Exploiting parallelism in many-core architectures: Lattice Boltzmann models as a test case

F Mantovani[1], M Pivanti[2], S F Schifano[3] and R Tripiccione[4]

[1] Department of Physics, Univesität Regensburg, Germany
[2] Department of Physics, Università di Roma *La Sapienza*, Italy
[3] Department of Mathematics and Informatics, Università di Ferrara and INFN, Italy
[4] Department of Physics and CMCS, Università di Ferrara and INFN, Italy

E-mail: `filippo.mantovani@regensburg.de, pivanti@fe.infn.it, schifano@fe.infn.it, tripiccione@fe.infn.it`

Abstract. In this paper we address the problem of identifying and exploiting techniques that optimize the performance of large scale scientific codes on many-core processors. We consider as a test-bed a state-of-the-art Lattice Boltzmann (LB) model, that accurately reproduces the thermo-hydrodynamics of a 2D-fluid obeying the equations of state of a perfect gas. The regular structure of Lattice Boltzmann algorithms makes it relatively easy to identify a large degree of available parallelism; the challenge is that of mapping this parallelism onto processors whose architecture is becoming more and more complex, both in terms of an increasing number of independent cores and – within each core – of vector instructions on longer and longer data words. We take as an example the Intel *Sandy Bridge* micro-architecture, that supports AVX instructions operating on 256-bit vectors; we address the problem of efficiently implementing the key computational kernels of LB codes – streaming and collision – on this family of processors; we introduce several successive optimization steps and quantitatively assess the impact of each of them on performance. Our final result is a production-ready code already in use for large scale simulations of the Rayleigh-Taylor instability. We analyze both raw performance and scaling figures, and compare with GPU-based implementations of similar codes.

1. Introduction

For several years, the main path to performance in large scale scientific computations has been that of introducing more parallelism in the associated algorithms and in their implementations. This trend reflects a matching evolution in system architectures that – over the years – have delivered more computing power by increasing the number of processing elements: today, high-end systems routinely have tens of thousands of interconnected cores.

Following this trend, application development has focused on identifying and exploiting a larger fraction of parallelism from the underlying algorithms and on mapping applications on a larger number of processing threads; several programming and runtime environments, such as MPI or openMP, are now available to support this trend. For some algorithms, whose parallelizable fraction is very large, the limits set by Amdahl's law have not prevented to reach good *strong* scaling figures. In many other cases, increased parallel computing resources have been used to tackle problems of larger scale, and, as summarized by Gustafson's law [1], at least good *weak* scaling figures have been obtained.

Content from this work may be used under the terms of the Creative Commons Attribution 3.0 licence. Any further distribution of this work must maintain attribution to the author(s) and the title of the work, journal citation and DOI.
Published under licence by IOP Publishing Ltd

Recent developments in processor architecture introduce yet a new feature: they try to exploit several *different* parallelization options within the *same* processor: not only is the number of processors of one processing element and the number of cores available inside one processor steadily increasing, but data-paths within each core have now more parallel features, such as SIMD or vector extensions. At the extreme edge of this spectrum one finds GP-GPU architectures, that can be seen as assemblies of several hundred "slim" cores, so typically one core processes one element of the data-set of the program.

Memory access has also become more complex: most current processing elements support a common addressing space for all their cores, and at least some level of cache is kept coherent across the node, but memory bandwidth and latency – as seen by one specific core – is not uniform across the addressing space, as it depends strongly on how "close" a given memory location is to the processing core.

Performance optimization for these recent architectures is becoming more complex, as some difficult balance between conflicting requirements has to be found: the problem has to be partitioned over many independent threads, but – within each thread – one has to identify data elements with no data-dependencies that can be operated upon by vector instructions; at the same time data allocation and data sharing policies may have a strong impact on cache performance and on memory bandwidth.

In this paper, we try to identify several optimization strategies to tackle and solve the problems described above, and to assess their impact on performance. We focus on one specific test-case, a state-of-the-art Lattice Boltzmann (LB) algorithm in 2 dimensions that has a large and easily identified available parallelism and a very regular algorithmic structure; we also limit ourselves to latest generation multi-core Intel architectures, that assemble a large number of "fat" cores within one processor.

While our main motivation for this work has been that of developing a very fast code for large scale fluid-dynamics simulations, here we offer to the reader an analysis of the problems that will be encountered in most large scale scientific applications and that must be overcome to sustain performance on present (and possibly future) processors. Most of our optimization steps match specific architectural features of these processors; this allows a close assessment of the potential for performance of each architectural feature, but on the other side makes programming difficult and strongly machine dependent; it seems obvious to us that the development of efficient (semi-)automatic tools able to carry out at least part of this work starting from high level programs is an important problem, that should be extensively addressed.

This paper is structured as follows: in section 2 we discuss the architecture of typical large scale HPC commodity systems, and general software optimization strategies suitable for recent processors; section 3 briefly describes our Lattice Boltzmann algorithm, while section 4 presents in detail all optimizations steps that we have developed for our LB code. Section 5 summarizes our performance results and draws some comparisons with different architectures, such as GPUs; the paper ends with our concluding remarks. This paper draws on and expands upon earlier results for a previous generation of Intel processors based on the *Nehalem* micro-architecture [2, 3], and on GP-GPU [9, 10] systems based on the Fermi processor.

2. Multi-core CPU Commodity Systems

The typical HPC system for scientific computing today is a large cluster of nodes based on commodity multi-core CPUs, and interconnected by a high-speed switched network, such as Infiniband.

In recent years multi-core architecture has become the standard approach to develop high performance processors. Several cores are integrated on a single chip device; this has allowed peak performance at node level to scale according to Moore's law. In many cases, the cores are x86-CPUs, usually including two levels of caches, and a floating-point unit able to execute

vector instructions; vector size increases steadily as newer processor versions are released. Today, 256 bit vectors have been adopted in the Intel Sandy Bridge micro-architecture, while the forthcoming Intel MIC architectures foresee 512 bits. Cores within a device share a large L3-cache and external memory.

Many system vendors provide so-called multi-socket platforms, based on several multi-core CPUs. Typically, two, four or eight sockets can be installed on a mother-board; each socket has its own memory-bank and is connected to the other sockets by dedicated busses. Appropriate protocols allow to share the global memory space and keep the content of caches coherent.

From a programming point of view these systems operate as *Symmetric Multiprocessors* (SMP), which means that they can be programmed as a single processor with N_c cores (N_c equal to the sum of cores of all sockets) sharing the full memory space and controlled by a single instance of the *Linux* operating system. Performances heavily rely on the ability of programmers, compilers and run-time support to carefully exploit parallelism on *all* available hardware features. Relevant optimizations for performances are:

- core parallelism: the code should allow all cores to work in parallel exploiting MIMD or SPMD multi-task parallelism; in the first case the application is decomposed in several sub-tasks and each one is executed by a different core; in the latter case, that can be also combined with the previous one, the data-set is typically partitioned among the cores, each one executing the same task.
- vector programming: each core has to process the data-set of the application using vector instructions and exploiting streaming-parallelism (SIMD); the number of data-items that can be processed by vector instructions depends on the architecture of the CPU; on the latest Sandy Bridge architecture the vector size is 256-bit, that is up to 8 single-precision or 4 double-precision floating point numbers.
- efficient use of caches: memory hierarchies of commodity systems are based on the concept of cache to minimize over-heads associated to accessing main memories. The application code has to exploit cache-reuse in order to save time in memory access; this may have a serious impact on the organization of the data-layout for the application.
- NUMA control: multi-socket platforms are *Non Uniform Memory Access* (NUMA) systems; this means that the time to access data in memory is related to the allocation of the threads that perform the memory access; the time is shorter to access data stored onto memory attached to the socket where the thread is running.

3. Lattice Boltzmann

The Lattice Boltzmann method is a computational approach that describes fluid dynamics by lattice discretization in position and momentum space [4]. On a discrete lattice (in $D = 2$ or 3 dimensions), the actual dynamics is replaced by a synthetic one, based on a set of lattice populations ($f_l(x,t)$); each population has a given fixed lattice velocity $\boldsymbol{c}_l$, associated to the fact that, at each time step, it drifts towards a nearby lattice site; populations evolve in (discrete) time according to the following equation:

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

Macroscopic observables (density ρ, velocity $\boldsymbol{u}$ and temperature T) are defined in terms of the $f_l(x,t)$:

$$\rho = \sum_l f_l, \tag{2}$$

$$\rho \boldsymbol{u} = \sum_l \boldsymbol{c}_l f_l, \tag{3}$$

$$D\rho T = \sum_l |\boldsymbol{c}_l - \boldsymbol{u}|^2 f_l, \tag{4}$$

and the equilibrium distributions ($f_l^{(eq)}$) are themselves function of these macroscopic quantities [4].

LB models are now available in many different versions, characterized by the dimension of the lattice, the number of populations that they use (they are usually labelled as DnQm, where n is the number of dimensions and m is the number of populations) and by the explicit functional form of the equilibrium distributions $f_l^{(eq)}$. Starting from early models (e.g., D2Q9 or D3Q19), able to describe an incompressible fluid at constant temperature, recent LB algorithms describe the thermo-hydrodynamics of multiple-phases or correctly treat compressible fluids with realistic equations of state.

In the following we focus on a recently proposed D2Q37 model that correctly reproduces the equation of state of the fluid, regarded as a perfect gas ($p = \rho T$); see [5, 6] for full details. For this model, one shows that, after appropriate shift and re-normalization of the velocity and temperature fields, one recovers, via a Taylor expansion in Δt, the following set of thermo-hydrodynamical equations:

$$D_t \rho = -\rho \partial_i u_i^{(H)} \tag{5}$$

$$\rho D_t u_i^{(H)} = -\partial_i p - \rho g \delta_{i,2} + \nu \partial_{jj} u_i^{(H)} \tag{6}$$

$$\rho c_v D_t T^{(H)} + p \partial_i u_i^{(H)} = k \partial_{ii} T^{(H)}; \tag{7}$$

where superscripts H flag renormalized (physical) quantities, $D_t = \partial_t + u_j^{(H)} \partial_j$ is the material derivative and we neglect viscous heating; c_v is the specific heat at constant volume for an ideal gas, $p = \rho T^{(H)}$, and ν and k are the transport coefficients; g is the acceleration of gravity. In this model, population velocities are associated to moves on the lattice that reach points up to three lattice points away, see figure 3.

¿From the point of view of the present paper, LB algorithms are a nice test case, as they offer a huge degree of available parallelism, that can be immediately and easily identified: defining $\boldsymbol{y} = \boldsymbol{x} + \boldsymbol{c}_l \Delta t$, one rewrites the main evolution equation as:

$$f_l(\boldsymbol{y}, t + \Delta t) = f_l(\boldsymbol{y} - \boldsymbol{c}_l \Delta t, t) - \frac{\Delta t}{\tau} \left(f_l(\boldsymbol{y} - \boldsymbol{c}_l \Delta t, t) - f_l^{(eq)} \right) \tag{8}$$

It is now easy to identify the overall structure of the computation that evolves the system by one time step Δt; for each point $\boldsymbol{y}$ in the discrete grid one:

(i) gathers from neighboring sites the values of the fields f_l corresponding to populations that drift towards $\boldsymbol{y}$ with velocity $\boldsymbol{c}_l$ and then

(ii) performs all mathematical processing needed to compute (*in a completely local fashion*) the quantities appearing in the equation above. This step is slightly more complex if one wants to take into account reactive effects (combustion); indeed, in that case the divergence of the velocity field has to be explicitly computed. This means that a further gather operation must be performed midway in this (otherwise local) compute intensive step.

The key remark is that both steps above are completely uncorrelated for different points of the grid, so they can be parallelized according to any convenient schedule, as long as one makes sure that, for each and all grid points, step 1 is performed before step 2. In principle, the available

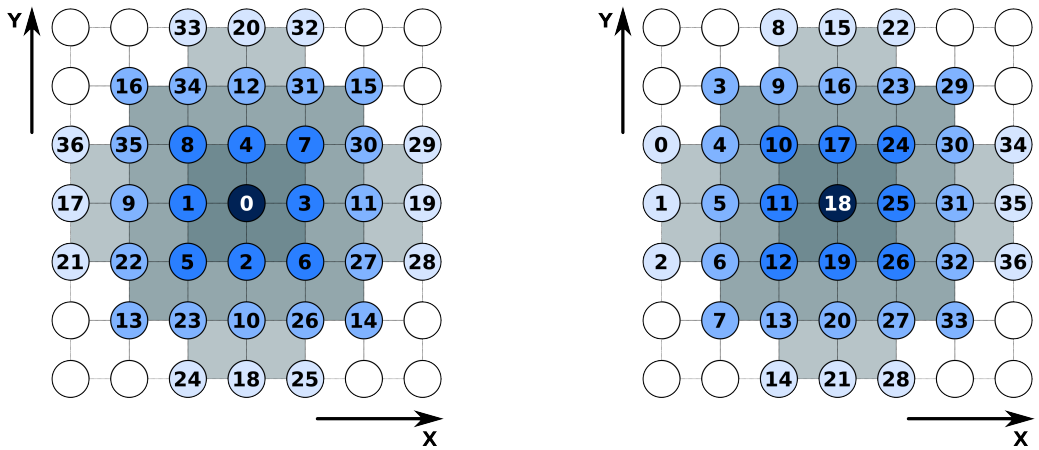

Figure 1. Velocity vectors for the LB populations in the D2Q37 model. Population labelling is arbitrary; populations are stored at consecutive memory locations, in the order given by their labels: at left our first labelling choice; at right, a different labeling , that improves cache reuse, as discussed in detail in section 4.

parallelism is as large as the number of sites of the lattice (that easily grows to hundreds of millions of sites).

A reference implementation of the LB algorithm repeatedly evolves the lattice-cells of the system for one time step. For each point in the grid, data needed to compute the new value of each population at each site in the grid is gathered from nearby sites, and then a fully local processing step is performed.

Reflecting this overall organization, the class of codes that we have developed processes every grid-point by applying in order the following three main computational phases:

- stream(): this phase gathers for each site 37 populations from neighbors according to the scheme of figure 3. This process does not make any floating-point computation but only accesses sparse blocks of memory locations. It collects at each site all the populations that will interact at the next computational phase, collide(). In this step, each site accesses the populations of neighbor cells at distance up to 3 in the physical grid.
- bc(): this phase adjusts values of the cells at the top and bottom edges of the lattice to enforce appropriate boundary conditions (e.g., a constant given temperature and zero velocity). At the right and left boundaries, we apply periodic boundary conditions. This is most easily done by allocating additional storage where copies of an appropriate number (3 in our case) of the rightmost and leftmost columns of the lattice are placed before the stream() step. Points at the right/left boundaries can then be processed as those in the bulk. If needed, boundary conditions could of course be enforced in the same way as it is done for the top and bottom edges.
- collide(): this phase performs all the mathematical steps associated to equation 8 and needed to compute the population values at each lattice site at the new time step (this is called "collision", in LB jargon). Input data for this phase are the populations gathered by the previous stream() phase. This step is the floating point most intensive section of the code; it uses only the population members of the site on which it operates, making the processing of different sites fully uncorrelated.

In the following section we describe in detail a family of implementations of these kernels that try to match at best the available parallelism with the architectural structure of a recent

many-core CPU.

4. D2Q37 Implementation and Optimization

Our codes are targeted for multi-socket systems based on the Intel Sandy Bridge micro-architecture. All performance results shown below refer to a pre-production system based on dual Intel Xeon E5-2680 sockets (2-SB, in short). Each socket is an eight-core processor operating at 2.7 GHz, with 20 MB of L3-cache and 32 GB of RAM. Each core supports the execution of the *Advanced Vector Extensions* instruction set, using 256-bit register operands, on which 8 single-precision or 4 double-precision numbers can be packed; it completes two AVX double precision operations – one add and one multiply – at each clock cycle, delivering a peak performance of 21.6 double precision Gflops per core. All in all, the system has a peak memory bandwidth of 85.3 GB/s, and a peak double precision performance of 345.6 Gflops.

Data organization schemes for LB codes are mainly of two kinds, namely array-of-structures (AoS) and structure-of-arrays (SoA). For cache-based CPUs the AoS scheme is preferable as it keeps all populations of each lattice-site close together in memory; this improves the locality of populations associated to each lattice point, and better suits the cache hierarchy of the processor. This is relevant for performance of the collision kernel, the most compute intensive part of the code, since all computation involved at each lattice site only depends on the populations associated to the site itself. In the AoS scheme each lattice-cell is stored as an array of 37 double-precision floating-point values representing the populations of the D2Q37 LB model. In our implementation we store the lattice in column-major order, and we keep in memory two copies of the lattice; each step reads inputs from one copy and writes results to the other. This solution is more memory expensive than having a single copy, but it makes the implementation of the code much easier, and on current systems memory size is not a limiting resource for this problem. Our LB code runs one thread per core, and a lattice of size $L_x \times L_y$ is split in *sub-lattices* along the X dimension, each associated to a different thread. The threads are topologically arranged in a ring, and each sub-lattice includes three left and three right *ghost columns*; at the beginning of each loop, threads copy the three leftmost and rightmost columns from the sub-lattice of the previous and next thread onto the ghost columns of its own sub-lattice. Within each core, K sites of the sub-lattice are processed in parallel, exploiting data-parallelism using vector instructions; in the case of Sandy Bridge architectures, four double precision number can be packet into an AVX vector, so $K = 4$.

Parallelism at core level is managed through the standard-POSIX Pthreads library, the closer approach to the *Linux* kernel, to avoid any overheads associated to high-level libraries such as openMP or openMPI.

We have developed two versions of the code. The first version (V1) keeps the computation of the stream and collide kernels separate; this is necessary if reactive dynamics is enabled because we have to compute the divergence of the velocity field which requires to access all lattice points and has to be performed by an additional step between stream and collide. Version V2 is used if reactive dynamics is not enabled; in this we can merge computation of stream and collide in just one step, applied in sequence to all cells of the lattice. This optimization step is well known in literature, see for example [7].

Code version V1 is organized in several computational steps, each synchronized through the use of a pthread-barrier:

(i) two threads (in our case threads 0 and 1) copy the left and right borders onto the appropriate frames. This operation is simply performed by memory-copies; after receiving data from the neighbor threads, both threads apply stream kernel to the three lattice columns close to the frame that they have just copied.

(ii) while step (i) is executed, the remaining threads compute in parallel the stream kernel on all columns far enough from their Y-edges.

(iii) at the end of step (i) and (ii) two threads compute the bc kernel on the cells close to the top and the bottom of the lattice.

(iv) after step (iii), all threads start computing the collide phase, each on a different portion of the sub-lattice.

The organization of code version V2 is not fully straightforward, as the correct enforcement of the boundary conditions (execution of the bc kernel) has to be done after stream but before collide. In this case the computation is organized in three main steps, each followed by a synchronization barrier:

step 1: two threads copy the border-columns from neighbor threads, while two other threads execute stream for the upper and lower rows of the grid, leaving out the first and the last 3 columns; for these columns stream will be computed by threads performing the copy of border columns, as soon as they have finished to copy data into the ghost columns.

step 2: two threads adjust the boundary conditions for the cells located at the top and bottom rows of the grid. After one step of synchronization, all threads start computing the collide kernel for the same cells.

step 3: all threads compute stream and collide for the remaining cells of grid (the ones in the bulk).

The organization of the codes described above allows us to exploit core parallelism. A further optimization aims to exploit data parallelism, using the AVX vector instructions available on the Sandy Bridge processor, and processing four cells of the lattice at the same time. Data parallelism can be exploited either by the compiler, or by using *intrinsic* functions. In the first case the program is a scalar code, and is compiled enabling the *auto-vectorization* mode of the compiler. This lets the compiler introduce vector instructions, e.g. unrolling a loop and processing two or more independent iterations (according to the type of variables involved) using SIMD AVX instructions. Alternatively, the program may be handcrafted to use vector variables which are processed using *intrinsic* functions mapped by the compiler directly onto vector instructions. For example a double precision vector sum can be performed by the code line `c = _mm256_add_pd (a, b)` where `a, b, c` are vector variables of type `__mm256d`. This approach is allowed both by the GCC and ICC compilers. A third, more elegant and transparent way to generate vector code is that of using custom data vector definitions, which is allowed by the GCC compiler. GCC allows to define a vector data-type where two or more primitive-types (int, float, double, ...) can be packed in a vector; this can be done using the vector_size attribute in the type definition. Operations on vector data-type variables can be performed using the usual infix notation, and are then translated by the compiler generating assembly code that uses streaming instructions for the micro-architecture selected by the appropriate compilation flag. We have used this approach for all our codes, and we have used the GCC compiler throughout; we have also verified in a few selected cases that the ICC compiler reaches similar performances ($\pm 10\%$). We use this feature to define a new vector data type called fourD_t, see figure 2 left, to pack four double variables in an AVX vector. Each element of the population array then stores populations of four lattice cells. In our case we pack four lattice cells at distance $L_y/4$, as shown in figure 2 right.

Figure 3 shows performances for the collide (left) and stream (right) benchmarks as a function of the numbers of threads, running on a dual-socket eight-core Sandy Bridge system. We compare performance of the scalar code (GCC autovec) compiled enabling the auto-vectorization mode of the compiler, with that obtained by the vector code using vector variables (GCC vec). Performance of the collide kernel scales well with the number of threads; the version using vector variables is a factor $2\times$ better than the scalar code with auto-vectorization, and a factor $3.5\times$

[t]

```
typedef double fourD_t
        __attribute__ ((vector_size(32)));

typedef struct {
  fourD_t p[37];      // populations array
} v_pop_t;
```

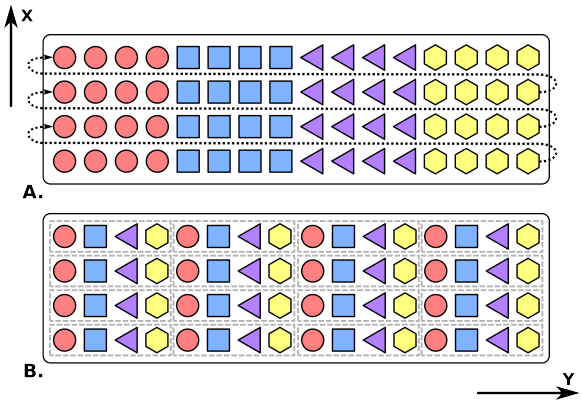

Figure 2. Left: definition of the structure holding four lattice cells at distance $NY/4$, that we use to exploit the AVX vector instructions of the Sandy Bridge processors. Right: visualization of how cells of a sample lattice of size 4×16 (A) have been packed to be processed by AVX vector instructions (B).

better than the scalar code compiled without auto-vectorization. Using 16 threads the kernel delivers 221 Gflops corresponding to 64% of peak.

Concerning the performance of the stream kernel, the compiler is not able to vectorize the code, so the compiled code executes loads and stores of scalar registers. Since loads and stores are actually made to cache, and eventually full cache lines are moved to/from main memory, the impact on performance should not be too large. However, a significant part of the workload associated to stream goes into the quite complex evaluation of addresses. In the vectorized version, this burden is reduced by a factor four, so one expects much better performance in this case. Indeed, the code version using vector variables – which already includes some of the optimizations that will be discussed later – is a factor approximately 1.7× better than the scalar code, and the memory bandwidth, saturates at 35 GB/s, ≈ 39% of peak.

We have introduced further optimizations to improve memory and cache efficiency; these optimizations are mainly relevant for the performance of the stream kernel. We remind that, in our implementation the cells of the lattice are stored in column major order; moreover, the

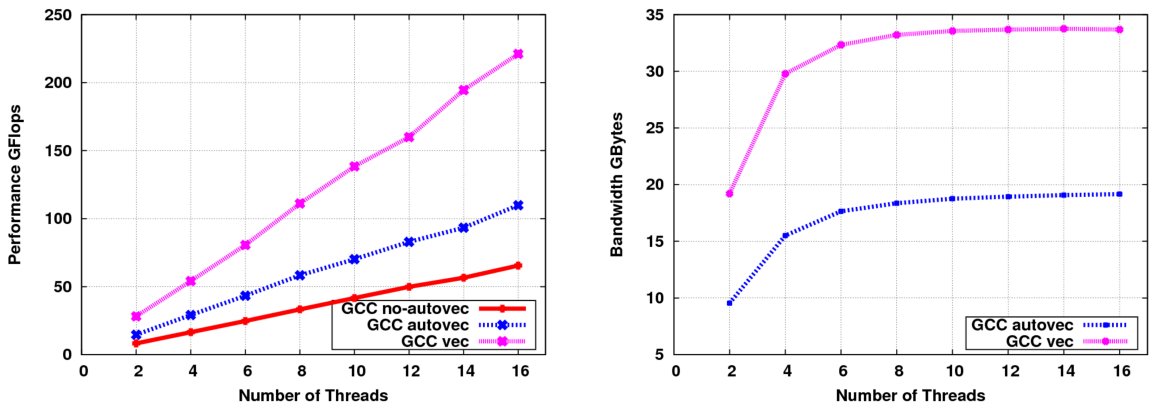

Figure 3. Performance figures for collide (left) and stream (right) kernels on a lattice of 1680×16000 points. For both kernels we show the performances obtained by the code compiled enabling the auto-vectorization mode of the compiler, and by the code that use intrinsic AVX instructions.

Table 1. Performance of the stream benchmark running on 16 threads, for several sizes of the lattice. We show the effective bandwidth, as we add more and more optimizations steps. Left to right, basic code, NUMA control (+NUMA), new population labelling (+New-Labelling), cache-blocking (+Cache-Blocking), non-temporal instructions (+NT).

Stream Bandwidth (GB/s))						
$L_x \times L_y$	Size (GB)	Base	+NUMA	+New-Labelling	+Cache-Blocking	+NT
256×32000	2.26	11.7	25.4	33.9	44.0	58.5
480×32000	4.23	12.8	25.5	33.7	44.3	58.2
1680×16000	7.41	11.6	25.4	33.7	43.8	58.5

stream() step performs scattered accesses in local memory to gather populations at distance 1, 2 and 3 in the grid; one expects that performances may depend on details of cache and memory-allocation policies, as described in [8]. To improve cache performance, we have first chosen an appropriate labeling for the populations, as shown in figure 3 right; this new labeling maximizes the number of population values loaded into cache when processing point (x, y) in the lattice, and still available in cache when considering points $(x, y+1)$ and $(x, y+2)$. Moreover, to improve cache re-use, we have also implemented *cache-blocking*; each core executes the stream operation on a block of $B_x \times B_y$ cells such that the data size associates to each block fits into the L3 cache. We combine cache-blocking with the use of *non-temporal* store instructions; non-temporal stores do not require a prior cache-line read for ownership (RFO), but write data to memory directly, improving memory bandwidth. Finally, we also managed memory and thread allocation using the Linux NUMA library. The lattice array allocated on each node is split into sub-arrays, and each one is mapped onto the memory-bank close to the CPU of the thread processing it. This avoid conflicts due to a thread running on a CPU and accessing memory locations allocated on the banks physically attached to a different CPU. For a detailed analysis of this allocation style, see [2, 3].

Table 1 lists the effective bandwidth obtained by the stream benchmark running on the dual socket Sandy Bridge system, as we introduce more and more optimization steps. The version including all optimizations improves by a factor ≈ 10 with respect to the `Base` version; it delivers a peak bandwidth of ≈ 58 GB/s corresponding to approximately 68% of the theoretical peak.

5. Results and Conclusions

In this section we summarize the results of our implementation, and compare performances of our full codes (versions V1 and V2) with that obtained on a GP-GPU system.

Table 2 lists performance figures of production-ready code versions V1 and V2, running on the 2-SB system; we compare performances with that of our previous implementation [11] developed for a NVIDIA C2050 card based on the Fermi processor. Code version V1 delivers ≈ 166 Gflops on the 2-SB system, corresponding to 48% of the peak, while Code version V2 delivers ≈ 216 Gflops on the the same system, faster than the corresponding version for the GP-GPU; this corresponds to 62% of peak.

Our LB code optimized for multi-core CPUs has been extensively used to study several features of the Rayleigh-Taylor instability in 2 dimension; physics results will be published elsewhere; in picture 4 we show the temperature maps from a simulation of a small Rayleigh-Taylor cell of size 512×1024, at several stages during time evolution; the simulation as been performed using code version V2 running on the 2-SB system.

In this paper, we have described several optimization-steps that we have applied to our LB

Table 2. Performances of production-ready code versions V1 (left) and V2 (right), on a C2050 board based on a NVIDIA Fermi GP-GPU, and on a dual-socket Sandy Bridge system. For each code version we show the execution time, and the corresponding performance figures; we also list the sustained performance P of the full code, its efficiency, and two metrics relevant for the users: the time to update one site and its inverse (MLUps).

	C2050	2-SB
propagate (ms)	29.11	42.12
collide (ms)	154.10	146.00
propagate (GB/s)	84	60
collide (GF/s)	205.4	220
T/site (ns)	44	46
MLUps	22	21.7
P (GF/s)	172	166
R_{max}	33%	48%

	C2050	2-SB
propagateCollide (ms)	167.2	144.0
propagateCollide (GF/s)	190	224
T/site (ns)	40	35
MLUps	25	28.2
P (GF/s)	188	216
R_{max}	36%	62%

code to boost performances for a class of high-end commodity multi-core processors widely used for scientific computing. Our optimizations have implied a rather complex restructuring of the original algorithm, in order to exploit all available parallelism on the target architecture. Several successive code transformation have been applied: we have carefully controlled data allocation in main memory (including careful consideration of NUMA issues), we have split the computational task in many threads trying to maximize concurrent execution of threads, we have organized the computation to exploit SIMD features of the processor, and cache data reuse to improve memory bandwidth.

In summary, we can say that rather complex applications can sustain high performance if carefully implemented; on the other hand, no single optimization step is able to exploit a major fraction of the theoretically possible performance of the system; rather, many steps have to be implemented, and a substantial restructuring of the code is needed to carefully match the hardware structure of the processor.

Obviously we hope that automatic optimization tools will be able to perform this job in the

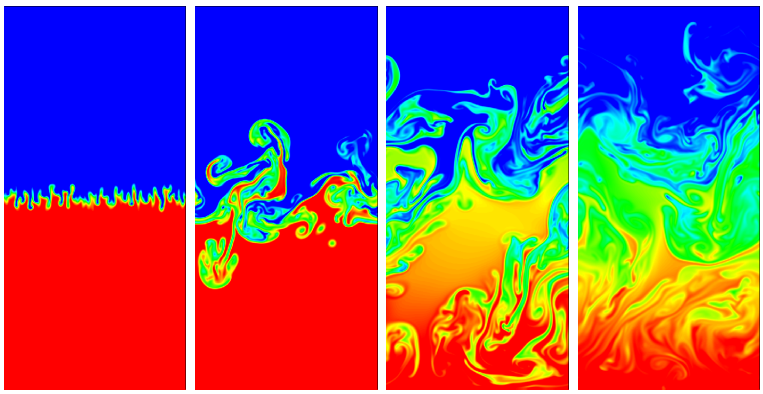

Figure 4. Temperature maps of a simulation of a Rayleigh-Taylor cell on a sample lattice size of 512×1024 points at several stages of its time evolution.

(not too far) future; we think that this is possible in principle, but it is hard to reach this goal starting from an implementation written in "low-level" languages such as C, where parallelism is not clearly expressed; a more high level description of the algorithm is probably necessary to help tools identify the available parallelism and adopt strategies to expose it.

Acknowledgments
This work has been done in the framework of the COKA and SUMA projects of INFN. We would like to thank INFN-CNAF and CINECA for providing access to their Sandy Bridge prototypes.

References
[1] Gustafson J L Reevaluating Amdahl's Law 2008 *Comm. of the ACM* **31** 532-3
[2] Biferale L et al 2011 Lattice Boltzmann Method Simulations on Massively Parallel Multi-core Architectures *Proc. of High Performance Computing Symposium* Boston MA
[3] Biferale L et al 2011 Optimization of Multi-Phase Compressible Lattice Boltzmann Codes on Massively Parallel Multi-Core Systems *Procedia Computer Science* **4** 994:1003
[4] Succi S 2001 *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond* (Oxford:University Press)
[5] Sbragaglia M et al 2009 Lattice Boltzmann method with self-consistent thermo-hydrodynamic equilibria *J. Fluid Mech.* **628** 299
[6] Scagliarini A et al 2010 Lattice Boltzmann methods for thermal flows: Continuum limit and applications to compressible Rayleigh-Taylor systems *Phys. Fluids* **22** 055101
[7] Pohl T et al 2003 Optimization and Profiling of the Cache Performance of Parallel Lattice Boltzmann Codes *Parallel Processing Letters* **13** 549
[8] Wellein G et al 2006 On the Single Processor Performance of Simple Lattice Boltzmann Kernels *Computers and Fluids* **35** 910
[9] Biferale L et al 2012 A multi-GPU implementation of a D2Q37 Lattice Boltzmann Code *Lecture Notes in Computer Science* vol 7203 (Heidelberg:Springer) pp 640-650
[10] Bertazzo A et al 2012 Implementation and Optimization of a Thermal Lattice Boltzmann Algorithm on a multi-GPU cluster *Proc. of Innovative Parallel Computing* San Jose CA
[11] Biferale L et al 2012 An Optimized D2Q37 Lattice Boltzmann Code on GP-GPUs *Computers and Fluids* special issue on computer and fluids