Multi-Kepler GPU *vs.* Multi-Intel MIC
A two test case performance study

Massimo Bernaschi
Istituto Applicazioni Calcolo
CNR
Rome, Italy
massimo.bernaschi@cnr.it

Francesco Salvadore
SuperComputing Applications and Innovation Department
Cineca
Rome, Italy
f.salvadore@cineca.it

***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 two physical systems. As a first benchmark we consider the time required to update a single spin of the 3D Heisenberg spin glass model by using the Over-relaxation algorithm. The second application we consider is a reactive fluid-dynamics problem for which we resolve the full Navier-Stokes compressible set of equations without resorting to a turbulence model. The results show that the performances of an Intel *MIC* change dramatically depending on (apparently) minor details. Another issue is that to obtain a reasonable scalability with the Intel *Phi* coprocessor in cluster configuration it is necessary to use the so-called *offload* mode which reduces the performances of the single system. All source codes are provided for inspection and for double-checking the results.**

***Keywords*—Spin Systems, Many Core, Asynchronous communication.**

I. 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. In the present paper we update and extend that work. First of all, we provide results for *Kepler*, the current Nvidia CUDA architecture and the new *Phi* coprocessor, one of the first implementations of the Intel Many Integrated Core architecture. Moreover, we consider a second benchmark, the solution of a reactive fluid-dynamics problem modeled by the classic Navier-Stokes equations. Since the computing kernels and the communication patterns we implemented are common to many applications, our results are of interest not-only for the community of people working in statistical mechanics or fluid dynamics.

The paper is organized as follows: Section II summarizes the main features of the platforms used for the numerical experiments and in particular of the Intel *Phi* coprocessor; Section III recalls the computational features of the 3D Heisenberg spin glass model and describes the multi-GPU and multi-MIC implementations. Section III-D presents the performances obtained for this first benchmark; Section IV describes our second benchmark application, the alternative parallel programming paradigm used for its implementation on the same platforms and the corresponding results. Section V concludes with a summary of the main findings.

II. MANY-CORE PLATFORMS

For the present work we use for both the single and the multi-GPU numerical experiments the Nvidia Tesla K20s having 2496 CUDA-core running at 0.706 GHz and equipped with 5GB of memory. The Tesla K20s is based on the NVIDIA "Kepler" architecture, widely described in [9]. The GPU has been programmed with the version 5.5 of the CUDA Toolkit.

The Xeon Phi is the Intel solution to improve the Xeon family, by focusing on performances and watt efficiency ([12]). The Intel Phi relies on a new microarchitectural design but the architecture is still based on P54c (Intel Pentium) cores enhanced by major improvements like:

- Vector units: the vector unit in an Intel 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.
- From multi-core to many-core: complex cores of the order of ten (multi-core) has been replaced by simpler low-frequency cores numbered in the hundreds (many-core) with no support to out-of-order execution.
- Interconnect and cache: the interconnect technology is a bidirectional ring topology connecting all the cores through a bidirectional path.
- System interconnect: coprocessors are often placed on Peripheral Component Interconnect Express (PCIe) slots to work with the host processors.

The tests reported in the present paper have been performed on an Intel Xeon Phi Coprocessor 5110P having 60 core running at 1.053 GHz and equipped with 8GB of memory, using the C Intel compiler version 14.0. Other features may be found at [10].

The GPU exhibits higher peak performances with 3.52 TFlops (single precision) and 1.17 TFlops (double precision)

978-1-4799-5313-4/14/$31.00 ©2014 IEEE

whereas the Xeon Phi has a peak performance of 1.97 TFlops and 0.98 TFlops. On the other hand, MIC features a higher theoretical memory bandwidth of 352 GB/sec compared to the 208 GB/sec of a GPU.

In principle, the main advantage of the Intel MIC technology, with respect to other coprocessors and accelerators, is the simplicity of the porting. Programmers compile their existing source codes specifying MIC as the target architecture. Classical programming languages used in High Performance Computing – Fortran/C/C++ – as well as well known parallel paradigms – OpenMP or MPI – may be directly employed regarding MIC as a "classic" x86 based (many-core) architecture. An Intel MIC may be accessed directly as a stand-alone system running executables in the so called *native* mode. 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. 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 III-D.

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

1) high scalability, to exploit all MIC cores and hardware threads (up to 240 threads/processes on a single-MIC).
2) highly vectorizable, 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 III-D.
3) ability of hiding I/O communications with host processor: the data exchange between host – responsible for data I/O and multi-node communication – and coprocessor should be minimized or overlapped with computations.

III. FIRST BENCHMARK: THE HEISENBERG SPIN GLASS MODEL

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 (HSG) model having Hamiltonian with the following form:

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

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 fact that the couplings may assume both positive and negative values determines the so called "frustration" effect which slows down dramatically the dynamics of the system. As a consequence, simulations require very long times up to that point that special purpose computers have been developed for the study of spin glass systems [7]. The sum in equation 1 runs on the 6 neighbors of each spin, 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

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

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 $[x, y, z]$,

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

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). That code has been the base of the Intel Phi implementation.

A. Multi-GPU and Multi-MIC code variants

When more than one computing system (either CPU or accelerator) is available, the most simple approach is to apply a domain decomposition along one axis. 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. Spins belonging to the boundaries between two subdomains need to be exchanged at each iteration. As already

mentioned, spins 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 $(i+1)\%N_{sys}$.

An approach like that, although correct, 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.

B. Effective Multi-GPU CUDA programming

CUDA supports concurrency within an application through *streams* [9]. 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. By using two streams on each GPU 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) start 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.

The overlap with this scheme, shown in figure 1, 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.

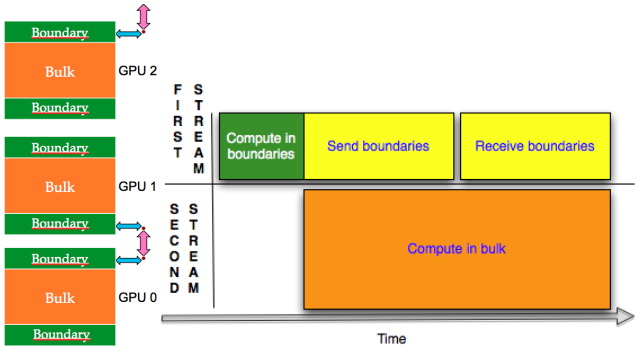

Figure 1: multi-GPU scheme using two streams

C. 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 Heisenberg spin glass 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 III-D, 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) start a new iteration.

In *offload* mode, the strategy is similar to that used in the CUDA programming model or OpenACC standard ([13]). 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 with respect 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) start 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 parts of the code without wasting time in managing the communication.

D. 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 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`

# of *GPUs*	T_{upd}	*Efficiency*
1	0.635 ns	n.a.
2	0.284 ns	112 %
4	0.138 ns	115 %
8	0.068 ns	116 %
16	0.034 ns	116 %

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

The results are reported in nanoseconds and correspond to the time required to update a single spin. 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 E5-2687W @ 3.10GHz
  - 16 GB RAM (5 nodes with 32GB)
  - 2 Nvidia Tesla K20s GPU, 5GByte RAM
  - 2 Intel Xeon Phi 5110P,8 GByte RAM
- Qlogic QDR (40Gb/s) Infiniband high-performance network

In Table I 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, in percentage, for P nodes as $100 \times \frac{T_s}{P \times T_P}$ where T_s is the serial execution time and T_P is the execution time on P nodes) 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 II 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 performances. Affinity specifications include the concept of being maximally spread apart to reduce contention (scatter), or to pack tightly (compact)

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 II: HSG - 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.

vectorization	3.046 ns
vectorization/padding/noprefetch	1.237 ns
vectorization/noprefetch/collapse	1.458 ns
vectorization/padding/collapse	0.765 ns
padding/collapse/noprefetch	7.700 ns
vectorization/padding/noprefetch/collapse	0.668 ns

TABLE III: HSG - The impact of different optimizations on a single Intel Phi in *native* mode.

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.

In Table III we report the results obtained by applying possible optimizations to the code either by using directives or by modifying directly the source code. In particular: (a) *collapse:* the triple loop (along the z, y and x directions) is "collapsed" to a double loop by fusing the z and y loops; (b) *padding:* dummy buffers are introduced among the main arrays (those containing the spins and the couplings) to avoid the effects of TLB trashing (see below); (c) *noprefetch:* by means of directives we instruct the compiler to avoid prefetching of variables that will be immediately over-written; (d) *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.

Other classical optimizations have been also attempted (see [11]) such as running many MPI processes for each MIC but no significant improvement has been obtained.

The reason why it is necessary to introduce dummy memory buffers between each two consecutive arrays that contain the values of spins and couplings is that the L2 TLB is a 4-way associative memory with 64 entries. 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. 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 IV shows the impact of padding the spin and coupling arrays. When the size of the padding buffer is such

# 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 IV: HSG - Effects of the padding among the main arrays. A page has a size of 2MByte.

that its address maps to the same TLB entries of the spin and couplings arrays there is no benefit from padding. An analysis carried out by using the Intel VTune profiler confirmed that the number of L2 TLB misses changes dramatically depending on the number of padding pages.

The next set of data shows the performances obtained by using up to 16 Intel Phi coprocessors. In Table V 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 threads 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, the speedup, defined as the ratio between the serial execution time and the parallel execution time, is much better in *offload* mode compared to *native* mode but remains far from being ideal (that is linear with respect to the number of computing nodes). 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*). We will

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

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

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

see that the situation changes looking at the *weak* scaling, i.e. when the amount of work computed by each system remains constant (the whole size is proportional to the number of systems). In the end, 16 Intel Phi coprocessors are more than three times slower with respect to 16 Nvidia K20s in multi-system configuration.

In Tables VI e VII we summarize the *best* results for single (Table VI) and multi (Table VII) system configurations. For single systems both Intel Phi and Nvidia Kepler perform much better with respect to a multi-core high-end traditional CPU (Intel Xeon E5-2687W @ 3.10GHz). As to the parallel 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 VIII we report the timings and the parallel efficiency measured in a *weak scaling* test where the total size of the simulated system increases so that each computing node does the same amount of work as in the single node 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%.

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

# 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 VIII: HSG - Timings and parallel efficiency obtained in a *weak scaling* test. The size of the system on each computing node remains 256^3.

IV. SECOND BENCHMARK: REACTIVE FLUID-DYNAMICS

The second application code we analyze is a reactive fluid-dynamics (RFD) problem focusing on the physics of turbulent combustion ([15]). A premixed flame, initially planar and stationary, is subjected to inlet turbulent boundary conditions which interact with the reacting front. As a consequence, the flame front speed increases its propagation speed. In order to avoid the flame front entering the inlet boundary, a constant additional streamwise velocity is dynamically adjusted and added to the flame velocity field, relying on the Galilean invariance of the problem. The code we are considering solves the full Navier-Stokes compressible set of equations without relying on any turbulence model, i.e. performing a so called Direct Numerical Simulation (DNS). On the other hand, the reacting model is oversimplified: only two species are considered and the reactive term is a basic Arrhenius term. The set of equations is summarized below.

$$\begin{aligned}
\frac{\partial \rho}{\partial t} + \frac{\partial (\rho\, u_j)}{\partial x_j} &= 0, \\
\frac{\partial (\rho\, u_i)}{\partial t} + \frac{\partial (\rho\, u_i u_j)}{\partial x_j} + \frac{\partial p}{\partial x_i} - \frac{\partial \sigma_{ij}}{\partial x_j} &= 0, \\
\frac{\partial (\rho\, E)}{\partial t} + \frac{\partial (\rho\, E u_j + p u_j)}{\partial x_j} - \frac{\partial (\sigma_{ij} u_i - q_j)}{\partial x_j} &= 0, \\
\frac{\partial \rho Y}{\partial t} + \frac{\partial (\rho\, u_j Y)}{\partial x_j} - D \frac{\partial Y}{\partial x_i^2} + A \exp(-\theta/\theta_A) &= 0,
\end{aligned}$$

where ρ is the density, u_i is the velocity component in the i-th coordinate direction ($i = 1, 2, 3$), $E = \rho\,[c_v(\theta - \theta_0) + u_i u_i/2 + Y \Delta H_0]$ is the total energy per unit mass, Y is the mass fraction of the first specie, p is the thermodynamic pressure, and

$$q_j = -\lambda \frac{\partial T}{\partial x_j}, \quad \sigma_{ij} = 2\,\mu S_{ij} - \frac{2}{3}\,\mu\, S_{kk}\, \delta_{ij}$$

are the heat flux vector and the viscous stress tensor respectively, and $S_{ij} = (u_{i,j} + u_{j,i})/2$ is the strain-rate tensor. The molecular viscosity μ, the heat conductivity k, the diffusion coefficient D, the Arrhenius terms A and θ_A are constant values.

The geometrical configuration is sketched in figure 2. Basically, we have two coupled problems: a non-reacting homogeneous triperiodic turbulence field (on the left) is simulated and a fixed plane is extracted and used as the turbulent

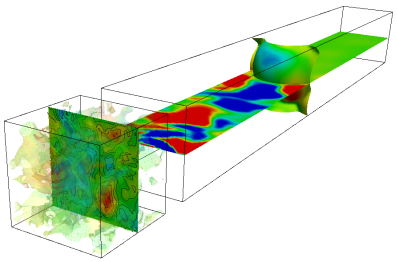

Figure 2: Sketch of the geometrical configuration for the turbulent premixed flame test case

inlet for the flame domain (on the right). For the flame domain, spanwise boundary conditions are still periodic whereas the outlet boundary is designed to avoid reflections from the artificial domain end.

The flow solver relies on a co-located finite-difference conservative approach which has been widely validated and used in the past to study compressible flows in various configurations (see [14]). Uniform grids and fourth-order centered finite difference schemes are employed. The code is written in Fortran and parallelization relies on both a three-dimensional MPI decomposition and OpenMP for multithreading ([8]). The code is available for inspection at `https://cube-flu.googlecode.com/svn/trunk/tars/ns_flame.tar`

A. *Programming model*

For the second benchmark, we decided to adopt a different approach to exploit the GPUs based on the OpenACC paradigm ([13]). OpenACC is a directive-based proposed standard aimed at simplifying the porting effort required to take advantage of modern accelerators and it is similar to the Intel Offload model. In principle, the resulting code is portable across different accelerators since the compilers may expand the OpenACC directives into an OpenCL code that is supported by most of modern accelerators. Unfortunately, the OpenACC support of compilers is still often incomplete and, at present, no open-source solution is available even though it has been officially announced that OpenACC support will be available for the GNU compilers in the near future. For the tests hereafter reported, we employed the PGI compiler (v14.1) and adopted OpenACC only to exploit GPUs,

An OpenACC code hardly achieves the same performances of a well-tuned direct CUDA implementation. However, to increase the performances, it is possible to exploit the opportunity of mixing the OpenACC and CUDA paradigms, by adopting a low-level programming model only for the most demanding (from the computational viewpoint) sections of a code.

B. *Results*

We present here some preliminary results for the second benchmark. The size of the base test case is $1024 \times 128 \times 128$ for the flame domain coupled to a 128^3 triperiodic domain. As performance metric we use the elapsed time for the execution of a simulation step. As a first study we tested the usage of the OpenMP collapse to maximize the scalability inside a single MIC. As in the HSG case, distributing the OpenMP threads over two loop nests is material to exploit the many-core architecture of the Xeon PHI. The results are summarized in table IX and have been obtained by running the code in native mode. Then, we performed tests of the single-system scalability obtained by mixing the MPI and OpenMP approaches. We executed the analysis for a standard CPU node (16 cores) as well as for the MIC system. It turned out that for both architectures the best performances are achieved with a suitable balance of the MPI and OpenMP parallelization: for the CPU runs, using 4 MPI processes and 4 OpenMP threads is the best choice, whereas for a MIC 4 MPI processes each one with 55 OpenMP threads is the best configuration. The performance gain, compared to a pure MPI or OpenMP parallelization is significant (more than 25%). It is worth noting that the best performances for a CPU node and a MIC accelerator are very close (1.27 s versus 1.35 s).

# MIC OMP-threads	No-collapse	Collapse(2)
1	108.7 s	109.26 s
8	15.27 s	14.59 s
16	7.67 s	7.52 s
30	5.24 s	4.51 s
60	3.08 s	2.51 s
120	2.60 s	1.87 s
180	1.89 s	1.77 s
240	2.20 s	1.67 s

TABLE IX: RFD - OpenMP scalability using a single MIC: comparison between OpenMP collapse(2) version versus the non-collapsed one.

# Size	CPU (16 cores)	GPU	MIC-native	MIC-offload
128x128x128	0.3 s	0.54 s	0.6 s	0.63 s
256x128x128	0.47 s	0.5 s	0.7 s	0.91 s
512x128x128	0.75 s	0.78 s	0.85 s	1.34 s
1024x128x128	1.27 s	1.34 s	1.35 s	2.38 s

TABLE X: RFD - Comparison of performances with respect to the size of flame domain for the standard CPU (16 cores), GPU, MIC-native and MIC-offload versions of the flame code.

A third study considered the size of the simulated domain. We performed a full comparison among the four versions of the code: CPU (16 cores), GPU, MIC-native, and MIC-offload. For MIC-native tests, we used 4 MPI processes per MIC whereas for the MIC-offload configuration 1 MPI process per MIC resulted the optimal choice. For the base size (1024x128x128) comparable performances have been obtained by using the CPU (16 cores), GPU and MIC-native versions whereas the MIC-offload version is significantly slower. For smaller sizes, the CPU tends to keep a better behavior. It is worth noting that, even though the performances of the code by using a CPU node or an accelerator are similar, two or

# Procs	CPU (8 cores)	GPU	MIC-native	MIC-offload
1	2.77 s	1.34 s	1.35 s	2.38 s
2	1.67 s	0.78 s	0.85 s	1.35 s
4	0.92 s	0.51 s	0.62 s	0.96 s
8	0.48 s	0.28 s	0.55 s	0.76 s

TABLE XI: RFD - Strong scalability analysis for flame code.

# Procs	Size	CPU (8 cores)	GPU	MIC-native	MIC-offload
1	1024x128x128	2.77 s	1.34 s	1.35 s	2.38 s
2	2048x128x128	3.14 s	1.50 s	1.44 s	2.36 s
4	4096x128x128	3.62 s	1.61 s	1.48 s	2.95 s
8	8192x128x128	4.05 s	1.81 s	1.75 s	3.67 s

TABLE XII: RFD - Weak scalability analysis for the flame code.

more accelerators are usually available on a node. Another major advantage of employing accelerators is their reduced power consumption, compared to standard CPUs.

The last study we conducted is a scalability analysis for multi-system configurations. Both strong and weak scaling cases have been considered. The strong scalability of MIC systems is very limited (parallel efficiency well below 50%) both in native and offload modes. CPUs and GPUs give better results. For CPUs the scalability between 1 and 2 processors is limited by the sharing of resources. The efficiency by using 8 GPUs is close to 60%. As expected, a naive management of MPI communications, with synchronous copies between the host and the device, limits dramatically the scalability. The weak scalability results are similar for all the platforms. It is worth noting that the domain is enlarged only along the dimension of the flame whereas the triperiodic domain is kept fixed.

V. CONCLUSIONS

We have presented a set of results obtained by using highly tuned multi-GPU and multi-MIC implementations of two applications that may be considered representative of a wide class of numerical simulations in physics.
Our findings can be summarized as follows:

1) Vectorization is absolutely required on both traditional Intel CPU and Intel Phi coprocessor so 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.
2) 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.
3) A careful tuning of the *source* code (*i.e.,* without resorting to vector *intrinsic* primitives) running on a single Intel Phi allows to achieve performances very close to those of a *Kepler* GPU. 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. Using GPU requires a porting to the CUDA architecture whose cost depends on the complexity of the application and the availability of *libraries* for common operations.
4) A directives-based approach makes easier to exploit accelerators and may represent a good trade-off for those parts of a code with a limited impact on the performances.
5) The main differences in performances between GPU and Intel Phi have been obtained in multi-system configurations. Here the *streams* mechanism offered by CUDA allows the hiding of the communication overhead, in particular of the copies between CPU and accelerator, provided that the computation executed concurrently with the communication is large enough. As a result, the parallel 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 parallel 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.

REFERENCES

[1] M. Bernaschi, G. Parisi, L. Parisi, Benchmarking GPU and CPU codes for Heisenberg spin glass overrelaxation, 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 multiquadratic 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] M. Baity-Jesi, Spin Glass Simulations on the Janus Architecture: A Desperate Quest for Strong Scaling, Lecture Notes in Computer Science Volume 7640, (2013), pp 528-537.
[8] F. Salvadore, A Generalized Directive-Based Approach for Accelerating PDE Solvers, Lecture Notes in Computer Science, OpenMP in a Heterogeneous World (2012), 258-261.
[9] NVIDIA CUDA Compute Unified Device Architecture Programming Guide `http://www.nvidia.com/cuda`
[10] Intel Xeon Phi Coprocessor 5110P `http://ark.intel.com/products/71992`
[11] J. Jeffers, J. Reinders, Intel Xeon Phi Coprocessor High Performance Programming, Morgan Jaufmann (2013)
[12] R. Ranman, Intel Xeon Phi Coprocessor architecture and Tools, Apress open (2013).
[13] OpenACC Directives for accelerators `http://www.openacc-standard.org/`
[14] S. Pirozzoli, Generalized conservative approximations of split convective derivative operators. J. Comput. Phys. 7180-7190 (2010)
[15] N. Chakraborty, M. Klein, R. S. Cant, Effects of Turbulent Reynolds Number on the Displacement Speed Statistics in the Thin Reaction Zones Regime of Turbulent Premixed Combustion, Journal of Combustion vol. 2011 (2011), Article ID 472679