Available on-line at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Multi-Kepler GPU *vs.* Multi-Intel MIC for spin systems simulations

M. Bernaschi[a,*], M. Bisson[a], F. Salvadore[a,b]

[a]Istituto Applicazioni Calcolo, CNR, Viale Manzoni, 30 - 00185 Rome, Italy
[b]CINECA, via dei Tizii, 6 - 00185 Rome, Italy

Abstract

We present and compare the performances of two *many-core* architectures: the Nvidia *Kepler* and the Intel *MIC* both in a single system and in cluster configuration for the simulation of spin systems. As a benchmark we consider the time required to update a single spin of the 3D Heisenberg spin glass model by using the Over-relaxation algorithm. We present data also for a traditional high-end multi core architecture: the Intel *Sandy Bridge*. The results show that although on the two Intel architectures it is possible to use basically the same code, the performances of a Intel *MIC* change dramatically depending on (apparently) minor details. Another issue is that to obtain a reasonable scalability with the Intel *Phi* coprocessor (*Phi* is the coprocessor that implements the MIC architecture) in cluster configuration it is necessary to use the so-called *offload* mode which reduces the performances of the single system. As to the GPU, the *Kepler* architecture offers a clear advantage with respect to the previous *Fermi* architecture maintaining exactly the same source code. Scalability of the multi-GPU implementation remains very good by using the CPU as a communication co-processor of the GPU. All source codes are provided for inspection and for double-checking the results.

Project ID: PA1727

1. Introduction

In recent papers [1] [2] [3] we presented different techniques for single and multi-GPU implementation of the Over-relaxation technique [4] applied to a typical statistical mechanics system: the classic Heisenberg spin glass model. The main target of those works was the Nvidia *Fermi* architecture. Besides us, other authors [5] [6] confirmed that GPUs provide a sizeable advantage with respect to traditional architectures for the simulation of spin systems. In the present paper we update and extend our work. In particular, we tested *Kepler*, the latest Nvidia CUDA architecture and the new Intel *Phi* coprocessor, one of the first implementations of the Many Integrated Core architecture. Our results are of interest not-only for the community of people working in statistical mechanics but, more in general, for those who need to run, on clusters of accelerators, large scale simulations in which the computation of a 3D stencil plays a major role (*e.g.,* in the numerical solution of Partial Differential Equations). As for our previous works, all source codes we developed are available for inspection and further tests.

The paper is organized as follows: Section 2. contains a summary of the approach plus a short description of the main features of the platforms used for the experiments and in particular of the Intel *Phi* coprocessor; Section 3. describes the features of the multi-GPU and multi-MIC implementations of the 3D Heisenberg spin glass model; Section 4. presents the performances obtained and concludes with a perspective about possible future extensions of the present work.

2. The Heisenberg spin glass model as a benchmark

In [1] and [2] we presented several highly-tuned implementations, for both GPU and CPU, of the over-relaxation technique applied to a 3D Heisenberg spin glass model having Hamiltonian with the following form:

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

where σ_i is a 3-component vector such that $\overrightarrow{\sigma_i} \in \mathbb{R}^3, |\sigma_i| = 1$ and the couplings J_{ij} are Gaussian distributed with average value equal to 0 and variance equal to 1. The sum in equation 1 runs on the 6 neighbors of each spin,

*Corresponding author.
tel. +0-000-000-0000 fax. +0-000-000-0000 e-mail. massimo.bernaschi@gmail.com

so the contribution to the total energy of the spin $\overrightarrow{\sigma_i}$ with coordinates x, y, z such that $i = x + y \times L + z \times L^2$ (with L being the linear dimension of the lattice) is

$$\begin{aligned} J_{x+1,y,z}\overrightarrow{\sigma}_{x,y,z} \cdot \overrightarrow{\sigma}_{x+1,y,z} &+ J_{x-1,y,z}\overrightarrow{\sigma}_{x,y,z} \cdot \overrightarrow{\sigma}_{x-1,y,z} + \\ J_{x,y+1,z}\overrightarrow{\sigma}_{x,y,z} \cdot \overrightarrow{\sigma}_{x,y+1,z} &+ J_{x,y-1,z}\overrightarrow{\sigma}_{x,y,z} \cdot \overrightarrow{\sigma}_{x,y-1,z} + \\ J_{x,y,z+1}\overrightarrow{\sigma}_{x,y,z} \cdot \overrightarrow{\sigma}_{x,y,z+1} &+ J_{x,y,z-1}\overrightarrow{\sigma}_{x,y,z} \cdot \overrightarrow{\sigma}_{x,y,z-1} \end{aligned} \quad (2)$$

where $\cdot$ indicates the scalar product of two $\overrightarrow{\sigma}$ vectors. The expression 2 is a typical example of a 3D *stencil* calculation. The memory access pattern for that calculation is very similar to that found in many other applications.
In the present work, we maintain the same performance metrics as in [1] and [2] that is the time required for the update of a single spin by using the Over-relaxation method [4] in which:

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

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

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

The update of a spin with this method requires only simple floating point arithmetic operations (20 sums, 3 differences and 28 products) plus a single division. We are not going to address other issues, like efficient generation of random numbers, even if we understand their importance for the simulation of spin systems.
The result of our previous numerical experiments showed that, on *Fermi* GPUs, the most effective approach to evaluate expression 2 is splitting the spins in two subsets (*e.g.*, red and blue spins) and loading them from the GPU global memory to the GPU registers by using the *texture* memory path of the CUDA architecture. Texture memory provides cached read-only access that is optimized for spatial locality and prevents redundant loads from global memory. When multiple blocks request the same region, the data are loaded from the cache. For the CPU we developed a vectorized/parallel code that could run on both shared memory systems (by using OpenMP) and in cluster configuration (by using MPI).

2.1. GPU and MIC architectures

For the present work we use for both the single and the multi-GPU numerical experiments the Nvidia Tesla K20s. The Tesla K20s is based on the latest architecture ("Kepler") introduced by NVIDIA. We report, in table 1, its key characteristics. The GPU has been programmed with the version 5.0 of the CUDA Toolkit. Further information about the features of the NVIDIA GPUs and the CUDA programming technology can be found in [7].

GPU model	Tesla K20s
Number of Multiprocessors	13
Number of cores	2496
Shared memory per multiprocessor (in bytes)	up to 49152
L1 Cache (in bytes)	up to 49152
L2 Cache (in Megabytes)	1.25
Number of registers per multiprocessor	65536
Max Number of threads per block	1024
Clock rate	0.706 Ghz
Memory bandwidth (ECC off)	208 GB/sec
Error Checking and Correction (ECC)	Yes

Table 1. Main features of the NVIDIA K20s GPU

As for MICs, all the tests reported in the present paper have been performed on an Intel Xeon Phi Coprocessor 5110P (8GB, 1.053 GHz, 60 core). The essential features are reported in table 2

It is worth noting that for architectural reasons, the maximum memory bandwidth which is actually available is around 50%-60% of the theoretical "electric" value.

Even if the core of Intel MIC architecture is based on P54c (Intel Pentium) cores, major improvements have been added ([9], [8]):

- Vector units: the vector unit in an Xeon Phi can work on 16 single-precision floating point values at the same time.
- Hardware multithreading: four hardware threads share the same core as if they were four processors connected to a shared cache subsystem.

Intel Xeon Phi Coprocessor model	5110P
Number of cores	60
Memory size	8 GB
Peak performance (single precision)	$\sim$ 2 TFlops
Peak performance (double precision)	$\sim$ 1 TFlops
Core multithreading	4-way
L1 Cache per core	32 Kb
L2 Cache per core	512 Kb
L2 Translation Lookaside Buffer	64 entries
Clock rate	1.053 GHz
Max Memory bandwidth (theoretical)	352 GB/sec
Error Checking and Correction (ECC)	Yes

Table 2. Main features of the Intel Xeon Phi Coprocessor 5110P

- High-frequency complex cores of the order of ten (multi-core) have been replaced by simpler low-frequency cores numbered in the hundreds (many-core) not supporting out-of-order execution.
- The interconnect technology chosen for Intel Xeon Phi is a bidirectional ring topology connecting all the cores through a bidirectional path. The cores also access data and code residing in the main memory through the interconnect ring connecting the cores to memory controller.
- Xeon Phi is usually placed on Peripheral Component Interconnect Express (PCIe) slots to work with the host processors, such as Intel Xeon processors.

In principle, the main advantage of using Intel MIC technology, with respect to other coprocessors and accelerators, is the simplicity of the porting. Programmers do not have to learn a new programming language but may compile their source codes specifying MIC as the target architecture. The classical programming languages used for High Performance Computing – Fortran/C/C++ – as well as the parallel paradigms – OpenMP or MPI – may be directly employed regarding MIC as a "classic" x86 based (many-core) architecture. Actually, in addition to OpenMP, other threading models are supported, such as Cilk and Threading Building Block (TBB) paradigms.

An Intel MIC may be accessed directly as a stand-alone system running executables in the so called *native* mode with no difference compared to an execution on a standard CPU. In other respects, like I/O, the Xeon Phi coprocessor cannot replace a traditional processor which is still needed as host processor.

However, another *offload* execution mode is available. Adopting the offload execution model, a code runs mostly on the host but selected regions of the code are marked through directives or APIs to be executed on the MIC coprocessor. The choice between the usage of directives or APIs depends on programmer's preferences. The coding strategy is similar to that devised in the OpenACC standard ([10]). Recently, OpenMP 4.0 support to manage the offloading has been made available, thus giving the users the opportunity to rely on a widely used and supported standard. At first glance, the native mode seems to be more attractive not only because the porting effort is minimized but also because there is no need to manage explicitly data movements between CPU and MIC which, if not accurately managed, may produce a major performance loss. However, the choice between native and offload mode is not so straightforward as we will discuss in Section 4..

Even if the porting effort to run on an Intel MIC appears quite limited, an optimal exploitation of its computing power may be far from being simple and, at least, a basic knowledge of the architecture is required.

In general terms, an application must fulfill three requirements to run efficiently on MIC:

1. high scalability, to exploit all MIC multithreaded cores: *e.g.,* scalability up to 240 processors (threads/processes) running on a single MIC and even higher running on multiple-MIC.
2. highly vectorizable, the cores must be able to exploit the vector units. The penalty when the code can not be vectorized is very high as we are going to show in Section 4..
3. ability of hiding I/O communications with host processor: the data exchange between host – responsible for data input and output to permanent storage – and coprocessor should be minimized or overlapped with computations.

As to the software, we used the C Intel compiler version 14.0 and the VTune profiler.

3. Multi-GPU and Multi-MIC code variants

When more than one computing system (either CPU or accelerator) is available, is quite natural to apply a simple domain decomposition along one axis like depicted in Fig. 1 so that each system is in charge of a subset of the whole lattice. We understand that a 1D decomposition could be sub-optimal but our goal here is to compare different techniques for data exchange among systems so what is important is that the decomposition is the same for all the tests. The main advantage of the 1D decomposition is that the computational kernels originally developed for the single-system version can be immediately used. Spins belonging to the boundaries between two subdomains need to be exchanged at each iteration. As already mentioned in section 2., spins

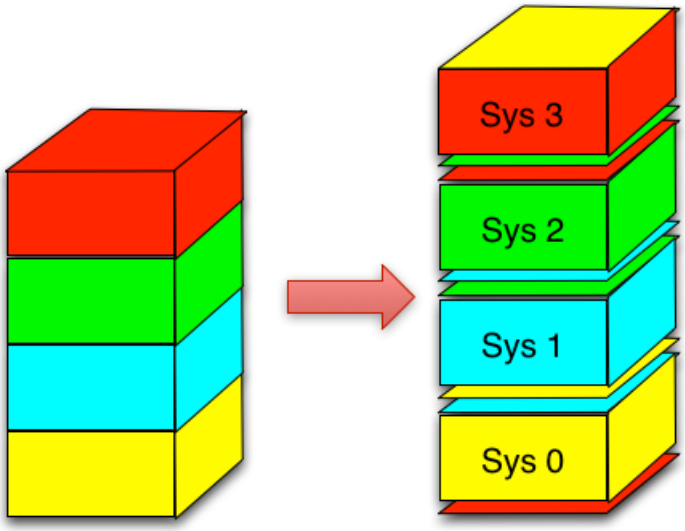

Fig. 1. Domain decomposition along one axis

are divided in two subsets: *red* and *blue* spins to allow concurrent update of all non-interacting spins. When N_{sys} systems are available, since the update of the *blue* (*red*) spins requires that the *red* (*blue*) spins of the boundaries have been exchanged, at each iteration, each system i must:

1. update the *red* spins of the planes of its subdomain i (both bulk and boundaries);
2. send the *red* spins of its bottom plane to system $(i-1)\%N_{sys}$;
   send the *red* spins of its top plane to system $(i+1)\%N_{sys}$
3. receive the *red* spins sent by system $(i-1)\%N_{sys}$;
   receive the *red* spins sent by system $(i+1)\%N_{sys}$.
4. update the *blue* spins of the planes of its subdomain i (both bulk and boundaries);
5. send the *blue* spins of its bottom plane to system $(i-1)\%N_{sys}$;
   send the *blue* spins of its top plane to system $(i+1)\%N_{sys}$;
6. receive the *blue* spins sent by system $(i-1)\%N_{sys}$;
   receive the *blue* spins sent by system $(i+1)\%N_{sys}$.

This "straightforward" scheme is represented in figure 2 for the case of three GPUs. Although it is correct, that approach imposes a severe limitation to the performances since computation and communication are carried out one after the other whereas they can be overlapped to a large extent when accelerators are used for the computation.

3.1. Effective Multi-GPU CUDA programming

CUDA supports concurrency within an application through *streams*. A stream is a sequence of commands that execute in order. Different streams, on the other hand, may execute their commands out of order with respect to each other or concurrently. The amount of execution overlap between two streams depends on the order in which the commands are issued to each stream. Further information about *streams* can be found in the CUDA documentation [7]. By using two streams on each GPU it is possible to implement a new scheme that assigns one stream to the bulk and one to the boundaries. Then, alternatively for *red* and *blue* spins:

1. starts to update the boundaries by using the first stream;
2. first stream:
   - copy data in the boundaries from the GPU to the CPU;
   - exchange data between nodes by using MPI;
   - copy data in the boundaries from the CPU to the GPU;
3. second stream:
   - updates the bulk;
4. starts a new iteration.

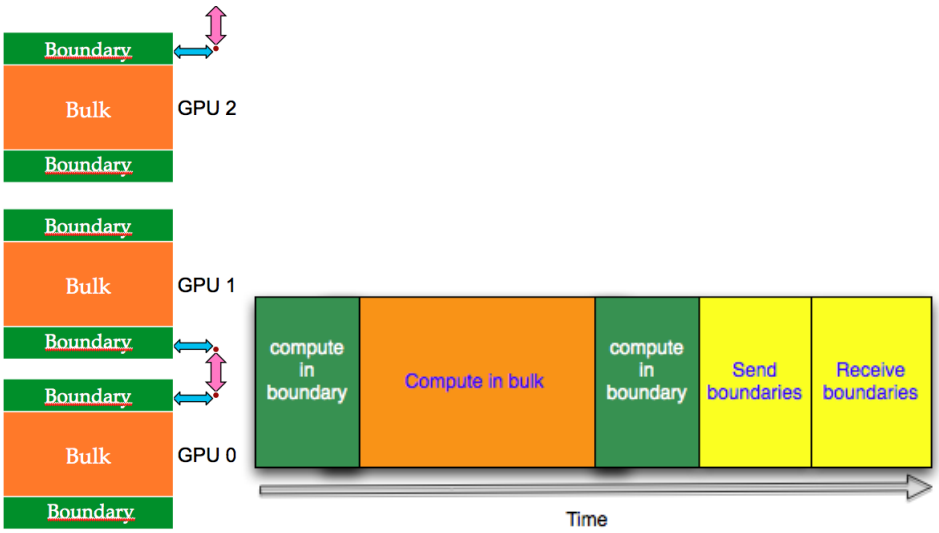

Fig. 2. Communication scheme among three GPUs with no overlap between communication and computation

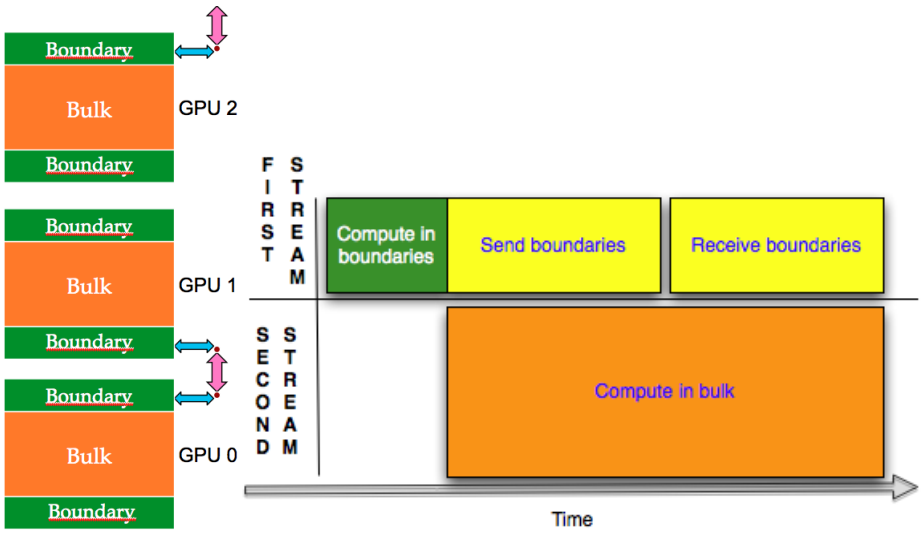

Fig. 3. multi-GPU scheme using two streams

The overlap with this scheme, shown in figure 3, is between the exchange of data within the *boundaries* (carried out by the first stream and the CPU) and the update of the bulk (carried out by the second stream). The CPU acts as a data-exchange-coprocessor of the GPU. Non-blocking MPI primitives should be used if multiple CPUs are involved in the data exchange.

Starting in CUDA 4.0, memory copies can be performed directly between two different GPUs. If such mechanism, named in CUDA as *peer-to-peer*, is enabled, then copy operations between GPUs no longer need to be staged through the CPU. The *peer-to-peer* mechanism works only if the source and the target GPU are connected to the same PCI-e root complex. We described in detail the features and the results obtained by using the *peer-to-peer* technique in [2].

3.2. Effective Multi-MIC programming

As mentioned in the introduction to the MIC programming models, classical CPU paradigms can be directly used when running on a MIC in native mode. This fact remains true even for multi-MIC programming. In the present work, for the multi-MIC version of the benchmark program we resort to the same simple 1D decomposition used for the multi-GPU variant and implement it according to the MPI paradigm. A pure MPI parallelization could be sufficient to handle single node multi-MIC and multi node multi-MIC work-load distribution but, as a matter of fact, it may be convenient to resort also to a shared-memory parallelization to overcome scalability limitations especially when dealing with a fixed-size problem.

Within the context of a hybrid parallel strategy, a *natural* approach is to assign one MPI process to each MIC while using OpenMP to distribute the computation across the MIC cores. A similar task organization minimizes the MPI communication cost, but its global performance strongly depends on the scalability across

OpenMP threads. As we are going to show in Section 4., single MIC scalability is excellent for our problem, especially when compared to a standard CPU (*e.g.,* an Intel Xeon).

As for the multi-GPU case, a major requirement for efficient multi-MIC coding is to minimize the cost of the MPI communication by overlapping data exchanges with the computations. To this purpose it is possible to resort to asynchronous communication primitives. Such strategy may be implemented for both *native* and *offload* programming models.

In *native* mode, either non-blocking MPI functions or OpenMP threads can be used for managing the asynchronous communications. For the former choice, a possible scheme is the following:

1. *red* spins: boundary updating
2. *red* spins: MPI Send/Recv non-blocking calls to exchange boundaries
3. *red* spins: bulk updating
4. *red* spins: MPI Wait calls
5. repeat step 1-4 for *blue* spins
6. starts a new iteration.

In *offload* mode, the strategy is similar to that used in the CUDA programming model or OpenACC standard ([10]). A block of code gets offloaded asynchronously (as it occurs for CUDA kernels), then MPI exchange primitives are invoked (either blocking or non-blocking calls with corresponding MPI wait calls). Finally, a synchronization primitive is invoked to ensure that offload computations have been completed. A fragment of pseudo code showing the structure used when updating red or blue bulk spins is given below.

```
#pragma offload target(mic:0) ... async(a)
<offload block a>
#pragma offload target(mic:0) ... async(b) wait(a)
<offload block b>
CPU operations (e.g., MPI calls)
#pragma offload_wait(b)
```

Actually, in the MIC offload model, the support for asynchronous operations is limited compared to CUDA. There is nothing similar to the CUDA *stream* entity and, as a consequence, the range of possible scenarios is narrower. For instance, it is not possible to serialize two asynchronous MIC-offloaded regions and have CPU operations overlapping with both regions. Hence, in the previous pseudo-code excerpt, CPU operations overlap only with offloaded block `b`.

However, an offloaded region is a wider concept compared to a CUDA kernel, since the compiler is capable of offloading complex block of codes. By taking into account this possibility, it is preferable to reduce the number of offloaded regions by merging them together, as shown in the following fragment of pseudo code:

```
#pragma offload target(mic:0) ... async(ab)
{
<offload block a>
<offload block b>
}
CPU operations (e.g., MPI calls)
#pragma offload_wait(ab)
```

Now, the execution of both blocks `a` and `b` overlaps with the CPU calls because only one offloaded region is defined.

Summarizing, the computing flow of our asynchronous offload version is:

1. *red* spins: boundary updating (offload synchronous)
2. *red* spins: bulk updating (offload asynchronous)
3. *red* spins: MPI calls to exchange boundaries
4. *red* spins: offload wait
5. repeat steps 1-4 for *blue* spins
6. starts a new iteration.

An *ex-ante* comparison between native and offload multi-MIC modes is not straightforward, but it is reasonable to assume that having the host in charge of MPI calls lets the MIC free to execute, at its best, the computing intensive part of the code without wasting time in managing the communication.

4. Results and Discussion

The test case used in the present paper is a cubic lattice with periodic boundary conditions along the X, Y and Z directions. Where not explicitly mentioned, the size along each direction is 512. The indexes for the access to the data required for the evaluation of the expression 2 are computed in accordance with the assumption that

# of *GPUs*	T_{upd}	*Efficiency*
1	0.635 ns	n.a.
2	0.284 ns	112 %
4	0.138 ns	115 %
8	0.68 ns	116 %
16	0.34 ns	116 %

Table 3. Time for spin over-relaxation on K20s GPUs. System size is 512^3. Single precision with ECC on.

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 can be found by browsing the source files available from
`http://www.iac.rm.cnr.it/~massimo/hsggpumic.tgz`.

The results are reported in nanoseconds and correspond to the time required to update a single spin. 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; Most of the tests have been carried out in single precision. For a discussion about the effects of using double precision we refer to [1].

The platform we used for the tests has the following features:

- 32 nodes with:
  - 2 eight-core Intel(R) Xeon(R) CPU E5-2658 @ 2.10GHz
  - 16 GB RAM
  - 2 Intel Xeon Phi 5110P,8 GByte RAM
- 32 nodes with:
  - 2 eight-core Intel(R) Xeon(R) CPU E5-2687W @ 3.10GHz
  - 16 GB RAM (5 nodes with 32GB)
  - 2 Nvidia Tesla K20s GPU, 5GByte RAM
- Qlogic QDR (40Gb/s) Infiniband high-performance network

In Table 3 we report the results obtained with up to 16 GPUs by using MPI and the CUDA *streams* mechanism for data exchange among the GPUs.

The parallel efficiency (defined for P GPU as $\frac{T_s}{P \times T_P}$ where T_s is the serial execution time and T_P is the execution time on P GPU) is outstanding and provides a clear indication that the overlap between communication of the boundaries and computation within the bulk hides very effectively the communication overhead. Actually, as we showed in [2], when the number of GPU increases, since the system size is fixed, the time required to update spins in the bulk reduces up to the point where it does not hide anymore the communication overhead that remains constant.

In Table 4 we report results obtained with a single Intel Phi in three different configurations of *affinity*. Affinity is a specification of methods to associate a particular software thread to a particular hardware thread usually with the objective of getting better or more predictable performance. Affinity specifications include the concept of being maximally spread apart to reduce contention (scatter), or to pack tightly (compact) to minimize distances for communication. As expected, the *scatter* specification works better with fewer threads whereas the performances of the *compact* and *balanced* specifications steadily increases with the number of threads.

MIC-OMP Threads	compact	scatter	balanced
30	4.885 ns	2.23 ns	2.089 ns
60	2.944 ns	1.13 ns	1.104 ns
120	1.305 ns	0.841 ns	0.768 ns
180	0.777 ns	0.884 ns	0.689 ns
200	0.695 ns	0.1445 ns	0.713 ns
220	0.638 ns	1.300 ns	0.651 ns
240	0.615 ns	1.456 ns	0.626 ns

Table 4. Time for spin using a single Intel Phi running one MPI process in *native* mode: scaling with respect to the number of OpenMP Threads using different *affinity* specification.

To evaluate the impact of using a hybrid configuration on a single Intel Phi, we ran also by using different combinations of MPI tasks and OpenMP threads so that the total number of *computing units* remains close to the full-load theoretical value (240). Table 5 shows the corresponding results. When the number of MPI tasks is limited there is no much difference with respect to a "pure" OpenMP implementation, however when the

MIC-MPI processes	MIC-OMP Threads	Time per spin
1	240	0.739 ns
2	120	0.733 ns
4	60	0.737 ns
8	30	0.778 ns
16	15	0.866 ns
32	7	17.054 ns
64	3	27.118 ns
128	1	49.492 ns

Table 5. Single Intel Phi in *native* mode: results obtained by using 1/2/4/8/16/32/64/128 MPI and (respectively) 240/120/60/30/15/7/3/1 OpenMP threads. The results show that the impact of a hybrid configuration on the performances is quite limited up to 16 MPI processes, whereas it becomes dramatic for higher number of MPI processes.

number of MPI tasks increases (and the number of OpenMP threads decreases) the performances reduce in a way that makes a pure MPI implementation absolutely unacceptable.

In Table 6 we report the results obtained by applying possible optimizations to the code either by using directives or by modifying directly the source code. The meaning of the different optimizations can be briefly summarized as follow:

collapse: the triple loop (along the z, y and x directions) is "collapsed" to a double loop. The loops along z and y are fused by using an OpenMP directive;

padding: dummy buffers are introduced among the main arrays (those containing the spins and the couplings) to avoid the effects of TLB trashing (see also text below and Table 7;

noprefetch: by means of directives we instruct the compiler to avoid prefetching of variables that will be immediately over-written.

vectorization: the inner loop where we implement the Over-relaxation algorithm is written in such a way that the compiler vectorizes it with no need to introduce vectorization primitives. So to assess the effect of vectorization, we had to turn off vectorization at compile time.

without collapse (c)	1.237 ns
without padding (p)	1.458 ns
without noprefetch (n)	0.765 ns
without c/p/n	3.046 ns
best (with c/p/n)	0.668 ns
best (with c/p/n) but without vectorization (v)	7.700 ns

Table 6. The impact of different optimizations on a single Intel Phi in *native* mode:

As shown in Table 6, it is necessary to introduce dummy memory buffers between each two consecutive arrays that contain the values of spins and couplings. The reason is that the L2 TLB has 64 entries and it is a 4-way associative memory. When there are more than 4 arrays that map to the same entries, there is no room in the TLB (although the other entries are unused) and as a consequence there is a L2 TLB *miss* that is managed by the operating system. Since there are 10 arrays involved in the update of a single spin, with no padding we would have six L2 TLB misses (the size of the system is a power of two and with no padding all the 12 main arrays, the six arrays containing the spins and the six arrays containing the couplings, map to the same TLB entries). Unfortunately the penalty for a L2 TLB miss is, at least, 100 clocks, so it is apparent that it is absolutely necessary to avoid them. Table 7 shows the impact of padding the spin and coupling arrays. When the size of the padding buffer is such that its address maps to the same TLB entries of the spin and couplings arrays there is no benefit from using it. An analysis carried out by using the Intel VTune profiler

# padding pages	Time per spin
0	1.458 ns
1	0.737 ns
4	0.764 ns
8	1.222 ns
16	1.537 ns
32	1.543 ns

Table 7. Effects of the padding among the main arrays; a page has a size of 2MByte.

confirmed that the number of L2 TLB misses changes dramatically depending on the number of padding pages.

In table 8, we report the general exploration metrics provided by VTune for a detailed comparison between the padded and the non-padded version of the code. The majority of the counters share the same order of

Parameter	Non-padded	Padded
CPU Time	33459.268	31783.926
Clockticks	35199150000000.000	33436690000000.000
CPU_CLK_UNHALTED	35199150000000.000	33436690000000.000
Instructions Retired	724220000000	417970000000
CPI Rate	48.603	79.998
Cache Usage		
L1 Misses	13680450000	19016200000
L1 Hit Ratio	0.954	0.900
Estimated Latency Impact	2516.588	1732.644
Vectorization Usage		
Vectorization Intensity	10.913	10.248
L1 Compute to Data Access Ratio	3.138	4.783
L2 Compute to Data Access Ratio	68.417	47.795
TLB Usage		
L1 TLB Miss Ratio	0.033	0.064
L2 TLB Miss Ratio	0.026	0.000
L1 TLB Misses per L2 TLB Miss	1.252	1052.121
Hardware Event Count		
L2_DATA_READ_MISS_CACHE_FILL	464800000	548100000
L2_DATA_WRITE_MISS_CACHE_FILL	361900000	357000000
L2_DATA_READ_MISS_MEM_FILL	7220500000	7918400000
L2_DATA_WRITE_MISS_MEM_FILL	176400000	180600000

Table 8. General Exploration Metrics from VTune report comparing the non-padded versus the padded version of the code.

magnitude comparing the two versions. However, the values of the L2 Miss Ratio are dramatically different. To detail such difference, since in the padded version the value is approximated to zero, we inspected the low-level hardware event LONG_DATA_PAGE_WALK, and it turned out that the padded version features the value 11550000 whereas the corresponding non-padded value is 7860650000, thus providing a quantitative base to the macroscopic difference in performances observed.

The next set of data shows the performances obtained by using up to 16 Intel Phi coprocessors. In Table 10 we report the results obtained by using 220 OpenMP threads with four different configurations:

Native-Sync: uses MPI blocking primitives running on Intel Phi;

Native-ASync: uses MPI non-blocking primitives (MPI_Irecv and MPI_Wait) running on Intel Phi;

Offload-Sync: uses MPI blocking primitives running on host CPU;

Offload-ASync: uses MPI non-blocking primitives (MPI_Irecv and MPI_Wait) running on host CPU.

We ran with 220 threads instead of 240 because we found that the performance dropped dramatically using 240 and MPI probably because there were no enough resources on the Intel Phi to support, at the same time, the maximum number of computing threads (240) and the MPI communication. Even with that expedient, the scalability using more than one Intel Phi in *native* mode is very limited (the efficiency with 16 systems is below 25%) despite of the use of non blocking MPI primitives that should support an asynchronous communication scheme. Actually, to achieve a real overlap between computation in the bulk and communication of the boundaries one has to change the execution model from *native* to *offload* so that computation in the bulk can still be carried out by the Intel Phi while the CPU manages the MPI primitives used for data exchange among the boundaries. As a matter of fact, the efficiency with this configuration improves significantly and, for 16 systems, reaches 60%. However, the single system performance in *offload* mode is significantly worse (about 40%) compared to an execution in *native* mode. Besides that, there is another issue, scalability is much better in *offload* mode compared to *native* mode but remains far from being ideal. The problem, more than in the communication, appears to be in the reduced amount of work that each system does when the total size of the problem is fixed (*strong scaling*). To double check this hypothesis we measured the performances for different sizes of the domain. The results are reported in Table 9 and show how the GPU and the MIC in *native* mode behave in the same way: the performances are similar and quite stable until the size of the domain becomes too small and the performance drops significantly. For the MIC in *offload* mode, the performance degradation is more gradual but also more remarkable. It is not surprising that the strong scaling is limited in this case.

As a matter of fact we will see that the situation changes looking at the *weak* scaling. In *offload* mode the penalty for (relative) small problem sizes is higher so, in the end, 16 Intel Phi coprocessors are more than three times slower with respect to 16 Nvidia K20s in multi-system configuration.

Although the subject of the present work are the *accelerators*, we feel important to provide, for the sake of comparison, some data produced by using state-of-art traditional CPUs like the Intel Xeon E5-2687W @

Domain size	Nvidia K20s	Intel Phi (native)	Intel Phi (offload)
64^3	3.51 ns	1.858 ns	14.289 ns
128^3	0.793 ns	0.920 ns	2.774 ns
256^3	0.671 ns	0.784 ns	1.337 ns
512^3	0.635 ns	0.739 ns	1.083 ns

Table 9. Performances of Nvidia K20s and of Intel Phi (in *native* mode) for different sizes of the domain.

3.10GHz. Table 11 show the results obtained by using up to 16 threads (for OpenMP) or processes (for MPI). In both cases the scalability is pretty poor but interestingly is better with MPI. The situation improves significantly by using multiple CPUs, as shown in Table 12: the efficiency reaches 70% when MPI asynchronous primitives are used for data exchange but the aggregate performance remains significantly lower compared to both the 16 K20s and the 16 Intel Phi configurations.

# MICS	Native-Sync	Native-Async	Offload-Sync	Offload-Async
1	0.709 ns	0.717 ns	1.049 ns	1.078 ns
2	0.484 ns	0.431 ns	0.558 ns	0.527 ns
4	0.445 ns	0.325 ns	0.335 ns	0.281 ns
8	0.376 ns	0.246 ns	0.219 ns	0.167 ns
16	0.343 ns	0.197 ns	0.154 ns	0.113 ns

Table 10. Results with four different Multi MIC configurations: one MPI process *per* MIC. Number of MIC-OpenMP threads for each MPI process equal to 220.

# processors	OpenMP-splitted	MPI-Async
1	0.007231 ns	0.00725 ns
2	0.003876 ns	0.003065 ns
4	0.00308 ns	0.00185 ns
8	0.003417 ns	0.001566 ns
16	0.003597 ns	0.001629 ns

Table 11. Single CPU node scalability using OpenMP and MPI.

In Tables 13 and 14 and Figure 4 we summarize the *best* results for single (Table 13) and multi (Table 14) system configurations. For single systems both Intel Phi and Nvidia Kepler perform much better with respect to a multi-core high-end traditional CPU. As to the efficiency using multiple-systems, the multi-GPU configuration is always better (the efficiency is almost double compared to the multi-MIC configuration using 16 units), so that the performance gap increases with the number of systems. As shown in [2] this remains true up to the point in which the computation in the bulk takes, at least, the same time of the boundaries exchange.
Finally, in Table 15 we report the timings and the efficiency measured in a *weak scaling* test where the total size of the simulated system increases so that each computing unit does the same amount of work as in the single unit case. Here the efficiency of traditional CPUs, GPUs and MICs in *offload* mode is outstanding ($> 95\%$) whereas the efficiency of MICs in *native* mode remains well below 50%.

5. Conclusions

We have presented a large set of results obtained by using highly tuned multi-CPU, multi-GPU and multi-MIC implementations of the 3D Heisenberg spin glass model whose computing core is the evaluation of a 3D stencil and, as such, it is representative of a wide class of numerical simulations. Our findings can be summarized as follows:

1. The performances of the single system may change significantly depending on the size of the problem under study.
2. Vectorization is absolutely required on both traditional Intel CPU and Intel Phi coprocessor. This entails that, on a single system, two levels of concurrency must be maintained (vectorization and threads parallelism). In CUDA there are both *blocks* of threads and *grids* of blocks but there is a unique *Single Program Multiple Threads* programming model.
3. The Intel Phi is sensible to a number of potential performance limiting factors. The most apparent we found is the need to *pad* arrays to avoid dramatic performance drops due to L2 TLB trashing effects.
4. A careful tuning of the C *source* code (*i.e.,* without resorting to vector *intrinsic* primitives) running on a single Intel Phi allows to achieve performances very close to that of a latest generation (*Kepler*) GPU.

# CPUs	Sync	Async
1	3.693 ns	3.444 ns
2	2.033 ns	1.841 ns
4	1.059 ns	0.968 ns
8	0.549 ns	0.485 ns
16	0.392 ns	0.287 ns

Table 12. Multi CPU scalability using 2 MPI processes per node (that is 1 per CPU) and 8 OpenMP threads.

CPU (MPI)	1.569 ns
MIC (*native*)	0.668 ns
GPU	0.635 ns

Table 13. Comparison single system: CPU (eight-core Intel Xeon CPU E5-2687W @ 3.10GHz), MIC (Intel Xeon Phi 5110P) in *native* mode, GPUs (Kepler K20s).

# Procs	CPU	GPU	MIC (o)	CPU effic.	GPU effic.	MIC (o) effic.
1	3.444 ns	0.635 ns	1.078 ns	n.a.	n.a.	n.a.
2	1.841 ns	0.284 ns	0.527 ns	93.5%	112%	102.5%
4	0.968 ns	0.138 ns	0.281 ns	89%	115%	96%
8	0.485 ns	0.068 ns	0.167 ns	88.7%	116.7%	80.7%
16	0.287 ns	0.034 ns	0.113 ns	75%	116.7%	60%

Table 14. Comparison multi: CPUs (eight-core Intel Xeon CPU E5-2687W @ 3.10GHz), GPUs (Kepler K20s), MICs (Intel Xeon Phi 5110P) in *offload* mode.

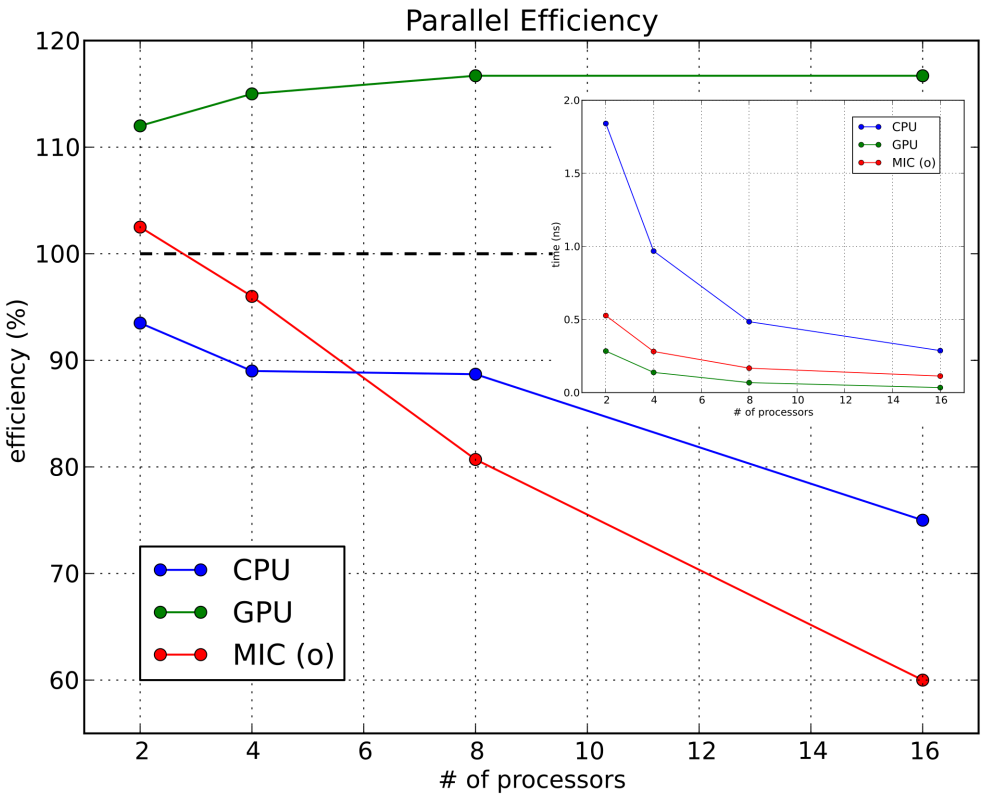

Fig. 4. Efficiency and timings (in the inset) for multi-system configurations

Although the source code for a traditional Intel CPU and an Intel Phi may be very similar, the effect of (apparently) minor changes may be dramatic so that the expected main advantage of Intel Phi (code portability) remains questionable. Obviously, GPU requires a porting to the CUDA architecture whose

# Procs	Size	CPU	GPU	MIC (native)	MIC (offload)
1	256	3.733 ns	0.671 ns	0.784 ns	1.340 ns
8	512	0.485 ns	0.068 ns	0.246 ns	0.167 ns
Efficiency		96.2%	123%	39.9%	100%

Table 15. Timings and efficiency obtained in a *weak scaling* test. The size of the system on each computing node remains 256^3.

cost depends on the complexity of the application and the availability of *libraries* for common operations.

5. The main differences in performances between GPU and Intel Phi have been obtained in multi-system configurations. Here the *stream* mechanism offered by CUDA allows to hide the communication overhead provided that the computation executed concurrently with the communication is long enough. As a result, the efficiency is always outstanding. Running on an Intel Phi in *offload* mode it is possible to emulate, somehow, the CUDA *stream* mechanism that supports independent execution flows. In *offload* mode the efficiency of a multi Intel Phi configuration is better, but remains low compared to a multi GPU configuration with the same number of computing units for a problem of fixed size. For problems where the size for computing unit is constant (so that the total size of the problem increases with the number of computing units), the efficiency of Intel Phi is close to ideal but the performance of the single system (due to the *offload* execution mode), at least for our reference problem, are significantly lower with a degradation of about 40%.
6. In a nutshell we can state that, for the problem we studied:
   - 1 GPU $\simeq$ 1 MIC $\simeq$ 5 High-end CPU;
   - strong scaling: 1 GPU $\simeq$ 1.5 - 3.3 MICs depending on the number;
   - weak scaling: 1 GPU $\simeq$ 2 MICs.

Clusters of accelerators are a very promising platform for large scale simulations of spin systems (and similar problems) but code tuning and compatibility across different architectures remain open issues. In the near future we expect to extend this activity to more general domain de-compositions and problems in higher dimensions.

Acknowledgments

We acknowledge PRACE for awarding us access to resource *Eurora* based in Italy at CINECA. We thank Massimiliano Fatica for useful discussions and Isabella Baccarelli for support in running the tests.

This work was financially supported by the PRACE-1IP project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-261557. The work was achieved using the PRACE Research Infrastructure resources at the EURORA (CINECA) supercomputer.

References

1. M. Bernaschi, G. Parisi, L. Parisi, Benchmarking GPU and CPU codes for Heisenberg spin glass over-relaxation, Computer Physics Communications, 182 (2011) 6.
2. M. Bernaschi, M. Fatica, G. Parisi, L. Parisi, Multi-GPU codes for spin systems simulations, Computer Physics Communication, 183 (2012), 1416.
3. M. Bernaschi, M. Bisson, D. Rossetti, Benchmarking of communication techniques for GPUs, Journal of Parallel and Distributed Computing, 73 (2013) 250.
4. S. Adler, Over-relaxation method for the Monte Carlo evaluation of the partition function for multi-quadratic actions. Phys. Rev. D 23, 29012904 (1981).
5. T. Preis, P. Virnau, W. Paul and J. Schneider, GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model. Journal of Computational Physics 228 (2009) 4468-4477.
6. M. Weigel, Simulating spin models on GPU, Preprint arXiv:1006.3865 (2010).
7. NVIDIA CUDA Compute Unified Device Architecture Programming Guide `http://www.nvidia.com/cuda`
8. J. Jeffers, J. Reinders, Intel Xeon Phi Coprocessor High Performance Programming, Morgan Jaufmann (2013)
9. R. Ranman, Intel Xeon Phi Coprocessor architecture and Tools, Apress open (2013).
10. OpenACC Directives for accelerators `http://www.openacc-standard.org/`