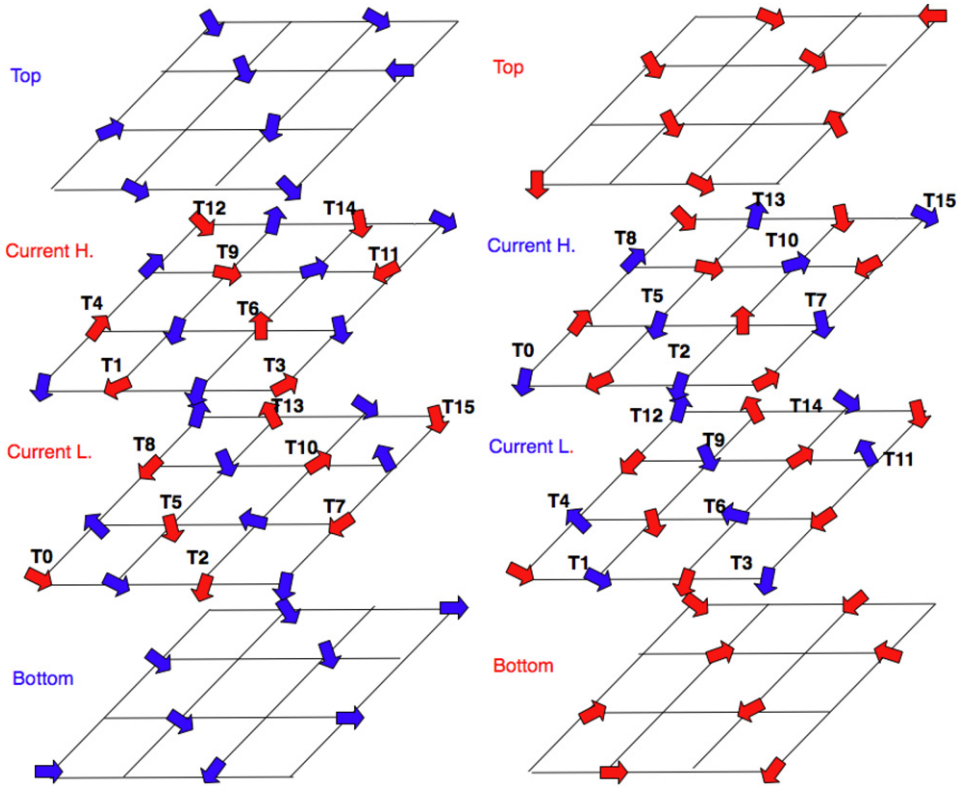

Fig. 4. The "four-planes" mechanism. In the first substep (left panel) red spins of the planes "Current H" and "Current L" are updated. In the second substep (right panel) blue spins of the same planes are updated. (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.)

Table 2
Main features of the multi-core CPUs used for the tests. The memory bandwidth shown is the "Triad" result from the *stream* benchmark.

	Intel X5680	AMD 6174	IBM Power7
Clock	3.3 GHz	2.2 GHz	3.3 GHz
# of threads	12	24	32
L1 cache	$2 \times 6 \times 32$ KB	$2 \times 12 \times 64$ KB	$2 \times 8 \times 32$ KB
L2 cache	6×256 KB	12×512 KB	8×256 KB
L3 cache	12 MB "Smart"	12 MB	8×4 MB "Fluid"
Vector instructions	SSE4.2	SSE3 (SSE4?)	VMX/AltiVec
Memory B/W	44.2 GB/s	35.6 GB/s	45.6 GB/s

For the Fermi architecture a further scheme has been implemented to measure the advantage provided by the cache. This scheme is quite similar to the three-plane shared memory scheme, so we update all the red spins of a plane and then all the blue spins of the same plane. If the cache works as expected, three planes should be loaded when the red spins are updated. Then for the update of the blue spins no new data should be loaded. Finally, a version where the loading of data from the *global* memory is replaced with texture fetches has been also developed. Texture memory provides cached read-only access that is optimized for spatial locality and it should prevent redundant loads from global memory. When multiple blocks request the same region, the data are loaded from the cache. We wanted to test whether the texture helps with a memory access pattern like that required for the evaluation of the expression in formula (2).

4.2. Multi-core CPUs

To compare in a fair way the GPU performances with those achievable on a standard architecture, we implemented also a highly tuned CPU version that we tested on three latest generation multi-core CPUs whose features are reported in Table 2. We used different C compilers on these three architectures (Intel *icc 11.1*, PGI 10.6, xlC 11.1, respectively on the Intel, AMD and Power7) but the same source code. The main loop of the update procedure has been carefully modified so that the compilers may exploit the vector instructions available on the three architectures. Multiple cores are activated by resorting to *OpenMP* directives supported by all the three compilers we tested.

5. Results and discussion

All the tests have been carried out for a cubic lattice with periodic boundary conditions along the X-, Y- and Z-directions. The indexes for the access to the data required for the evaluation of the expression (2) are computed in accordance with the assumption that the linear size L of the lattice is a power of 2. In this way bitwise operations and additions suffice to compute the required indexes with no multiplications, modules or other costly operations. All technical details about the implementation of the different approaches can be found by looking directly at the source code available from http://www.iac.rm.cnr.it/~massimo/hsgfiles.tgz. Most of the tests have been carried out on a lattice with linear size $L = 128$ but we tested that there are no significant differences on larger lattices (*e.g.*, for $L = 256$). The time we report is in nanoseconds and corresponds to the time required to update a single spin. It is important to note that the order in which the spins are updated changes depending on the different imple-

Table 3
Timings for simulating a single Heisenberg spin glass system with continuous (Gaussian distributed) isotropic couplings. The lattice size is set equal to $L = 128$. T_{upd} is the time in nanoseconds to process 1 spin. We ran 100 iterations and report the total time divided by the number of iterations and then divided by the number of spins in the system.

Platform	Number of threads	T_{upd}
Tesla C1060 GM	320	1.9 ns
Tesla C1060 CA	320	2.0 ns
Tesla C1060 SM	128	2.5 ns
Tesla C1060 SM4P	64	2.2 ns
Tesla C1060 TEXT	320	1.8 ns
GTX 480 (Fermi) GM	320	0.66 ns
GTX 480 (Fermi) CA	320	0.70 ns
GTX 480 (Fermi) SM	256	1.3 ns
GTX 480 (Fermi) SM4P	192	0.86 ns
GTX 480 (Fermi) TEXT	480	0.63 ns
Intel X5680 (SSE instr.)	1	$\sim$ 13.5 ns
Intel X5680 (SSE instr. + OpenMP)	8	$\sim$ 3.4 ns

mentations. For instance, in the global memory based versions, all red spins of the lattice are updated before updating blue spins whereas in the shared memory based versions, all spins in a plane (with the *caveat*, described in Section 4.1, for the spins in the boundaries among the subdomains) are updated before proceeding to the next plane. However, the correctness of all implementations is confirmed by the fact that the energy remains the same (as expected since the dynamics of the Over-relaxation process is micro-canonical) even if the spin configuration changes step-by-step. Most of the tests have been carried out in single precision with the exception of those whose results are reported in Table 7.

The main results are reported in Table 3. GM stands for Global GPU Memory. This is the most simple version which starts a very large number of threads and blocks by exploiting the fact that the Over-relaxation GPU kernel we wrote needs very few registers. The GPU kernel updates all red spins and then all blue spins. The synchronization required between the updating of red and blue spins is guaranteed by two distinct kernel invocations. To reduce the corresponding overhead, we could resort to a different synchronization mechanism although the lack of a native mechanism to synchronize, within a single kernel, the thread-blocks each other makes a bit tricky any alternative approach.

SM stands for Shared GPU Memory. This is the version which keeps three planes in shared memory and updates before red and then blue spins of a plane before proceeding to the next plan. SM4P stands for Shared Memory version with 4 planes in memory. The difference with the SM version is that we update two planes concurrently. For the two shared memory versions (SM and SM4P) the number of threads is determined implicitly by the size of the available shared memory (so it varies between the Tesla C1060 and the GTX 480 or the C2050).

CA stands for "Cache". This is the version developed for the Fermi architecture where a 48 KB cache is available on each MultiProcessor. The results in the table have been obtained by using the *hint* for L1 cache that sets it equal to 48 KB per each GPU MultiProcessor. We tested this version also on the Tesla architecture, although the Tesla does not have a cache, just to measure the overhead introduced by processing a single plane in each kernel. The TEXT implementation is very similar to the GM one but the fetches from the Global Memory are replaced by *texture* fetches.

We also measured the *overhead* of invoking a CUDA kernel. This is basically independent on the method and represents a lower bound of the time that the update of a spin requires. The value we found is 0.1 ns per spin for a 128^3 lattice. Such value is clearly volume-dependent and decreases by increasing the lattice size.

In Table 4 we report the results of a test in which the spins of the version SM4P are updated multiple times ("multi-hit") before moving to the next two planes. Since the updates after the first one do not require new accesses to the global memory (all data are already in the shared memory) this technique offers the chance to measure the overhead introduced by the initial loading of the data as the difference between the total time required by the single-hit and the time required by an infinite number of hits. Obviously it is not possible to carry out an infinite number of hits but it is easy to extrapolate the corresponding value by using a simple fit. The time we found is $\sim$0.6 ns for both the compilation setups used for the tests. It is worthy of note also the sizeable effect of the two additional compiler options that force, respectively, denormalized numbers to zero and the use of a less precise division. In general, for an optimal tuning, it is absolutely necessary to take into account the specific features of each implementation of the CUDA architecture like the different size of the shared memory or the new integer multiplication capability. For instance, if on the Fermi architecture the multiplication between two integers is carried out by using the `__mul24` (as it was suggested on the previous generation GPUs when the result of the multiplication fitted in 24 bits) there is a penalty of about 5% for our code.

Table 4
Timings with respect to the number of hits for the SM4P version on a GTX 480. Timings are reported for the compilation with two different setups ("-O3" for standard optimizations, central column) and with the additional options "-ftz=true -prec-div=false" (right column) that, respectively, force denormalized numbers to zero and the usage of a less precise division. The test case is the same as in Table 3.

Number of hits	T_{upd} -O3	T_{upd} -O3 -ftz=true -prec-div=false
1	0.911 ns	0.846 ns
2	0.606 ns	0.519 ns
5	0.429 ns	0.335 ns
10	0.373 ns	0.280 ns
∞ (extrapolated)	0.313 ns	0.246 ns

If we consider the time (0.63 ns per spin update) required by the best "single-hit" technique that makes use of the texture, and use the estimate of 60 floating point operations per spin update (that does not take into account the index arithmetics), we obtain a sustained performance very close to 100 GFlops. However, the most interesting observation is that, despite the much higher latency of the global memory, the simple versions (the GM one and the TEXT one) that use neither the shared memory nor the cache (where available) are significantly faster unless a "multi-hit" variant of the shared memory version is used. A possible explanation of this behavior is that the limited size of the shared memory (and of the cache) allows to start fewer threads with respect to the case of a global memory based version where the only limitation is the number of registers available on each multiprocessor. Moreover, although the shared memory version allows to reduce the number of global memory transactions when loading data (spins and couplings), it offers no advantage when the updated spins are stored back in global memory. The advantage of using texture fetches instead of simple load operations from the global memory is quite limited (about 5%). However, only very minor changes are required to the code to take this advantage so it is worth it. It is more puzzling that, apparently the cache does not offer advantages at all. A possible explanation is that the overhead introduced by calling the computational kernel a number of times equal to twice the linear dimension of the lattice (instead of calling it just twice as we do in the GM or TEXT version) is as large and possibly larger than the saving in time that the cache may offer. The requirement of calling the kernel multiple times arises because on each plane it is necessary to update all red spins and then all blue spins. So, for each step in the Z-direction, two invocations are required.

As shown in Table 5, the Error Checking and Correction introduces a sizeable overhead for the methods working in global memory (GM, TEXT, CA). For reasons that we could not determine,

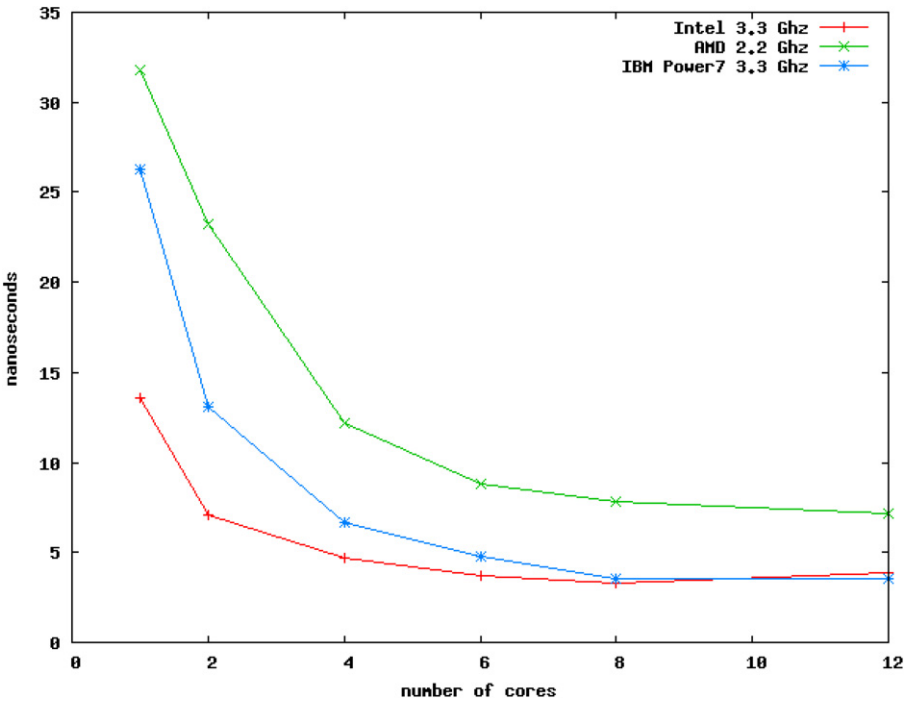

Fig. 5. Best spin update time on multi-core CPUs. Lattice size is 128.

Table 5
Timings with and without Error Checking and Correction on a C2050 (the GTX 480 does not have ECC). The test case is the same as in Table 3.

Platform	T_{upd} ECC on	T_{upd} ECC off
C2050 (Fermi) GM	1.0 ns	0.84 ns
C2050 (Fermi) CA	1.1 ns	0.87 ns
C2050 (Fermi) SM	1.63 ns	1.66 ns
C2050 (Fermi) SM4P	1.4 ns	1.3 ns
C2050 (Fermi) TEXT	1.0 ns	0.78 ns

Table 6
Timings for spin update on a single core without and with vectorization. Lattice size is 128^3.

Platform	T_{upd} scalar code	T_{upd} vectorized code
Intel X5680 3.33 GHz	32.6 ns	13.5 ns
IBM Power7 3.33 GHz	30.5 ns	26.3 ns
AMD 6174 2.2 GHz	76 ns	31.8 ns

the methods working in shared memory undergo a much more limited penalty when the ECC is on.

The results for multi-core CPUs are summarized in Fig. 5 which shows the best results obtained on each platform by running the vectorized version on multiple cores. The Intel CPU performs remarkably better than the AMD and Power7 processors specially on one and two cores. However, the IBM Power7 scales much better than both Intel and AMD. Both Intel and AMD get a very high performance boost by the vectorization. This is apparent by looking at Table 6, whereas the vectorization improves marginally the performances of the Power7. We recall that we used *exactly* the same code on the three platforms and checked that the compilers had vectorized the main loop.

All timings reported so far are for the single-precision version of the codes. Table 7 shows a subset of the results obtained with a double precision version of the same codes. In general the time (approximately) doubles for both GPU and CPU going from single to double precision. However the shared memory version for the GPU performs worse (the time increases of a factor 2.6) whereas the Power7 performs better (the time increases of a factor 1.3 only).

Table 7
Single (s.p.) vs. double precision (d.p.) timings.

Platform	Number of threads	T_{upd} (s.p.)	T_{upd} (d.p.)
GTX 480 (Fermi) GM	320	0.66 ns	1.3 ns
GTX 480 (Fermi) CA	320	0.70 ns	1.5 ns
GTX 480 (Fermi) SM4P	192 (s.p.) 96 (d.p.)	0.86 ns	2.2 ns
Intel X5680 3.33 GHz	1	13.5 ns	26.6 ns
Intel X5680 3.33 GHz	8	3.5 ns	6.8 ns
Power7 3.33 GHz	1	26.3 ns	36.3 ns
Power7 3.33 GHz	8	3.5 ns	5.6 ns

6. Conclusions

We have presented the results obtained with different GPU and CPU implementations of the Over-relaxation algorithm for the 3D Heisenberg spin glass model. We showed how only the multi-hit variant of a sophisticated shared memory based version outperforms a much more simple implementation that uses only global memory. Moreover we showed that, at least for the specific case we studied, the cache introduced in the Fermi architecture offers limited advantages.

Besides that, it is worth to highlight that: (i) a well tuned GPU implementation can achieve a significant speedup with respect to a well tuned vector–multi-core implementation (0.63 ns per spin update for the TEXT version on the GTX 480 *vs.* 3.5 ns per spin update on a high-end multi-core Intel *X5680*; (ii) the performances of the GPUs keep on to increase significantly in a pretty short time (the performances of the new architecture are double and in some cases even more than double with respect to the previous generation cards that are only two years old) making it a very interesting platform for the numerical simulation of spin systems; (iii) vectorization is absolutely required to get the best performances on standard multi-core CPU but it is far from being easy to write code that compilers consider “safe” for auto-vectorization.

For the future, we expect to extend our tests to multi-GPU implementations and to mixed (OpenMP/MPI) implementations for multi-core CPUs.

Acknowledgements

We thank Luis Antonio Fernandez, Filippo Mantovani, Victor Martin-Mayor, Sergio Perez, Fabio Schifano and Raffaele Tripiccione for useful discussions.

We thank Massimiliano Fatica for the *hint* about the three-planes technique and for running preliminary tests on Fermi architecture based GPUs.

We thank Luca Leuzzi and Edmondo Silvestri for the access to their GTX 480 GPU.

We thank Giorgio Richelli for the access to the IBM Power7 platform.

Special thanks to José Manuel Sanz González for providing us with his program for simulating the Heisenberg spin glass (including anisotropic couplings) and sharing his preliminary performance results.

References

[1] S. Adler, Over-relaxation method for the Monte Carlo evaluation of the partition function for multiquadratic actions, Phys. Rev. D 23 (1981) 2901–2904.
[2] L. Hower, D. Thomas, Efficient random number generation and application using CUDA, http://http.developer.nvidia.com/GPUGems3/gpugems3_ch37.html.
[3] 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 (2009) 4468–4477.
[4] M. Weigel, Simulating spin models on GPU, preprint arXiv:1006.3865, 2010.
[5] J.M. Sanz González, Heisenberg spin glass on graphic cards: preliminary tests, private communication.
[6] NVIDIA CUDA compute unified device architecture programming guide, http://www.nvidia.com/cuda.
[7] D. Landau, Vectorisation of Monte Carlo programs for lattice models using supercomputers, in: The Monte Carlo Method in Condensed Matter Physics, Top. Appl. Phys. 71 (1992).
[8] M. Giles, Jacobi iteration for a Laplace discretisation on a 3D structured grid, http://people.maths.ox.ac.uk/~gilesm/codes/laplace3d/laplace3d.pdf.
[9] M. Fatica, E.H. Phillips, Implementing the Himeno benchmark with CUDA on GPU clusters, in: IEEE International Parallel & Distributed Processing Symposium, 2010.