Accelerating Lattice Boltzmann Applications with OpenACC

Enrico Calore[1], Jiri Kraus[2], Sebastiano Fabio Schifano[3(✉)], and Raffaele Tripiccione[1]

[1] Dip. di Fisica e Scienze Della Terra, Univ. di Ferrara and INFN, Ferrara, Italy
{calore,tripiccione}@fe.infn.it
[2] NVIDIA GmbH, Würselen, Germany
jkraus@nvidia.com
[3] Dip. di Matematica e Informatica, Univ. di Ferrara and INFN, Ferrara, Italy
schifano@fe.infn.it

Abstract. An increasingly large number of HPC systems rely on heterogeneous architectures combining traditional multi-core CPUs with power efficient accelerators. Designing efficient applications for these systems has been troublesome in the past as accelerators could usually be programmed only using specific programming languages – such as CUDA – threatening maintainability, portability and correctness. Several new programming environments try to tackle this problem; among them OpenACC offers a high-level approach based on directives. In OpenACC, one annotates existing C, C++ or Fortran codes with compiler *directive* clauses to mark program regions to offload and run on accelerators and to identify available parallelism. This approach directly addresses code portability, leaving to compilers the support of each different accelerator, but one has to carefully assess the relative costs of potentially portable approach versus computing efficiency. In this paper we address precisely this issue, using as a test-bench a massively parallel Lattice Boltzmann code. We implement and optimize this multi-node code using OpenACC and OpenMPI. We also compare performance with that of the same algorithm written in CUDA, OpenCL and C for GPUs, Xeon-Phi and traditional multi-core CPUs, and characterize through an accurate time model its scaling behavior on a large cluster of GPUs.

Keywords: OpenACC · OpenMPI · Lattice Boltzmann methods · Accelerator computing · Performance analysis

1 Introduction and Background

Lattice Boltzmann (LB) methods are widely used in computational fluid dynamics, to simulate flows in two and three dimensions. From the computational point of view, LB methods have a large degree of available parallelism so they are suitable for massively parallel systems.

Over the years, LB codes have been written and optimized for large clusters of commodity CPUs [1], for application-specific machines [2–4] and even for

J.L. Träff et al. (Eds.): Euro-Par 2015, LNCS 9233, pp. 613–624, 2015.
DOI: 10.1007/978-3-662-48096-0_47

FPGAs [5]. More recently work has focused on exploiting the parallelism of powerful traditional many-core processors [6], and of power-efficient accelerators such as GPUs [7,8] and Xeon-Phi processors [9].

As diversified HPC architectures emerge, it is becoming more and more important to have robust methodologies to port and maintain codes for several architectures. This need has sparked the development of frameworks, such as the *Open Computing Language* (OpenCL), able to compile codes efficiently for several accelerators. OpenCL is a low level approach: it usually obtains high performances at the price of substantial changes in the code and large human efforts, seriously posing a threat to code correctness and maintainability. Other approaches start to emerge, mainly based on directives: compilers generate offload-functions for accelerators, following "hints" provided by programmers as annotations to the original – C, C++ or Fortran – codes [12]. Examples along this direction are OpenACC [13] and OpenMP4 [14]. Other proposals, such as the Hybrid Multi-core Parallel Programming model (HMPP) proposed by CAPS, hiCUDA [15], OpenMPC [16] and StarSs [17] follow the same line. OpenACC today is considered the most promising approach. In many ways its structure is similar to OpenMP (Open Multi-Processing) [18]: both frameworks are directive based, but while OpenMP is more prescriptive, e.g. one maps work-loads explicitly using distribute constructs, OpenACC is more descriptive. Indeed, with OpenACC the programmer only specifies that a certain loop should run in parallel on the accelerator and leaves the exact mapping to the compiler. This approach leaves more freedom to the compiler and the associated runtime support, offering in principle more space for performance portability.

So far very few OpenACC implementations of LB codes have been described in literature: [19] focus on accelerating through OpenACC a part of a large CFD application optimized for CPU; several other works describe CUDA [21] or OpenCL [10,11] implementations; also scalability on GPU clusters has been rarely addressed [22]. In this paper we focus on the design and optimization of a multi-GPU LB code analyzing performances between a portable high level approach like OpenACC and lower level approaches like CUDA.

This paper is structured as follows: Sect. 2 gives a short overview of LB methods; Sect. 3 describes in details our OpenACC implementation, and Sect. 4 analyzes performance results and compares with similar codes written in CUDA, OpenCL and C for GPUs, Xeon-Phi accelerators and traditional multi-core CPUs.

2 Lattice Boltzmann Models

Lattice Boltzmann methods (LB) are widely used in computational fluid dynamics, to describe flows in two and three dimensions. LB methods [23] 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. Over the years, many different LB models have been devised, in 2 and 3 dimensions with

different degrees of accuracy [24]. LB models in n dimensions with y populations are labeled as $DnQy$; in this paper, we consider a state-of-the-art $D2Q37$ model that correctly reproduces the thermo-hydrodynamical equations of motion of a fluid in two dimensions and automatically enforces the equation of state of a perfect gas ($p = \rho T$) [25,26]; this model has been extensively used for large scale simulations of convective turbulence (see e.g., [27–29]).

From a computational point of view this physically very accurate LB scheme is more complex than simpler LB models; this translates into higher requirements in terms of storage (each lattice points has 37 populations), memory bandwidth and floating-point throughput (at each time step, ≈ 7600 double-precision floating point operations are performed per lattice point).

Populations ($f_l(\boldsymbol{x}, t)$ $l = 1 \cdots 37$) are defined at the sites of a discrete and regular 2-D lattice; each $f_l(\boldsymbol{x}, t)$ has a given lattice velocity $\boldsymbol{c}_l$; populations evolve in (discrete) time according to the following equation:

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

Macroscopic quantities, 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):

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

the equilibrium distributions ($f_l^{(eq)}$) are known function of these macroscopic quantities [23], and τ is a suitably chosen relaxation time. In words, (1) stipulates that populations drift from lattice site to lattice site according to the value of their velocities (*propagation*) and, on arrival at point $\boldsymbol{x}$, they interact among one another and their values change accordingly (*collision*). One can show that, in suitable limiting cases and after appropriate renormalizations are applied, the evolution of the macroscopic variables defined in (2) obey the thermo-hydrodynamical equations of motion of the fluid.

An LB code takes an initial assignment of the populations, in accordance with a given initial condition at $t = 0$ on some spatial domain, and iterates (1) for all points 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 population values at and close to the boundaries.

The LB approach offers a huge degree of easily identified parallelism. Indeed, (1) shows that the *propagation* step amounts to gathering the values of the fields f_l from neighboring sites, corresponding to populations drifting towards $\boldsymbol{x}$ with velocity $\boldsymbol{c}_l$; the following step (*collision*) then performs all mathematical processing needed to compute the quantities in the r.h.s. of (1), for each point in the grid. One sees immediately from (1), that both steps above are fully uncorrelated for different points of the grid, so they can be executed in parallel according to any schedule, as long as step 1 precedes step 2 for all lattice points.

In practice, an LB code executes a loop over time steps, and at each iterations applies three kernels: `propagate`, `bc` and `collide`.

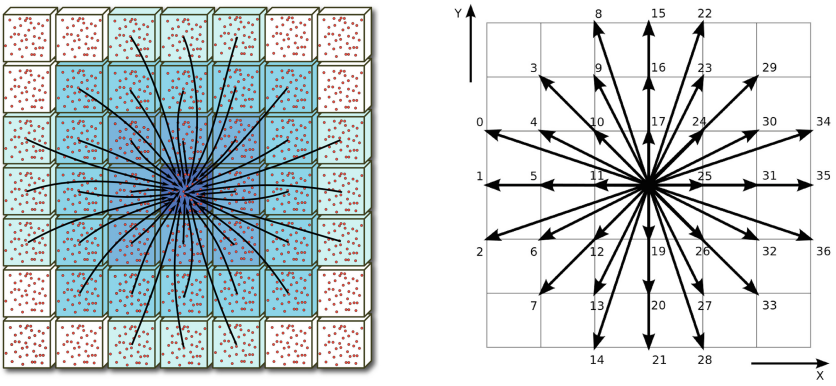

Fig. 1. Left: LB populations in the D2Q37 model, hopping to nearby sites during the `propagate` phase. Right: populations f_l are identified by an arbitrary label; for each l population data is stored contiguously in memory.

`propagate` moves populations across lattice sites according to the pattern of Fig. 1, collecting at each site all populations that will interact at the next phase (`collide`). In our model populations move up to three lattice sites per time step. Computer-wise, `propagate` moves blocks of memory locations allocated at sparse addresses, corresponding to populations of neighbor cells.

`bc` executes *after* propagation and adjusts populations at the edges of the lattice, enforcing appropriate boundary conditions (e.g., constant temperature and zero velocity at the top and bottom edges of the lattice). For the left and right edges, we usually apply periodic boundary conditions. This is conveniently done by adding *halo* columns at the edges of the lattice, where we copy the rightmost and leftmost columns (3 in our case) of the lattice before starting the `propagate` step. After this is done, points close to the boundaries are processed as those in the bulk.

`collide` performs all mathematical steps needed to compute the population values at each lattice site at the new time step, as per (1). Input data for this phase are the populations gathered by the previous `propagate` phase. This step is the most floating point intensive part of the code.

3 Implementation and Optimization of the D2Q37 Model

Our implementation uses CUDA-aware MPI and start one MPI rank per GPU to have GPU-to-GPU transfers transparently handled by the MPI library. The lattice is copied on the accelerator memory at the beginning of the loop over time-steps, and then all three kernels – `propagate`, `bc` and `collide` – run on the accelerator. Data is stored in memory in the Structure-of-Array (SoA) format scheme, where arrays of all populations are stored one after the other. This helps exploit data-parallelism and enables data-coalescing when accessing data needed by work-items executing in parallel. On each MPI-rank the physical lattice is

```
// processing of bulk
propagateBulk( f2, f1 ); // async execution on queue (1)
bcBulk( f2, f1 );        // async execution on queue (1)
collideInBulk( f2, f1 ); // async execution on queue (1)
// execution of pbc step
#pragma acc host_data use_device(f2) {
  for ( pp = 0; pp < 37; pp++ ) {
    MPI_Sendrecv ( &(f2[...]), 3*NY, ... );
    MPI_Sendrecv ( &(f2[...]), 3*NY, ... );
  } }
// processing of the three leftmost columns
propagateL( f2, f1 );    // async execution on queue (2)
bcL( f2, f1 );           // async execution on queue (2)
collideL( f1, f2 );      // async execution on queue (2)
// processing of the three rightmost columns
propagateR( f2, f1 );    // async execution on queue (3)
bcR( f2, f1  );          // async execution on queue (3)
collideR( f1, f2  );     // async execution on queue (3)
```

Fig. 2. Scheduling of operations started by the host at each time step. Kernels processing the lattice bulk run asynchronously on the accelerator, and overlap with MPI communications executed by the host.

surrounded by halo columns and rows: for a physical lattice of size $L_x \times L_y$, we allocate $NX \times NY$ points, with $NX = H_x + L_x + H_x$ and $NY = H_y + L_y + H_y$.

We split our 2-D physical lattice of size $L_x \times L_y$ on N accelerators along the X dimension; GPUs are connected in a ring-scheme and each one hosts a sub-lattice of $L_x/N \times L_y$ points. With this splitting, halo-columns are allocated at successive memory locations, so we do not need to gather halo data on contiguous buffers before communication. At the beginning of each time-step left- and right-halos are updated: we copy population data coming from the three adjoining physical columns of the neighbor nodes in the ring to the left and right *halos*. This is done by an MPI node-to-node communication step that we call *periodic boundary condition* (`pbc`). Once this is done, all remaining steps are local to each MPI-rank so they run in parallel.

At each iteration of the loop over time steps, each MPI-rank first update its halo columns using `pbc()`, and then runs in sequence `propagate()`, `bc()` and `collide()` on its local lattice.

As lattice data is stored in the SoA format, `pbc` exchanges 37 buffers, each of 3 columns, with its left and right neighbors. It executes a loop over the 37 populations and each iteration performs two MPI send-receive operations, respectively for the left and the right halo (see Fig. 2). On GPUs, we exploit *CUDA-aware* MPI features, available in the OpenMPI library and use data pointers referencing GPU-memory buffers as source and destination, making the code more compact and readable. In OpenACC this is controlled by the `#pragma acc host_data use_device(p)` clause, that maps a GPU memory pointer `p` into

```
inline void propagate (
  const data_t* restrict prv, data_t* restrict nxt ) {
  int ix, iy, site_i;
  #pragma acc kernels present(prv) present(nxt)
  #pragma acc loop gang independent
  for ( ix=HX; ix < (HX+SIZEX); ix++) {
  #pragma acc loop vector independent
    for ( iy=HY; iy<(HY+SIZEY); iy++) {
      site_i = (ix*NY) + iy;
      nxt[      site_i] = prv[      site_i-3*NY+1];
      nxt[NX*NY+site_i] = prv[NX*NY+site_i-3*NY  ];
      ....
    } } }
```

Fig. 3. OpenACC pragmas in the body of the `propagate()` function; pragmas before the loops instruct the compiler to generate corresponding accelerator kernel and configure the grid of threads and blocks.

host space, so it can be used as an argument of the MPI send and receive functions. Also, communications between GPUs are optimized in the library and implemented according to physical location of buffers and the capabilities of the devices involved, also enabling *peer-to-peer* and *GPUDirect RDMA* features. Figure 3 shows the code of the *propagate* function. For each lattice site we update the values of the populations, copying from the `prv` array onto the `nxt` array. The body of *propagate* is annotated with several OpenACC directives telling the compiler how to organize the kernel on the accelerator. `#pragma acc kernels present(prv) present(nxt)` tells the compiler to run the following instructions on the accelerator; it also carries the information that the `prv` and `nxt` arrays are already available on the accelerator memory, so no host-accelerator data transfer is needed; `#pragma acc loop gang independent` states that each iteration of the following loop (over the X-dimension) can be run by different gangs or block of threads; `#pragma acc loop vector independent` tells the compiler that iterations of the loop over Y-dimension can likewise be run as independent vectors of threads. Using these directives the compiler structures the thread-blocks and block-grids of the accelerator computation such that: one thread is associated to and processes one lattice-site; each thread-block processes a group of lattice sites lying along the Y-direction, and several blocks process sites along the X-direction. This allows to expose all available parallelism.

We split `bc()` in two kernels, processing the upper and lower boundaries. They run in parallel since there is no data dependencies among them. We have not further optimized this step because its computational cost is small compared to the other phases of the code.

The `collide()` kernel sweeps all lattice sites and computes the collisional function. The code has two outer loops over X and Y dimensions of the lattice, and several inner loops to compute temporary values. We have annotated the

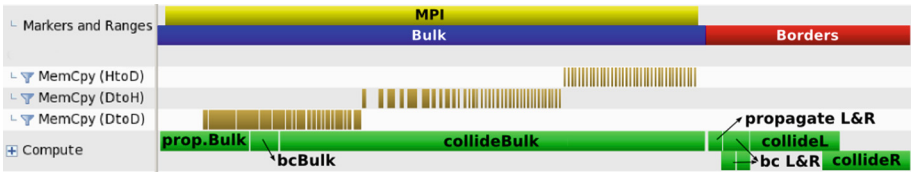

Fig. 4. Profiling of one time step. `pbc` (yellow line marked as "MPI") and the kernels processing the bulk of the lattice (blue line marked as "Bulk") fully overlap (Color figure online).

outer loops as we did for `propagate()`, making each thread to process one lattice site. Inner loops are computed serially by the thread associated to each site.

Performance wise, `pbc()` is the most critical step of the code, since it involves node-to-node communications that can badly affect performance and scaling. We organize the code so node-to-node communications are (fully or partially) overlapped with the execution of other segments of the code. Generally speaking, `propagate`, `bc` and `collide` must execute one after the other, and they cannot start before `pbc` has completed. One easily sees however that this dependency does not apply to all sites of the lattice outside the three leftmost and rightmost border columns (we call this region the *bulk* of the lattice). The obvious conclusion is that processing of the bulk can proceed in parallel with the execution of `pbc`, while the sites on the three leftmost and rightmost columns are processed only after `pbc` has completed. OpenACC abstracts concurrent execution using queues; function definitions flagged by `#pragma acc async(n)` directive enqueue the corresponding kernels *asynchronously* on queue `n`, leaving the host free to perform other tasks concurrently. In our case, this happens for `propagateBulk`, `bcBulk` and `collideBulk`, which start on queue 1 (see Fig. 2), while the host concurrently executes the MPI transfers of `pbc`. After communications complete, the host starts three more kernels on two different queues (2 and 3) to process the right and left borders, so they can execute in parallel if sufficient resources on the accelerator are available. This structure allows to overlap `pbc` with all other steps of the code, most importantly with `collideBulk`, which is the most time consuming kernel, giving more opportunities to hide communication overheads when running on a large number of nodes.

Figure 4 shows the profiling of one time step on one GPU on a lattice of 1080×2048 points split across 24 GPUs. MPI communications started by `pbc` are internal (MemCopy DtoD), moving data between GPUs on the same host, or external (MemCopy DtoH and HtoD) moving data between GPUs on different hosts. The actual scheduling is as expected: both types of GPU-to-GPU communications fully overlap with `propagate`, `bc` and `collide` on the bulk.

4 Results and Conclusions

We start our performance analysis comparing OpenACC code with CUDA, OpenCL and C implementations of the same LB algorithm developed for NVIDIA

Table 1. Performance comparison between single (1CPU) and dual (2CPU) Intel 18-core CPUs (Haswell-v3 micro architecture), NVIDIA K40 GPUs and Intel Xeon-Phi 7120; the lattice size is 1920×2048 points. All quantities are defined in the text.

	Tesla K40			Xeon-Phi 7120	E5-2699-v3	
Code Version	CUDA	OCL	OACC	OCL	1CPU C	2CPU C
$T_{\text{Pbc+Prop}}$ [msec]	13.78	15.80	13.91	30.46	120.71	61.40
GB/s	168.91	147.33	167.37	76.42	19.53	37.91
$\mathcal{E}_p$	59 %	51 %	58 %	22 %	29 %	28 %
T_{Bc} [msec]	4.42	6.41	2.76	3.20	1.62	0.80
T_{Collide} [msec]	39.86	136.93	78.65	72.79	136.24	67.95
$\mathcal{E}_c$	45 %	13 %	23 %	34 %	34 %	34 %
T_{WC}/iter [msec]	58.07	159.14	96.57	106.45	259.79	131.88
MLUPS	68	25	41	37	15	30

GPUs, Intel Xeon-Phi and Intel traditional multi-core CPUs. For OpenACC we have used PGI compiler version 14.10, while for GPUs we have used CUDA version 6.5, and for Xeon-Phi and multicore-CPUs the Intel compiler version 14.

We started with an early version for Intel commodity CPUs, using OpenMP to handle parallelism over all available cores (18 in this case) of each CPU, and controlling vectorization via intrinsics functions [30]. We then developed a CUDA version [20,21], optimized for Fermi and Kepler architectures, and an OpenCL version that we have run on NVIDIA and Intel Xeon-Phi accelerators [11,31].

Table 1 summarizes performance figures on a reference lattice of 1920×2048 sites. The first lines refer to the `propagate` kernel; we show the execution time, the effective bandwidth, and the efficiency $\mathcal{E}_p$ computed w.r.t. the peak memory bandwidth; the table then lists execution times of the `bc` function; For the `collide` kernel, we show the execution time and the efficiency $\mathcal{E}_c$ as a fraction of peak performance. Finally, we show the wall-clock execution time (WcT) and the corresponding *Millions Lattice UPdate per Second* (MLUPS) – counting the number of sites handled per second – of the full production-ready code. For `propagate`, which is strongly memory bound, the CUDA and OpenACC versions run at $\approx 60\,\%$ of peak, while the OpenCL version is 10 % slower. For the `collide` kernel – the most computationally intensive part of the code – the OpenACC code has an efficiency of 23 % while the CUDA version doubles this figure, running at 45 % of peak. The lower performance of the OpenACC code can likely be explained by latency overheads caused by two factors: (i) population values are used several times within the computation of the collide, and repeatedly read from global-memory; (ii) constant values like the coefficients of the Hermite polynomial expansion are stored on global memory. On the other hand the CUDA version [21] is more performing because explicit control with `cudaMemcpyToSymbol` allows to store constant values on low-latency constant-memory; OpenACC is not able to do that due to the large number of terms and their dependencies and then they are computed at run time. CUDA also uses

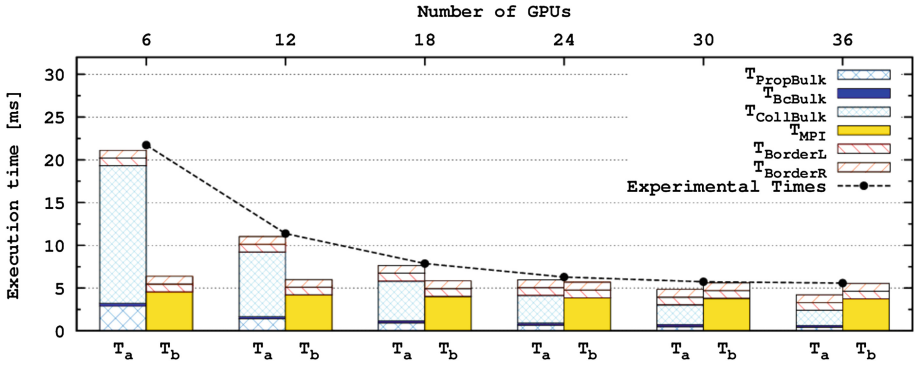

Fig. 5. T_a and T_b for the time model defined in the text on a lattice of 1080×5736 points as a function of the number of GPUs. The black points are the execution times of the code with all asynchronous steps enabled.

registers more efficiently, allowing to fully unroll inner loops, while on OpenACC this has a negative effect. A CUDA version which does not use these two optimizations matches the performance of the OpenACC code.

The OpenCL version is respectively 2X and 3X slower than OpenACC and CUDA. This reflects that the current version of the NVIDIA OpenCL driver does not optimize for the Kepler architecture (it can not exploit features introduced with the Kepler architecture, e.g. the capability to address 255 registers per thread) [11]. Comparing performances of our code across all architectures – CPUs, GPUs and Xeon-Phi accelerators – we see that on GPUs, using CUDA, `collide` runs $\approx 3.5X$ faster than on a single-CPU and $1.7X$ than on a dual-CPU using C, and $\approx 1.8X$ faster than the Xeon-Phi using OpenCL.

We now discuss in details the scaling behaviour of our implementation for a large number of GPUs. We model the execution time of the whole program as $T \approx \max\{T_a, T_b\}$, with T_a and T_b defined as:

$$T_a = T_{\text{bulk}} + T_{\text{borderL}} + T_{\text{borderR}}, \qquad T_b = T_{\text{MPI}} + T_{\text{borderL}} + T_{\text{borderR}}$$

and $T_{\text{bulk}}, T_{\text{borderL}}, T_{\text{borderR}}$ are respectively the sums of the execution times of `propagate`, `bc` and `collide` on the bulk, and on the left and right halos, while T_{MPI} refers to MPI communications.

This model is in good agreement with data measured on an Infiniband-interconnected cluster with 6 GPUs on each node: we first profile the execution time of each kernel and MPI communication running them in sequence, i.e. without any overlap, and then measure the execution time of the whole program with all asynchronous steps enabled. Figure 5 shows the values of T_a and T_b for a lattice of 1080×5736 points. The histograms show the times taken by each part of the code when running serially while the black dots show the time taken by the asynchronous code. For this choice of the lattice size, we see that $T \approx T_a$ up to 24 GPUs as communications are fully hidden behind the execution of the program on the bulk; as long as this condition holds, the code enjoys full

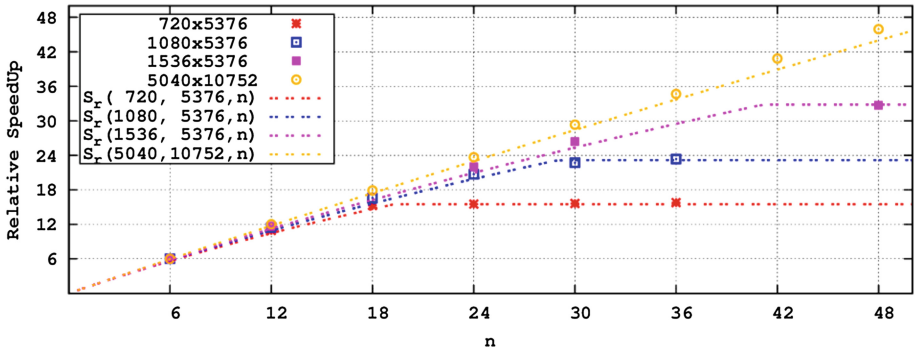

Fig. 6. Strong scaling behavior of the OpenACC code as a function of the number of GPUs (n) for several lattice sizes. Points are experimental data and dashed lines are the predictions of our timing model.

scalability. As we increase the number of GPUs (≥ 30) $T \approx T_b$, communications become the bottleneck and the scaling behavior necessarily degrades.

We further characterize the execution time assuming, to first approximation, that bulk processing is proportional to $(L_x \times L_y)$, boundary conditions scale as L_x, and communication and borders processing scales as L_y; so, on n GPUs

$$T(L_x, L_y, n) = \max\left\{\alpha\frac{L_x}{n}L_y + \beta\frac{L_x}{n}, \quad \gamma L_y\right\} + \delta L_y$$

We extract the parameters (α, β, γ and δ) from the profiling data of Fig. 5, and define the function $S_r(L_x, L_y, n) = \frac{T(L_x,L_y,1)}{T(L_x,L_y,n)}$ to predict the relative speedup for any number of GPUs and any lattice size. Figure 6 shows the (strong) scaling behaviour of our code for several lattice sizes relevant for physics simulations; dots are measured values and dashed lines are plots of $S_r()$ for different values of L_x and L_y. Values of $S_r()$ are in good agreement with experimental data, and predict the number of GPUs for which the code does not scale any more. For large lattices (5040×10752) the code has an excellent scaling behavior up to 48 GPUs, slightly underestimated by our model as constants are calibrated on smaller lattices and then more sensitive to overheads.

In conclusion, we have successfully ported, tested and benchmarked a complete lattice Boltzmann code using OpenACC, and characterized its performances through an accurate time model. Our experience with OpenACC is very positive from the point of view of porting and programmability. The effort to port existing codes to OpenACC is reasonably limited and easy to handle; we started from an existing `C` version and marked through *pragmas* regions of code to offload and run on accelerators instructing the compiler to identify and exploit available parallelism. However, major changes in the structure of the code cannot be handled by compilers, so the overall organization must be (at least partially) aware of the target architectures; for example, in our case it is crucial to organize data as *Structure of Arrays* to allow to coalesce performance-critical memory accesses.

Concerning performance results, one is ready to accept that the use of a high level programming model trades better programmability with computing efficiency: we consider a performance drop of $\leq 20\,\%$ a satisfactory result. While the performance is not as good as we would like, we understand the reasons behind this gap and expect that compiler improvements will be able to narrow it. We believe that our analysis provides important feedbacks to help improve the performance of OpenACC. As an interim step, interoperability between OpenACC and CUDA allows to foster the high productivity of OpenACC and still get full performance by using CUDA for the most performance critical kernels.

Acknowledgements. This work was done in the framework of the COKA and Suma projects of INFN. We thank INFN-Pisa (Pisa, Italy), and the NVIDIA Jülich Application Lab (Jülich Supercomputer Center, Germany) for allowing us to use their computing systems.

References

1. Pohl, T., et al.: Performance evaluation of parallel large-scale lattice boltzmann applications on three supercomputing architectures. In: Proceedings of the Conference on Supercomputing (2004). doi:10.1109/SC.2004.37
2. Belletti, F., et al.: Multiphase lattice boltzmann on the cell broadband engine. Nuovo Cimento Soc. Ital. Fis., C **32**(2), 53–56 (2009). doi:10.1393/ncc/i2009-10379-6
3. Biferale, L., et al.: Lattice boltzmann fluid-dynamics on the QPACE supercomputer. Proc. Comp. Sci. **1**, 1075–1082 (2010). doi:10.1016/j.procs.2010.04.119
4. Pivanti, M., Mantovani, F., Schifano, S.F., Tripiccione, R., Zenesini, L.: An optimized lattice boltzmann code for bluegene/q. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2013, Part II. LNCS, vol. 8385, pp. 385–394. Springer, Heidelberg (2014)
5. Sano, K., et al.: FPGA-based streaming computation for lattice boltzmann method. In: Proceedings of Field-Programmable Technology, pp. 233–236 (2007). doi:10.1109/FPT.2007.4439254
6. Biferale, L., et al.: Optimization of multi-phase compressible lattice boltzmann codes on massively parallel multi-core systems. Proc. Comp. Sci. **4**, 994–1003 (2011). doi:10.1016/j.procs.2011.04.105
7. Biferale, L., et al.: A multi-GPU implementation of a D2Q37 lattice boltzmann code. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2011, Part I. LNCS, vol. 7203, pp. 640–650. Springer, Heidelberg (2012)
8. Bailey, P., et al.: Accelerating lattice Boltzmann fluid flow simulations using graphics processors. In: Proceedings of Parallel Processing, pp. 550–557 (2009). doi:10.1109/ICPP.2009.38
9. Crimi, G., et al.: Early experience on porting and running a lattice boltzmann code on the Xeon-phi co-processor. Proc. Comp. Sci. **18**, 551–560 (2013). doi:10.1016/j.procs.2013.05.219
10. McIntosh-Smith, S.N., Curran, D.: Evaluation of a performance portable lattice Boltzmann code using OpenCL. In: International Workshop on OpenCL (2014)
11. Calore, E., et al.: A portable OpenCL lattice boltzmann code for multi-and many-core processor architectures. Proc. Comp. Sci. **29**, 40–49 (2014). doi:10.1016/j.procs.2014.05.004

12. Wienke, S., Springer, P., Terboven, C., an Mey, D.: OpenACC — first experiences with real-world applications. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 859–870. Springer, Heidelberg (2012)
13. OpenMP. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf
14. OpenACC directives for accelerators. http://www.openacc-standard.org/
15. Han, T., Abdelrahman, T.: hiCUDA: high-level GPGPU programming. IEEE Trans. Par. Distr. Syst. **22**(1), 78–90 (2011). doi:10.1109/TPDS.2010.62
16. Lee, S., Eigenmann, R.: OpenMPC: extended OpenMP programming and tuning for GPUs. In: Proceedings of SC, pp. 1–11 (2010). doi:10.1109/SC.2010.36
17. Ayguadé, E., et al.: Extending OpenMP to survive the heterogeneous multi-core era. Int. J. Parallel Prog. **38**(5–6), 440–459 (2010). doi:10.1007/s10766-010-0135-4
18. Wienke, S., Terboven, C., Beyer, J.C., Müller, M.S.: A pattern-based comparison of OpenACC and OpenMP for accelerator computing. In: Silva, F., Dutra, I., Santos Costa, V. (eds.) Euro-Par 2014. LNCS, vol. 8632, pp. 812–823. Springer, Heidelberg (2014)
19. Kraus, J., et al.: Accelerating a C++ CFD code with OpenACC. In: Proceedings of the First Workshop on Accelerator Programming Using Directives, pp. 47–54 (2014). doi:10.1109/WACCPD.2014.11
20. Biferale, L., et al.: An optimized D2Q37 lattice boltzmann code on GP-GPUs. Comput. Fluids **80**, 55–62 (2013). doi:10.1016/j.compfluid.2012.06.003
21. Kraus, J., et al.: Benchmarking GPUs with a parallel lattice-boltzmann code. In: Proceedings of Computer Architecture and High Performance Computing (SBAC-PAD), pp. 160–167 (2013). doi:10.1109/SBAC-PAD.2013.37
22. Obrecht, C., et al.: Scalable lattice boltzmann solvers for CUDA GPU clusters. Parallel Comput. **39**(6–7), 259–270 (2013). doi:10.1016/j.parco.2013.04.001
23. Succi, S.: The Lattice-Boltzmann Equation. Oxford University Press, Oxford (2001)
24. Aidun, C.K., Clausen, J.R.: Lattice-boltzmann method for complex flows. Annu. Rev. Fluid Mech. **42**(1), 439–472 (2010). doi:10.1146/annurev-fluid-121108-145519
25. Sbragaglia, M., et al.: Lattice boltzmann method with self-consistent thermo-hydrodynamic equilibria. J. Fluid Mech. **628**, 299–309 (2009). doi:10.1017/S002211200900665X
26. Scagliarini, A., et al.: Lattice boltzmann methods for thermal flows: continuum limit and applications to compressible Rayleigh-Taylor systems. Phys. Fluids **22**(5), 055101 (2010). doi:10.1063/1.3392774
27. Biferale, L., et al.: Second-order closure in stratified turbulence: simulations and modeling of bulk and entrainment regions. Phys. Rev. E **84**(1), 016305 (2011). doi:10.1103/PhysRevE.84.016305
28. Biferale, L., et al.: Reactive Rayleigh-Taylor systems: front propagation and non-stationarity. EPL (Europhys. Lett.) **94**(5), 54004 (2011). doi:10.1209/0295-5075/94/54004
29. Ripesi, L., et al.: Evolution of a double-front Rayleigh-Taylor system using a graphics-processing-unit-based high-resolution thermal lattice-Boltzmann model. Phys. Rev. E **89**(4) (2014). doi:10.1103/PhysRevE.89.043022
30. Mantovani, F., et al.: Performance issues on many-core processors: a D2Q37 lattice boltzmann scheme as a test-case. Comput. Fluids **88**, 743–752 (2013). doi:10.1016/j.compfluid.2013.05.014
31. Calore, E., Schifano, S.F., Tripiccione, R.: On portability, performance and scalability of an MPI OpenCL lattice boltzmann code. In: Lopes, L., et al. (eds.) Euro-Par 2014, Part II. LNCS, vol. 8806, pp. 438–449. Springer, Heidelberg (2014)