On Portability, Performance and Scalability of an MPI OpenCL Lattice Boltzmann Code

Enrico Calore[1], Sebastiano Fabio Schifano[2], and Raffaele Tripiccione[3]

[1] Istituto Nazionale di Fisica Nucleare (INFN), Ferrara, Italy
[2] Dip. di Matematica e Informatica, Università di Ferrara and INFN, Ferrara, Italy
[3] Dip. di Fisica e Scienze della Terra, Università di Ferrara and INFN, Ferrara, Italy

Abstract. High performance computing increasingly relies on heterogeneous systems, based on multi-core CPUs, tightly coupled to accelerators: GPUs or many core systems. Programming heterogeneous systems raises new issues: reaching high sustained performances means that one must exploit parallelism at several levels; at the same time the lack of a standard programming environment has an impact on code portability. This paper presents a performance assessment of a *massively parallel* and *portable* Lattice Boltzmann code, based on the Open Computing Language (OpenCL) and the Message Passing Interface (MPI). Exactly the same code runs on standard clusters of multi-core CPUs, as well as on hybrid clusters including accelerators. We consider a state-of-the-art Lattice Boltzmann model that accurately reproduces the thermo-hydrodynamics of a fluid in 2 dimensions. This algorithm has a regular structure suitable for accelerator architectures with a large degree of parallelism, but it is not straightforward to obtain a large fraction of the theoretically available performance. In this work we focus on portability of code across several heterogeneous architectures preserving performances and also on techniques to move data between accelerators minimizing overheads of communication latencies. We describe the organization of the code and present and analyze performance and scalability results on a cluster of nodes based on NVIDIA K20 GPUs and Intel Xeon-Phi accelerators.

1 Introduction

High performance computer architectures are becoming more and more heterogeneous, heavily relying on *accelerators*, which commonly deliver a major fraction (e.g., $\simeq 70\%$) of the full system computing power. Virtually all currently available accelerators (GPUs, many-core CPUs, FPGAs) are independent processing units, connected to commodity CPUs via standard busses, such as PCI-Express. The CPU orchestrates the coarse-grained harness of a complex computation, while accelerators handle compute intensive kernels. In order to use accelerators efficiently, one must partition an algorithm on many processing cores, each core in turn heavily using SIMD features: one is then forced to concurrently exploit several levels of parallelism. Furthermore, accelerators use their own memory hierarchy, so data transfers between host and accelerators have to be carefully

L. Lopes et al. (Eds.): Euro-Par 2014 Workshops, Part II, LNCS 8806, pp. 438–449, 2014.

scheduled. This is an important issue for code performances as also highlighted in [1].

Fortunately enough, several large scale computer codes in the scientific and engineering domain have sufficiently large available parallelism and an algorithmic structure that allows to split the code on the compute elements available on accelerators and to schedule the full computation in a way that tames the problems highlighted above: some handcrafted codes have delivered unexpectedly high performance figures.

An obvious and relevant question is whether a similar level of performance can be obtained using the *same* programming environment and the *same* code for different accelerator architectures and if this approach is also viable when large scale parallelism (involving many nodes with many accelerators) is needed.

In this paper we address this problem, using OpenCL, a software framework able to provide a common abstraction over the underlying computing resources, and MPI, a de-facto standard for multi-node parallel processing. We consider a fluid-dynamics code based on a state-of-the-art massively parallel Lattice-Boltzmann method that we have re-written using OpenCL and MPI. We describe the structure and implementation of the code and present our performance and scaling results on several state-of-the-art heterogeneous architectures, comparing with handcrafted versions of the code for the same algorithm.

We find that the performance of our OpenCL implementation is comparable with that of architecture-specific optimizations, granting, on the other hand, code portability. Moreover, we eventually study the bottlenecks limiting the extent of the scaling window for massively parallel implementations.

Our paper is structured in this way: we first introduce the OpenCL framework and in section 3 we give a short introduction to Lattice Boltzmann methods; section 4 follows, giving details of our implementation and of our optimization results. Section 5 contains an analysis of our performance results, followed by our conclusions and outlook.

2 OpenCL

OpenCL (Open Computing Language) [2] aims to provide a single framework to develop portable code executable across heterogeneous platforms; it is a hardware oblivious open standard, maintained by the non-profit *Kronos Group* and supported by a large set of vendors. OpenCL offers a standard API, providing an extension of the C99 language to write functions that run on heterogeneous platforms (CPUs, GPUs or other accelerators) exploiting a task-based or data-based parallel approach. Manufacturer are responsible for providing an OpenCL API implementation for their devices, following the OpenCL open standard specification. OpenCL codes run on commodity computers, which may or may not host accelerators as GPUs, DSPs (Digital Signal processors), FPGAs (Field-Programmable Gate Arrays), or other processors, in addition to ordinary CPUs. In order to generalize across different architectures, OpenCL provides an abstraction of the actual hardware defining a *platform*, a *memory* and an *execution*

```
__kernel void saxpy( __global double *A, __global double *B,
                     __global double *C, const double s) {

  int id = get_global_id(0); // get global thread ID
  C[id] = s * A[id] + B[id]; // compute the id-th element
}
```

Fig. 1. Sample OpenCL code, computing a saxpy kernel on two vectors

model. How the models map onto the actual hardware is device dependent and is defined by the corresponding implementation; these aspects can be neglected by programmers from the point of view of ensuring program correctness, but they are relevant for performance tuning.

In the OpenCL Platform model, all OpenCL enabled devices in the host are seen as containers of Compute Units (CU); in turn, each CU is made up of different Processing Elements (PE). On the other side, the OpenCL Execution model is made up of two main components: a host program and one or more kernel functions which run on devices. Where each kernel runs depends on the so called OpenCL context, which is defined by the host program as consisting of one or more devices and one or more command queues associated to them. Commands (such as kernel launches or memory transfers) submitted to a command queue may be executed in-order or, optionally, out-of-order; it is possible to define multiple queues for the same device to issue not synchronized commands, which may execute concurrently, if the device is able to do so. The main idea behind OpenCL is the possibility to define an n-dimensional problem domain and then to run a kernel function for each point of it. Each instance, running on each domain point, in the OpenCL taxonomy, is called a work-item and can be thought as a single thread, executing on a processing element within a device. Multiple work-items are commonly grouped in what is called a work-group, which runs on a CU. Each work-item has a global ID and a local ID. The global ID is unique among all work-items of a Kernel. The local ID identifies a work-item within a work-group.

Concerning the OpenCL Memory model, a first distinction is made between the host memory (commonly the host RAM memory) and the device memory (e.g. a GPU memory bank); the device memory in its turn is divided into four address spaces which commonly differ for size and access time. *Global* memory is commonly the largest area; it is visible by all work-items running on the device, but it has the highest access latency. *Constant* memory stores read-only data and is commonly a relatively small cached part of the Global memory. *Local* memory is meant for data sharing by work-items within the same work-group; it is usually faster than Global memory, but smaller and not globally accessible. *Private* memory is accessible only by individual work-items; it is the fastest, but also smallest available storage (e.g. the registers of a CPU).

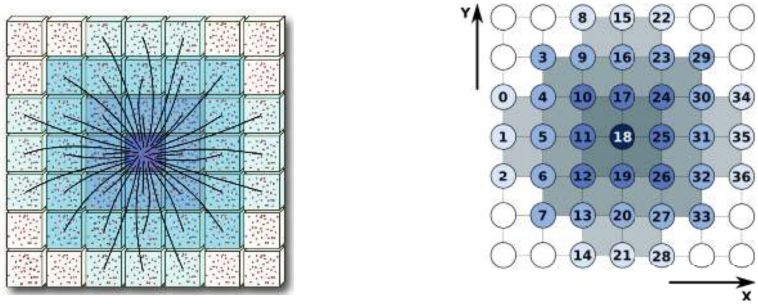

Fig. 2. Left: Velocity vectors for the LB populations in the D2Q37 model. Right: populations labels identify the lattice hop that they perform in the *propagate* phase.

Let us assume that a code is broken down into N_{wg} work-groups and each work-group has N_{wi} work-items. When this code executes on a device with N_{cu} compute units, each able to compute on N_d data items, at any given time $N_{cu} \times N_d$ work-items will execute; iterations will be needed to perform all globally required $N_{wg} \times N_{wi}$ work-items. For example, the Xeon-Phi has 60 physical cores, each supporting up to 4 threads, for a total of 240 virtual cores; it supports AVX 256-bit operations that process 8 double-precision or 16 single-precision floating-point data. In this case, up to 240 work-groups execute on all cores, each core in turn processing up to 8 (or 16) work-items in parallel. Similar mappings of the available parallelism on the computing resources can be worked out for other architectures.

In Fig.1 we show an OpenCL implementation of the *saxpy* operation of the *Basic Linear Algebra Subprogram* (BLAS) set. The parameters of the kernel are three arrays, `A`, `B` and `C` and one double precision number. Pointers to the arrays are marked as `__global` because they are allocated on the global memory of the device. Each work-item executes the *saxpy* kernel computing just one data-item of the output array: first it computes its unique global identifier `id` and then uses it to address the `id`*th* data-item of arrays `A`, `B` and `C`.

3 Lattice Boltzmann Methods

Lattice Boltzmann methods (LB) are widely used in computational fluid dynamics, to describe flows in two and three dimensions. LB methods (see, e.g. [3] 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 incoming populations *collide* among one another, that is, they mix and their values change accordingly.

LB models in x dimensions with y populations are labeled as $DxQy$; we consider a state-of-the-art $D2Q37$ model that correctly reproduces the thermo-hydrodynamical equations of motions of a fluid in two dimensions, automatically enforcing the equation of state of a perfect gas ($p = \rho T$) [4,5]; this model has been extensively used for large scale simulations of convective turbulence (see e.g., [6,7]).

In the algorithm, a set of populations ($f_l(x,t)$ $l = 1 \cdots 37$), defined at the points of a discrete and regular lattice and each having a given lattice velocity $\boldsymbol{c}_l$, evolve in (discrete) time according to the following equation:

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

The macroscopic variables, density ρ, velocity $\boldsymbol{u}$ and temperature T are defined in terms of the $f_l(x,t)$ and of the $\boldsymbol{c}_l$s (D is the number of space dimensions):

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

and the equilibrium distributions ($f_l^{(eq)}$) are themselves function of these macroscopic quantities [3]. In words, populations drift from different lattice sites (*propagation*), according to the value of their velocities and, on arrival at point y, they change their values according to Eq. 1 (*collision*). One can show that, in suitable limiting cases, the evolution of the macroscopic variables obey the thermo-hydrodynamical equations of motion of the fluid.

An LB code starts with an initial assignment of the populations, in accordance with a given initial condition at $t = 0$ on some spatial domain, and iterates Eq. 1 for each point in the domain and for as many time-steps as needed; boundary-conditions at the edges of the integration domain are enforced at each time-step by appropriately modifying the population values at and close to the boundaries.

The LB approach offers a huge degree of easily identified parallelism. Indeed, Eq. 1 shows that the *propagation* step amounts to gathering the values of the fields f_l from neighboring sites, corresponding to populations drifting towards $\boldsymbol{y}$ with velocity $\boldsymbol{c}_l$; the following step (*collision*) then performs all mathematical processing needed to compute the quantities appearing in the r.h.s. of Eq. 1, for each point in the grid. Referring again to Eq. (1), one sees immediately that both steps above are completely uncorrelated for different points of the grid, so they can be computed in parallel according to any convenient schedule, if one ensures that for all grid points step 1 is performed before step 2.

In practice, an LB code executes the following three main steps at each iteration of the loop over time:

- propagate moves populations across lattice sites according to the pattern of Fig.2 left, collecting at each site all populations that will interact at the next phase (collide). Consequently, propagate moves blocks of memory locations allocated at sparse addresses, corresponding to populations of neighbor cells. propagate can either use a pull scheme or a push scheme; in the first case populations are gathered at one site as shown in Fig.2; while in the latter case populations are pushed from one lattice-site towards a set of neighbors. Which of the two is best to use depends on the capability of processor memory-controller.

– bc (Boundary Conditions) adjusts the populations at the top and bottom edges of the lattice to enforce appropriate boundary conditions (e.g., a constant given temperature and zero velocity). This is done *after* propagation, since the latter changes the value of the populations close to the boundary points and hence the macroscopic quantities that must be kept constant. At the right and left boundaries, we apply periodic boundary conditions. This is conveniently done by adding *halo* columns at the edges of the lattice, where we copy the 3 (in our case) rightmost and leftmost columns of the lattice before performing the propagate step. Points close to the right/left boundaries can then be processed as those in the bulk. If needed, boundary conditions could be enforced in the same way as done for the top and bottom edges.

– collide performs all the mathematical steps associated to equation 1 and needed to compute the population values at each lattice site at the new time step. Input data for this phase are the populations gathered by the previous propagate phase. This step is the floating point intensive step of the code.

4 Code Implementation

At top level, our code is based on MPI processes, each managing one OpenCL (OpenCL) device. Actual devices are attached to the host nodes of the cluster, so MPI communications are either fully within the host or across a commodity network, such as Infiniband. This is managed transparently and in a uniform way by the MPI run-time support, so our code runs both on single-host and multi-host multi-device systems.

We split a lattice of size $L_x \times L_y$ on N devices along the X dimension; each device allocates a *sub-lattice* of size $L_x/N \times L_y$. On each device the lattice is stored using the SoA (Structure of Arrays) scheme, where arrays of populations are stored in memory one after the other. This allows to exploit data-parallelism and enable data-coalescing in accessing data when executing several work items in parallel. Each array of population is stored in columns-major order, and we keep in memory two copies of it, prv and nxt. Each kernel reads from prv and update results on the nxt copy; nxt and prv swap their roles at each iteration. This solution needs more memory, but it allows to map one work-item per lattice site, and then to process many sites in parallel. The lattice splitting implies a virtual ordering of the MPI-processes along a ring, so each process exchanges its borders of its own sub-lattice with its adjacent processes. One could consider a different decomposition (e.g. $L_y/N \times L_x$, reducing communication overheads if $L_y \geq L_x$); however, since we plan to use our code for physics simulations in a wide range of aspect-ratios (both $L_x > L_y$ and $L_x < L_y$), we arbitrarily select only one of the two possibilities. Moreover, since our lattice is stored in column-major order, splitting along X means that lattice columns are allocated sequentially in memory, improving memory access time when copying halos.

Each device allocates a *sub-lattice* of $NX \times NY$ lattice points, $NX = H_x + L_x + H_x$, and $NY = H_y + L_y + H_y$, including vertical and horizontal halos of size H_x and H_y. Left and right halos keep copies of the three rightmost and

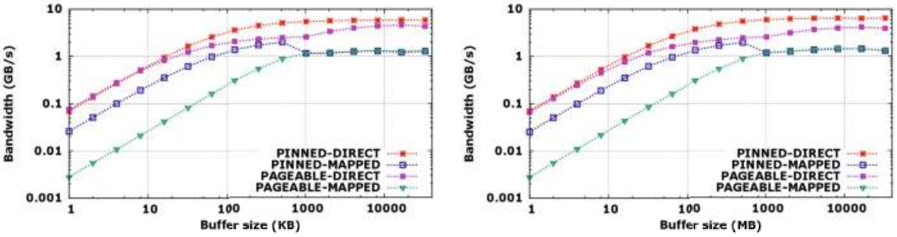

Fig. 3. Bandwidth vs. buffer size for h2d (left) and d2h (right) transfers between host and an NVIDIA K20 GPU device.

leftmost columns of the sub-lattices allocated on the neighbor nodes. This makes the computation uniform for all lattice sites, avoiding divergences of work-items which lead to performance degradation. Bottom and top halos are adjusted to keep memory accesses by work-items aligned, enabling memory coalescing.

Each MPI process runs a loop over time; at each iteration it executes four main-steps: first pbc (Periodic Boundary Conditions) updates the left and right halo columns, and then three kernels – propagate, bc and collide – run on the device to perform the required computational tasks.

Based on previous results in coding with CUDA [8], a language for GPUs not widely different from OpenCL, we configure the OpenCL kernels for propagate, bc and collide as a grid of $(L_y \times L_x)$ work-items; each work-group is a uni-dimensional array of N_{wi} work-items, processing data at successive locations in memory. In this way memory coalescing can be easily exploited.

In the following we describe in details the combination of pbc and propagate, which is critical to scalability when running on multi-device multi-host systems configuration. The key point to consider is that the propagate step for the bulk of the lattice (all lattice points except for three columns at right and left) has no data dependency with pbc (while propagate on the edges depend on fresh data moved to the halos by pbc). Our strategy therefore leverages on i) speeding up data transfers and ii) overlapping as much as possible data transfers with propagate (on the bulk). Let us consider these two points in order. pbc copies the three leftmost and rightmost columns of the lattice respectively into the right and left halos of the neighbor sub-lattices. In a multi-device implementation this implies moving data between OpenCL devices. This task implies the following steps:

1. copy data corresponding to the left and right borders from the device to two host buffers;
2. send data to the previous and the next node in the ring;
3. receive data from neighbors and store them into two host buffers;
4. copy the just received data from host buffers into the halo columns of the device.

All these steps are performance critical, as they use data paths with limited bandwidth and large latency (see later for accurate figures). MPI communi-

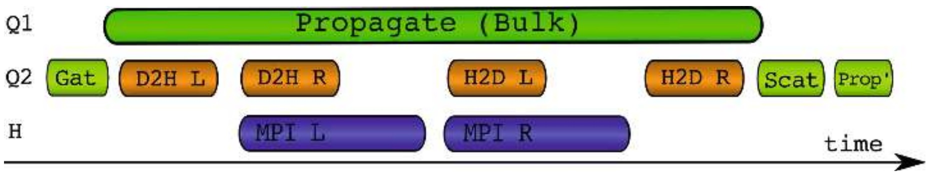

Fig. 4. Concurrent scheduling of the various steps of the `propagate` and `pbc` kernels

cations are handled by the MPI run-time support, so there is not much the programmer can do there. On the other hand, OpenCL has several options to allocate memory and to perform device-to-host (d2h) and host-to-device (h2d) copies.

OpenCL has routines to allocate memory in *pageable* or *pinned* mode; the former option is a standard allocation in virtual space that can be swapped out of physical memory by the operating system, while the latter mode forces memory to be always resident in real memory; the OpenCL function `clCreateBuffer()` function with the `CL_MEM_ALLOC_HOST_PTR` flag performs this operation. Memory access can be *mapped* or *direct*. In mapped mode, buffers on the device are mapped onto the address space of the host node, while in direct mode, data is moved by specific OpenCL routines such as `clEnqueueReadBuffer()` to read from the device and `clEnqueueWriteBuffer()` to write data into it.

We have tested all four combination of allocation and access modes; Fig. 3 shows the bandwidth as a function of the buffer size for `h2d` and `d2h` transfers between a host and an NVIDIA GPU K20 device. Perhaps not unexpectedly, one obtains the best performance using *pinned* memory allocation and *direct* memory access. In this case, the transfer time (μsec) as a function of the data block size s (bytes) is well fitted by the following expressions:

$$T_{\mathsf{h2d}}(s) = 14.16 + 0.00017 \times s, \qquad T_{\mathsf{d2h}}(s) = 14.21 + 0.00015 \times s$$

corresponding to a latency of ≈ 14 μsec (in both directions), and an asymptotic bandwidth of ≈ 6 GB/s for `h2d` and ≈ 6.6 GB/s for d2h. The asymptotic bandwidth is $\approx 75\%$ of the aggregate raw bandwidth of a 16 lanes (16X) GEN2 PCi-Express bus (8 GB/s). The large value for the latency means that it is useful to gather all data into one block before starting the `d2h` operation (and scatter back at destination), rather than paying the latency overhead 37 times.

We now consider how to schedule operations in order to overlap (bulk) `propagate` and `pbc`. We define two OpenCL queues `Q1` and `Q2`: the first schedules the execution of (bulk) `propagate`, while `Q2` schedules the sequence of operations corresponding to `pbc`. There is no data dependency between `Q1` and `Q2`, so both queues can in principle fully overlap in time. In practice, we have seen that this option cannot be fully exploited because the execution over the bulk uses all resources of the device; the best it can do is to overlap host-device transfers and computations on the device. According to our measurements the best scheduling is indeed that shown in Fig. 4:

1. the host starts the gather kernel; this operation collects the 37 left and right borders into two contiguous buffers allocated on the device (Q2 queue).
2. the host starts propagate on the bulk of the lattice (Q1 queue)
3. as soon as gather completes, the host starts the D2H L and D2H R operations in asynchronous mode to copy the two buffers on the host side memory; these operations do not fully overlap because they use the same channel bus, but the host is not blocked (Q2 queue);
4. as each of the two D2H transfers finishes the host starts the corresponding MPI communication – first MPI L and then MPI R – to send and receive border data to/from the left and right neighbours;
5. as each MPI communication completes, the host starts the corresponding H2D L or H2D R steps and moves back the buffers onto its device (Q2 queue);
6. the scatter kernel moves the content of the buffers onto the left and right halos (Q2 queue).
7. propagate executes on the lattice columns not handled by Q1, using fresh halo data (Prop', in Fig. 4). This is a Q2 step, but in practice it does not start before propagate on Q1 finishes.

Inspection of Fig. 4 shows that all data transfer overheads can be hidden behind the execution of (bulk) propagate. The effective time for the combined pbc and propagate steps on the whole lattice is given by $\max\{T\alpha, T_\beta\}$, where

$$T_\alpha = T_{\mathsf{Gath}} + T_{\mathsf{Prop}} + T_{\mathsf{Scat}} + T_{\mathsf{Prop'}}$$
$$T_\beta = T_{\mathsf{Gath}} + T_{\mathsf{D2h(L)}} + T_{\mathsf{MPI(L)}} + T_{\mathsf{MPI(R)}} + T_{\mathsf{H2d(R)}} + T_{\mathsf{Scat}} + T_{\mathsf{Prop'}}$$

As we split the lattice on more and more devices, propagate becomes faster and faster, while data transfers are approximately constant in time, so hiding will be partial. We assess this quantitatively in the next section.

5 Results

We have tested our OpenCL code on the *Eurora* cluster, installed at CINECA (Italy). *Eurora* is a cluster of nodes interconnected through a standard Infiniband network. Each node has two Intel processors of the Xeon-E5 family, based on the Sandybridge micro-architecture, and two accelerators, either two Kepler K20s NVIDIA GPUs or two Intel Xeon-Phi 5100 devices. The double-precision peak

Table 1. Performance comparison of the main critical kernels of the code, using a *common* OpenCL (OCL) code or architecture-specific CUDA and C versions; execution times are in μsec.

	OCL - GPU	CUDA - GPU	OCL - PHI	C - PHI
$T_{\mathsf{Pbc+Prop}}$	17.64	15.40	39.40	37.70
T_{Collide}	104.65	83.33	81.12	79.14

Table 2. Time break-down of all steps of our OpenCL code running on two K20s GPUs for lattice sizes of $L_x \times 2048$. All times are milli-seconds and the lattice is sliced along X-dimension. Values in bold identify the performance limiting factor for scalability.

L_x	3840	1920	960	480	240	120	64	32	16
$T_{\mathsf{Pbc+Prop}}$	17.64	8.93	4.56	2.39	2.07	2.10	2.11	2.03	2.06
T_{Bc}	7.91	3.98	2.02	1.04	0.56	0.30	0.20	0.11	0.11
T_{Collide}	104.65	52.35	26.61	13.15	6.64	3.35	1.82	0.94	0.49
T_{tot}	130.21	65.25	33.19	16.58	9.27	5.74	4.13	3.08	2.66
T_{Gath}	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07
$T_{\mathsf{D2h(L)}}$	0.29	0.29	0.29	0.29	0.29	0.29	0.29	0.29	0.29
$T_{\mathsf{D2h(R)}}$	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28
$T_{\mathsf{MPI(L)}}$	0.70	0.67	0.61	0.60	0.60	0.62	0.64	0.61	0.63
$T_{\mathsf{H2d(L)}}$	0.32	0.32	0.32	0.32	0.32	0.32	0.32	0.32	0.32
$T_{\mathsf{MPI(R)}}$	0.57	0.58	0.58	0.58	0.58	0.58	0.58	0.59	0.59
$T_{\mathsf{H2d(R)}}$	0.32	0.31	0.31	0.31	0.31	0.31	0.31	0.32	0.32
T_{Scat}	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07
$T_{\mathsf{Prop'}}$	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07
T_{Prop}	17.38	8.65	4.31	2.13	1.04	0.50	0.25	0.11	0.04
T_α	**17.60**	**8.86**	**4.53**	**2.35**	1.25	0.71	0.46	0.32	0.25
T_β	2.10	2.06	2.01	2.00	**2.00**	**2.02**	**2.03**	**2.04**	**2.04**

performance of both accelerators is ≈ 1 Tflops. We first assess the performance penalty, if any, of an OpenCL code w.r.t. architecture-optimized codes written using programming languages closer to the specific architecture (i.e. CUDA for GPUs, C and intrinsics for commodity CPUs and Xeon-Phi).

In Table 1 we compare the execution times of the two most critical kernels, propagate and collide, for our OpenCL code and for highly optimized codes written in CUDA for GPUs [10] and C for Xeon-Phis [9]. The GPU CUDA code was compiled using the same configuration options supported by the current NVIDIA OpenCL library. We remark that other options (not supported by the current version of the OpenCL library) allows significantly better performances for the collide kernel [10]. Data is for a lattice large enough (1920 × 2048 per device) that communications are well overlapped with computation. We see that the performances of the *same* OpenCL code are only slightly worse than those of codes specifically optimized for each device. We also notice that the two accelerators have roughly the same performance in the computing intensive kernel (collide), while the PHI processor is slower in the propagate step; this will have an impact on scalability, that we discuss next.

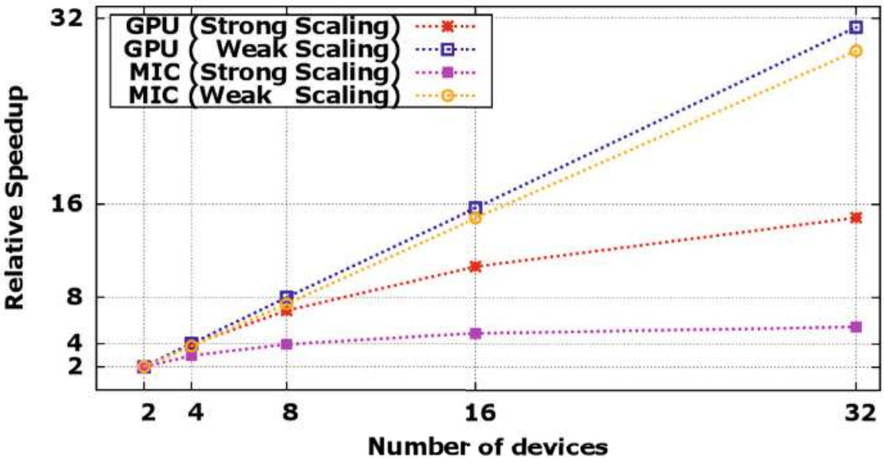

Fig. 5. Weak and strong scalability of our OpenCL code on the EURORA cluster for GPUs and MICs. In the strong regime, the code runs on a lattice of 1024 × 8192 cells; in the weak regime each device handles a sub-lattice of constant size (256 × 8192 cells).

Table 2 contains the time break-down of all operations of our OpenCL code on a dual-K20s system (results are qualitatively similar for the Intel Xeon-Phi) as we reduce the value of L_x. The first part shows the execution time of the main steps of our code. $T_{\mathsf{Pbc+Prop}}$ refers to the execution of `pbc` and `propagate` scheduled as discussed in the previous section; while T_{tot} is the total execution time of the code. The following section of the table shows the full break down of all steps associated to `pbc` and `propagate`, while the last section shows the values of T_α and T_β appearing in the time model of the previous section. Note that our time model describes very well the behavior of $T_{\mathsf{Pbc+Prop}}$ in terms of the contributions of all steps involved. As expected, as we vary the sub-lattice size all operations belonging to `Q2` take approximately the same time, as they handle the same amount of data. However, as the sub-lattice size becomes smaller and smaller ($L_x \leq 480$), T_{Prop} for the bulk is too short to successfully hide communication latencies, so violations to scaling start to appear.

Fig. 5 shows scalability results obtained for both GPUs and PHIs. We have measured strong scalability on a lattice of 1024 × 8192 points. On this purposely small lattice, we see that communications quickly become the major bottlenecks, so there is no real advantage in using more than 32 GPU devices. For PHIs the situation is even worse and we have a performance improvement only up to 16 devices. Here the major bottleneck comes from data transfers between host and PHIs that are slower than for GPUs. For weak scaling we have allocated a lattice of 256 × 8192 on each device, a typical size for physics simulations. In this case communication overheads are fully overlapped with computation of propagate, and the code enjoys perfect scalability both for GPUs and PHIs in the whole range considered, up to 32 devices.

6 Conclusions and Outlook

An important result of this work is that the same OpenCL code runs on different accelerators, either based on GPUs or MICs, with single-node performance similar to that obtained with programming languages closer to each architecture. This provides an higher portability w.r.t. architecture specific implementations. However, in today heterogeneous cluster architectures, performance scalability of codes is seriously limited by the poor integration at hardware level between accelerators, the host node and the network; this translates to high latencies to move data between accelerators. In our implementation we have shown how computation and commnication can be efficiently overlapped in order to minimize impact of transfer latencies. In a future work we plan to use these results to design and optimize a portable 3D Lattice Boltzmann code using the OpenCL framework, or higher level languages such as OpenACC.

Acknowledgements. This work has been done in the framework of the COKA and Suma projects, supported by INFN. We have used the computing facilities of INFN-CNAF (Bologna, Italy) and CINECA (Bologna, Italy). We thank J. Kraus for useful suggestions and comments.

References

1. Obrecht, C., et al.: Scalable lattice Boltzmann solvers for CUDA GPU clusters. Parallel Computing 39 (2013)
2. Kronos Group, The open standard for parallel programming of heterogeneous systems, `http://www.khronos.org/opencl`
3. Succi, S.: The Lattice Boltzmann Equation for Fluid Dynamics and Beyond, Oxford University Press (2001)
4. Sbragaglia, M., et al.: Lattice Boltzmann method with self-consistent thermo-hydrodynamic equilibria. J. Fluid Mech. 628, 299–309 (2009), doi:10.1017/S002211200900665X
5. Scagliarini, A., et al.: Lattice Boltzmann methods for thermal flows: Continuum limit and applications to compressible Rayleigh-Taylor systems. Phys. Fluids 22, 055101 (2010), doi:10.1063/1.3392774
6. Biferale, L., et al.: Second-order closure in stratified turbulence: Simulations and modeling of bulk and entrainment regions. Phys. Rev. E 84, 1, 2, 016305 (2011), doi:10.1103/PhysRevE.84.016305
7. Biferale, L., et al.: Reactive Rayleigh-Taylor systems: Front propagation and non-stationarity. EPL 94, 5, 54004 (2011), doi:10.1209/0295-5075/94/54004
8. Biferale, L., et al.: An Optimized D2Q37 Lattice Boltzmann Code on GP-GPUs. Comp. and Fluids 80 (2013), doi:10.1016/j.compfluid.2012.06.003
9. Crimi, G., et al.: Early Experience on Porting and Running a Lattice Boltzmann Code on the Xeon-phi Co-Processor. Proc. Comp. Science 18 (2013), doi:10.1016/j.procs.2013.05.219
10. Kraus, J., et al.: Benchmarking GPUs with a Parallel Lattice-Boltzmann Code. In: Proc. of 25th Int. Symp. on Computer Architecture and High Performance Computing, SBAC-PAD (2013), doi:10.1109/SBAC-PAD.2013.37