Computers & Fluids 80 (2013) 55–62

Contents lists available at SciVerse ScienceDirect

Computers & Fluids

journal homepage: www.elsevier.com/locate/compfluid

An optimized D2Q37 Lattice Boltzmann code on GP-GPUs

Luca Biferale [a], Filippo Mantovani [b,1], Marcello Pivanti [c,2], Fabio Pozzati [d], Mauro Sbragaglia [a], Andrea Scagliarini [e], Sebastiano Fabio Schifano [c,*], Federico Toschi [f], Raffaele Tripiccione [g]

[a] University of Tor Vergata and INFN, I-00173 Roma, Italy
[b] Deutsches Elektronen Synchrotron (DESY), D-15738 Zeuthen, Germany
[c] University of Ferrara and INFN, I-44124 Ferrara, Italy
[d] Fondazione Bruno Kessler Trento, I-38122 Trento, Italy
[e] University of Barcelona, S-08007 Barcelona, Spain
[f] Eindhoven University of Technology, Eindhoven, The Netherlands and CNR-IAC, I-00185 Roma, Italy
[g] University of Ferrara, INFN and CMCS, I-44124 Ferrara, Italy

ARTICLE INFO

Article history:
Received 22 September 2011
Received in revised form 1 June 2012
Accepted 4 June 2012
Available online 19 June 2012

Keywords:
Computational fluid-dynamics
Lattice Boltzmann methods
GP-GPU computing

ABSTRACT

We describe the implementation of a thermal compressible Lattice Boltzmann algorithm on an NVIDIA Tesla C2050 system based on the *Fermi* GP-GPU. We consider two different versions, including and not including reactive effects. We describe the overall organization of the algorithm and give details on its implementations. Efficiency ranges from 25% to 31% of the double precision peak performance of the GP-GPU. We compare our results with a different implementation of the same algorithm, developed and optimized for many-core Intel *Westmere* CPUs.

1. Introduction

The Lattice Boltzmann method (LB) is a computational approach that describes fluid dynamics by lattice discretization in both position and momentum space [1]. Key advantages are the relative ease with which complex physics can be implemented in the model, as well as good computational efficiency on massively parallel computer architectures.

Recently, the use of General Purpose Graphics Processing Units (GP-GPUs) to accelerate non-graphics computations has drawn much attention, as the computational power delivered by just one GPU is of the order of several hundreds Gigaflops, exceeding by almost one order of magnitude that of a computing system based on standard CPUs. In general terms, GP-GPUs architectures extensively exploit internal parallelism to boost their performance level. This makes it potentially rewarding to use GPU-systems for Lattice Boltzmann (LB) algorithms. Several groups have recently proposed GPU-based implementations of a variety of LB algorithms in two and three dimensions; see for example [2] for a D2Q9 and [3] for a D3Q13 model (LB algorithms are usually labeled as DnQm, where n is the number of space dimensions and m is the number of *populations*; see the following section for a definition of *population*).

In the present contribution, we focus on a substantially more complex D2Q37 LB scheme able to correctly describe a compressible thermal fluids that obeys the equation-of-state of a perfect gas. From the point of view of a computer implementation, the LB D2Q37 model that we consider has substantially more severe requirements than earlier schemes, both for the floating-point units and the memory interface of the processor. We describe an efficient implementation of this algorithm on state-of-the-art GP-GPUs; we also compare performance results on these processors and on recent many-core CPU architectures.

2. Lattice Boltzmann methods

In this section, we introduce the computational methods that we adopt, based on an advanced D2Q37 LB scheme, that correctly reproduces the equation of state of the fluid, regarded as a perfect gas ($p = \rho T$); full details of the algorithm are given in [4,5].

The lattice description of the dynamics that we want to study is given in terms of an LB discretization on a regular lattice, based on a set of lattice populations ($f_l(x,t)$); each has a given lattice velocity $\boldsymbol{c}_l$; populations evolve in (discrete) time according to the following equation:

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

* Corresponding author. Tel./fax: +39 0532974614.
E-mail address: schifano@fe.infn.it (S.F. Schifano).
[1] Now at University of Regensburg, Regensburg, Germany.
[2] Now at "La Sapienza" University, Roma, Italy.

0045-7930/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.compfluid.2012.06.003

In our case we study a 2-dimensional system ($D = 2$ in the following); the set of populations has 37 elements (hence the D2Q37 acronym), corresponding to (pseudo-) particle moves up to three lattice points away, as shown in Fig. 1. Macroscopic 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 [1].

In extreme conciseness (full details can be found in [4,5]), one makes contact between this synthetic dynamics and the true dynamics of a compressible gas, starting with a kinetic and thermal description of a system of variable density, velocity and internal energy $\mathscr{K}$, subject to a local body force $\boldsymbol{g}$ (gravity); one then is able to show that, after appropriate shift and re-normalization of the velocity and temperature fields, one recovers, through a Taylor expansion in Δt, the correct 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.

3. LB implementation

The LB approach to computational fluid-dynamics offers the advantage of a huge degree of available parallelism. We discuss now how it can be easily identified and exploited by emerging HPC architectures.

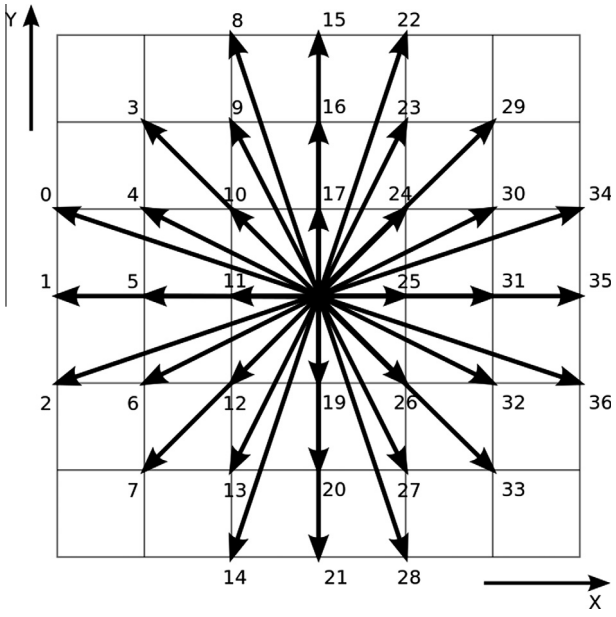

Fig. 1. Visualization of the stream phase for the D2Q37 LB scheme. Populations at distance 1, 2, 3 lattice-points and moving with velocities $\boldsymbol{c}_l$, shown by the arrows, are gathered at the lattice-point at center.

Defining $\boldsymbol{y} = \boldsymbol{x} + \boldsymbol{c}_l \Delta t$ and rewriting the main evolution equation as:

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

one easily identifies the overall structure of the computation; in order to evolve the system by Δt, one has to perform the following steps at each point $\boldsymbol{y}$ in the discrete grid:

1. gather the fields f_l corresponding to populations that drift towards $\boldsymbol{y}$ with velocity $\boldsymbol{c}_l$; these data words are stored at neighboring sites in the lattice;
2. perform all mathematical processing needed to compute (*in a completely local fashion*) the quantities appearing in Eq. (8).

Step 2 is slightly complicated if one wants to take into account reactive effects (combustion), as 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 we can parallelize according to any convenient schedule, as long as we make sure that, for all grid points, step 1 is performed before step 2.

This approach is well suited for HPC architectures based on a large number of processing nodes, each one involving a large number of computing cores, the cores themselves often containing SIMD data-paths. The challenge then rests in matching data parallelism with available computing resources.

In our implementation, at each time step, each grid-point is processed applying three main phases, stream (), bc () and collide ():

- stream () gathers for each site the populations according to the scheme of Fig. 1. 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 ()). This step has rather irregular memory access patterns.
- bc () adjusts the values of the cells at the top and bottom edges of the lattice to enforce appropriate boundary conditions (e.g., one might want a constant given temperature and non-slip boundary conditions for 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 we do for the top and bottom edges.
- collide () performs all the mathematical steps associated to Eq. (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 most floating point intensive kernel of the code; it uses only the population members of the site on which it operates, making the processing of different sites fully uncorrelated.

Our first implementation of the code (we call it V1) keeps the computation of stream and collide as separate kernels. This scheme is used if reactive dynamics (combustion) are enabled, because in this case the divergence of the velocity field has to be explicitly computed; in order to do so, we need a further step in which data is gathered from memory; this step must be performed at some

intermediate point in the (otherwise local) mathematical processing of collide.

So far, our code moves data from/to memory two times, first during stream and then during collide. If reactive dynamics are not enabled, we can structure the code in a slightly different way, improving performance: we can merge the stream and collide phases in just one computational step applied in sequence to all cells of the grid (this optimization technique was first suggested in [6]). Note that we have to take into account that the computation of the boundary conditions (execution of the bc kernel) has to be done after stream but before collide. This requires a careful intertwined scheduling, described in detail later.

In the following sections we describe the implementation of our code. First we review the implementation on a state-of-the-art HPC system based on multi-core CPUs. This implementation is used as a reference for comparing the performance of our GPU code.

4. LB CPU implementation

In this section we summarize the implementation of the LB algorithm for multi-core CPUs described in details in [8,9]. We developed our code on a commodity system based on dual Intel six-core X5680 CPUs (code-name Westmere). Each CPU operates at 3.3 GHz, and has 12 MB of L3-cache. The system has 6 + 6 GB of RAM. The theoretical peak double-precision performance of this system is 160 Gflops. We consider this system a state-of-the-art platform not relying on GP-GPUs, and we use the performance of this code as a reference-point for further implementations on massively parallel and multi-core architectures, relying both on CPUs and GPUs.

Each lattice cell is represented in memory by a structure with several members as shown in Fig. 2. The array of 37 double-precision floating point values contains the *populations* of our D2Q37 LB code. Other members store macroscopic variables, such as velocity and temperature (and the divergence of velocity, when needed). We keep in memory two copies of the same lattice. Each step reads inputs from one copy and writes results to the other. The three computational phases are called inside a loop over time steps, and operate on the array of the lattice-cell structure, as shown in Fig. 2.

To reach high computational performance, parallelism has to be exploited at all possible levels: cores and instructions. To do that we have organized our computation as follows:

1. the lattice allocated into the memory of a single node is split over the cores, each one working on a portion of it, called *sub-lattice*,
2. the cores of each node process their sub-lattice in parallel, each one operating on a different part of it,
3. within each core, two sites of the sub-lattice are handled in parallel, exploiting data-parallelism through vector SSE instructions.

```
typedef struct {
  double p[37];    // population array
  double u;        // velocity_x
  double v;        // velocity_y
  double rho;      // density
  double temp;     // temperature
} pop_type;

for ( step = 0; step < MAXSTEP; step++ ) {
  stream();
  bc();
  collide();
}
```

Fig. 2. Key data-structures and overall organization of the code. The pop_type structure is used for the variables describing each lattice-cell. The p[] member is the array of populations (see the text for details). Other members are used as temporary storage of macroscopic variables.

Since each node operates as a *Symmetric Multiprocessor* (SMP) the two CPUs can be programmed as a single CPU with 12 cores; they are controlled by a single instance of the *Linux* operating system. Parallelism at core level is handled through the standard pthread library, the approach closer to the *Linux* kernel; this avoids overheads associated to, e.g. openMP or openMPI. To get the best performance from the node, we also run one thread per core, avoiding overheads due to the scheduling of threads among the cores. Fig. 3 details the code executed on the node; it is organized as follows:

1. Two threads (threads 0 and 1) copy the left and right borders onto the appropriate frames. In a single node implementation this step (that we call comm ()) is simply performed by memory-copies; after receiving data from the neighbors nodes, both threads apply stream () to the three lattice columns close to the frame that they have just copied.
2. While step 1 is executed, the remaining threads compute in parallel the stream () phase on all columns far enough from their *Y*-edges.
3. As soon as 1 and 2 complete, two threads compute the bc () phase on the three rows of cells close to the top and the bottom of the lattice.
4. After step 3, if reactive dynamics is enabled, we have to compute the divergence of the velocity field at all lattice points; this requires an additional step, similar to stream () and referred to as comm2 (), to propagate velocities to neighbor nodes. After this is done, all threads start computing collide (), each on a different portion of the sub-lattice.

All the above steps are synchronized by the use of pthread barriers.

A further level of parallelism entails processing two sites at a time. We pair the data of two cells at distance $NY/2$ by using vector variables, see Fig. 4. The gcc compiler, through the custom vectorsize extension, allows the definition of a new variable type where two primitive types can be packed together; we use this feature to define vectors of two doubles. Operations on vector variables are then translated by the compiler, generating code that uses streaming SSE instructions.

We also apply a simple optimization step that significantly improves memory and cache efficiency for stream. Remember that stream () performs scattered accesses in local memory to gather populations with distances of 1, 2 and 3 in the grid; one expects that performance may depend on details of cache and memory-allocation policies, as analyzed in details in [7]. To improve cache performance, we have chosen an appropriate relabeling of the population set; this forces a memory allocation order of the population elements that maximizes the number of population elements brought to cache when handling point (x, y) in the grid that remain available in the cache when considering points $(x, y+1)$ and $(x, y+2)$. We also manage allocation of memory using the Linux NUMA library: we explicitly map lattice points onto the memory bank close to the CPU of the thread that accesses them. This avoids conflicts due to a thread running on a CPU and accessing memory physically attached to the other CPU. For more details on this points, see [8,9].

5. LB GPU implementation

In this section we move onto the implementation of our LB algorithm on an NVIDIA Tesla C2050 system, powered by the *Fermi* General Purpose Graphics Processing Unit (GP-GPU).

The Fermi chip has 14 cores that operate at 1.15 GHz. Each core is named *Streaming Multiprocessor (SM)* in NVIDIA jargon; it supports the execution of up to 32 CUDA-threads, for a total of 448

```
for ( step = 0; step < MAXSTEP; step++ ) {

  if ( tid == 0 || tid == 1 ) {
    comm();    // exchange borders
    stream(); // apply stream to left- and right-border
  } else {
    stream(); // apply stream to the inner part
  }

  pthread_barrier_wait(...);

  if ( tid == 0 )
    bc(); // apply bc() to the three upper row-cells

  if ( tid == 1 )
    bc(); // apply bc() to the three lower row-cells

#ifdef DIVERGENCE
  pthread_barrier_wait(...);
  comm2();
#endif

  pthread_barrier_wait(...);
  collide();   // compute collide()
  pthread_barrier_wait(..);
}
```

Fig. 3. Code executed by each core of the node. The various phases of the code are executed by a variable number of threads, and are synchronized through barriers.

CUDA-threads per device. Unlike CPU-threads, CUDA-threads are extremely lightweight, meaning that context switch between two threads is not a costly operation; typically one thread processes one element of the data-set of the program. At each clock cycle the SM schedules and executes a *warp*, a bunch of 32 instructions processed in SIMD fashion. In case all the operations of the warp are fused-multiply–add (FMA), the device delivers ≈1 TFlops of peak-performance. For double-precision operands, an FMA takes two cycles, and peak-performance is ≈500 GFlops. For more details see [10].

Our code has been written using CUDA [11], the NVIDIA programming language for GP-GPUs. CUDA supports data-parallelism through the *Single Instruction Multiple Thread* (SIMT) programming paradigm. A CUDA program consists of one or more functions that are executed on either the host, a standard CPU, or on a GPU. The functions that exhibits no (or limited) parallelism stay on the host, while those exhibiting a large degree of parallelism run on the GPU. A CUDA program is a relatively standard C program including keyword extensions for executing data-parallel functions, called *kernels*, on the GPU device. Kernel functions typically generate a large number of threads, i.e. a large number of independent operations, to exploit data parallelism. For example, we generate a thread for each lattice-site, and process them in parallel. All the threads generated by a kernel are collectively called a *grid*. The grid can be configured as a 1- or 2-dimensional array of blocks, each one executed independently on each SM processor. Each block can be configured as a 1-, 2- or 3-dimensional array of threads; they execute on the same SM in lock-step, and cooperate by sharing data through a fast shared memory. In our program we often use the possibility to define dynamically the configuration of the CUDA grid. When all threads of a kernel complete their execution, the corresponding grid terminates and execution continues on the hosts.

Codes running on GP-GPUs approach the peak performance of the system if:

- the execution of the program generates many blocks; this allows to keep active as long as possible each SM processor, and hide external memory access latencies,
- a large number of *warps* – a group of 32 instructions in the case of Fermi – per SM processor can be activated to exploit *memory coalescing*; this improves the bandwidth between the SM processor and external memory.

Our LB D2Q37 algorithm requires far more floating-point operations per lattice point than simpler models, so it is a good candidate for GPU processing; on the other hand, its populations move up to three grid points for each time step, which implies rather sparse memory access patterns, that may make memory coalescing not so efficient as in simpler models.

As already remarked, each lattice point is described by a set of 37 double-precision floating point values representing the populations of the D2Q37 model. On GP-GPU, this data structure becomes a structure of arrays, as shown in Fig. 5. Each CUDA thread executes the same code and processes one lattice site. Threads processing adjacent lattice-sites address the same population-array at the

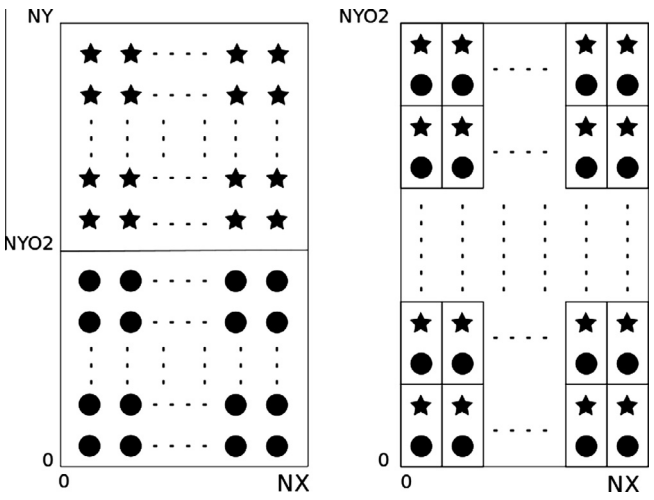

Fig. 4. Cells at distance $NY/2$ are paired together, in order to exploit the streaming vector instructions available in the processors.

same time. This arrangement has been chosen to help the hardware *coalesce* memory accesses by the threads into one single memory transaction, so efficient access to external memory is possible. Note that, in the CPU-optimized version described in the previous section, we used a different strategy for memory allocation (the data structure was an array of structure), that better suited the cache structure of those machines.

As we did in the case of the CPU implementation, for easy programming, we duplicate each population array on the GPU memory. Each step of the algorithm reads values from one copy and write results to the other.

At each iteration of the loop over time steps, the host executes the main four steps described in Section 3, as shown in Fig. 6. First it executes a device-to-device copy operation to exchange data corresponding to the border frames, and then calls the three kernels, stream, bc and collide, passing to them pointers to the input and output lattices. Each phase of the algorithm is implemented by a separate CUDA kernel:

- The comm phase is a copy operation. This is performed by a CUDA memory-copy function invoked by the host; it moves data from locations corresponding to the right border to locations corresponding to the left border and vice versa. We use the device-to-device flag which allows to make an internal copy of data without involving the host, substantially increasing performance.
- In the computation of the move and collide kernels for a lattice size of NX × NY, the layout of each block is (1 × NTHREAD × 1) and the grid of blocks is defined as (NY/N_THREAD, NX) (N_THREAD is the number of thread per block), see Fig. 7 left-side. In our case we adopt N_THREAD = 128; this allows to fit the number of registers required by threads on a single SM core, and allows to overlap memory accesses and computation.
- during the computation of the bc kernel the layout of each block is changed into (NX × 1 × 1) and the grid of blocks is (1 × NY), see Fig. 7 right-side. The bc kernel runs only on the threads corresponding to lattice-sites with coordinate y = 0,1,2 and y = NY-1,NY-2,NY-3.

As we explained earlier for the CPU-optimized version, even in this case, if reactive dynamics are not enabled, we can structure the code in such a way that the computation of stream and collide phases are merged in one computational step. In this case the computation of the bc kernel has to be done after stream but before collide. To fit this constraint, the program organization has been structured in a different way. As in the previous case the host performs a loop over time steps, and, at each iterations, it does the following:

step 1: exchanges the Y frames through a cudaMemCopy operation;
step 2: executes the stream kernel for the three topmost and lowermost rows of the grid;
step 3: adjusts the boundary conditions for the cells located at the top and bottom rows of the grid. It then runs collide for the same cells;
step 4: executes a kernel that jointly computes stream and collide for the bulk cells of the lattice.

This new schedule corresponds to another version of the code that we call V2.

```
typedef struct {
  double p1[NSITES]; // population 1 array
  double p2[NSITES]; // population 2 array
  ...
  double p37[NSITES];// population 37 array
} pop_type;
```

Fig. 5. The main data structure of the GPU code for lattice variables.

6. Performance results

In this section we present performance results and compare our GPU-code, running on a Tesla C2050 board, with that of the CPU-code running on the commodity system described in Section 4.

We start with a simple estimate of the performance that we may expect; this may help us assess the quality of our results. All together, the workload W associated to mathematical processing for each lattice point amounts to $W \approx 7800$ double-precision floating-point operations. Processing one site requires 37 data elements, and produces 37 updated population variables. To very first approximation, we can estimate the computing time T as:

$$T \geqslant \max(W/F, D/B) \tag{9}$$

where F is the peak performance of the processor, $D = 37 \times 2 \times 8 = 592$ Bytes is the amount of data exchanged with memory, and B is the peak memory bandwidth. Remember that $F \approx 500$ Gflops and $B \approx 144$ GBytes/s for the GPU that we use, while the corresponding values for the Intel Westmere processor are $F \approx 160$ Gflops and $B \approx 64$ GBytes/s; we then obtain:

$$T_{CPU} \geqslant \max\left(\frac{7800}{160}, \frac{592}{64}\right) ns = \max(48.8, 9.25)\, ns. \tag{10}$$

$$T_{GPU} \geqslant \max\left(\frac{7800}{500}, \frac{592}{144}\right) ns = \max(15.6, 4.1)\, ns. \tag{11}$$

In other words, if peak values for performance and bandwidth apply, our application is strongly compute bound (as opposed to bandwidth bound) for processors, so one can hope to reach high efficiency. This estimate is a crude upper bound, implying that still we should be able to reach high efficiency for this code if:

- all vector data paths in all SMs are used sustainedly,
- data is not moved from memory to processor and vice versa more often than needed,
- the complex addressing pattern associated to sparse memory gather operations generated by stream, does not degrade memory bandwidth too much (this is probably the most questionable assumption).

Table 1 shows the performance results for a lattice of 252×16,000 cells, the largest lattice we can allocate on our GPU. We have used NVIDIA CUDA version 3.2, and the GCC compiler version 4.1.2. In this case, stream () and collide () kernels are configured as a grid of 125 blocks of 128 threads each. The bc () kernel is a grid of 16,000 blocks of 252 threads each, but, as already underlined, only the three topmost and lowermost blocks perform the computation. Some remarks are in order:

- Table 1 refers to the best choice for some available optimization choices: in the GPU code we can configure the 64 KBytes on-chip memory either as 48 KBytes of shared-memory and 16 KBytes of L1-cache or as 16 KBytes of shared-memory and 48 KBytes of L1-cache; the latter option is approximately 15% better than the former;
- similarly for CPUs, one can let the compiler auto-vectorize the code, or explicitly pair data items and use intrinsic function to introduce in the code SSE assembly instructions, as explained in Section 4. In the latter case the GCC and ICC compiler give

```
foreach ( step=0; step < MAX_STEP; step++ ) {
  comm ( );                                  // exchange Y borders
  move    <<< grid, threads >>> ( );         // run stream
  bc      <<< grid, threads >>> ( );         // run bc
  collide <<< grid, threads >>> ( );         // run collide
}
```

Fig. 6. Main loop executed by the host. At each iteration it calls four kernel functions which run on the GPU. Parameters (grid and threads) specify the configuration of the CUDA grid which is different for each kernel.

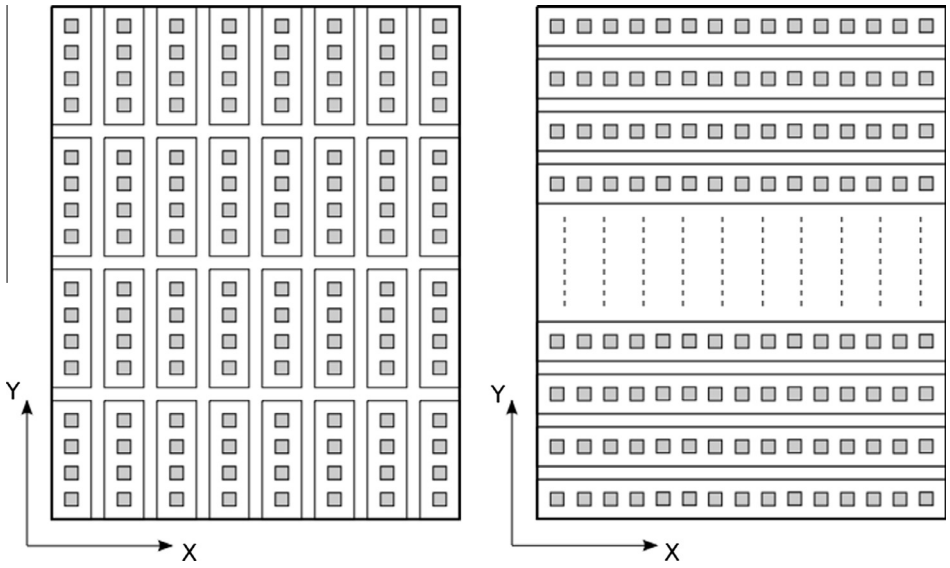

Fig. 7. Left: grid configuration used for the stream and collide kernels on a physical lattice of 8×32 points; the grid is configured as 8×4 blocks, and each block is a grid of $1 \times 4 \times 1$ threads (N_THREAD is 4). Right: grid configuration for the same physical lattice for the bc kernel.

approximately (within 2%) the same performance results. In the former case ICC is 1.5X better than GCC, but performance (for ICC) is still only 75% of explicit vectorization;

- the single GPU code performs roughly 2X better than the optimized code on multi-core CPU, delivering $\approx 25-30\%$ of peak performance;
- even if efficiency on GPU, as fraction of the peak, is lower than multi-core CPUs, sustained performance is still remarkably high for a production ready code;
- sustained performance is satisfactory from the point of view of physics application; however the value that we reach is significantly lower than one would expect from the estimates of Eq. (11); this is partly due to the fact that the mathematical structure of the algorithm makes it very difficult to fuse multiply and adds into FMAs assembly instructions. In fact, looking into the PTX assembly file produced by the CUDA compiler, we find ≈ 2300 FMA instructions, and ≈ 3000 *non-fused* floating-point instructions. Moreover, the intrinsic dependencies of the code, and memory accesses to load constants are still causing pipeline stalls. Increasing kernel occupancy, i.e. the number of active warps, which is ≈ 0.33 in the current version, to further hide memory access latency would require more registers; this causes register-spilling overheads and outweighs the potential performance gain;
- fine tuning the CPU-program has required accurate programming efforts. This is mainly due to the fact that on standard CPUs we tend to use a C-like programming approach which hardly allows to exploit parallelism. On GP-GPU efficient programming is easier, as coders are forced to (re-) write programs using the CUDA model that naturally allows to exploit data-parallelism.

7. Preliminary physics results

Our LB code optimized for multi-core CPUs has been extensively used to study several features of the Rayleigh–Taylor instability in 2 dimension; results are described in [12]. The GPU-optimized

Table 1
Performance comparison for the GPU and CPU codes, for versions V1 and V2 (defined in the text). Tests have been performed on a grid of $252 \times 16,000$ lattice-points. We show the performance in GFlops and as a fraction of peak (R_{max}); we also present two performance metrics relevant to the user, that is T/site (the execution time spent on each lattice site) and its inverse (MLups, e.g. the number of lattice sites updated per second).

	GPU code V1	CPU code V1
Comm	0.20 ms	10.00 ms
Stream	47.85 ms	140.00 ms
bc	0.60 ms	0.20 ms
Collide	194.69 ms	360.00 ms
GFLOps	129.23	60.17
R_{max}	25%	38%
Time/site	0.06 μs	0.13 μs
MLUps	16.56	7.71
	GPU code V2	CPU code V2
STEP 1	0.19 ms	7.00 ms
STEP 2	1.18 ms	0.64 ms
STEP 3	0.99 ms	0.62 ms
STEP 4	193.45 ms	410.00 ms
GFLOps	160.59	72.41
R_{max}	31%	45%
Time/site	0.04 μs	0.11 μs
MLUps	20.59	9.28

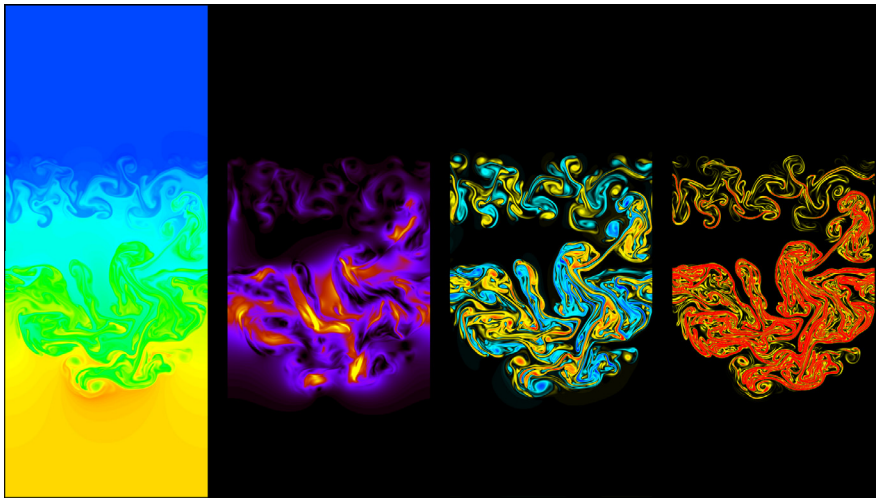

Fig. 8. Snapshot of a Rayleigh–Taylor (RT) system at a later stage of the evolution of the RT instability. From left to right, color-coded maps of the temperature, kinetic energy, vorticity, and temperature gradient. The dynamics are visibly different in the regions close to the two different initial temperature drops. A movie of this simulation is available at [13].

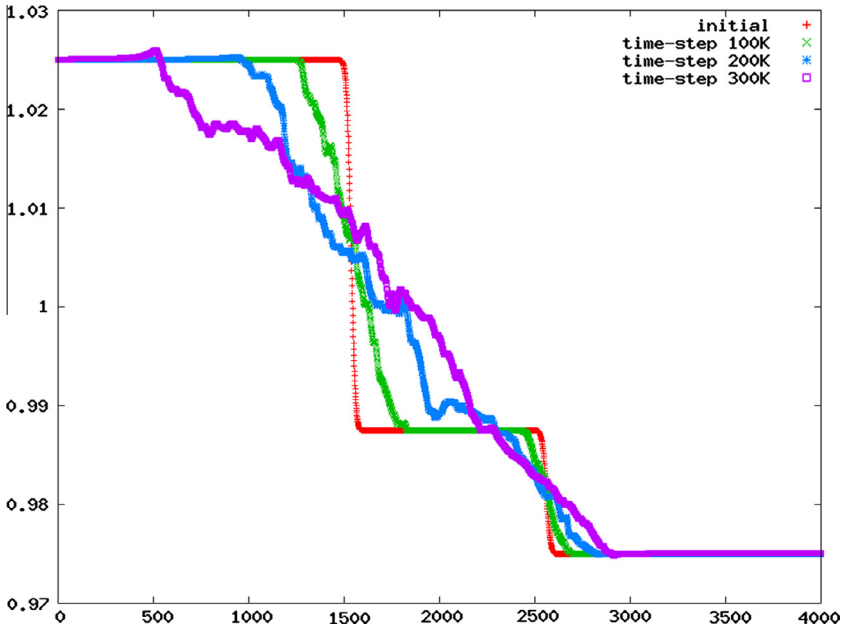

Fig. 9. Average vertical profile of the temperature for a double step Rayleigh–Taylor cell at various stages of the time evolution. The horizontal axis is the y coordinate and the vertical axis is the average temperature $T(y)$.

implementation described in this paper has been used in summer 2011 for a large simulation campaign of a double Rayleigh–Taylor instability (a system initially configured as three regions at different uniform temperatures, separated by two lines where sharp temperature changes occur). We have studied both the symmetric case (the temperature drop ΔT is the same at both interfaces, and the asymmetric case (different values for ΔT at the two interfaces)). We have considered a large lattice of 1600×4096 grid points; we have followed the time evolution of the system for 600 K time steps and we have accumulated statistics performing 15 independent runs. All runs have been performed on the *Judge* system at the Jülich Supercomputer Center.

Physics analysis has just started, and results will be published elsewhere. Here we only show (Fig. 8) a snapshot of a typical configuration, well ahead in the time evolution, when the plumes and puffs arising close to the asymmetric interfaces start to interact among one another. We also show (Fig. 9) the mean temperature profile along the vertical direction, at different points in the time evolution.

8. Conclusions and outlook

In this paper we have described in details an implementation of a complex LB algorithm for compressible fluids, optimized for GP-GPU architectures. Our implementation tries to use all parallelization opportunities made available by the algorithmic structure of the computation, and tailors the structure of the code to the architectural organization of the target processor.

The sustained peak performance is large ($\approx$130–160 Gflops), even if this is only approximately one third of peak. By way of comparison, multi-core CPUs allow to reach higher relative performance, even if absolute peak performance is lower by approximately a factor 2. Reasons for this behavior have been discussed above.

Available performance is large enough to simulate large physical systems in 2D within acceptable wall-clock times, but a study of very large lattices has still to wait for an efficient multi-GPU parallel implementation; performance in this case can be badly affected by the overhead required to move data between GPUs and CPUs, and among GPUs; careful overlap of communication and computation is necessary to allow the code to scale over tens of GPUs; work is in progress in this direction. At present a 3D version of a code that describes physics with comparable accuracy is still in the early development phase [15], so a strong optimization effort is premature. This new code, when available for physics, will require 105 populations, so its computing requirements in terms of storage, memory accesses and floating-point capabilities will be much more severe. The experience that we are doing now on a parallel implementation of this 2D code, will be very valuable at that point.

Acknowledgments

We would like to thank the Jülich Supercomputing Center (JSC) for providing access to the *Judge* system [14]. The support of Wilhelm Homberg and Jochen Kreutz is gratefully acknowledged.

References

[1] Succi S. The lattice Boltzmann equation for fluid dynamics and beyond. Oxford University Press; 2001.
[2] Tolke J. Implementation of a lattice Boltzmann kernel using the compute unified device architecture developed by nVIDIA. Comput Visual Sci 2008.
[3] Tolke J, Krafczyk M. TeraFLOP computing on a desktop PC with GPUs for 3D CFD. J Comput Fluid Dynam 2008;22(7):443–56.
[4] Sbragaglia M et al. Lattice Boltzmann method with self-consistent thermo-hydrodynamic equilibria. J Fluid Mech 2009;628:299.
[5] Scagliarini A et al. Lattice Boltzmann methods for thermal flows: continuum limit and applications to compressible Rayleigh-Taylor systems. Phys Fluids 2010;22:055101.
[6] Pohl T et al. Optimization and profiling of the cache performance of parallel lattice boltzmann codes. Parallel Process Lett 2003;13(4):549.
[7] Wellein G, Zeiser T, Hager G, Donath S. On the single processor performance of simple Lattice Boltzmann kernels. Comput Fluids 2006;35:910.
[8] Biferale L et al. Lattice Boltzmann method simulations on massively parallel multi-core architectures. in: Watson LT, Howell G, Thacker WI, Seidel S, editors. Proceedings of the 2011 spring simulation multiconference, high performance computing symposium 2011 (HPC 2011). Vista, CA: Society for Modeling and Simulation International; 2011.
[9] Biferale L et al. Optimization of multi-phase compressible lattice boltzmann codes on massively parallel multi-core systems. In: Proceedings of the international conference on computational science, ICCS 2011. Procedia computer science, vol. 4; 2011. p. 994–1003.
[10] http://www.nvidia.com/object/fermi_architecture.html.
[11] NVIDIA CUDA Programming Guide. http://developer.download.nvidia.com
[12] see, for instance Biferale L et al. Second-order closure in stratified turbulence: simulations and modeling of bulk and entrainment regions. Phys Rev E 2011;84:016305. and references therein.
[13] http://www.youtube.com/user/filimanto#p/u/0/jITKAku-PeY.
[14] http://www2.fz-juelich.de/jsc/judge.
[15] Succi S. Private communication.