Original Article

International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

*The International Journal of High Performance Computing Applications*
1–16
© The Author(s) 2017
Reprints and permissions:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/1094342017703771
journals.sagepub.com/home/hpc
SAGE

# Optimization of lattice Boltzmann simulations on heterogeneous computers

**E Calore, A Gabbana, SF Schifano and R Tripiccione**

**Abstract**
High-performance computing systems are more and more often based on accelerators. Computing applications targeting those systems often follow a host-driven approach, in which hosts offload almost all compute-intensive sections of the code onto accelerators; this approach only marginally exploits the computational resources available on the host CPUs, limiting overall performances. The obvious step forward is to run compute-intensive kernels in a concurrent and balanced way on both hosts and accelerators. In this paper, we consider exactly this problem for a class of applications based on lattice Boltzmann methods, widely used in computational fluid dynamics. Our goal is to develop just one program, portable and able to run efficiently on several different combinations of hosts and accelerators. To reach this goal, we define common data layouts enabling the code to exploit the different parallel and vector options of the various accelerators efficiently, and matching the possibly different requirements of the compute-bound and memory-bound kernels of the application. We also define models and metrics that predict the best partitioning of workloads among host and accelerator, and the optimally achievable overall performance level. We test the performance of our codes and their scaling properties using, as testbeds, HPC clusters incorporating different accelerators: Intel Xeon Phi many-core processors, NVIDIA GPUs, and AMD GPUs.

**Keywords**
Lattice Boltzmann methods, accelerators, performance modeling, heterogeneous systems, performance portability

## 1 Background and related works

The architecture of high-performance computing (HPC) systems is increasingly based on accelerators; typical HPC systems today—including several leading entries in the TOP500 list (TOP500, 2016)—are large clusters of processing nodes interconnected by a fast low-latency network, each node containing standard processors coupled to one or more accelerator units.

Accelerators promise higher computing performance and better energy efficiency, improving on traditional processors by up to one order of magnitude; in fact, they (i) allow massive parallel processing by a combination of a large number of processing cores and vector units, (ii) allow for massive multi-threading, in order to hide memory access latencies, and (iii) trade a streamlined control structure (e.g. executing instructions in-order only) for additional data-path processing for a given device or energy budget.

Typical accelerated applications usually follow a host-driven approach, in which the host processor offloads (almost) all compute-intensive sections of the code onto accelerators; the host itself typically only orchestrates global program flow or processes sequential segments of the application; this approach wastes a non-negligible amount of the available computational resources, reducing overall performance.

The obvious step forward is that even compute-intensive application kernels should be executed in a balanced way on both hosts and accelerators. This improvement has been hampered so far by several non-trivial obstacles, especially because CPUs and accelerators often present different architectures, so efficient accelerated codes may involve different data structures and operation schedules. Moreover, the lack of well-established performance-portable programming heterogeneous frameworks has, so far, required the use of

University of Ferrara and INFN Ferrara, Italy

**Corresponding author:**
SF Schifano, University of Ferrara and INFN Ferrara, via Saragat 1, I-44122 Ferrara, Italy.
Email: schifano@fe.infn.it

specific programming languages (or at least proprietary variants of standard languages) for each different accelerator, harming portability, maintainability, and possibly even correctness of the application.

Improvements on this latter aspect come with the recent evolution of *directive*-based programming environments, allowing programmers to annotate their codes with hints to the compiler about available parallelization options. Several frameworks of this type have been proposed, such as the Hybrid Multi-core Parallel Programming model (HMPP) proposed by CAPS, hiCUDA (Han and Abdelrahman, 2011), OpenMPC (Lee and Eigenmann, 2010), and StarSs (Ayguadé et al., 2010). However, the most common compiler frameworks currently used for scientific codes are OpenMP (OpenMP, 2016a) and OpenACC (OpenACC, 2016). Both frameworks allow annotation of codes written in standard languages (e.g. C, C++, and Fortran) with appropriate *pragma* directives characterizing the available parallelization space of each code section. This approach leaves compilers to apply all optimization steps specific to each different target architecture and consistent with the directives, enabling, in principle, portability of codes between any supported host and accelerator device. This process is still immature, and significant limits to portability still exist. The OpenMP standard, version 4 (OpenMP, 2016b), has introduced support for—in principle—any kind of accelerator, but compilers supporting GPUs are not yet available, and the Intel Xeon Phi is, de facto, the only supported accelerator. Conversely OpenACC supports several different GPUs and, more recently, also multi-core CPU architectures, but not the Xeon Phi, and does not allow compilation of codes able to spread parallel tasks concurrently on both GPU and CPU cores. Also, neither standards address processor-specific hardware features, so non-portable proprietary directives and instructions are often necessary; for example, performance optimization on Intel CPU processors often requires Intel-proprietary compiler directives.

In spite of these weaknesses, and in the hope that a converging trend is in progress, one would like to (i) understand how difficult it is to design one common code using common domain data structures running concurrently on hosts and accelerators, which is portable and also performance-portable across traditional processors and different accelerators, and (ii) quantify the performance gains made possible by concurrent execution on host and accelerator.

To explore this problem, one has to understand the impact on performance that different data layouts and execution schedules have for different accelerator architectures. One can then define a common data layout and write a common code for a given application with optimal (or close to optimal) performance on several combinations of host processors and accelerators. Assuming that one can identify a common data layout giving good performance on several combinations of host processors and accelerators, then one has to find efficient partitioning criteria to split the execution of the code among hosts and processors.

In this paper, we tackle exactly these issues for a class of applications based on lattice Boltzmann (LB) methods, which are widely used in computational fluid dynamics. This class of applications offers a large amount of easily identified available parallelism, making LB an ideal target for accelerator-based HPC systems. We consider alternate data layouts, processing schedules, and optimal ways to compute concurrently on host and accelerator. We quantify the impact on performance, and use these findings to develop production-grade massively parallel codes. We run benchmarks and test our codes on HPC systems whose nodes have dual Intel Xeon processors and a variety of different accelerators, namely the NVIDIA K80 (NVIDIA, 2015) and AMD Hawaii GPUs (AMD, 2016), and the Intel Xeon Phi (Chrysos, 2012).

Over the years, LB codes have been written and optimized for large clusters of commodity CPUs (Pohl et al., 2004) and for application-specific machines (Belletti et al., 2009; Biferale et al., 2010; Pivanti et al., 2014). More recent work has focused on exploiting the parallelism of powerful traditional many-core processors (Mantovani et al., 2013) and of power-efficient accelerators. such as GP-GPU (Bailey et al., 2009; Biferale et al., 2013; Calore et al., 2016b) and Xeon Phi processors (Crimi et al., 2013), and even FPGAs (Sano et al., 2007).

Recent analyses of optimal data layouts for LB have been made (Wittmann et al., 2011; Shet et al., 2013a,b). However, Wittmann et al. (2011) focuses only on the *propagate* step, one of the two key kernels in LB codes, while Shet et al. (2013b) does not take vectorization into account; in Shet et al. (2013a), vectorization is considered using intrinsic functions only. None of these papers considers accelerators. In Calore et al. (2016a), we have started preliminary investigations considering only the Xeon Phi as an accelerator. Here, we extend these results in several ways: first, we take into account both *propagate* and *collision* steps used in LB simulations. Then we use a high-level approach based on compiler directives, and we also take into account NVIDIA and AMD accelerators commonly used in HPC communities. Very recently, Valero-Lara et al. (2015); Valero-Lara and Jansson (2016) have explored the benefits of LB solvers on heterogeneous systems considering different memory layouts and systems based on both NVIDIA GPUs and Xeon Phi accelerators. In our contribution, we consider a more complex LB solver (D2Q37 instead of D2Q9), a wider analysis of data layouts, and an automatic analytic way to find the optimal partitioning of lattice domains between host CPU and accelerators.

This paper is structured as follows: Section 2 introduces the main hardware features of CPUs and GPUs that we have taken into account in this paper and Section 3 gives an overview of LB methods. Section 4 discusses design and implementation options for data layouts suitable for LB codes and measures the impact of different choices in terms of performances, while Section 5 describes the implementation of codes that we have developed concurrently running on both host and accelerator. Section 6 describes two important optimization steps for our heterogeneous code and defines a performance model, while Section 7 analyzes our performance results on two different clusters, one with K80 GPUs and one with Xeon Phi accelerators. Finally, Section 8 summarizes our main results and highlights our conclusions.

## 2 Architectures of HPC systems

Heterogeneous systems have recently enjoyed increasing popularity in the HPC landscape. These systems combine, within a single processing node, commodity multi-core architectures with off-chip accelerators, GPUs, Xeon Phi many-core units, and (sometimes) FPGAs. This architectural choice comes from an attempt to boost overall performance by adding an additional processor (the accelerator) that exploits massively parallel data paths to increase performance (and energy efficiency) at the cost of reduced programming flexibility. In this section, we briefly review the architectures of the state-of-the-art accelerators that we have considered—GPUs and Xeon Phi many-core processors—focusing on the impact that their diverging architectures have on the possibility of developing a common HPC code able to run efficiently on both of them.

GPUs are multi-core processors with a large number of processing units, all executing in parallel. The NVIDIA K80 (NVIDIA, 2015) is a dual GK210 GPU, each containing 13 processing units, called streaming multiprocessors (SMX). Each processing unit has 192 compute units, called CUDA-cores, concurrently executing groups of 32 operations in a SIMT (single instruction, multiple thread) fashion; much like traditional SIMD processors, cores within a group execute the same instruction at the same time but are allowed to take different branches (at a performance penalty). The AMD FirePro W9100 (AMD, 2016) is conceptually similar to the K80 NVIDIA GPU; it has 44 processing units, each with 64 compute units (stream processors).

The clock of an NVIDIA K80 has a frequency of 573 MHz, which can be boosted up to 875 MHz. The aggregate peak performance is then 5.6 TFlops in single precision and 1.87 TFlops in double precision (only one-third of the SMXs work concurrently when performing double-precision operations). Working at 930 MHz, the processing units of the AMD FirePro W9100 deliver up to 5.2 TFlops in single precision and 2.6 TFlops in double precision.

In general, GPUs sustain their huge potential performance thanks to large memory bandwidth—to avoid starving the processors—and massive multi-threading—to hide memory access latency. Consequently, register files are huge in GPUs, as they have to store the states of many different threads, while data caches are less important. For example (see also Table 1), a K80 GPU has a combined peak memory bandwidth of 480 GB/s, while each SMX has a register file of 512 KB and just 128 KB L1 cache/shared memory; SMX units share a 1536 KB L2 cache. Similarly, the AMD W9100 has a peak memory bandwidth of 320 GB/s and its last-level cache is just 1 MB.

The architecture of Xeon Phi (Chrysos, 2012) processors—the other class of accelerators that we consider—builds on a very large number of more traditional x86 cores, each optimized for streaming parallel processing, with a streamlined and less versatile control part and enhanced vector processing facilities. For instance, the currently available version of this processor family—the Knights Corner (KNC)—integrates up to 61 CPU cores, each supporting the execution of four

**Table 1.** Selected hardware features of the systems tested in this work: Xeon E5-2630 is a commodity processor adopting the Intel Haswell micro-architecture; Xeon Phi 7120P is based on the Intel many integrated core (MIC) architecture; Tesla K80 is a NVIDIA GPU with two Tesla GK210 accelerators; FirePro W9100 is an AMD Hawaii GPU.

| | Xeon E5-2630 v3 | Xeon Phi 7120P | Tesla K80 | FirePro W9100 |
|---|---|---|---|---|
| Number of physical cores | 8 | 61 | 2 × 13 SMX | 44 |
| Number of logical cores | 16 | 244 | 2 × 2496 | 2816 |
| Clock (GHz) | 2.4 | 1.238 | 0.560 | 0.930 |
| Peak performance (double or single precision, GF) | 307/614 | 1208/2416 | 1870/5600 | 2620/5240 |
| SIMD unit | AVX2 256-bit | AVX2 512-bit | n/a | n/a |
| LL cache (MB) | 20 | 30.5 | 1.68 | 1.00 |
| Number of memory channels | 4 | 16 | – | – |
| Maximum memory (GB) | 768 | 16 | 2 × 12 | 16 |
| Memory bandwidth (GB/s) | 59 | 352 | 2 × 240 | 320 |

threads, for an aggregate peak performance of 1 TFlops in double precision, with a clock running at 1.2 GHz. Each core has a 32 KB L1-cache and a 512 KB L2-cache. The L2-caches, private at the core level, are interconnected through a bi-directional ring and data are kept coherent and indirectly accessible by all cores. The ring also connects to a GDDR5 memory bank of 16 GB, with a peak bandwidth of 352 GB/s. Each core has a vector processing unit (VPU), executing SIMD operations on vectors of 512 bits.

Present-generation GPUs and Xeon Phi processors are connected to their host through a PCI-express interface, allowing data exchange between the two processors. Typically, 16 PCI-express lanes are used, with an aggregate bandwidth of 8 GB/s, much smaller than the typical memory bandwidth of these processors, so processor-accelerator data exchanges may easily become serious performance bottlenecks.

For both processor classes, in this work we consider a host-driven heterogeneous programming model, with applications executing on both the host and the accelerator. (The Xeon Phi also supports "native mode," thus acting as an independent node capable of running applications independently. This approach will be enhanced in the next-generation Xeon Phi system, the Knights Landing, which will also be available as a stand-alone processor. We do not consider this mode of operation in this paper.) So far, different programming languages have been available for each specific accelerator; indeed, NVIDIA GPUs have a proprietary programming language, and AMD GPUs are supported by the OpenCL programming environment. On the contrary, the Xeon Phi uses the same programming environment as its Xeon multi-core counterpart, so one can develop codes following a directive-based approach (e.g. OpenMP), slightly reducing the effort of application migration. Conversely, many studies have shown that extracting a large fraction of the performance capabilities of the KNC, the first-generation Xeon Phi co-processors, still requires significant effort and fine restructuring of the code (Crimi et al., 2013; Gabbana et al., 2013; Liu et al., 2013).

Following recent improvements in directive-based programming environments, this work aims to explore ways of writing common codes that (i) have optimization features that can be exploited by traditional CPUs and both accelerator architectures, and (ii) are written using a common directive-based (e.g. OpenMP, OpenACC) programming environment.

## 3 Lattice Boltzmann methods

In this section, we sketchily introduce the computational method that we adopt, based on an advanced LB scheme. Lattice Boltzmann methods (see Succi (2001) for an introduction) are discrete in position and momentum spaces; they are based on the synthetic dynamics of *populations* sitting at the sites of a discrete lattice. At each time step, populations hop from lattice site to lattice site and then they *collide*, mixing and changing their values accordingly.

Over the years, several LB models have been developed, describing flows in two or three dimensions, and using sets of populations of different sizes (a model in $x$ dimensions based on $y$ populations is labeled $DxQy$). Populations ($f_l(x,t)$), each having a given lattice velocity $\boldsymbol{c}_l$, are defined at the sites of a discrete and regular grid; they evolve in (discrete) time according to the Bhatnagar–Gross–Krook equation

$$f_l(\boldsymbol{x} + \boldsymbol{c}_l\Delta t, t + \Delta t) = f_l(\boldsymbol{x}, t) - \frac{\Delta t}{\tau}\left(f_l(\boldsymbol{x}, t) - f_l^{(\mathrm{eq})}\right) \quad (1)$$

The macroscopic physics variables, density $\rho$, velocity $\boldsymbol{u}$, and temperature $T$ are defined in terms of $f_l(x, t)$ and $\boldsymbol{c}_l$

$$\rho = \sum_l f_l \qquad \rho\boldsymbol{u} = \sum_l \boldsymbol{c}_l f_l \qquad D\rho T = \sum_l |\boldsymbol{c}_l - \boldsymbol{u}|^2 f_l$$

the equilibrium distributions ($f_l^{(\mathrm{eq})}$) are themselves functions of these macroscopic quantities (Succi, 2001). With an appropriate choice of the set of lattice velocities $\boldsymbol{c}_l$ and of the equilibrium distributions ($f_l^{(\mathrm{eq})}$, one shows that, performing an expansion in $\Delta t$ and renormalizing the values of the physical velocity and temperature fields, the evolution of the macroscopic variables obeys the thermohydrodynamic equations of motion and the continuity equation

$$\begin{aligned} \partial_t\rho + \rho\partial_i u_i &= 0 \\ \rho D_t u_i &= -\partial_i p - \rho g\delta_{i,2} + \nu\partial_{jj}u_i \\ \rho c_v D_t T + p\partial_i u_i &= k\partial_{ii}T \end{aligned} \quad (2)$$

$D_t = \partial_t + u_j\partial_j$ is the material derivative and we neglect viscous heating; $c_v$ is the specific heat at constant volume for an ideal gas, $p = \rho T$, and $\nu$ and $k$ are the transport coefficients; $g$ is the acceleration of gravity, acting in the vertical direction. Summation of repeated indexes is implied.

In our case, we study a two-dimensional system ($D = 2$ in the following), and the set of populations has 37 elements (hence the D2Q37 acronym), corresponding to (pseudo-)particles moving up to three lattice points away, as shown in Figure 1 (Sbragaglia et al., 2009; Scagliarini et al., 2010). The main advantage of this recently developed LB method is that it automatically enforces the equation of state of a perfect gas ($p = \rho T$). Our optimization efforts have made it possible to perform large-scale simulations of convective turbulence in several physics conditions (see, e.g., Biferale et al., 2011a,b).

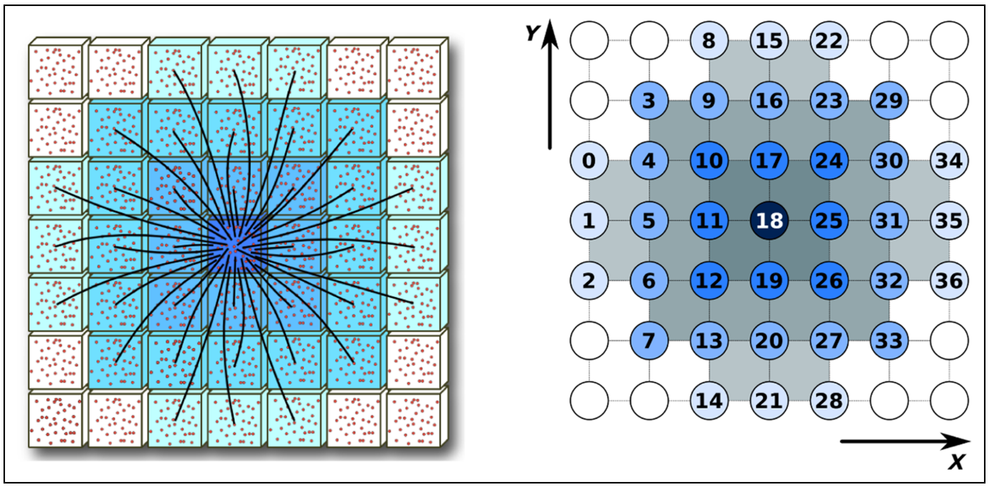


**Figure 1.** Left: Velocity vectors for the lattice Boltzmann populations of the D2Q37 model. Right: Populations are identified by an arbitrary label, associated with the lattice hop that they perform in the *propagate* phase.

An LB code starts with an initial assignment of the populations, corresponding to a given initial condition at $t = 0$ on some spatial domain, and iterates equation (1) for each population and lattice site and for as many time steps as needed; boundary conditions are enforced at the edges of the domain after each time step by appropriately modifying population values at and close to the boundary.

From the computational point of view, the LB approach offers a huge degree of easily identified available parallelism. Defining $\boldsymbol{y} = \boldsymbol{x} + \boldsymbol{c}_l \Delta t$ and rewriting the main evolution equation as

$$\begin{aligned} &f_l(\boldsymbol{y}, t + \Delta t) \\ &= f_l(\boldsymbol{y} - \boldsymbol{c}_l \Delta t, t) - \frac{\Delta t}{\tau}\left(f_l(\boldsymbol{y} - \boldsymbol{c}_l \Delta t, t) - f_l^{(\mathrm{eq})}\right) \end{aligned} \tag{3}$$

one easily identifies the overall structure of the computation that evolves the system by one time step $\Delta t$: for each point $\boldsymbol{y}$ in the discrete grid the code: (i) gathers from neighboring sites the values of the fields $f_l$ corresponding to populations drifting toward $\boldsymbol{y}$ with velocity $\boldsymbol{c}_l$ and then (ii) performs all mathematical steps needed to compute the quantities in the right-hand side of equation (3). One quickly sees that there is no correlation between different lattice points, so both steps can proceed in parallel on all grid points according to any convenient schedule, with the only constraint that step 1 precedes step 2.

As already remarked, our D2Q37 model correctly and consistently describes the thermohydrodynamic equations of motion and the equation of state of a perfect gas; the price to pay is that, from a computational point of view, its implementation is more complex than for simpler LB models. This translates to demanding requirements for memory bandwidth and floating point throughput. Indeed, step 1 implies accessing 37 neighbor cells to gather all populations, while step 2 implies $\approx$ 7000 double-precision floating point operations per lattice point, some of which can be optimized away, e.g. by the compiler.

## 4 Data layout optimization for lattice Boltzmann kernels

Our goal is to design a performance-portable code capable of running efficiently on recent Intel multi-core CPUs as well as on Xeon Phi and GPU accelerators.

Our intended common application is written in plain C and annotated with compiler directives for parallelization. For Intel architectures, we have used OpenMP and proprietary Intel directives, using the offload pragmas to run kernels on the Xeon Phi. For NVIDIA and AMD GPUs we have annotated the code with OpenACC directives, implemented by the PGI compiler, which supports both architectures. Mapping OpenACC directives on OpenMP directives is almost straightforward, so code divergence is limited at this point in time; it is expected that OpenMP implementations supporting both classes of accelerators will become available in the near future, so we hope to be able soon to merge the two versions into one truly common code.

As remarked in Section 1, data layout has a critical role in extracting performance from accelerators.

Data layouts for LB methods, as for many other stencil applications, have been traditionally based either on *array of structures* (AoS) or on *structure of arrays* (SoA) schemes. Figure 2 shows how population data are stored in the two cases; in the AoS layout, population data for each lattice site are stored one after the other at neighboring memory locations. This scheme exhibits locality of populations at a given lattice site,

```
#define N (LX*LY)
typedef struct {
  data_t  p1; // population  1
  data_t  p2; // population  2
   . . .
  data_t p37; // population 37
} pop_t;

pop_t lattice[N];
```

```
#define N (LX*LY)
typedef struct {
  data_t  p1[N]; // population  1
  data_t  p2[N]; // population  2
   . . .
  data_t p37[N]; // population 37
} pop_t;

pop_t lattice;
```

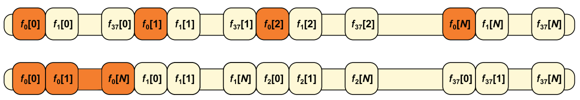


**Figure 2.** Top and Middle: C definitions of lattice populations using the array of structures and structure of arrays data layouts. Bottom: Graphic representation of the array of structures and structure of arrays layouts.

while populations of common index *i* at different lattice sites are stored in memory at non-unit-stride addresses. Conversely, in the SoA scheme for a given index *i*, populations of all lattice sites are stored contiguously, while the various populations of each lattice site are stored far from each other at non-unit-stride addresses.

In our implementation, we have arbitrarily chosen to store the lattice in column-major order (*Y* spatial direction), and to keep in memory two copies that are alternatively read and written by each kernel routine. This option, termed in the literature *ping-pong buffering* or *A-B pattern*, allows processing of all lattice sites in parallel. The *A-A pattern*, proposed in Bailey et al. (2009), is more efficient in terms of memory consumption since it stores only one copy of the lattice. However, since memory occupation is not an issue for our simulations, we have not yet implemented this further optimization.

## 4.1 Data layout optimization for `propagate`

In our LB code, the `propagate` kernel is applied to each lattice site, moving populations according to the patterns of Figure 1. For each site, `propagate` reads and writes populations from lattice cells at distances up to 3 in the physical lattice; for this reason, locality of memory accesses plays an important role for performance.

This kernel can be implemented using either a *push* or a *pull* scheme (Wittmann et al., 2011). The former moves all populations of a lattice site toward appropriate neighboring sites, while the latter gathers to one site populations stored at neighbor sites. Relative advantages and disadvantages of these schemes are not obvious and depend to some extent on the hardware features of the target processor. While the *push* scheme performs an aligned-read followed by a misaligned-write, the opposite happens if the *pull* scheme is used, and it is well known that reading or writing from or to (non-)aligned memory addresses may have a large impact on the sustained memory bandwidth of modern processors (Kraus et al., 2013). However, on cache-based Intel architectures (both standard CPUs and Xeon Phi), aligned data can be stored directly to memory using *non-temporal* write instructions. If data to be stored are not resident at any cache level, standard semantics of ordinary memory-writes require a prior read for ownership (RFO); the *non-temporal* version of store avoids the RFO read, improving effective memory bandwidth and saving time. This feature can be used in the *pull* scheme, reducing the overall memory traffic by a factor 1/3, so we adopt it.

The `propagate` kernel can, in principle, be vectorized, applying each move shown in Figure 1 to several lattice sites in parallel, e.g. moving populations with the same index *i* and belonging to two or more sites. The number of sites processed in parallel depends on the size of the vector instructions of the target processor; the vector size is 4 double words for the Xeon CPU, 8 for the Xeon Phi, 32 or multiples thereof for GPUs. However, using the AoS scheme, populations of different sites are stored at non-contiguous memory addresses, preventing vectorization. Conversely, when using the SoA layout, access to several populations of index *i* has unit stride, allowing movement in parallel populations of index *i* for several sites and allowing vectorization. This discussion suggests that the SoA layout should perform better than the AoS one. Figure 3 (left) shows the C-code written for Intel processors, adopting the SoA layout. The code sweeps all lattice sites with two loops in the *X* and *Y* spatial directions. The inner loop is on *Y*, as elements are stored in column-major order. We have annotated this loop with the `#pragma omp simd` OpenMP pragma to introduce SIMD vector instructions. Since values of populations written into `nxt` array are not re-used within this kernel, we also enable *non-temporal* stores; since this feature is not yet part of the OpenMP standard, we have used the specific Intel directive `#pragma vector nontemporal`. The equivalent code for GPUs, replacing Intel and OpenMP directives with corresponding OpenACC directives, is shown in Figure 3 (right).

The first two rows of Table 2 compare the performance obtained with the two layouts on all the processors that we consider. As expected, the SoA layout is much more efficient on GPUs, but this is not true for both Intel processors; inspection of the assembly codes and compiler logs shows that read operations are not vectorized on Intel processors, owing to *unaligned* load

```
typedef data_t double;
typedef struct { data_t s[LX*LY]; } pop_soa_t;

pop_soa_t prv[NPOP], nxt[NPOP];


#pragma omp parallel for
for ( ix = STARTX; ix < ENDX; ix++ ) {
  #pragma vector nontemporal
  for ( iy = STARTY; iy < ENDY; iy++ ) {
    idx = IDX(ix, iy);
    for ( ip = 0; ip < NPOP; ip++ ) {
      nxt[ip].s[idx] = prv[ip].s[ idx + OFF[ip] ];
    }
  }
}
```

```
typedef data_t double;
typedef struct { data_t s[LX*LY]; } pop_soa_t;

pop_soa_t prv[NPOP], nxt[NPOP];

#pragma acc kernels present(prv, nxt)
#pragma acc loop gang independent
for ( ix = STARTX; ix < ENDX; ix++ ) {
  #pragma acc loop vector independent
  for ( iy = STARTY; iy < ENDY; iy++ ) {
    idx = IDX(ix, iy);
    for ( ip = 0; ip < NPOP; ip++ ) {
      nxt[ip].s[idx] = prv[ip].s[ idx + OFF[ip] ];
    }
  }
}
```

**Figure 3.** Codes of the `propagate` kernel for (left) Intel architectures and (right) GPUs. This kernel moves populations as shown in Figure 1. `OFF` is a vector containing the memory address offsets associated to each population hop. The `prv` and `nxt` arrays use the SoA layout.

**Table 2.** Execution time (milliseconds per iteration) of the `propagate` and `collide` kernels on several architectures using different data layouts. The lattice size is 2160 × 8192 points.

| Data structure | Haswell | Xeon Phi | Tesla K80 | AMD Hawaii |
|---|---|---|---|---|
| Propagate | | | | |
| AoS | 408 | 194 | 326 | 649 |
| SoA | 847 | 224 | 36 | 57 |
| CSoA | 247 | 78 | 32 | 45 |
| CAoSoA | 286 | 89 | 33 | 50 |
| Collide | | | | |
| AoS | 1232 | 631 | 767 | 2270 |
| SoA | 1612 | 1777 | 171 | 1018 |
| CSoA | 955 | 445 | 165 | 452 |
| CAoSoA | 812 | 325 | 166 | 402 |

AoS: array of structures; CSoA: cluster structure of array; CAoSoA: clustered array of structure of array; SoA: structure of arrays.

addresses. Lacking vectorization, the AoS layout exhibits a better performance as it has a better cache hit rate.

The reason why compilers fail to vectorize the code is that load addresses are computed as the sum of the destination-site address—which is memory-aligned—and an offset, so they point to the neighbor sites from which populations are read. This does not guarantee that the resulting address is properly aligned to the vector size, i.e., 32 bytes for CPUs and 64 bytes for the Xeon Phi. Store addresses, however, are always aligned if the lattice base address is properly aligned and $Y$ is a multiple of 32 or 64.

A simple modification to the data layout solves this problem. Starting from the lattice stored in the SoA scheme, we cluster `VL` consecutive elements of each population array, with `VL` a multiple of the hardware vector size supported by the processor (e.g. 4 for the Haswell CPU, 8 for the Xeon Phi, and 32 for GPUs).

```
#define LYOVL (LY / VL)
typedef data_t double;
typedef struct { data_t c[VL]; } vdata_t;

typedef struct { vdata_t s[LX*LYOVL]; } vpop_soa_t;

vpop_soa_t prv[NPOP], nxt[NPOP];

for ( ix = startX; ix < endX; ix++ ) {
  idx = ix*LYOVL;
  for( ip = 0; ip < NPOP; ip++) {
    for ( iy = startY; iy < endY; iy++ ) {
      #pragma vector aligned nontemporal
      for(k = 0; k < VL; k++) {
        nxt[ip].s[idx + iy].c[k] =
              prv[ip].s[idx + iy + OFF[ip]].c[k];
      }
    }
  }
}
```

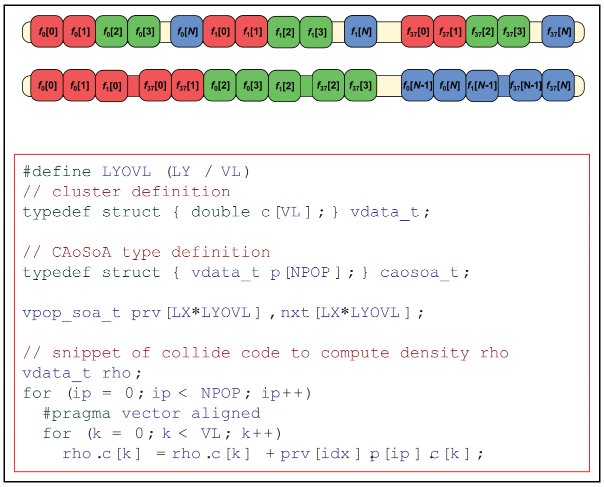
**Figure 4.** Source code of the `propagate` kernel for Intel architectures using the cluster structure of array data layout. `OFF` is a vector containing the memory address offsets associated to each population hop. `VL` is the size of a cluster (see text for details). To properly vectorize the inner loop with SIMD instructions, the value of `VL` should match the width of vector-registers supported by the target architecture.

We call this scheme *cluster structure of array* (CSoA); Figure 4 shows the corresponding `C` type definitions, `vdata_t` and `vpop_soa_t`. `vdata_t` holds `VL` data words corresponding to the same population of index $i$ at `VL` different sites that can be processed in SIMD fashion. `vpop_soa_t` is the type definition for the full lattice data. Using this scheme, move operations generated by `propagate` apply to clusters of populations and not to individual population elements. Since clusters have the same size as hardware vectors, all read operations are now properly aligned. As in the case of the SoA data layout, write operations always have aligned accesses and non-temporal stores can be used. Figure 4 shows the corresponding code; in this case, we have also rearranged the order of the loops in a way which reduces the pressure on the translation lookaside

**Table 3.** Profiling results provided by the Intel VTune profiler for the `collide` kernel on a lattice of $2160 \times 8192$ points, comparing the CSoA and the CAoSoA schemes on the Xeon CPU and Xeon Phi processors.

| Metric | CSoA | CAoSoA | Threshold |
|---|---|---|---|
| Xeon Phi | | | |
| L1 TLB miss ratio | 2.66% | 0.06% | 1.0% |
| L2 TLB miss ratio | 2.00% | 0.00% | 0.1% |
| Xeon CPU | | | |
| LLC miss count | 787,647,256 | 177,010,620 | n/a |
| Average latency (cycles) | 13 | 9 | n/a |

CSoA: cluster structure of array; CAoSoA: clustered array of structure of array; LLC: last-level cache; TLB: translation lookaside buffer.


$f_0[0]$ $f_0[1]$ $f_0[2]$ $f_0[3]$ $f_0[N]$ $f_1[0]$ $f_1[1]$ $f_1[2]$ $f_1[3]$ $f_1[N]$ $f_{37}[0]$ $f_{37}[1]$ $f_{37}[2]$ $f_{37}[3]$ $f_{37}[N]$
$f_0[0]$ $f_0[1]$ $f_1[0]$ $f_{37}[0]$ $f_{37}[1]$ $f_0[2]$ $f_0[3]$ $f_1[2]$ $f_{37}[2]$ $f_{37}[3]$ $f_0[N-1]$ $f_0[N]$ $f_1[N-1]$ $f_{37}[N-1]$ $f_{37}[N]$

```
#define LYOVL (LY / VL)
// cluster definition
typedef struct { double c[VL]; } vdata_t;

// CAoSoA type definition
typedef struct { vdata_t p[NPOP]; } caosoa_t;

vpop_soa_t prv[LX*LYOVL], nxt[LX*LYOVL];

// snippet of collide code to compute density rho
vdata_t rho;
for (ip = 0; ip < NPOP; ip++)
  #pragma vector aligned
  for (k = 0; k < VL; k++)
    rho.c[k] = rho.c[k] + prv[idx].p[ip].c[k];
```

**Figure 5.** Top: data arrangement for the CAoSoA layout; for illustration purposes, we take VL = 2. Bottom: sample code for `collide` using this layout.

buffer (TLB) cache. As before, code for GPUs can be obtained by replacing directives with *OpenAcc* ones.

Table 2 (see the first three rows of the `propagate` section) quantifies the impact of the data layout on performance, showing benchmark results using the three different data layouts, AoS, SoA, and CSoA. The advantages of using the CSoA data layout are large for Intel architectures, while improvements are marginal for GPUs, as they are less sensitive to misaligned memory reads (Kraus et al., 2013). The relevant result is, however, that using the CSoA format we have one common data layout that maximizes performance for `propagate` on all processors.

## 4.2 Data layout optimization for `collide`

The `collide` kernel is computed after the `propagate` step, reading, at each lattice site, populations gathered by the `propagate` phase. It updates their values, applying the *collisional* operator and performing all mathematical operations associated with equation (1). For each lattice site, this floating point intensive kernel uses only population data associated with the site on which it operates; lattice sites are processed independently of each other making processing of the lattice fully parallelizable. In this case, in contrast with the `propagate` kernel, locality of populations plays an important role for performances. Vectorization of this step is implemented, as for `propagate`, trying to process different sites in parallel.

Following results obtained in Section 4.1, we consider first the CSoA data layout, which—as seen before—gives very good performance results with the `propagate` kernel. The log files of the compiler show that the CSoA scheme allows vectorization of the code, but profiling the execution of the code on Xeon CPUs and Xeon Phi accelerators, we have observed a large number of TLB and (last-level cache) LLC misses, suggesting that further improvements could be put in place. Table 3 shows the results (see CSoA column) provided by the Intel VTune profiler for both Xeon CPU and Xeon Phi; for the latter, the miss ratios are much bigger than the threshold values provided by the profiler itself. These penalties arise because, in the CSoA scheme, different populations associated with the same lattice site are stored at memory addresses far from each other, so several non-unit-stride memory accesses are necessary to gather all relevant data words.

To overcome these problems, we further modify the CSoA scheme, and define a new data layout, which takes into account the locality requirements of both `propagate` and `collide` kernels. In this scheme, for each population array, we divide each $Y$ column into `VL` partitions, each of size `LY/VL`; all elements sitting at the $i$th position of each partition are then packed together into an array of `VL` elements called a *cluster*. We call this layout, a *clustered array of structure of array* (CAoSoA); Figure 5 shows how data are arranged in the memory. This data layout still allows vectorization of inner structures (clusters) of size `VL`, and at the same time improves locality—with respect to the CSoA—of populations, as it keeps all population data needed to process each lattice site at close and aligned addresses. Figure 5 shows the definition of the `vdata_t` data type, corresponding to a *cluster*, and representative small sections of the `collide` code for Intel processors. Cluster variables are processed iterating on all elements of the cluster through a loop over `VL`; `pragma vector aligned` instructs the compiler to fully vectorize the loop, since all iterations are independent and memory accesses are aligned. This data layout combines the benefits of the CSoA scheme, allowing aligned memory accesses and vectorization (relevant for the *propagate* kernel) and at the same providing population locality (together relevant for the *collide* kernel).

Table 3 shows the impact of the CAoSoA data layout on memory misses: on Xeon Phi, the TLB misses have been reduced well below the threshold values, and on Xeon CPU have been reduced by a factor of 4.5 with respect to the CSoA scheme. Table 2 shows the execution time of the `collide` kernel run using all data data layouts defined so far. As we see, the CAoSoA improves performances over the CSoA on Intel and AMD processors, while for NVIDIA GPU the two layouts give marginal differences in performance. These gains in the performance of `collide` come at a limited cost (12–16%) for *propagate* on all architectures except for the K80, so CAoSoA maximizes the combined performances of the two kernels.

The overall performance gain for the `propagate` and `collide` kernels using the CAoSoA scheme is approximately 1.5 × over AoS, 2 × over SoA, and 1.1 × over CSoA for Intel (Haswell) CPUs. For the Xeon Phi, the speed-up is 2 × over SoA, 4.8 × over AoS, and 1.2 × over CSoA.

## 5 Heterogeneous implementation

In this section, we describe the implementation of our code designed to involve and exploit compute capabilities of both host and accelerators. We only consider the CAoSoA layout, as it grants the best overall performances on all the processors and accelerators that we have studied.

Our implementation uses MPI libraries and each MPI process manages one accelerator. The MPI process runs on the host CPU; part of the lattice domain is processed on the host itself, and part is offloaded and processed by the accelerator. Using one MPI process per accelerator makes it easy to extend the implementation to a cluster of accelerators installed on either the same or different hosts.

A lattice of size `GSIZEX` × `GSIZEY` is partitioned among `NMPI` MPI processes, along one direction, in our case the *X*-direction, and each slice of size `SIZEX` × `GSIZEY` is assigned to a different MPI process (with `SIZEX=GSIZEX/NMPI`). Within each MPI process, each partition of size `SIZEX` × `SIZEY` is further divided between host and accelerator. We define three regions, namely left border, bulk, and right border, as shown in Figure 6. The right and left borders include *M* columns and are allocated to the host memory while the remaining SIZEX − 2*M* columns stay on the accelerator memory. As `propagate` stencils require neighbor sites at distances up to 3 to be accessed (see Figure 1), each region is surrounded by a halo of three columns and rows. Each halo stores a copy of lattice sites of the adjacent region either allocated on the host, on the accelerator, or on the neighbor MPI process. The use of halos allows the `propagate` kernel to be applied uniformly to all sites, avoiding divergences

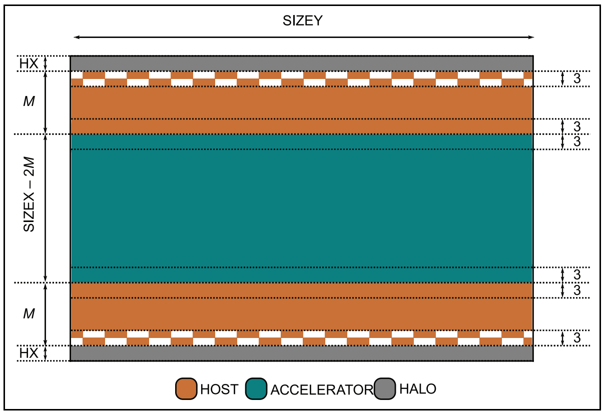


**Figure 6.** Logic partitioning of the lattice domain among host and accelerator. The central (dark-green) region is allocated on the accelerator, side (orange and gray) regions on the host. Checkerboard textures flag lattice regions involved in MPI communications with neighbor nodes.

in the computation. Allocation of lattice borders (*M* columns each) on the host avoids dependencies between MPI and device-to-host and host-to-device data transfers. This allows full overlap of the computation performed on the accelerator, MPI communications, and processing performed on the CPU.

Each MPI process performs a loop over time steps, and at each iteration it launches, in sequence, the `propagate`, `bc`, and `collide` kernels on the accelerator, processing the bulk region. To allow the CPU to operate in overlapped mode, kernels are launched asynchronously on the same logical execution queue to ensure in-order execution. After launching the kernels on the accelerator, the host first updates halos with adjacent MPI processes and then starts to process its left and right borders, applying the same sequence of kernels. After the processing of borders completes, the host updates the halo regions shared with the accelerator; this step moves data between host and accelerator. The control flow of the code executed by the MPI process is shown in Figure 7, where the `bc` kernel applies the physical boundary conditions at the three uppermost and lowermost rows of the lattice.

The code for KNC is implemented using the offload features available in the Intel compiler and runtime framework. In this case, we have a unique code and the compiler produces the executable codes for both Xeon Phi and CPU. For GPUs, the situation is somewhat different. In fact, we have to use two different compilers, one for GPUs and one for CPUs. We have written the code for GPUs, both NVIDIA and AMD, using *OpenACC* directives (Calore et al., 2016c) and compiled using the PGI 16.5 compiler, which supports both architectures. The kernels running on CPUs are written using standard `C` and compiled using Intel compiler ICC v16.0. Then we have linked the two codes in one single executable.

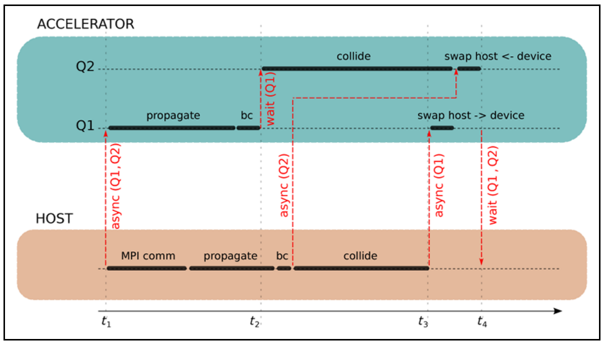


**Figure 7.** Control flow executed by each MPI process. The schedule executed on the accelerator is on the upper band, while that executed by the host is on the lower band. Execution on the accelerator runs on two concurrent queues. Synchronization points are marked with red lines.

## 6 Parameter optimization

In this section, we describe two important optimization steps for our heterogeneous code. The first is about the optimal partitioning of the computation between host and accelerator, and the second is about the optimal cluster size for the CAoSoA data layout.

### *6.1 Workload partitioning*

Hosts and accelerators have different peak (and sustained) performance, so careful workload balancing between the two concurrent processors is necessary. We model the execution time $T_{\mathrm{exe}}$ of our code with the following set of equations

$$T_{exe} = \max\{T_{\mathtt{acc}}, T_{\mathtt{host}} + T_{\mathtt{mpi}}\} + T_{\mathtt{swap}} \tag{4}$$

$$T_{\mathtt{acc}} = (LX - 2M)LY \cdot \tau_{\mathrm{d}} \tag{5}$$

$$T_{\mathtt{host}} = (2M)LY \cdot \tau_{\mathrm{h}} \tag{6}$$

$$T_{\mathtt{mpi}} = \tau_{\mathrm{c}} \tag{7}$$

where $T_{\mathtt{acc}}$ and $T_{\mathtt{host}}$ are the execution times of the accelerator and host, respectively, $T_{\mathtt{swap}}$ is the time required to exchange data between host and accelerator at the end of each iteration, and $T_{\mathtt{mpi}}$ is the time to move data between two MPI processes in a multi-accelerator implementation. As $T_{\mathtt{swap}}$ is independent of $M$, $T_{\mathrm{exe}}$ is minimal for a value $M^*$, for which the following equation holds

$$T_{\mathtt{acc}}(M^*) = T_{\mathtt{host}}(M^*) + T_{\mathtt{mpi}}(M^*) \tag{8}$$

Our code has an initial auto-tuning phase, in which it runs a set of mini-benchmarks to estimate approximate values of $\tau_{\mathrm{d}}$, $\tau_{\mathrm{h}}$, and $\tau_{\mathrm{c}}$. These are then inserted in equation (8) to find $M^*$, an estimate of the value of $M$ that minimizes the time to solution.

Figure 8 shows the performance of our code for three different lattice sizes as a function of $2M/LX$, the fraction of lattice sites that we map on the host CPU. We have run our tests on two different machines, the *Galileo* HPC system installed at *CINECA* and the *Etna* machine. *Galileo* has two different partitions, one with K80 GPUs and one with KNC accelerators. The *Etna* machine is part of the *COKA* experimental cluster at the University of Ferrara, and has two AMD Hawaii GPUs. Both machines use an 8-core Intel Xeon

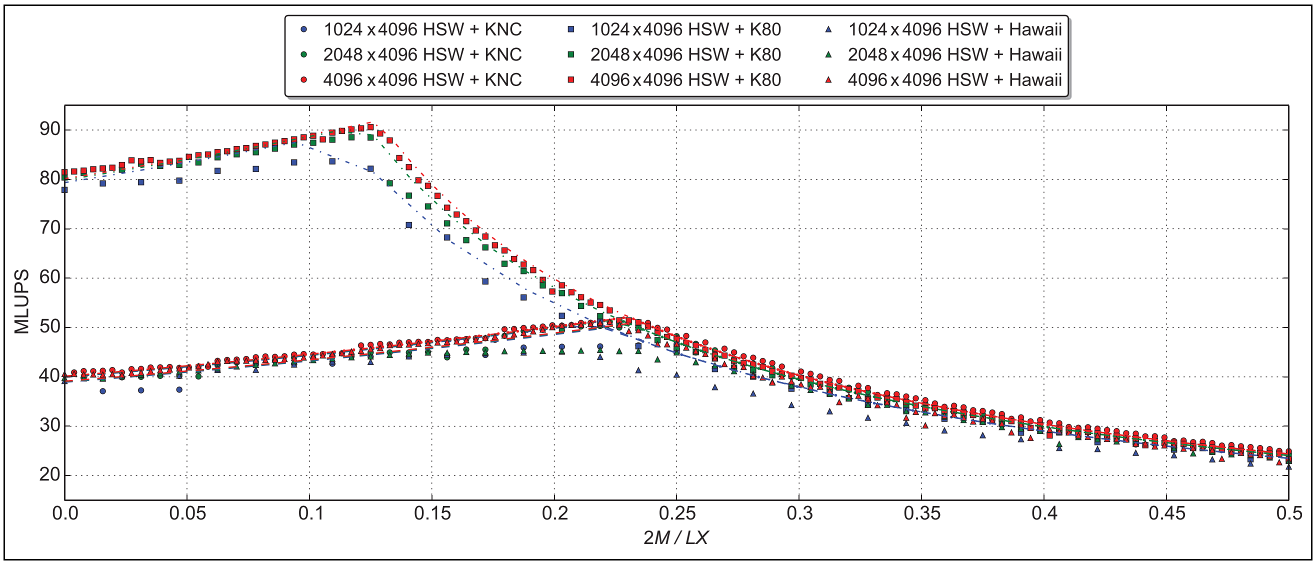


**Figure 8.** Performance of the heterogeneous code (measured in MLUPS (million updates per second)) for all three platforms, as a function of the fraction of lattice sites (*2M / LX*) mapped on the Haswell (HSW) host CPU. KNC is the Intel Knights Corner accelerator, K80 is the NVIDIA Tesla GPU, and Hawaii is the AMD GPU. Dots are measured values, dashed lines are the prediction of our model.

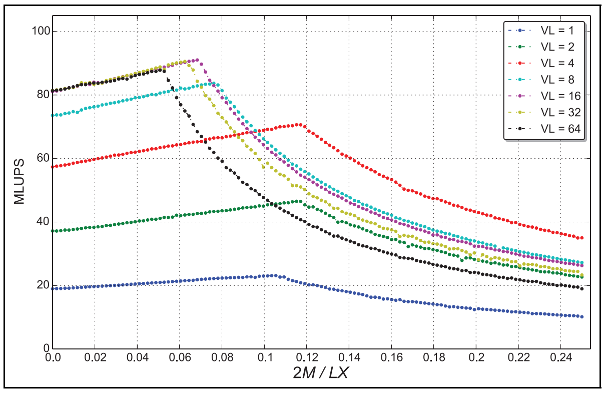


**Figure 9.** Performance (measured in MLUPS (million updates per second)) for different values of the `VL` parameter in the CaoSoA data layout; results are for a Tesla K80 GPU.

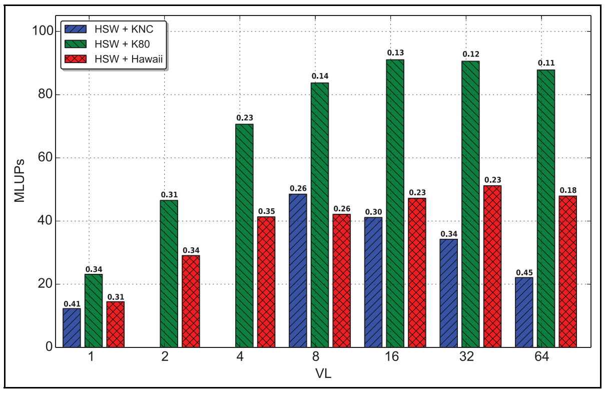


**Figure 10.** Impact on performance of different cluster sizes (`VL`) in the CaoSoA data layout for several accelerator choices.

E5-2630v3 CPU based on the *Haswell* micro-architecture as host processor, and each host has one attached accelerator. Performance is measured using the *million updates per second* (MLUPS) metric, a common option for this class of applications. In Figure 8, dots are measured values, while lines are our predictions. Our auto-tuning strategy predicts performance with good accuracy, and estimates the workload distribution between host and device for which the execution time reaches its minimum. An interesting features of this plot is the fact that the optimal point is—for each accelerator—a function of $2M / LX$. As expected, for values of $M < M^*$ and $M > M^*$ performances decrease because the workload is unbalanced on either the accelerator or the host side; results at $2M / LX = 0$ correspond to earlier implementations in which critical kernels are fully offloaded to accelerators; we see that running these kernels concurrently on host and accelerators (KNC and K80) increases performances by approximately $10 - 20\%$. Finally, as $M$ becomes much larger than $M^*$, all lines in the plot fall on top of each other, as in this limit the host CPU handles the largest part of the overall computation.

## 6.2 Fine-tuning of data layout cluster size

An important parameter of the CAoSoA layout is the cluster size `VL`, as performance depends significantly on its value. This parameter, whose optimal value correlates with the hardware features of the target processor, affects data allocation in memory and must be fixed at compile time.

Figure 9 shows the impact on performances (measured again in MLUPS) of our code running on a node with K80 GPUs and using different choices for `VL`. We see that, for this processor, a wrong choice may reduce performance by large factors ($\simeq 5$); the good news is that there is a reasonably large interval `VL` $= 16, 32, 64$ for which performance is close to its largest value. We have made the same measurements for all kind of nodes available with KNC and AMD GPUs, and then picked, for each node and each value of `VL`, the best operating point in terms of $2M / LX$. In this way, we select the highest performance for each combination of host and accelerator. These results are collected in Figure 10 as a function of `VL`; on top of each bar we report the corresponding value of $2M / LX$. We see that GPUs are more robust than the KNC against a non-optimal choice of `VL`: for the former processors, performance remains stable as long as `VL` is large enough, while for the latter only one or two `VL` values allow the highest performance to be attained. Fortunately enough, there is a window of `VL` values for which all systems are close to their best performance.

## 6.3 Performance prediction on new hardware

A further interesting result of the model developed in Section 6.1 is that we can use it to predict to which extent the performance of our codes is affected if either the host CPU or the accelerator is replaced by a different processor; in particular, one may ask what happens if announced but not yet available processors or accelerators are adopted. One such exercise replaces the host processor that we have used for our previous tests with the new Intel multi-core Xeon E5-2697v4, based on the latest *Broadwell* micro-architecture. We have run our code on a *Broadwell* processor with no attached accelerators and measured the host-related performance parameters used in equation (4); we have then used these parameters to estimate the expected performance of a would-be machine whose nodes combine Broadwell hosts with either K80 or KNC accelerators. Results are shown in Figure 11, which compares the measured performance on the current hardware (dots) with the predictions of our model (dashed lines). We see that, using this new more powerful processor, performance for our code would improve by

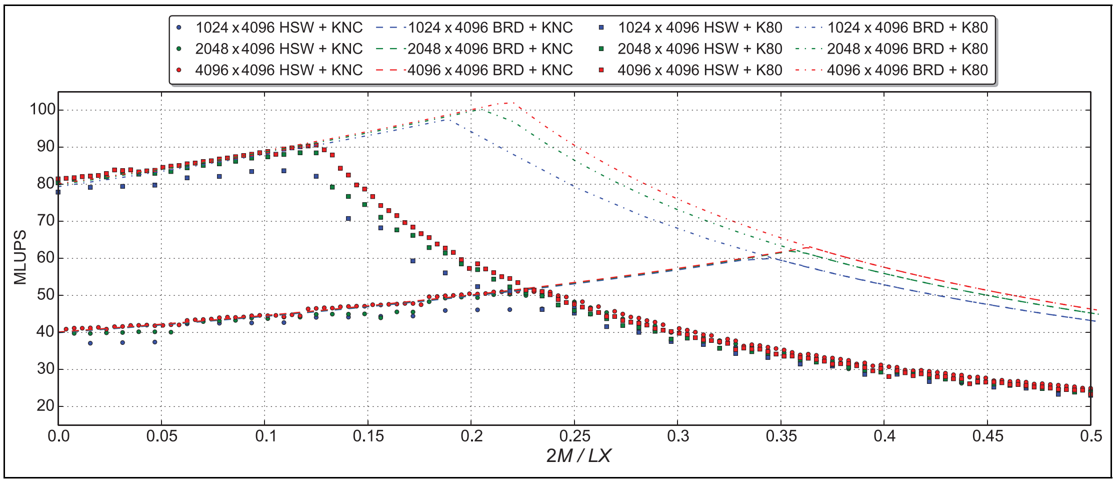


**Figure 11.** Performance predictions (dashed lines) of our model for a would-be system using as host the recently released Broadwell (BRD) CPU compared with measured data on a Haswell (HSW) CPU (dots). Measurements refer to three different lattice sizes.

approximately 20%, when perfectly balancing the workload between host and accelerator. As expected, the improvement is the same for both types of accelerator, since we have only virtually replaced the host processor. Another interesting feature of the plot is that the model predictions overlap with measured data when $2M / LX$ values tend to zero; this is expected, since in this case the fraction of lattice sites mapped on the host CPU tends to zero and the execution time is dominated by the accelerator. A similar analysis might be performed, for instance, to assess the overall performance gains to be expected when next-generation GPUs become available.

## 7 Scalability performances

In this section, we analyze scalability performances of our codes running on the *Galileo* machine, both on the K80 and on the KNC partition.

In Figure 12 (top left), we show the performance of our code running on larger and larger KNC partitions of the *Galileo* cluster, and for several physically relevant sizes of the physical lattice, showing the scaling results of this code. We compare with a previous `v1` implementation of the code running all kernels on the KNC accelerator. Figure 12 (top right) shows the same data as the previous picture, except that it shows the speed-up factor as a function of the number of accelerators. One easily sees that the new heterogeneous version of the code is not only faster than its accelerator-only counterpart but also has a remarkably better behavior from the point of view of hard scaling. This is because data moved through MPI communications between different processing nodes is always resident on the host, saving time to move them to and from KNC accelerator. All in all, for massively parallel runs on many accelerators, the heterogeneous code extracts from the same KNC-based hardware system roughly twice the performance of the earlier version.

In Figure 12 (bottom left), we show the performance of our code running on the K80 partition. In this case, owing to the larger difference in performance between the host CPU and the accelerator, we have a different behavior. The newer version (`v2`) is faster than the earlier version (`v1`), but the gain is smaller than for the KNC because the gap in performance between the K80 GPU and the host CPU is larger. Scalability (see Figure 12, bottom right) of `v2`, is as good as version `v1` because in both implementations MPI communications are fully overlapped with computation (Calore et al., 2015).

## 8 Analysis of results and conclusions

In this section, we analyze our results and outline our conclusions.

The first important contribution of this work highlights the critical role played by data layouts in the development of a common LB code that runs concurrently on CPUs and accelerators and is also *performance*-portable onto different accelerator architectures. The crucial finding in this respect is that data memory organization should support, at the same time, efficient memory accesses and vector processing. This challenge is made more difficult because different kernels (in our case, the critical `propagate` and `collide` routines) have conflicting requirements. Table 2 substantiates the relevance of this problem and quantifies the improvements that we have achieved. Previously used data structures are the AoS layout, supporting data locality, and the *SoA layout*, already known to exploit more vectorization, especially for GPUs. The problem with

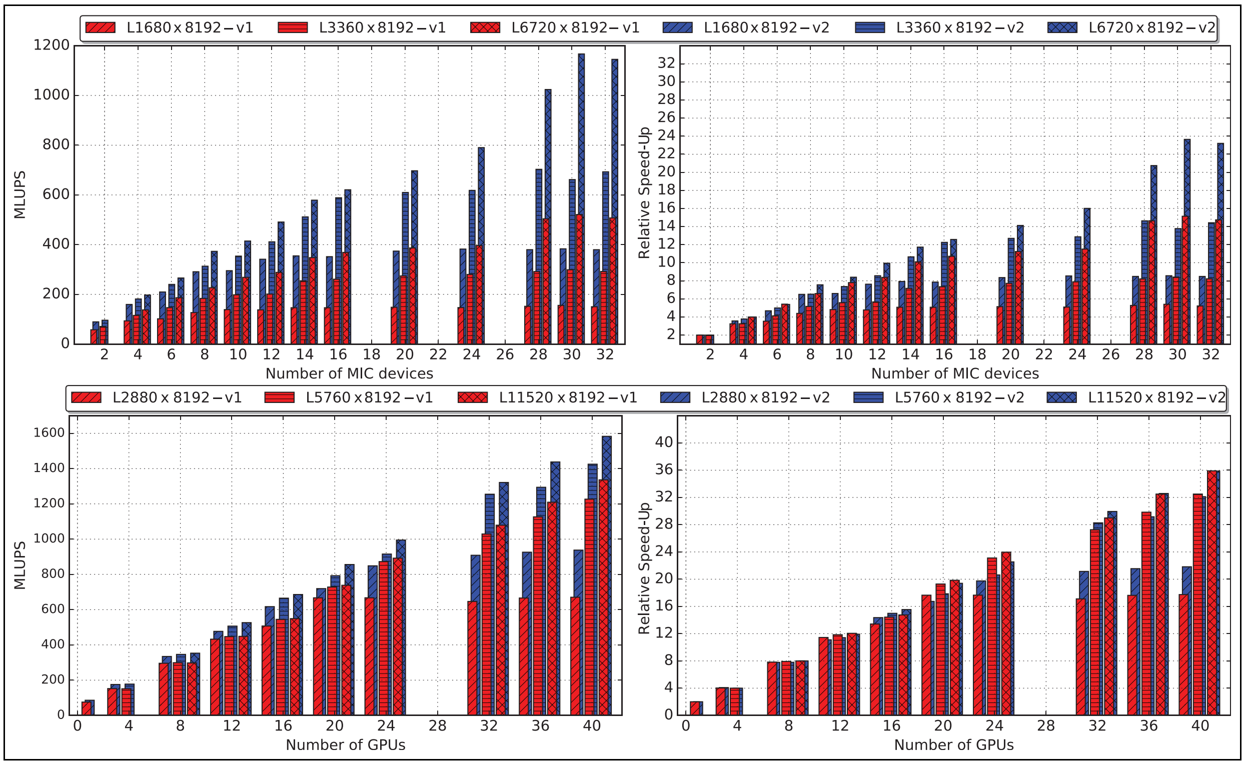


**Figure 12.** Multi-node scalability results measured on the KNC (top) and K80 (bottom) partitions of the Galileo cluster. We compare performances—MLUPS and relative speed-up—on three different lattice sizes of the heterogeneous code described in this paper (v2) with an earlier version (v1), with all critical computational kernels running on accelerators only.

these two layouts is that the former allows better performances for Intel architectures but is dramatically bad for GPUs, with performance losses that are up to $10\times$ for `propagate` and $2\times-5\times$ for `collide`. The latter scheme is very efficient on GPUs but fails on CPUs, as it limits code vectorization and causes memory management overheads (such as TLB and cache misses).

This paper introduces two slightly more complex data layouts, the CSoA and the CAoSoA, to exploit vectorization on both classes of architecture while still guaranteeing efficient data memory accesses and vector processing for both critical kernels. These data structures differ from those used in Shet et al. (2013a) and Valero-Lara et al. (2015) in the way that populations are packed into SIMD vectors; this allows operations involved in the `propagate` kernel to be properly translated into SIMD instructions, and performance of aligned memory accesses.

The CSoA layout improves performance on Intel architectures by factors of 2 over the AoS for `propagate` and 1.5 for `collide`. On GPUs, it also shows marginal improvements on the already very good results that GPUs have with the original SoA layout. A final improvement is given by the CAoSoA layout, which further increases data locality without introducing vectorization penalties. Again, performance remains substantially stable on GPUs in this case, while there are still further improvements on Intel architectures for `collide` (≈10–20%) and a corresponding penalty for `propagate`; since the former routine has a larger impact on overall performance, the CAoSoA layout is the most efficient to be used for the whole code.

It can be interesting to compare our results with those obtained by Shet et al. (2013a), who have also analyzed the relative advantages of several data layouts. This comparison is necessarily partial, since Shet et al. (i) use three-dimensional lattices and a different LB scheme (D3Q27); (ii) run on earlier processor architectures (Intel E5-2670, using the SandyBridge microarchitecture); and (iii) do not use accelerators, so they map the whole lattice on the CPU. In spite of these differences, the comparison can still be meaningful for the `propagate` kernel. On a lattice of $256^3$ sites adopting their hybrid data structure, Shet et al. (2013a) report a performance of 30 MLUPS. This corresponds to a memory bandwidth of approximately 13 GB/s, that is, 25% of the E5-2670 raw peak (51.2 GB/s). In our code, we measure a bandwidth of approximately 36 GB/s, that is, $\simeq$ 60% of the E5-2630v3 raw peak, on a $2160\times8192$ lattice and adopting the CAoSoA data structure. A final remark is about code portability: the implementation of Shet et al. (2013a) uses Intel intrinsics functions to vectorize the code, which are not portable across different architectures, while we use a directive-based approach able to run the same code on different platforms.

The definition of the appropriate layouts has allowed us to code our LB application in a common program that executes concurrently on host CPUs and all kinds of accelerators, obviously improving the portability and long-term support of this code.

Another important contribution of this paper is the development of an analytic model (see Section 6.1) that is able to predict the optimal partitioning of the workload among host CPU and accelerators; using this model, we can automatically tune this parameter for best performance on running systems, or predict performances for not yet available hardware configurations (see Section 6.3).

The final result of this contribution is that single node performances can be improved by $\approx$10–20% with respect to earlier implementations that simply wasted the computational power offered by the host CPU. Our implementation also significantly improves the (hard)-scaling behavior on relatively large clusters (see Figure 12). This follows from the fact that node-to-node communications in our code do not imply host-accelerator transfers. This improvement is very large on KNC-accelerated clusters, while on GPUs, owing to the larger performance unbalance between host and accelerator, the observed improvement is smaller.

In conclusion, an important result of this work is that it is possible to design and implement directive-based codes that are performance-portable across different present—and hopefully future—architectures. This requires an appropriate choice of the data layout, which is able to meet conflicting requirements of different parts of the code and to match hardware features of different architectures. Defining these data layouts is today in the hands of programmers, and still out of the scope of currently available and stable compilers commonly used in the HPC context. For this reason, on a longer time horizon, we look forward to further progress in compilers allowing data definitions that abstract from the actual in-memory implementation, and of tools able to make appropriate allocation choices for specific target architectures.

Another related open question, that we leave for further investigation, is whether our proposed hybrid approach has a positive impact on energy-related aspects (e.g. reducing the value of *energy-to-solution* for the given computation).

## Acknowledgements

We thank CINECA (Bologna, Italy) for access to the Galileo computing cluster.

## Declaration of Conflicting Interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

## Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work was conducted within the framework of the COKA, COSA and Suma projects of INFN. AG was supported by the European Union's Horizon 2020 research and innovation program under the Marie Sklodowska-Curie grant agreement (no 642069).

## References

AMD (2016) AMD FirePro™ W9100 workstation graphics. Technical report. Houston, TX: Advanced Micro Devices, Inc.

Ayguadé E, Badia RM, Bellens P, et al. (2010) Extending OpenMP to survive the heterogeneous multi-core era. *International Journal of Parallel Programming* 38(5–6): 440–459.

Bailey P, Myre J, Walsh SDC, et al. (2009) Accelerating lattice Boltzmann fluid flow simulations using graphics processors. In: *International conference on parallel processing, ICPP*, Vienna, Austria, 22–25 September, pp.550–557. Piscataway, NJ: IEEE.

Belletti F, Biferale L, Mantovani F, et al. (2009) Multiphase lattice Boltzmann on the cell broadband engine. *Nuovo Cimento della Societa Italiana di Fisica C* 32(2): 53–56.

Biferale L, Mantovani F, Pivanti M, et al. (2010) Lattice Boltzmann fluid-dynamics on the QPACE supercomputer. *Procedia Computer Science* 1(1): 1075–1082.

Biferale L, Mantovani F, Pivanti M, et al. (2013) An optimized D2Q37 lattice Boltzmann code on GP-GPUs. *Computers & Fluids* 80: 55–62.

Biferale L, Mantovani F, Sbragaglia M, et al. (2011a) Reactive Rayleigh–Taylor systems: Front propagation and non-stationarity. *Europhysics Letters* 94(5): 54004.

Biferale L, Mantovani F, Sbragaglia M, et al. (2011b) Second-order closure in stratified turbulence: Simulations and modeling of bulk and entrainment regions. *Physical Review E* 84(1): 016305.

Calore E, Demo N, Schifano SF, et al. (2016a) Experience on vectorizing lattice Boltzmann kernels for multi- and many-core architectures. In: Wyrzykowski R, Deelman E, Dongarra J, et al. (eds) *Parallel Processing and Applied Mathematics*. Cham: Springer International Publishing.

Calore E, Gabbana A, Kraus J, et al. (2016b) Massively parallel lattice Boltzmann codes on large GPU clusters. *Parallel Computing* 58: 1–24.

Calore E, Gabbana A, Kraus J, et al. (2016c) Performance and portability of accelerated lattice Boltzmann applications with OpenACC. *Concurrency and Computation: Practice and Experience* 28(12): 3485–3502.

Calore E, Marchi D, Schifano SF, et al. (2015) Optimizing communications in multi-GPU lattice Boltzmann simulations. In: *International conference on high performance* *computing simulation (HPCS)*, Netherlands, Amsterdam, 20–24 July 2015, pp.55–62. Piscataway, NJ: IEEE.

Chrysos G (2012) Intel Xeon Phi X100 family coprocessor—the architecture. Technical report. 12 November. Santa Clara, CA: Intel Corp.

Crimi G, Mantovani F, Pivanti M, et al. (2013) Early experience on porting and running a lattice Boltzmann code on the Xeon-Phi co-processor. *Procedia Computer Science* 18: 551–560.

Gabbana A, Pivanti M, Schifano SF, et al. (2013) Benchmarking MIC architectures with Monte Carlo simulations of spin glass systems. In: *20th annual international conference on high performance computing*, Bangalore, India, 18–21 December 2013, pp.378–385. Piscataway, NJ: IEEE.

Han T and Abdelrahman T (2011) hiCUDA: High-level GPGPU programming. *IEEE Transactions on Parallel and Distributed Systems* 22(1): 78–90.

Kraus J, Pivanti M, Schifano SF, et al. (2013) Benchmarking GPUs with a parallel lattice-Boltzmann code. In: *25th international symposium on computer architecture and high performance computing (SBAC-PAD)*, Porto de Galinhas, Brazil, 23–26 October 2013, pp.160–167. Piscataway, NJ: IEEE.

Lee S and Eigenmann R (2010) OpenMPC: Extended OpenMP programming and tuning for GPUs. In: *Proceedings of the 2010 ACM/IEEE international conference for high performance computing, networking, storage and analysis*, New Orleans, LA, 13–19 November 2010. Washington DC: IEEE Computer Society.

Liu X, Smelyanskiy M, Chow E, et al. (2013) Efficient sparse matrix-vector multiplication on x86-based many-core processors. In: *Proceedings of the 27th international ACM conference on supercomputing*, Eugene, OR, 10–14 June 2013, pp.273–282. New York, NY: ACM.

Mantovani F, Pivanti M, Schifano SF, et al. (2013) Performance issues on many-core processors: A D2Q37 lattice Boltzmann scheme as a test-case. *Computers & Fluids* 88: 743–752.

NVIDIA (2015) *Tesla K80 GPU accelerator-board specification*. Technical report. Parsippany, NJ: NVIDIA.

OpenACC (2016) OpenACC directives for accelerators. Available at: http://www.openacc-standard.org (accessed 1 September 2016).

OpenMP (2016a) The OpenMP API specification for parallel programming. Available at: http://www.openmp.org/ (accessed 1 September 2016).

OpenMP (2016b) OpenMP application program interface version 4.0. Available at: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf (accessed 1 September 2016).

Pivanti M, Mantovani F, Schifano SF, et al. (2014) An optimized lattice Boltzmann code for BlueGene/Q. In: Wyrzykowski R, Dongarra J, Karczewski K, et al. (eds) *Parallel Processing and Applied Mathematics*. Berlin: Springer, pp.385–394.

Pohl T, Deserno F, Thurey N, et al. (2004) Performance evaluation of parallel large-scale lattice Boltzmann applications on three supercomputing architectures. In: *Proceedings of the 2004 ACM/IEEE conference on supercomputing*, Pittsburgh, PA, 6–12 November 2004. Washington, DC: IEEE Computer Society.

Sano K, Pell O, Luk W, et al. (2007) FPGA-based streaming computation for lattice Boltzmann method. In: *International conference on field-programmable technology, 2007*, Kitakyushu, Japan, 12–14 December 2007, pp.233–236. Piscataway, NJ: IEEE.

Sbragaglia M, Benzi R, Biferale L, et al. (2009) Lattice Boltzmann method with self-consistent thermo-hydrodynamic equilibria. *Journal of Fluid Mechanics* 628: 299–309.

Scagliarini A, Biferale L, Sbragaglia M, et al. (2010) Lattice Boltzmann methods for thermal flows: Continuum limit and applications to compressible Rayleigh–Taylor systems. *Physics of Fluids* 22(5): 055101.

Shet AG, Siddharth K, Sorathiya SH, et al. (2013a) On vectorization for lattice based simulations. *International Journal of Modern Physics C* 24: 1340011.

Shet AG, Sorathiya SH, Krithivasan S, et al. (2013b) Data structure and movement for lattice-based simulations. *Physical Review E* 88: 013314.

Succi S (2001) *The Lattice Boltzmann Equation: For Fluid Dynamics and Beyond*. Oxford University Press, Oxford.

TOP500 (2016) Top 500 the list. Available at: http://top500.org (accessed 1 September 2016).

Valero-Lara P and Jansson J (2016) Heterogeneous CPU + GPU approaches for mesh refinement over lattice-Boltzmann simulations. *Concurrency and Computation: Practice and Experience* 29(7): e3919.

Valero-Lara P, Igual FD, Prieto-Matas M, et al. (2015) Accelerating fluid–solid simulations (lattice-Boltzmann & immersed-boundary) on heterogeneous architectures. *Journal of Computational Science* 10: 249–261.

Wittmann M, Zeiser T, Hager G, et al. (2011) Comparison of different propagation steps for the lattice Boltzmann method. *Computers & Mathematics with Applications* 65(6): 924–935.

## Author biographies

*E Calore* received his BSc and MSc degrees in Computer Engineering from the Università degli Studi di Padova (Italy) in 2006 and 2010, respectively. In the meanwhile, he worked at the Italian National Institute of Nuclear Physics, thanks to technical and later research fellowship programs. He received his PhD in Computer Science from the Università degli Studi di Milano in 2014 and is now a postdoctoral researcher at the Università degli Studi di Ferrara and INFN Ferrara Associate. His main interests are in the fields of high-performance computing, parallel and distributed computing, scientific computing, code optimization, and energy-efficient computing.

*A Gabbana* graduated from the University of Ferrara (Italy), with a bachelor's degree in computer science. He continued his studies at the University of Umeå (Sweden) with a master's degree in computational science and engineering. For his master's thesis he worked on a multi-GPU implementation of the D3Q19 lattice Boltzmann model. His current research interests lie in the field of high-performance computing, in particular in the evaluation of new highly parallel computing architectures for the solution of computational physics problems. He is enrolled in a European joint doctorate (HPC-LEAP project), with joint degree-awarding institutions Università degli Studi di Ferrara and Bergische Universität Wuppertal. The project, entitled "Optimized implementations of the lattice Boltzmann method in two and three dimensions on highly parallel computing devices" is supervised by Professor Raffaele Tripiccione and Professor Matthias Ehrhardt.

*SF Schifano* graduated in Computer Science from the University of Pisa (Italy). He spent his early scientific carrier at IEI-CNR and at INFN, as researcher and senior researcher; since 2006 he has been an assistant professor at University of Ferrara (Italy). He had a major role in several projects for the development of parallel systems optimized for scientific applications, such as lattice gauge theory (LQCD), fluid dynamics, and spin glasses. In 2007 he was co-author of the proposal for the QPACE project, a German project to develop a massively parallel system based on IBM Cell-BE processors, interconnected by a custom three-dimensional mesh network, awarded as best Green500 system in November 2009 and June 2010. More recently, his research activities have been focused on the design and optimization of LQCD and lattice Boltzmann applications for multi- and many-core processor architectures. He is author of more than 90 articles and conference papers.

*R Tripiccione* was born in Florence (Italy) in 1956. He graduated in physics (summa cum laude) in 1980 and then attended the graduate school at the Scuola Normale Superiore, Pisa. He is a professor of Physics at Università di Ferrara (Italy). His early scientific interests were in the areas of quantum field theory and the phenomenology of elementary particle physics. More recently, he has mainly focused on several facets of computational theoretical physics, such as lattice gauge theories, computational fluid dynamics and spin systems. Over the years, he has worked on large simulations in lattice gauge theories, on the study of the statistical properties and scaling laws of fluids in the turbulent regime, on the simulation, with Monte Carlo techniques, of spin glasses and on the development of lattice Boltzmann computational methods. He is author or co-author of approximately 200 scientific papers, conference proceedings, and reports.