Experience on Vectorizing Lattice Boltzmann Kernels for Multi- and Many-Core Architectures

Enrico Calore[1,2], Nicola Demo[1], Sebastiano Fabio Schifano[1,2(✉)], and Raffaele Tripiccione[1,2]

[1] Università di Ferrara, Ferrara, Italy
[2] INFN Ferrara, Ferrara, Italy
schifano@fe.infn.it

Abstract. Current development trends of fast processors calls for an increasing number of cores, each core featuring wide vector processing units. Applications must then exploit both directions of parallelism to run efficiently. In this work we focus on the efficient use of vector instructions. These process several data-elements in parallel, and memory data layout plays an important role to make this efficient. An optimal memory-layout depends in principle on the access patterns of the algorithm but also on the architectural features of the processor. However, different parts of the application may have different requirements, and then the choice of the most efficient data-structure for vectorization has to be carefully assessed. We address these problems for a Lattice Boltzmann (LB) code, widely used in computational fluid-dynamics. We consider a state-of-the-art two-dimensional LB model, that accurately reproduces the thermo-hydrodynamics of a 2D-fluid. We write our codes in C and expose vector parallelism using directive-based programming approach. We consider different data layouts and analyze the corresponding performance. Our results show that, if an appropriate data layout is selected, it is possible to write a code for this class of applications that is automatically vectorized and performance portable on several architectures. We end up with a single code that runs efficiently onto traditional multi-core processors as well as on recent many-core systems such as the Xeon-Phi.

Keywords: Directive based compilation · Memory data layout · Vectorization · Accelerator processors

1 Introduction

Lattice Boltzmann (LB) methods are widely used in computational fluid dynamics. This class of applications – discrete in time and momenta and living on a discrete and regular grid of points, see later for details – offers a large amount of easily identified parallelism, making LB an ideal target for modern HPC systems [1–4]. However, exploiting available parallelism is becoming more and more difficult on recent processor architectures, exhibiting a large number of cores, each core being in turn able to execute SIMD instructions; both levels of parallelism have to be used.

R. Wyrzykowski et al. (Eds.): PPAM 2015, Part I, LNCS 9573, pp. 53–62, 2016.
DOI: 10.1007/978-3-319-32149-3_6

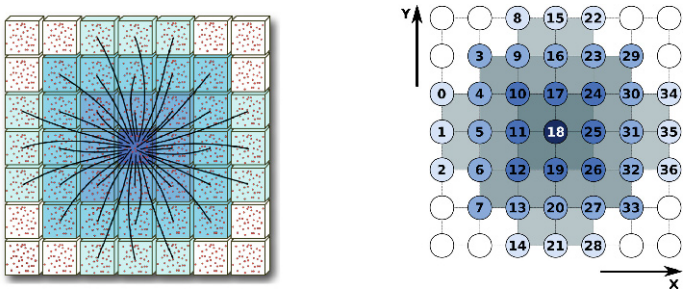


Fig. 1. Left: velocity vectors for the LB populations in the D2Q37 model. Right: populations are identified by an arbitrary label, identifying the lattice hop that they perform in the *propagate* phase.

For regular LB applications, it is easy to apply core parallelism assigning tiles of the physical lattice to different cores. However, exploiting vectorization requires additional care, and in particular two aspects are relevant: how to introduce and expose vector instructions in the code, and memory data-layout to enable efficient vector processing. Vector instructions can be explicitly introduced defining vector variables and processing them with specific functions – called *intrinsics* – which are mapped by the compiler onto the corresponding assembly instruction. However, even if potentially efficient, this approach prevents compiler to make all possible optimizations, and codes are not portable. Moreover, unskilled use can make the code less efficient than plain C code. In this work, we use the *directive* based approach provided by OpenMP 4.0 and supported by the Intel compiler, which allows to annotate standard C-codes with *pragma* directives to specify regions of the code to vectorize; this approach leaves to the compiler all optimization steps specific for the target architecture, and makes the code portable. Finding the best data structure layout to enable vector processing is relevant to ensure that data operated upon by SIMD operations are allocated on contiguous memory addresses so read (and write back) of data is fast enough not to starve the processing engine. This involves to find the best compromise between conflicting requirements between different parts of the code. Currently this is difficult to achieve automatically by programming tools, even if some experimental stencil compilers, such as PLUTO [5] ad POCHOIR [6] are promising solutions.

In this work, we use the directive approach for programming, and experiment with several memory data layouts. We assess the corresponding performance results, and we end up with just one implementation of our LB application that can be automatically and efficiently vectorized onto traditional multi-core processors and many-core processors such as the Xeon-Phi accelerator. Analyses of optimal data layouts for LB have been made in [7–9]. However, [7] focuses only on the *propagate* step, one of the two key kernels in LB codes, while [8] does not take into account vectorization; in [9] vectorization is considered using *intrinsics* functions only. None of these papers consider accelerators. 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 take into account also accelerators.

This paper is structured as follows: Sect. 2 gives an overview of LB methods, while Sect. 3 describes in details our implementations. Finally, Sect. 4 analyzes our performance results and compare them with those of earlier codes for the same LB application we have written in CUDA for GPU, and using *intrinsics* for traditional multi-core CPUs and the Xeon-Phi processor.

2 Lattice Boltzmann Methods

In this section, we sketchily introduce the computational method that we adopt, based on an advanced Lattice Boltzmann (LB) scheme. LB methods (see, e.g. [10] 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 2 or 3 dimensions, and using sets of populations of different size (a model in x dimensions based on y populations is labeled as $DxQy$). 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 (BGK) equation:

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

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

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

the equilibrium distributions ($f_l^{(eq)}$) are themselves functions of these macroscopic quantities [10]. With an appropriate choice of the set of lattice velocities $\boldsymbol{c}_l$ and of the equilibrium distributions $f_l^{(eq)}$, one shows that, performing an expansion in Δt and renormalizing the values of the physical velocity and temperature fields, the evolution of the macroscopic variables obeys the thermo-hydrodynamical equations of motion and the continuity equation:

$$\partial_t \rho + \rho \partial_i u_i = 0 \tag{3}$$

$$\rho D_t u_i = -\partial_i p - \rho g \delta_{i,2} + \nu \partial_{jj} u_i, \qquad \rho c_v D_t T + p \partial_i u_i = k \partial_{ii} T; \tag{4}$$

where $D_t = \partial_t + u_j \partial_j$ 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 ν 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 2-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 Fig. 1. This LB model, that automatically enforces the equation of state of a perfect gas ($p = \rho T$), was recently developed in [11,12];. Our optimization efforts have made it possible to perform large scale simulations of convective turbulence in several physics regimes (see e.g., [13,14]);

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 Eq. 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. Inspecting Eq. (1) one easily identifies the overall structure of the computation that evolves the system by one time step Δt: for each point $\boldsymbol{x}$ in the discrete grid the code: (i) gathers from neighboring sites the values of the fields f_l corresponding to populations drifting towards $\boldsymbol{x}$ with velocity $\boldsymbol{c}_l$ (`propagate` step) and then, (ii) performs all mathematical operations associated to the r.h.s. of Eq. (1) (`collide` step). One quickly sees that there is no correlation between different lattice points, so both steps can be performed in parallel on all grid points according to any convenient schedule, with the only constraint that step 1 precedes step 2. All other steps of a complete simulations have a negligible computational cost.

As already remarked, our D2Q37 model correctly and consistently describes the thermo-hydrodynamical equations of motion and the equation of state of a perfect gas; this translates into a more complex implementation than earlier 2D LB models as well as in demanding hardware requirements for memory bandwidth and floating-point throughput. Indeed, `propagate` implies accessing 37 neighbor cells to gather all populations, while `collide` executes ≈7000 double-precision floating point operations per lattice point.

3 Implementation and Optimization of LB Kernels

As already remarked, data organization plays a key role; popular layouts for LB methods are arrays of structures (AoS) or structure of arrays (SoA). In the AoS layout, population data for each lattice site are stored one after the other at successive memory locations, while in SoA, for each population of index i, all sites are stored one after the other. AoS enjoys locality for all data of each site; on the other hand, same-index populations of different sites are stored at non-unit strided addresses. We store the lattice in column-major order (Y direction) and instantiate two copies, that are alternatively read and written. Although this solution allocates more memory than having a single copy, it is required to process all lattice sites in parallel.

```
typedef struct {
  double *p[NPOP];
} pop_soa_t;

  // snippet of propagate code to move population index 0
  for ( xx = XMIN; xx < stopx; XMAX++ ) {
    #pragma vector nontemporal
    for( yy = YMIN; yy < YMAX; yy++ ) {
      idx = IDX(xx,yy);
      (nxt->p[0])[idx] = (prv->p[0])[idx+OXM3YP1];
    }
  }
```

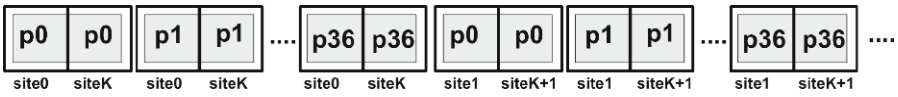
Fig. 2. Snippet of sample code for `propagate`, moving f_0 according to Fig. 1, right. `OXM3YP1` is the memory address offset associated to the population hop. The lattice is stored using the SoA layout.

3.1 Optimization of `propagate`

The `propagate` kernel moves populations for each lattice site according to the pattern of Fig. 1 (left) involving accesses at lattice-cells at distance up to 3 in the physical grid. Earlier works – e.g. [7] – has considered two implementation schemes: *push* and *pull*. The *push* scheme moves all populations of one site to appropriate neighbor sites, while *pull* gathers to one site populations belonging to neighbor sites. However, *push* performs aligned-reads and misaligned-writes, while *pull* does the opposite, and this results more efficient on modern architectures since aligned writes can bypass the cache hierarchy.

Vectorization of this kernel using SIMD instructions apply each move shown in Fig. 1 to several lattice sites in parallel depending on the size of vectors supported by the target processor, e.g. 4 and 8 for those we have considered. This simple vectorization is not possible if one uses an AoS layout, as populations of different sites are stored at non-contiguous memory addresses, requiring multiple memory accesses. In contrast, an SoA layout has unit stride for populations of fixed index i belonging to different sites, so data can be fetched in parallel from memory using vector instructions. This is the main reason for selecting SoA rather than AoS memory arrangement.

Figure 2 shows the C-code moving population of index 0 which – see Fig. 1 (right) – comes from sites three steps left and one up in the physical lattice. All other populations are moved in the same way. The code sweeps all lattice with two loops in X and Y; the inner loop is on Y, as elements are stored in column-major order. The `#pragma vector` directive notifies the compiler that the next loop can be vectorized; this is an Intel-specific directive, corresponding to `#pragma omp simd` in the OpenMP standard. This directive vectorize the inner loop in chunks of 4 or 8 words for the Haswell CPU and the Xeon-Phi accelerator respectively. We add the `nontemporal` attribute specifying that data is not used again by this kernel, so *non-temporal* stores can be used. The latter bypass the cache hierarchy, and reduce memory traffic by 1/3, with a corresponding save in time. Unfortunately OpenMP currently does not support this directive.




```
typedef struct { double c[VL]; } vdata_t;  // cluster

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

// snippet of propagate code to move population index 0
for ( xx = startx; xx < stopx; xx++ ) {
  for ( yy = 0; yy < SIZEYOVL; yy++, idx++ ) {
    idx =  IDX(xx,yy);
    #pragma vector aligned nontemporal
    for(tt = 0; tt < VL; tt++) {
      nxt[idx].p[0].c[tt] = prv[idx+OPOVL0].p[0].c[tt];
    }
  }
}

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

Fig. 3. Top: data arrangement for the CAoSoA layout; for illustration purposes, we take VL = 2. Bottom: snippet of sample codes for `propagate` and `collide` using this layout.

3.2 Optimization of `collide`

The `collide` kernel updates populations at each lattice site at the next time step, performing all mathematical operations associated to Eq. 1 (this is called *collision* in LB jargon). Input data are the populations gathered by the previous `propagate` phase. For each lattice site, this floating point intensive kernel uses only data belonging to the site on which it operates – previously gathered by `propagate` – so the available parallelism equals the size of the full lattice.

The SoA layout allows in principle to vectorize the code: to do so, the machine code should load in sequence blocks of words for all populations, each block as long as the vector size, and then perform all operations in SIMD mode using these blocks as operands. In practice, this implies reading relatively short data chunks from scattered memory locations, that the memory controller is not able to do efficiently. We therefore anticipate a limited performance as the compute-unit in the processor becomes data starved.

We then define a new data layout, offering to compilers a different handle to vectorization. We divide each population array in VL parts along Y (`VL` is the vector size), pack the populations at each site of each part into an array of `VL` elements (we call this block a *cluster*) and store clusters for all 37 populations one after the other, see Fig. 3. We call this layout a *Clustered Array of Structure of Array* (CAoSoA). This layout obviously allows vectorization of inner structures (clusters) of size VL, and at the same time improve loacality of population

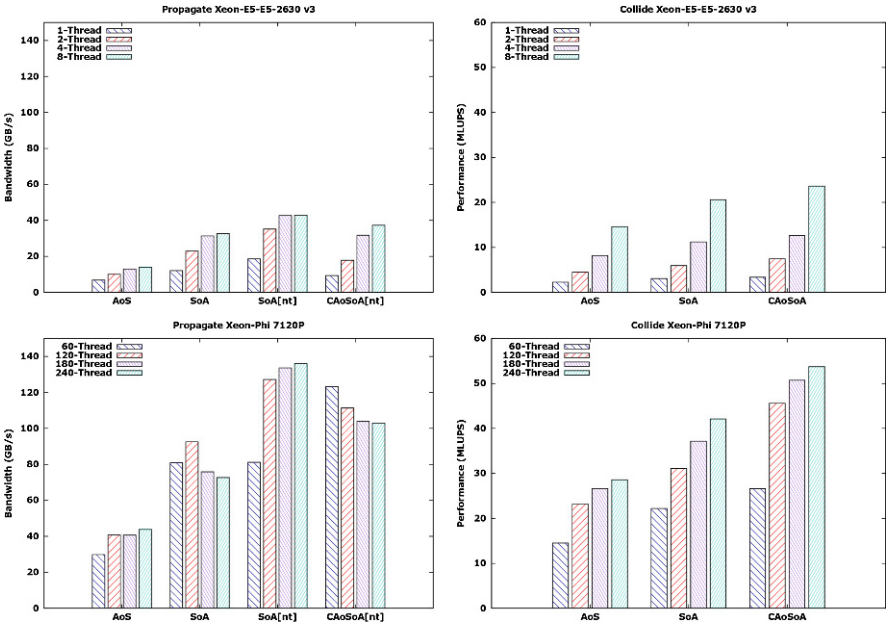


Fig. 4. Benchmark results for `propagate` and `collide` on a lattice of 2160 × 8192 points, using different data layouts. Top: Xeon E5-2630-v3 CPU; bottom: Xeon-Phi 7120P accelerator.

keeping at contiguous and aligned addresses all data items needed to handle each lattice point in the kernel. Figure 3 shows the definition of the `vdata_t` data type, corresponding to a *cluster*, and a snippet of representative small sections of the code. 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 layout is probably less efficient for *propagate*: in the next section we discuss the corresponding tradeoffs.

Table 1. Performance comparison of propagate and collide kernels using the CaAoSoA scheme with intrinsic functions and pragma directives. The rightmost column lists performance figures for an NVIDIA K40 GPU. Data refer to a lattice of 2160 × 8192 points.

	E5-2630 v3 CAoSoA		Intel Xeon 7120 CAoSoA		NVIDIA K40 SoA
	Intrinsics	Directives	Intrinsics	Directives	CUDA
Propagate (GB/s)	36	37	86	103	187
Collide (MLUPS)	14	24	55	54	108
Collide (GFs)	94	154	365	351	703
Global (MLUPS)	12	17	40	41	81

4 Results and Conclusions

We have tested our code on an 8-core Xeon E5-2630-v3 CPU, based on the Haswell micro-architecture running at 2.4 GHz, and on a 61-core Xeon-Phi 7120P accelerator based on the *Many Integrated Core* architecture.

Figure 4 shows performance results for all data layouts that we have considered. For the *propagate* kernel we measure the effective bandwidth, while for *collide* we use *Millions Lattice Updates per Second* (MLUPS) figures, a standard performance metric for this application.

We first consider CPU results. Both kernels have been run using 1 thread per core since use hyperthreading does not improve performances. For *propagate*, the bandwidth increases with the number of threads used, as using only one thread does not saturate the bandwidth made available by all memory channels. The AoS scheme has poor performance since it prevents vectorization and use of non-temporal stores, while the SoA scheme reaches the best results with 4 and 8 threads and non-temporal stores enabled; in this case the bandwidth is ≈43 GB/s, that is ≈70 % of theoretical peak (59 GB/s) and ≈86 % of bandwidth measured with STREAM benchmark [15] (≈50 GB/s). As expected, the CAoSoA layout reduces performance by ≈15 % (≈37 GB/s). The performance of *collide* grows with the number of threads. The best performance value we measure is 24 MLUPS using 8 cores and the CAoSoA layout; this is 1.2X faster than SoA and 1.6X faster than AoS.

On the Xeon-Phi the situation is somewhat different. In this case we have used hyperthreading since this is required to hide the huge latency in accessing external memory. For *propagate*, performance depends weakly on the number of threads per core, reflecting a complex interaction between the complexity of the memory system and the role of multi-threading to mitigate latency effects. Using the SoA[nt] layout the peak value of bandwidth is 136 GB/s. Using the CAoSoA[nt] layout, the bandwidth decreases to 103 GB/s, after prefetching hints have been added to the code. These figures are 30 % or less of the theoretical peak (≈350 GB/s), but an encouraging ≈62 % of the STREAM benchmark [15], a widely acknowledged reference test for the Xeon-Phi, that measures an effective bandwidth of ≈165 GB/s. In much the same way as for the CPU case, *collide* scales in performance with the number of threads up to 240 threads for all storage layouts. The best value is obtained using the CAoSoA layout, approximately 1.3X faster than SoA, and 1.8X faster than the AoS.

Table 1 summarizes our results, comparing performances for different data layouts on different processors; it considers implementations using intrinsics instructions [16–18] and directive-based ones (this work). We also show global MLUPS figures, a reasonably accurate performance metric of a complete code assuming that `propagate` and `collide` kernels are executed and applied to the lattice one after the other; moreover we also quote previous results for GPUs using the SoA layout [19–21].

In summary, we verify that `propagate` and `collide` have conflicting requirements for data layout. The hybrid CAoSoA scheme combines the benefits of SoA since clusters can be easily vectorized, with those of AoS where population

locality boosts the performance of the `collide` kernel. All in all, considering the overall performance the CAoSoA layout is the preferred one for both strands of processor architectures. It is interesting to note that our directive-based code is as fast or even faster than those using intrinsic functions, as in the former case the compiler is able to apply a larger set of optimization strategies.

Coming now to a comparison of performance across different processors, we see that for this class of codes – typical of a large class of HPC applications – Xeon-Phi accelerators offer roughly 2X better performance than state-of-the-art CPUs, while a NVIDIA K40 GPU is approximately 4X better. We also remark that GPUs have a more efficient memory interface, so outstanding performance is obtained without a careful adjustment of the SoA data layout.

In conclusion, an important result of this work is that it is possible to write directive-based programs that are: (i) as efficient or even more efficient than handcrafted codes and, (ii) reasonably performance-portable across different present available (and hopefully future) architectures. This requires to invest a significant amount of effort in the definition of an appropriate data layout, choosing a non-trivial optimal (or almost optimal) layout among conflicting requirements. This step is clearly out of the scope of current programming environment; considering a longer time horizon, we see here a potential large space for data definitions that do not prescribe a fixed memory layout, leaving the compiler free to make appropriate choices during the compilation process.

Acknowledgements. This work was done in the framework of the COKA, COSA and SUMA projects of INFN. We would like to thank CINECA (Italy) for access to their HPC systems.

References

1. Williams, S., et al.: Lattice Boltzmann simulation optimization on leading multicore platforms. In: IEEE International Symposium on Parallel and Distributed Processing (2008). doi:10.1109/IPDPS.2008.4536295
2. Williams, S., et al.: Optimization of a Lattice Boltzmann computation on state-of-the-art multicore platforms. J. Parallel Distrib. Comput. **69**, 762–777 (2009). doi:10.1016/j.jpdc.2009.04.002
3. Bernaschi, M., et al.: A flexible high-performance Lattice Boltzmann GPU code for the simulations of fluid flows in complex geometries. Concurrency Comput. Pract. Experience **22**(1) (2010). doi:10.1002/cpe.1466
4. Ye, Z.: Lattice Boltzmann based PDE solver on the GPU. Vis. J. **24**(5), 323–333 (2008). doi:10.1007/s00371-007-0191-y
5. Bondhugula, U., et al.: A practical and automatic polyhedral program optimization system. In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (2008). doi:10.1145/1375581.1375595
6. Tang, Y., et al.: The pochoir stencil compiler. In: Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (2011). doi:10.1145/1989493.1989508
7. Wittmann, M., et al.: Comparison of different Propagation Steps for the Lattice Boltzmann Method, 3 November 2011. arXiv:1111.0922vI

8. Shet, A.G., et al.: Data structure and movement for lattice-based simulations. Phys. Rev. E **88**, 013314 (2013). doi:10.1103/PhysRevE.88.013314
9. Shet, A.G., et al.: On vectorization for lattice based simulations. Int. J. Mod. Phys. C **24**, 1340011 (2013). doi:10.1142/S0129183113400111
10. Succi, S.: The Lattice-Boltzmann Equation. Oxford University Press, Oxford (2001)
11. Sbragaglia, M., et al.: Lattice Boltzmann method with self-consistent thermo-hydrodynamic equilibria. J. Fluid Mech. **628**, 299–309 (2009). doi:10.1017/S002211200900665X
12. 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
13. 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
14. 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
15. McCalpin, J.: The STREAM Benchmark: Computer Memory Bandwidth. http://www.streambench.org/
16. Mantovani, F., et al.: Exploiting parallelism in many-core architectures: Lattice Boltzmann models as a test case. J. Phys. Conf. Ser. **454**, 012015 (2013). doi:10.1088/1742-6596/454/1/012015
17. Mantovani, F., et al.: Performance issues on many-core processors: a D2Q37 Lattice Boltzmann scheme as a test-case. Comp. Fluids **88** (2013). doi:10.1016/j.compfluid.2013.05.014
18. Crimi, G., et al.: Early experience on porting and running a Lattice Boltzmann code on the Xeon-phi co-processor. Proc. Comput. Sci. **18**, 551–560 (2013). doi:10.1016/j.procs.2013.05.219
19. Biferale, L., et al.: An optimized D2Q37 Lattice Boltzmann code on GP-GPUs. Comput. Fluids **80** (2013). doi:10.1016/j.compfluid.2012.06.003
20. 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)
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