ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

MATHICSE

Mathematics Institute of Computational Science and Engineering

School of Basic Sciences - Section of Mathematics

MATHICSE Technical Report
Nr. 06.2011
July 2011

A modular lattice Boltzmann solver for GPU computing processors

Matteo Astorino, Julian Becerra Sagredo and Alfio Quarteroni

http://mathicse.epfl.ch

Phone: +41 21 69 32555

Address:
EPFL - SB - MATHICSE (Bâtiment MA)
Station 8 - CH-1015 - Lausanne - Switzerland

Fax: +41 21 69 32545

A modular lattice Boltzmann solver for GPU computing processors

M. Astorino[*] J. Becerra Sagredo[*] A. Quarteroni[*†‡]

Abstract

During the last decade, the lattice Boltzmann method (LBM) has been increasingly acknowledged as a valuable alternative to classical numerical techniques (e.g. finite elements, finite volumes, etc.) in fluid dynamics. A distinguishing feature of LBM is undoubtedly its highly parallelizable data structure. In this work we present a general parallel LBM framework for graphic processing units (GPUs). After recalling the essential programming principles of the CUDA C language for GPUs, the details of the implementation will be provided. The modular and generic framework here devised guarantees a flexible use of the code both in two- and three-dimensional problems. In addition, a careful implementation of a memory efficient formulation of the LBM algorithm has allowed to limit the high memory consumption that typically affects this computational method. Numerical examples in two and three dimensions illustrate the reliability and the performance of the code.

Keywords: *lattice Boltzmann method, GPU programming, CUDA, parallel computing, Navier-Stokes equations*

1 Introduction

The lattice Boltzmann method (LBM) is a kinetic-based approach for the numerical simulation of fluid dynamics problems [34]. In the last two decades, this method has proven to be quite efficient in the simulation of various transport phenomena (see [1] for a review) and can now be considered a valuable complementary approach to more classical and consolidated techniques (e.g. finite differences, finite elements or finite volumes) [8, 26, 33, 18, 21]. Differently from these techniques, that directly stem from a continuum mechanics formulation of flow problems, LBM adopts a bottom-up approach in which the macroscopic behavior is described by modeling the interactions of moving particles at a mesoscopic level and then retrieving the macroscopic quantities as weighted sums of the corresponding particle distribution functions. Depending on the way the particles propagate (or *stream*) and interact (or *collide*), different macroscopic behaviors can be described.

From the computational point of view, LBM is recognized to be both computational expensive and memory demanding [23]. However the explicit nature of the method and its noteworthy spatial

[*] CMCS, Chair of Modelling and Scientific Computing, MATHICSE, Mathematics Institute of Computational Science and Engineering, École Polytechnique Fédérale de Lausanne, Station 8, CH-1015 Lausanne, Switzerland

[†] MOX, Modeling and Scientific Computing, Department of Mathematics, Politecnico di Milano, Via Bonardi 9, 20133 Milano, Italy

[‡] Corresponding author: Alfio.Quarteroni@epfl.ch

locality of data access (in fact, for each lattice node only nearest neighbor information are needed) make LBM an excellent candidate for parallel implementations.

With the recent advances in parallel computing capabilities of the contemporary graphics processing units (GPUs), there has been a growing interest of computational scientists in using this hardware to accelerate computationally demanding numerical simulations. Nowadays two different programming languages for GPUs are available: the Compute Unified Device Architecture (CUDA C) developed and supported by NVIDIA [10] and the Open Computing Language (OpenCL) [28].

In the framework of LBM, efficient CUDA C implementations have been recently proposed in various works, see e.g. [16, 35] and [36, 3, 4, 27] respectively for two- and three-dimensional fluid problems. In order to fully benefit of the computational capabilities of GPU devices, most of the mentioned works describe implementations that are highly optimized for a given type of macroscopic problem and lattice structure. As a consequence, they are characterized by a limited flexibility.

In this work a modular and efficient implementation of the LBM for GPU is presented. The implementation is based on the CUDA C programming language and offers a modular structure that allows for easy modification. A new memory layout based on three Structure-of-Arrays (SoA) is used to store the unknown particle distribution functions and their access is controlled by an *ad hoc* semi-indirect addressing scheme. The resulting code is also optimized in terms of memory requirements adopting the swapping technique proposed in [22]. The proposed implementation exploits very well the characteristics of the GPU architecture still preserving a general formulation. As a matter of fact, even though the focus is given on the simulation of fluid flows, our general formulation allows the (re-)use of the same code for different kind of macroscopic problems at the expense of a slight reduction in performance.

The remainder of the paper is organized as follows. In Section 2 we briefly review the LBM and the corresponding discretization, focusing in particular on the model for fluid flows. In Section 3 we provide the basic notions of GPU architectures as well as the essential features of the CUDA C programming language. We present the implementation details and the optimization strategy of our lattice Boltzmann framework in Section 4. Test cases in two and three dimensions are reported in Section 5 in order to validate the code and discuss its performance. Finally, some concluding remarks are given in Section 6.

2 The lattice Boltzmann method

Historically the lattice Boltzmann method has been derived from the lattice gas automata (LGA) [24]. The fundamental idea behind LGA is that the physics of macroscopic phenomena can be retrieved by defining suitable interactions among microscopic particles, which essentially consist in the propagation and the collision of particles lying on a discrete regular grid (the *lattice*).

Although relying on the same idea, differently from LGA, the LBM has its roots in the kinetic theory of the Boltzmann-Maxwell equation [6]. Because of the strengthened mathematical foundations of kinetic theory, LBM remedies to some of the shortcomings of LGA, while preserving the same local nature and simplicity.

The lattice Boltzmann equation provides a minimal discrete form of the Boltzmann-Maxwell equation

$$\big(\partial_t + \mathbf{e} \cdot \nabla_{\mathbf{x}} + \mathbf{F} \cdot \nabla_{\mathbf{e}}\big) f(\mathbf{e}, \mathbf{x}, t) = J(f)(\mathbf{x}, t), \tag{1}$$

a conservation equation for the *particle distribution function* $f(\mathbf{e}, \mathbf{x}, t)$, so that $f(\mathbf{e}, \mathbf{x}, t)\mathrm{d}\mathbf{e}\mathrm{d}\mathbf{x}$ gives the total mass of particles inside the infinitesimal volume element $\mathrm{d}\mathbf{e}\mathrm{d}\mathbf{x}$ at a fixed time t, position $\mathbf{x}$

and velocity $\mathbf{e}$. The quantity $\mathbf{F}$ represents the external force while the term J, also called *collision operator*, takes into account the effects of inter-particle collisions.

The classical form of lattice Boltzmann equation (see [24]) can be retrieved from (1) upon discretization of the time and the phase space (position and velocity). Following [33], first the velocity space is approximated by projecting the distribution function f onto a Hilbert subspace H^N spanned by the first N Hermite polynomials, where the order N is dictated by the macroscopic behavior one wants to recover (e.g. $N = 2$ for the Navier-Stokes equations). Then the resulting discrete velocity equation is integrated along the characteristics. An approximation with finite differences leads to the following lattice Boltzmann equation

$$f_i(\mathbf{x} + \mathbf{e}_i \delta x, t + \delta t) - f_i(\mathbf{x}, t) = J_i(f)(\mathbf{x}, t) + \delta t F_i \qquad \forall i = 0, ..., Q, \tag{2}$$

where δx and δt are respectively the space- and time-step, while $E = \{\mathbf{e}_0, ..., \mathbf{e}_Q\}$ denotes the discrete velocity set obtained by the projection onto H^N.

Remark 1 *Note that in Equation* (2) *the evolution of the particle distributions f_i is restricted to velocities that belong to the pre-assigned discrete set E. In other words the particles move always on the lattice nodes and only along the links defined by the discrete set of velocities $\{\mathbf{e}_0, ..., \mathbf{e}_Q\}$. For this reason the set of velocities is also commonly addressed as the* lattice structure *of the model.*

On the right hand side, the quantity F_i is a discrete forcing term, approximating the external force $\mathbf{F}$. The form of F_i depends on the particular macroscopic behavior we want to simulate, or equivalently on the choice of the lattice structure. The term $J_i(f)$ defines a general collision operator and represents a discrete approximation for J. The most common choice for $J_i(f)$ is undoubtedly the well known single relaxation time Bhatnagar-Gross-Krook (BGK) model [30], where

$$J_i(f) \approx J_i^{BGK}(f) = -\frac{1}{\tau}(f_i - f_i^{eq}) \qquad \forall i = 0, ..., Q, \tag{3}$$

τ being the so-called *relaxation time* and f_i^{eq} an appropriate discrete approximation of the Maxwell-Boltzmann equilibrium distribution [33]. The BGK model has been considered throughout this work, nonetheless other collisional operators have been proposed in literature. For instance, Equation (3) can be considered a particular approximation of the quasilinear form

$$J_i(f) \approx J_i^{QL}(f) = -A_{ij}(f_j - f_j^{eq}) \qquad \forall i = 0, ..., Q, \tag{4}$$

where summation convention over repeated indices is implied [13]. In the last equation, A_{ij} is the quasilinear scatter matrix defined as

$$A_{ij} = \frac{\partial J_i(f)}{\partial f_j}|_{f^{eq}}.$$

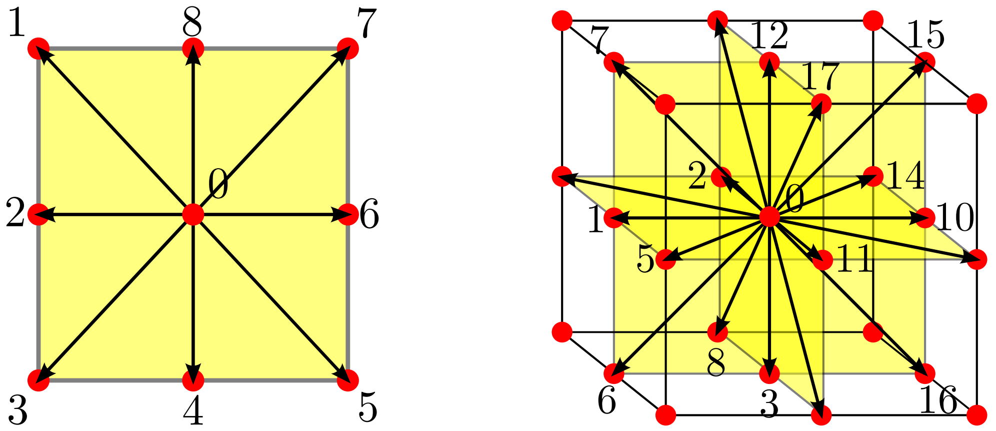
Other choices for $J_i(f)$ are the multiple relaxation times (MRT) model [11], the entropic model [2] and the regularized model [19].

Depending on the choice of the discrete velocity set E and on the collision operator $J_i(f)$ different macroscopic behaviors can be modeled [18]. Let us consider for example the BGK model for isothermal low Mach number fluid flows in two and three dimensions. In these cases the equilibrium distribution function in $J^{BGK}(f)$ and the forcing term F_i take respectively the forms

$$\begin{aligned} f_i^{eq} &= \rho w_i \left(1 + \frac{1}{c_s^2}\mathbf{e}_i \cdot \mathbf{u} + \frac{1}{2c_s^4}(\mathbf{e}_i\mathbf{e}_i - c_s^2 I) : \mathbf{u}\mathbf{u}\right) \qquad \forall i = 0, ..., Q, & (5) \\ F_i &= \left(1 - \frac{1}{2\tau}\right) w_i \left(\frac{\mathbf{e}_i - \mathbf{u}}{c_s^2} + \frac{\mathbf{e}_i \cdot \mathbf{u}}{c_s^4}\mathbf{e}_i\right) \cdot \mathbf{F} \qquad \forall i = 0, ..., Q, & (6) \end{aligned}$$

where ρ and $\mathbf{u}$ are the density and the velocity of the fluid, respectively. The *speed of sound* c_s is defined as $\beta\frac{\delta x}{\delta t}$ and represents the sound-propagation velocity of the model. The weights w_i and β are suitable constants that depend on the choice of the discrete velocity set E.

In two dimensions, the nine-velocity square lattice structure –also referred as D2Q9– is undoubtedly the most popular choice for the discrete velocity set E, Figure 1.a. In three dimensions, various choices of the discrete velocity set are available, among them we recall for instance the 15 velocities lattice structure (D3Q15), the 19 velocities lattice structure (D3Q19) (also reported in Figure 1.b) and the 27 velocities lattice structure (D3Q27). We refer to Appendix A for more details on the mentioned lattice structures as well as on the associated coefficients (the weights w_i and the speed of sound c_s).



(a) D2Q9

(b) D3Q19

Figure 1: Examples of lattice structures. Dots represent the lattice nodes $\mathbf{x}_i, \forall i = 0, ..., Q$ in the unit cell. The arrows identify the links $\mathbf{e}_i, \forall i = 0, ..., Q$ in the lattice structure, their lengths are given by $\|\mathbf{x}_0 - \mathbf{x}_i\|, \forall i = 0, ..., Q$.

In the limit of low Mach numbers ($\mathbf{u}/c_s$ small enough), assuming small space and time discretizations, the numerical model defined by Equations (2), (3), (5), (6) recovers asymptotically the dimensionless Navier-Stokes equations with dynamic shear viscosity[1]

$$\nu_d = c_s^2 \left(\tau - \frac{1}{2}\right) \delta t. \tag{7}$$

Note finally that the macroscopic variables, such as density and velocity, can be locally recovered as moments of the distribution functions:

$$\rho(\mathbf{x}, t) = \sum_{i=0}^{Q} f_i(\mathbf{x}, t), \tag{8}$$

$$\mathbf{u}(\mathbf{x}, t) = \frac{1}{\rho} \left(\sum_{i=0}^{Q} \mathbf{e}_i f_i(\mathbf{x}, t) + \frac{\delta t}{2} \mathbf{F} \right), \tag{9}$$

and that the fluid pressure scales linearly with the density by the ideal gas law $p = c_s^2 \rho + p_0$, p_0 being the reference pressure. A comprehensive presentation of these results can be found in [9, 38, 34].

[1] The dynamic shear viscosity ν_d in the dimensionless Navier-Stokes equation corresponds to the quantity $1/\mathrm{Re}$, where $\mathrm{Re} = u_0 l_0/\nu$ is the Reynolds number, l_0 and u_0 being respectively the reference length and velocity and ν the dynamic shear viscosity in physical units.

From a computational point of view, one can think of Equation (2) being split into two parts:

$$\text{collision step :} \qquad \tilde{f}_i(\mathbf{x},t) = f_i(\mathbf{x},t) + J_i(f)(\mathbf{x},t) + \delta t F_i \qquad \forall i = 0,...,Q, \tag{10}$$

$$\text{streaming step :} \qquad f_i(\mathbf{x}+\mathbf{e}_i\delta x, t+\delta t) = \tilde{f}_i(\mathbf{x},t) \qquad \forall i = 0,...,Q. \tag{11}$$

The collision step is a local update of the distribution functions on each lattice node, while the streaming step moves the data across the lattice. This set of equations is eventually supplemented with appropriate initial and boundary conditions for the distribution functions, for which multiple formulations exist (see [25, 15, 20] and references therein). The treatment and implementation of complex initial and boundary conditions goes beyond the scope of this work. For this reason in the numerical experiments of Section 5, we simply initialize the computations with the equilibrium distribution f^{eq} and we adopt the full-way bounce back rule for solid fixed walls:

$$f_i(\mathbf{x}, t+\delta t) = f_{\hat{i}}(\mathbf{x}, t-\delta t), \qquad \forall i = 0,...,Q, \tag{12}$$

$f_{\hat{i}}$ denoting the distribution anti-parallel to f_i. Equilibrium distribution boundaries, zero-gradient boundaries and periodic boundaries have also been implemented. An accurate description and analysis of these and other boundary conditions for LBM can be found in [7].

3 An overview on GPUs and NVIDIA CUDA

This section briefly introduces the basics of GPU architecture and CUDA programming in order to provide the reader with the essential understanding of this particular computational framework. The focus is given on NVIDIA cards since we adopted the CUDA C programming language.

3.1 NVIDIA GPU architecture

In Figure 2 we illustrate the most important architectural elements of a GPU device. NVIDIA GPUs are made of several *Streaming Multiprocessors* (SMs), each of which consists of a certain number of *Scalar Processors* (SPs). There are also an instruction unit and three fast access memories: the *shared memory*, the *constant memory* and the *texture memory*. All these memories have streaming multiprocessor scope however the last two are read-only. The GPU device has also a fourth memory, the device (or *global*) memory, and it is common to all the SMs. Differently from the others this memory can be accessed by the CPU and it is characterized by a higher capacity but slower access.

From a practical point of view, the parallel computing capabilities of a device are often identified with the number of GPU cores, which is given by the total number of SPs on the device. The number of scalar processors per SM as well as the number of streaming multiprocessors depend on the series and model of the device.

Each multiprocessor runs in parallel with the others and it is responsible for creating, managing, and dispatching concurrent group of threads on the associated scalar processors. These groups of threads are named *blocks* and are actually executed on a multiprocessor in subgroups of 32 threads (what NVIDIA calls a *warp*). Depending on the number of cores per SM, a different number of clock cycles may be required to complete the operations on the warp. As soon as all the threads in a block terminate, a new block is launched on the vacated multiprocessors until all the blocks have been executed.

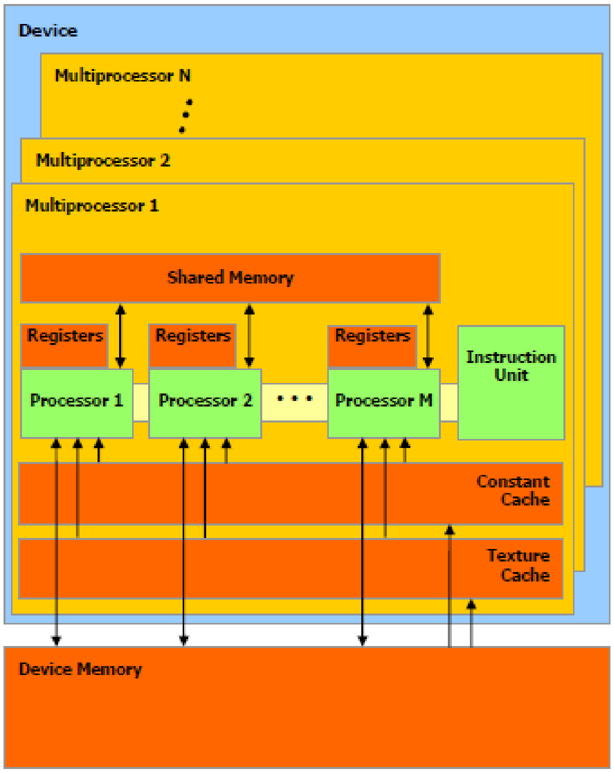


Figure 2: NVIDIA GPU architecture. Source [10].

3.2 CUDA programming

The CUDA programming model is built around the multithreaded structure presented above. A CUDA code consists in functions that can be mainly classified in two groups: functions run by the CPU –the *host*– and functions run by the GPU –the *device*. Functions running on the device are also called *kernels*. When a CUDA program on the host CPU invokes a kernel, a grid of threads is generated. This grid is made of blocks and each block is made of threads; a sketch of it is given in Figure 3.

The layout of the grid (i.e. the number of blocks and the number of threads) is specified at run time during the kernel execution. The syntax adopted in the CUDA C programming language for the kernel is the following

```
kernelName <<< gridSize , blockSize >>> (inputParameters);
```

where `gridSize` and `blockSize` prescribe respectively the dimensions of the grid and of the block. The blocks in the grid can be organized in a one or two-dimensional layout while the threads in the block may have up to three dimensions. Considering for example the grid in Figure 3 we have six blocks organized in a two-dimensional layout and within each block a two-dimensional arrangement of twelve threads. The number of blocks per grid and of threads per block may vary according to the kernel and/or the application of interest. It is however recommended to set the number of threads per block as a multiple of the half-warp size in order to avoid scalar processors being idle [10]. The maximum number of blocks per grid is 65535 in each dimension, while the maximum number of threads vary according to the *compute capability* of the GPU device.

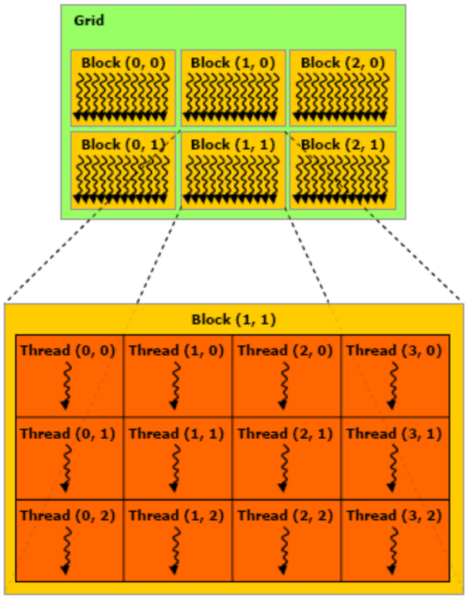
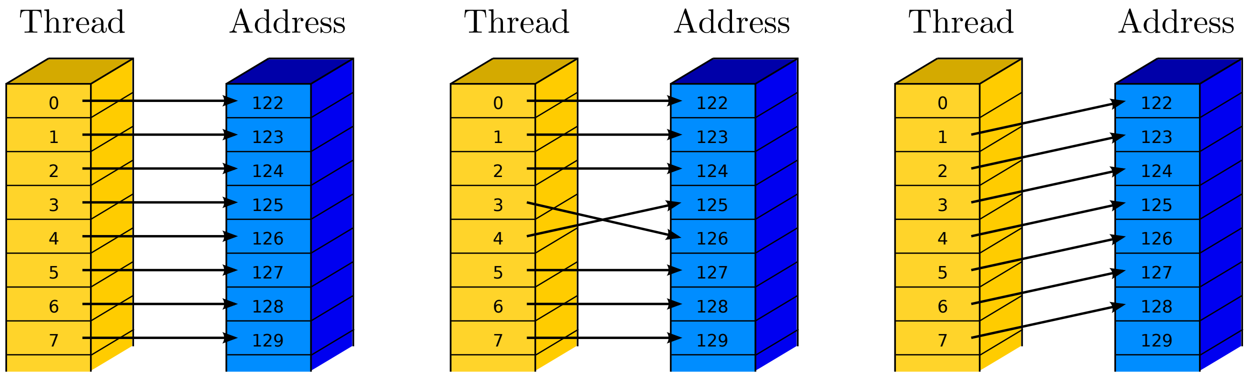


Figure 3: Threads, blocks and grids in CUDA. Source [10].

It is worth to notice that the development of a performant CUDA code depends strongly on threads communications and memory access pattern. Threads can efficiently communicate within a block by passing data through the shared memory. However inter-blocks communications are much slower, since they are based on global memory. It is therefore important to limit them whenever possible. Optimal memory access patterns are also critical, as a matter of fact memory bandwidth may degrade up to one order of magnitude if they aren't properly optimized. In order to avoid these dramatic effects on GPU performance various programming rules are recommended in [10]. Here we summarize a few of them:

- Concurrent reading/writing of threads on the same memory address has to be avoided in order to prevent code serialization.
- Threads should be organized in groups of 16 (half-warp size) so that all the threads within the group perform the very same operations. In this way, reading/writing operations on the group are done into a single memory access. When threads perform different operations within the same group, the so-called *thread divergence*, a reduction of performance is experienced.
- Data should be aligned in such a way that their reading/writing can be coalesced into a continuous aligned memory access: the `N`-th thread of a block should access the `N`-th element at address `BaseAddress+N` (see Figure 4.a). Index `N` starts from zero and it is local within a block, `BaseAddress` is the memory address of the zero-th thread. Two examples where this is not respected are given in Figure 4.b and Figure 4.c.

Remark 2 *As it will be described in the next section, a* naive *implementation of the streaming step on GPUs may lead to misaligned accesses (Figure 4.c).*



(a) Coalesced

(b) Uncoalesced: non-sequential

(c) Uncoalesced: misaligned

Figure 4: Different accesses to GPU memory. The `BaseAddress` corresponding to the zero-th thread is 122.

4 A modular and memory efficient GPU framework for LBM

In this section we describe the main features of our GPU framework for the lattice Boltzmann method. Differently from other works (e.g. [36, 16, 27, 35]) which provide a very specific and performant GPU implementation of the method, here we present a memory saving framework that offers a high level of generality with good computational performances. The code, based on a set of GPU procedures written in CUDA C language, adopts a modular structure that allows for easy modification.

The presentation of the framework is divided in three parts. First the basic LBM algorithm is illustrated. Then the data layout and the corresponding CUDA grid structure are described. Finally a description of the optimizations made on the routines follows. Comments on the algorithmic similarities and differences respect to other GPU implementations are also pointed out through the section.

LBM algorithm. In a classical LBM algorithm at least six milestones routines can be identified:

- `initProblem` for problem initialization,
- `computeMacro` for computation of macroscopic quantities from the particle distributions,
- `collideParticles` for particle collision,
- `streamParticles` for particle streaming,
- `applyBCs` for the enforcement of boundary conditions,
- `exportResults` to export results.

In our code these procedures have been implemented according to Algorithm 1.

In the routine `initProblem` the initialization procedure is implemented. As already mentioned, in this work we consider for the sake of simplicity an initialization based on the values of the equilibrium distribution f^{eq}, nonetheless other approaches exists in literature (see [5] for a review). The functions `computeMacro` and `collideParticles` are implemented within the same routine `computeMacroAndCollideParticles` since the macroscopic quantities are locally needed for the evaluation of the equilibrium distribution (5). For each node, first the density and velocity computations are carried out according to Equations (8)-(9), then the collision step (10) follows. Eventually, the procedure `streamParticles` implements step (11) and the routine `applyBCs` enforces the various boundary conditions (e.g. Equation (12)).

Algorithm 1: Basic LBM algorithm

```
geoData = input(GeometryData);
physicsData = input(PhysicsData);
initProblem(geoData, physicsData);
for time ← 0 to finalTime do
    computeMacroAndCollideParticles();
    streamParticles();
    applyBCs();
    if time = postprocessTime then
        exportResults();
    end
end
```

Each one of the routines reported in Algorithm 1 has been implemented in independent CUDA kernels and wrapped in C++ functions. Algorithm 2 provides an example of the C++ wrapper for the `streamParticlesKernel` routine. Note that in literature other implementations propose a unique GPU kernel containing all the different routines (e.g. collision, streaming and boundary treatment). This approach exploits at best the memory resources in the GPU limiting communications across global memory, nonetheless it remains constraint in terms of modularity.

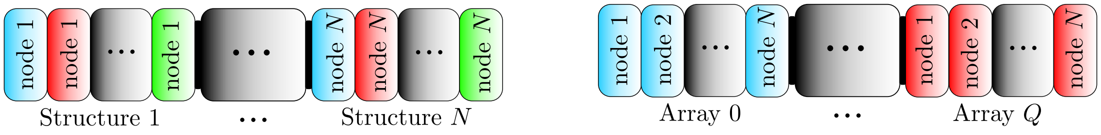
Remark 3 *As we will notice in the following, higher modularity is not the only advantage of our approach. As a matter of fact, the use of separate kernels for different routines allows us to independently select for each one the grid layout which offers the best performance.*

Algorithm 2: `streamParticles` wrapper

```
void streamParticles(){
... // code executed on the host;
streamParticlesKernel <<<gridSize, blockSize>>> (inputParameter1,
inputParameter2); // code executed on the device;
... // code executed on the host;
}
```

Data layout and CUDA grid organization. The data layout of the particle distributions and the grid structure of the kernels have substantial impact on the performance in GPU codes. Even though the two are intrinsically related, the data layout remains in general unique in the whole code, while the grid structure may be different among the kernel functions.

In literature two main data layouts can be identified [37]. The first, conventionally called "Array-of-Structures" (AoS) arrangement, stores contiguously the $Q + 1$ particle distribution functions for each lattice node (Figure 5.a). The second arrangement, named "Structure-of-Arrays" (SoA), stores contiguously the N lattice nodes for each particle distribution functions of the lattice model (Figure 5.b). As pointed out in [37] the two arrangements, AoS and SoA, are computationally optimized for two different steps of the LBM algorithm, respectively collision and streaming. On the one hand, the AoS layout allows for an optimized computation of the macroscopic quantities during the `computeMacroAndCollideParticles` step. On the other hand, the SoA layout guarantees a consecutive memory access during the distributions update in the `streamParticles` step.

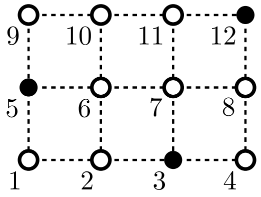


(a) Array-of-Structures

(b) Structure-of-Arrays

Figure 5: Example of Array-of-Structures and Structure-of-Arrays layout. Different colors identify different particle distribution functions (i.e. f_0, f_1,..., f_Q).

For each one of the arrangements different addressing schemes for accessing the cell's variables can be considered. Among the most common there are the direct addressing and the indirect addressing [23]. The former is easier to implement since the whole computational domain (fluid nodes and boundary nodes) and the associated particle distributions are accessed by enumeration (Figure 6.b). In this first approach an additional phase variable per node is usually needed to apply the correct dynamics to each cell. The latter, more involved, reduces memory consumption but requires an extra algorithmic stage to reconstruct the connectivity matrix [32]. In this second approach, having lost the natural ordering of the cell, one can take advantage to sort the distribution in order to optimize data flow during the computations (see Figure 6.c).



(a) Lattice

1	2	3	4	5	6	7	8	9	10	11	12
f	f	s	f	s	f	f	f	f	f	f	s

(b) Direct addressing

1	2	4	6	7	8	9	10	11	3	5	12
f	f	f	f	f	f	f	f	f	s	s	s

(c) Indirect addressing

Figure 6: Example of different addressing schemes for a lattice where fluid and boundary nodes are respectively white and black filled.

The choice of the layout and of the addressing scheme is essentially based on the specific needs of the final application/user and on the characteristics of the computational architecture employed. In the particular case of GPU architectures, the SoA layout is usually preferred since it may attain better performance allowing for coalesced access to global memory [31, 27].

In this work, in view of the development of a generic performant LBM framework, we propose an arrangement where the single SoA structure of $Q + 1$ arrays of Figure 5.b is split in three independent SoA structures. The first one is an array of N elements (the N nodes) associated to the rest distribution f_0, the other two structures are obtained by splitting the remaining distributions

f_i, $\forall i \in \{1, ..., Q\}$, in two groups based on the central symmetry of the lattice structure. Considering for example the D2Q9 lattice in Figure 7, distributions from f_1 to f_4 and from f_5 to f_8 belong to the first and second group, respectively. A similar separation in groups holds for the other lattice structures reported in Appendix A.

The advantages of the chosen layout are twofolds. Firstly, this is very general and admissible for all the lattice structures since the property of central symmetry holds for all of them. Secondly, it is particularly suited for all the computations involving a swap of distributions across opposite directions (e.g. full-way bounce back).

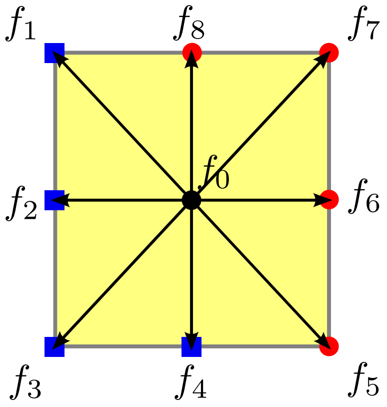


Figure 7: Example of layout for the D2Q9 lattice. Squares and circles identify the group of distributions belonging to different SoA.

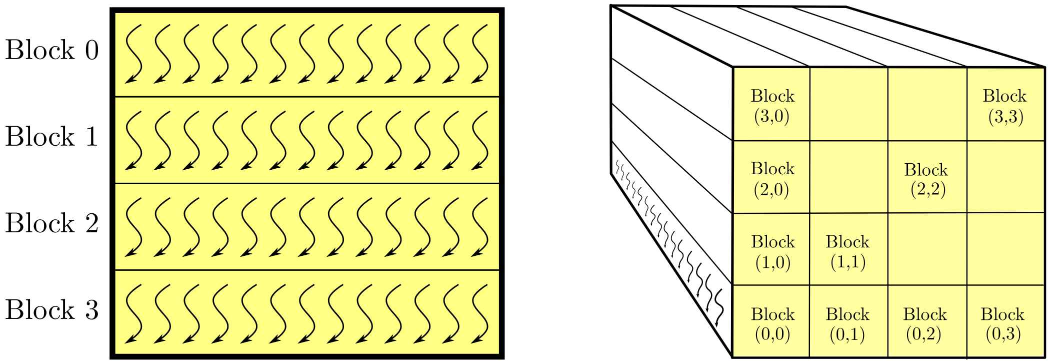
Concerning the addressing scheme, a semi-direct approach is adopted. The main idea behind this formulation is to reconstruct the connectivity matrix only for boundary nodes and to keep a direct addressing for the fluid ones. This could be efficiently implemented on GPUs making use of the stream-compaction algorithm provided by the open-source library Thrust [14].

Even though the total memory consumption is slightly increased (due to the repetition of the boundary information), the proposed approach is advantageous in terms of implementation and performance. As a matter of fact, on the one hand we exploit the simplicity of implementation of the direct addressing during the collision and streaming steps, on the other hand we improve the performance during the enforcement of boundary conditions.

Remark 4 *Further advantages of the semi-direct addressing are expected in the extension of the current LBM framework to deal with moving solids or for fluid-structure interaction. In these problems the connectivity matrix has to be reconstructed every time-step due to the movement of the solid. A fully indirect approach would require the reconstruction of the connectivity matrix for all the nodes (fluid and boundary). On the contrary, our semi-direct approach limits the computation to the solid nodes only, increasing computational efficiency.*

Remark 5 *Another semi-direct approach based on an exclusive enumeration of the fluid nodes has been proposed in [23], we refer to that work for more details.*

The choices of the distributions arrangement and of the addressing scheme have to be followed by an *ad hoc* grid organization in the GPU kernels. As already observed in Remark 3, the use of independent kernels for the different routines allows for independent grid organizations. In our implementation the grid layouts for the two- and three-dimensional problems are illustrated in Figure 8. For each kernel the number of threads in the grid coincides with the number of lattice nodes involved in the kernel computation (i.e. one thread for one lattice node). On the other hand, the number of threads per blocks and the number of blocks per grid may differ from one kernel to another.



(a) Two-dimensional grid

(b) Three-dimensional grid

Figure 8: Grid layout.

Code optimization. The optimization has been carried out aiming at improving the performance and reducing the memory consumption.

Concerning the first aspect, the focus has been given to the specific improvement of the various routines. As already observed in Section 2, the collision step is completely local and therefore perfectly suited for GPU implementation. On the other hand, as mentioned in Remark 2, a naive implementation of the streaming step may lead to uncoalesced memory accesses, affecting negatively the code performance. In order to avoid these uncoalesced operations an approach similar to the one proposed in [36] has been adopted. In the `streamParticles` kernel, the grid has been organized in such a way that one block contains all the N_x nodes along one line in the x-direction, this means that for each block the number of threads is equal to N_x. Then the grid of thread blocks is defined by the number of nodes N_y and N_z along the y- and z-direction ($N_z = 1$ in two dimensions). The considered layout limits the maximum number of admissible nodes along the x-direction to the maximum number of supported threads of the device (e.g. 512 or 1024), but it has the main advantage to allow a coalesced read and write of the distributions that propagates perpendicularly to the x-direction. The remaining distributions –the ones propagating to cells with different x-indices– are first buffered into shared memory and then transferred to the correct cell. Note that, differently from the global memory, an uncoalesced filling of the shared memory doesn't affect the performance. Eventually, surrounding the lattice with a layer of ghost cells, it has been possible to completely avoid in the same kernel the use of conditional statements, which are typically needed to select the streaming directions on boundary nodes. We remark that conditional statements may lead to thread divergence and therefore to a reduction of performance.

Remark 6 *The additional layer of ghost cells represents also a key element for the use of domain decomposition techniques on multicore environments [29]. In particular this will be extremely useful in view of the forthcoming implementation on hybrid CPU-GPU environments.*

The reduction of memory requirements is the second aspect that we have considered. As already mentioned, the lattice Boltzmann method is a highly memory demanding numerical method: a standard implementation requires to store $2qN_xN_yN_z$ scalar variables, that is all the f_i and $\tilde{f}_i$

for each lattice node. The need of this huge amount of unknowns per node may become critical in GPU devices, as a consequence in our current framework we adopted the *swapping technique* proposed in [22, 17] which allows to perform lattice Boltzmann simulations storing only $qN_xN_yN_z$ variables (i.e. only the f_i for each lattice node). We finally remark that in [3] a different memory access technique with a similar characteristics in terms of memory saving has been proposed and implemented on GPU devices.

5 Numerical experiments

In this section we present two numerical experiments respectively based on a two- and a three-dimensional lid driven cavity. The former has the aim of validating our code, the latter of evaluating its performance. Note that the validation of the code is limited to the two-dimensional case because of the generality of the implementation. As a matter of fact the routines used are the same for all the dimensions, the only difference is related to the lattice structure adopted.

All the simulations are performed on the graphic device Nvidia GeForce GTX 480 using the CUDA Toolkit 4.0 RC2. It has 480 CUDA cores (15 SMs with 32 SPs per multiprocessor) and its main features are: memory clock rate of 1848 Mhz, total amount of global memory of 1.6 GBytes and 32768 registers per block. Even though the compute capabilities of this card (= 2.0) allow for computations in single- and double-precision floating-point, here we will present only the single-precision computations, which are more significant for comparison with previous works.

5.1 Two-dimensional lid driven cavity

The two-dimensional lid driven cavity test is one of the most popular validation problems for fluid flow simulations. In this test, the fluid is contained in a unitary squared domain and it has Dirichlet boundary conditions on all sides: three stationary sides and one (at the top) moving side, characterized by a unitary tangent velocity (Figure 9).

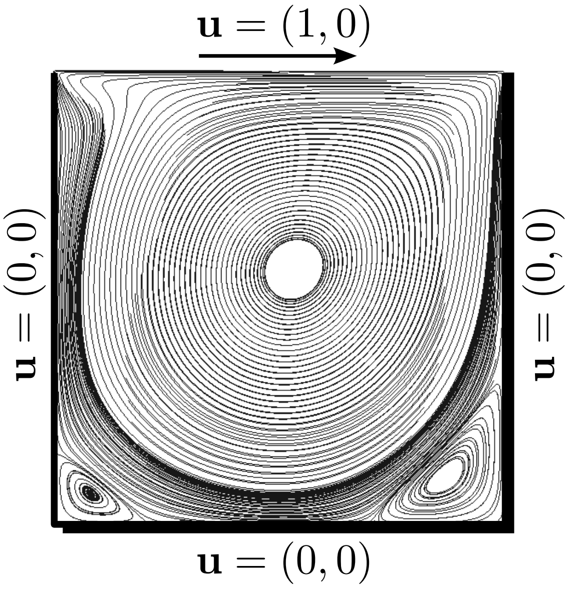
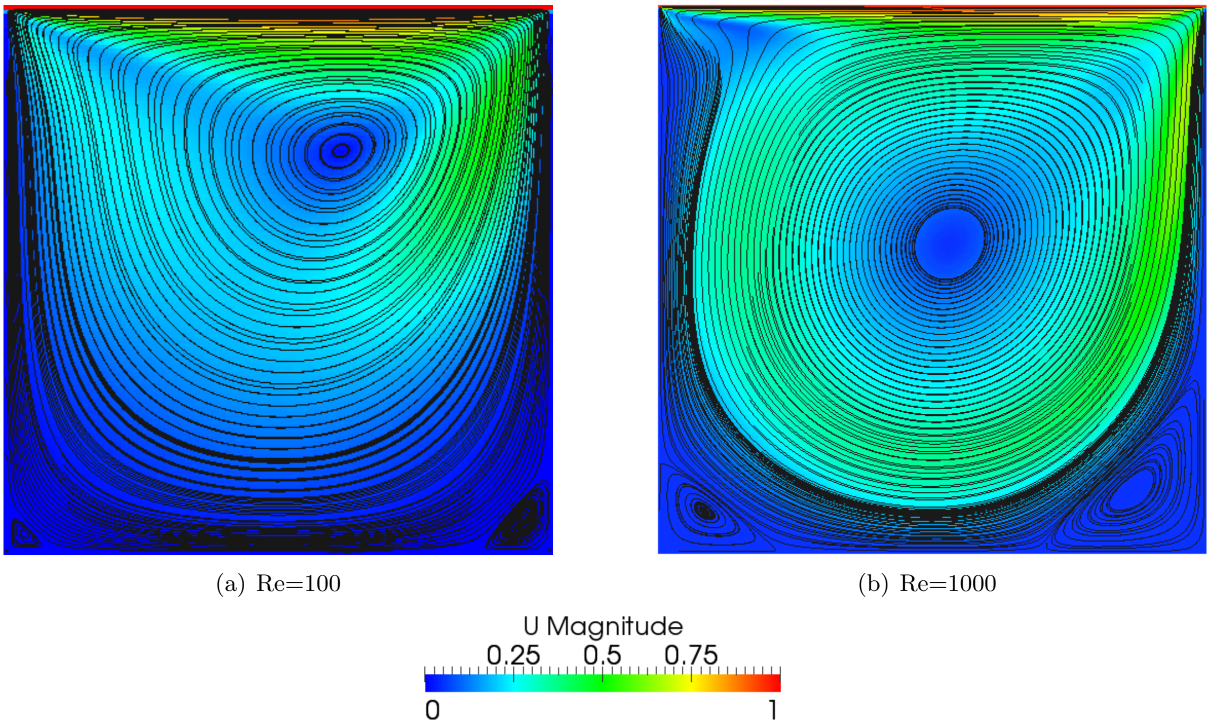


Figure 9: Two-dimensional sketch of the lid driven cavity problem.

Two different flow conditions have been simulated, the first for a Reynolds number of $Re = 100$, the other for $Re = 1000$. In both cases the D2Q9 lattice structure has been adopted. Given the Reynolds number, the relaxation time that recovers the desired macroscopic dynamics can be easily computed according to Equation (7). Choosing a time-step $\delta t = \delta x^2$ for both simulations, and a space-step $\delta x = 1/128$ in the first case (i.e. 128 nodes per unit length) and $\delta x = 1/256$ in the second case (i.e. 256 nodes per unit length), the two relaxation times are $\tau = 0.53$ and $\tau = 0.503$ respectively for $Re = 100$ and $Re = 1000$.

The results of the two simulations are reported in Figure 10. The velocity profiles in the horizontal and vertical midsections have been compared with those obtained by a finite differences discretization of the incompressible Navier-Stokes equation in its vorticity-stream function formulation [12]. From Figure 11 it can be observed that they nicely match, validating on the one hand the capability of the lattice Boltzmann method and on the other hand the correctness of our implementation.

Remark 7 *It must be observed that the number of nodes used in [12] for the test case at* $Re = 1000$ *(129 per unit length) is smaller than the one used here. The reason of our choice is that a fine space discretization usually improves both the accuracy and the stability of the method, especially when first order accurate boundary conditions (like the ones used here) are adopted for high Re simulations. For more details on the stability issues, we refer to [20, 7] where the stability of the LBM has been assessed with respect to the Reynolds number for different types of boundary conditions.*



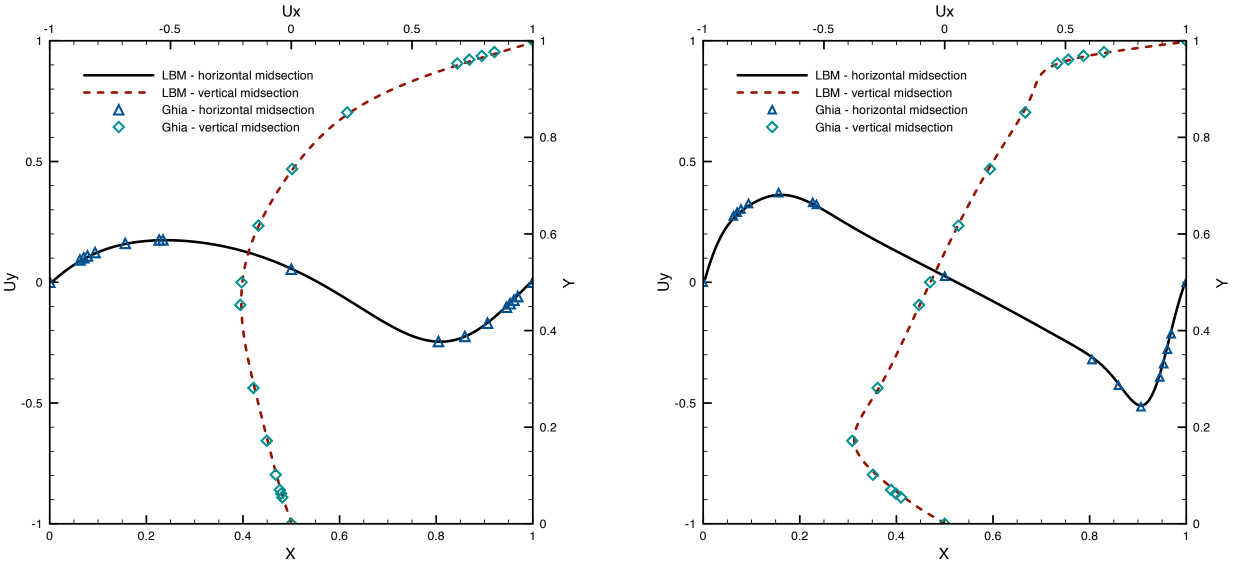
(a) Re=100

(b) Re=1000

Figure 10: Velocity magnitude and streamlines for two flow conditions.

5.2 Three-dimensional lid driven cavity

In this section the three-dimensional lid driven cavity test is adopted to investigate the performance of our LBM code. Similarly to the two-dimensional problem, the three-dimensional case is charac-



(a) Re=100

(b) Re=1000

Figure 11: Comparison between the velocity profiles given by our LBM code and those reported by Ghia et al. in [12] for the two-dimensional lid driven cavity problem.

terized by a fluid flow in a cubic domain driven by a tangential unitary velocity along one of the six boundary surfaces. Homogeneous Dirichlet conditions are adopted on all the other boundaries. Among the different lattice structures that can be adopted in three dimensions (see Appendix A), here we consider the D3Q15 and the D3Q19. An example of the computational results that can be obtained with the D3Q15 is given in Figure 12, similar results can be retrieved by using the D3Q19.

In order to evaluate the performance, two different tests have been analyzed. In the first case, we considered a unitary cubic domain discretized with different regular lattices made of a number of nodes ranging between 64^3 and 256^3. In the second case the lengths of the domain in the y- and z-directions are assumed equal and uniformly discretized with 128 nodes, while the discretization (and length) in the x-direction varies between 128 and 1024 nodes. For both tests three distinct grid layouts have been considered, based respectively on 64, 128 and 256 threads per block.

Remark 8 *Note that a different number of threads per block implies performance variations for all the kernels in the code but the one of the* `streamParticles` *routine. As a matter of fact for the latter, the number of threads per block is fixed and equal to the number of nodes in the x-direction.*

The performances, reported in Figure 13 for the first test and in Figure 14 for the second one, are expressed in Million of Lattice node Updates Per Second (MLUPS), which indicates the total number of lattice collision and streaming steps performed in one second.

As expected, in both tests the performance of the D3Q15 lattice structure are higher than those of the D3Q19, because of the smaller set of unknown needed per node. For both lattice structures the peak performance are reached in the second test, when at least 512 nodes are used in the x-direction (i.e. 512 threads per block are used in the kernel of the `streamParticle` routine). The D3Q15 and the D3Q19 structures attain respectively a maximum of 496 MLUPS and 375 MLUPS. Concerning the choice of the grid layout for the remaining kernels, it is interesting to observe that the best performances are reached for a number of threads per block larger than 128. A worthy

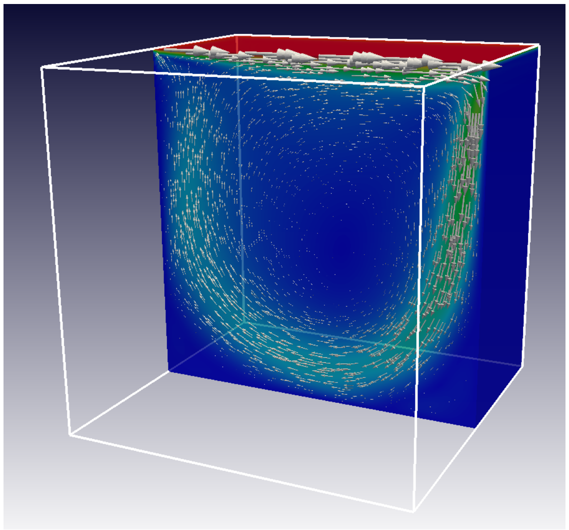
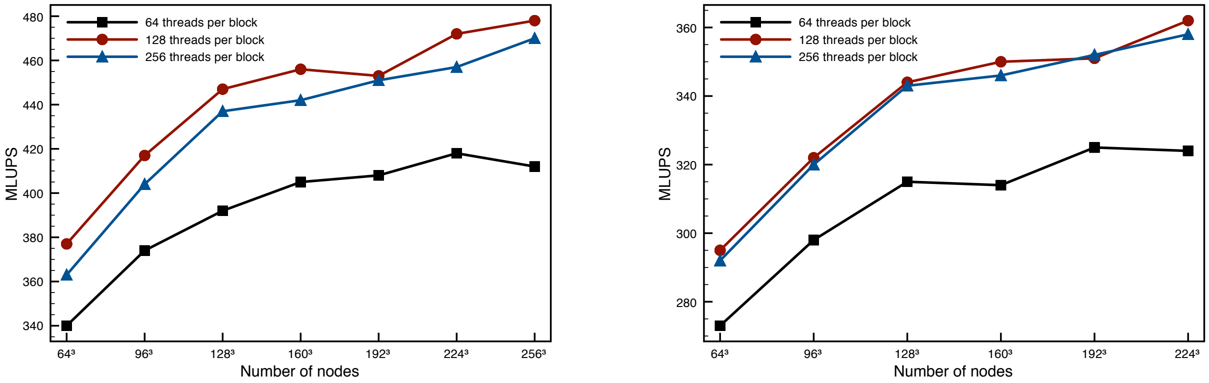


Figure 12: Velocity magnitude for the three-dimensional lid driven cavity at $Re = 300$.

remark is that the adopted memory-saving implementation allows the use of very fine lattices: up to 16 Mio. nodes for the D3Q15 and up to 8 Mio. for the D3Q19. To the best of authors knowledge this is a result that haven't been obtained yet with other single GPU implementations.

The performance of our code is comparable to the ones of other works [3, 4]. Nonetheless it must be observed that other works seem to reach even higher performance [36, 27]. This difference is mainly due to our constraint in having a general and memory saving implementation, two elements that slightly deteriorate the performance of a code.



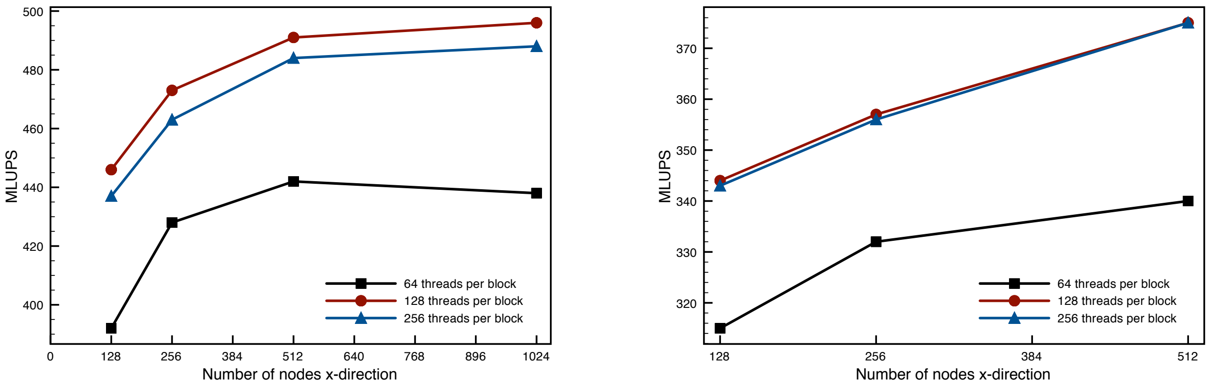
(a) D3Q15

(b) D3Q19

Figure 13: Performance of the code for different lattice discretizations of a unitary cubic domain.

6 Conclusion

We have presented a general modular framework for the lattice Boltzmann method on GPUs, valid for two- and three-dimensional problems. The proposed implementation is characterized by an *ad hoc* SoA formulation and a semi-indirect addressing scheme which allow an efficient computation of



(a) D3Q15

(b) D3Q19

Figure 14: Performance of the code for different lattice discretizations of the x-direction.

the solution both on fluid and boundary nodes. Moreover, the swapping technique proposed in [22] has been adopted to save up to the 50% of the memory required by a standard LBM implementation. The performance of the code is very good, achieving on a NVIDIA GeForce GTX 480 more than 490 MLUPS and 370 MLUPS respectively for the D3Q15 and the D3Q19 lattice structures. Future work will include the extension of the code to multiple GPUs and hybrid CPU-GPU clusters.

Acknowledgments

The authors would like to acknowledge the financial support of the Swiss Platform for High-Performance and High-Productivity Computing (HP2C) and of the European Research Council through the Advanced Grant Mathcard, Mathematical Modelling and Simulation of the Cardiovascular System, Project ERC-2008-AdG 227058.

References

[1] C.K. Aidun and J.R. Clausen. Lattice-Boltzmann method for complex flows. *Annu. Rev. Fluid Mech.*, 42:439–472, 2010.

[2] S Ansumali. *Minimal kinetic modeling of hydrodynamics.* PhD thesis, ETH Zürich, 2004.

[3] P. Bailey, J. Myre, S.D.C. Walsh, D.J. Lilja, and M.O. Saar. Accelerating lattice Boltzmann fluid flow simulations using graphics processors. In *2009 International Conference on Parallel Processing*, pages 550–557. IEEE, 2009.

[4] M. Bernaschi, M. Fatica, S. Melchionna, S. Succi, and E. Kaxiras. A flexible high-performance lattice Boltzmann GPU code for the simulations of fluid flows in complex geometries. *Concurrency Computat.: Pract. Exper.*, 22(1):1–14, 2010.

[5] A. Caiazzo. Analysis of lattice Boltzmann initialization routines. *Journal of statistical physics*, 121(1):37–48, 2005.

[6] C. Cercignani. *Mathematical methods in kinetic theory.* Plenum Press New York, 1969.

[7] P. Chen. The lattice Boltzmann method for fluid dynamics: theory and applications. Master's thesis, EPFL, 2011.

[8] S. Chen and G.D. Doolen. Lattice Boltzmann method for fluid flows. *Ann. Rev. Fluid Mech.*, 30:329–364, 1998.

[9] B. Chopard and M. Droz. *Cellular automata modeling of physical systems.* Cambridge University Press Cambridge, UK, 1998.

[10] NVIDIA CUDA. *Programming Guide, Version 3.2*, 2010.

[11] D. d'Humières, I. Ginzburg, M. Krafczyk, P. Lallemand, and L.S. Luo. Multiple-relaxation-time lattice Boltzmann models in three-dimensions. *Trans. R. Soc. Lond. A*, 360(1792):437–451, 2002.

[12] U. Ghia, K.N. Ghia, and C.T. Shin. High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method. *J. Comput. Phys.*, 48(3):387–411, 1982.

[13] F.J. Higuera and J. Jimenez. Boltzmann approach to lattice gas simulations. *EPL (Europhysics Letters)*, 9:663–668, 1989.

[14] J. Hoberock and N. Bell. Thrust: A parallel template library, 2010. Version 1.3.0.

[15] P.H. Kao and R.J. Yang. An investigation into curved and moving boundary treatments in the lattice Boltzmann method. *J. Comput. Phys.*, 227(11):5671–5690, 2008.

[16] F. Kuznik, C. Obrecht, G. Rusaouen, and J.J. Roux. LBM based flow simulation using GPU computing processor. *Comput. Math. Appl.*, 59(7):2380–2392, 2010.

[17] J. Lätt. How to implement your ddqq dynamics with only q variables per node (instead of 2q). Technical report, Tufts University, June 2007.

[18] J. Lätt. *Hydrodynamic limit of lattice Boltzmann equations.* PhD thesis, Geneva University, 2007.

[19] J. Lätt and B. Chopard. Lattice Boltzmann method with regularized non-equilibrium distribution functions. *Math. Comp. Sim.*, 72:165–168, 2006.

[20] J. Lätt, B. Chopard, O. Malaspinas, M. Deville, and A. Michler. Straight velocity boundaries in the lattice Boltzmann method. *Phys. Rev. E*, 77:056703, 2008.

[21] O. Malaspinas. *Lattice Boltzmann method for the simulation of viscoelastic fluid flows.* PhD thesis, EPFL, 2009.

[22] K. Mattila, J. Hyväluoma, T. Rossi, M. Aspnäs, and J. Westerholm. An efficient swap algorithm for the lattice Boltzmann method. *Comput. Phys. Commun.*, 176(3):200–210, 2007.

[23] K. Mattila, J. Hyväluoma, J. Timonen, and T. Rossi. Comparison of implementations of the lattice-Boltzmann method. *Comput. Math. Appl.*, 55(7):1514–1524, 2008.

[24] G.R. McNamara and G. Zanetti. Use of the Boltzmann equation to simulate lattice-gas automata. *Phys. Rev. Lett.*, 61(20):2332–2335, Nov 1988.

[25] R. Mei, L.-S. Luo, P. Lallemand, and D. d'Humières. Consistent initial conditions for lattice Boltzmann simulations. *Comp. Fluids*, 35:855–862, 2006.

[26] R.R. Nourgaliev, T.N. Dinh, T.G. Theofanous, and D. Joseph. The lattice Boltzmann equation method: theoretical interpretation, numerics and implications. *Int. J. Multiphase Flow*, 29(1):117 – 169, 2003.

[27] C. Obrecht, F. Kuznik, B. Tourancheau, and J.J. Roux. A new approach to the lattice Boltzmann method for graphics processing units. *Comput. Math. Appl.*, 2010.

[28] openCL. http://www.khronos.org/opencl/.

[29] B. Palmer and J. Nieplocha. Efficient algorithms for ghost cell updates on two classes of mpp architectures. In S.G. Akl and T. Gonzalez, editors, *PDCS International Conference on Parallel and Distributed Computing Systems*, pages 197–202. ACTA Press, Anaheim, CA, United States(US), 2002.

[30] Y.H. Qian, D. d'Humieres, and P. Lallemand. Lattice BGK models for Navier-Stokes equation. *EPL (Europhysics Letters)*, 17:479, 1992.

[31] S. Ryoo, C.I. Rodrigues, S.S. Baghsorkhi, S.S. Stone, David B. Kirk, and W.W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM.

[32] M. Schulz, M. Krafczyk, J. Tölke, and E. Rank. Parallelization strategies and efficiency of CFD computations in complex geometries using lattice Boltzmann methods on high-performance computers. In *High performance scientific and engineering computing: proceedings of the 3rd International FORTWIHR Conference on HPSEC, Erlangen, March 12-14, 2001*, page 115. Springer Verlag, 2002.

[33] X. Shan, X.-F. Yuan, and H. Chen. Kinetic theory representation of hydrodynamics: a way beyond the Navier-Stokes equation. *J. Fluid Mech.*, 550:413–441, 2006.

[34] S. Succi. *The lattice Boltzmann equation for fluid dynamics and beyond.* Oxford University Press, USA, 2001.

[35] J. Tölke. Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA. *Comput. Visual. Sci.*, 13(1):29–39, 2010.

[36] J. Tölke and M. Krafczyk. TeraFLOP computing on a desktop PC with GPUs for 3D CFD. *Int. J. Comput. Fluid. Dynam.*, 22(7):443–456, 2008.

[37] G. Wellein, T. Zeiser, G. Hager, and S. Donath. On the single processor performance of simple lattice Boltzmann kernels. *Computers & Fluids*, 35(8-9):910–919, 2006.

[38] D.A. Wolf-Gladrow. *Lattice-gas cellular automata and lattice Boltzmann models: an introduction*, volume 1725 of *Lecture Notes in Mathematics*. Springer, 2000.

A Lattice structures

Below we provide the various lattice structures implemented in our lattice Boltzmann code. For the sake of simplicity these are given in *lattice units*, i.e. assuming a unitary dimensionless space and time discretizations ($\delta t = 1$ and $\delta x = 1$). For each lattice structure we provide the speed of sound c_s, the lattice weights w_i and the lattice velocities $\mathbf{e}_i$. The weights are grouped according to their euclidean norm. Different subscripts identifies different groups: 0 for the weight associated to the rest distribution, s for small, m for medium and l for long.

D2Q5

$c_s^2 = \frac{1}{2}$

$w_0 = 0$ $w_s = \frac{1}{4}$

$\mathbf{e}_0 = (0, 0)$
$\mathbf{e}_1 = (-1, 0)$ $\mathbf{e}_2 = (0, -1)$ $\mathbf{e}_3 = (1, 0)$ $\mathbf{e}_4 = (0, 1)$

D2Q9

$c_s^2 = \frac{1}{3}$

$w_0 = \frac{4}{9}$ $w_s = \frac{1}{9}$ $w_l = \frac{1}{36}$

$\mathbf{e}_0 = (0, 0)$
$\mathbf{e}_1 = (-1, 1)$ $\mathbf{e}_2 = (-1, 0)$ $\mathbf{e}_3 = (-1, -1)$ $\mathbf{e}_4 = (0, -1)$
$\mathbf{e}_5 = (1, -1)$ $\mathbf{e}_6 = (1, 0)$ $\mathbf{e}_7 = (1, 1)$ $\mathbf{e}_8 = (0, 1)$

D3Q15

$c_s^2 = \frac{1}{3}$

$w_0 = \frac{2}{9}$ $w_s = \frac{1}{9}$ $w_l = \frac{1}{72}$

$\mathbf{e}_0 = (0, 0, 0)$
$\mathbf{e}_1 = (-1, 0, 0)$ $\mathbf{e}_2 = (0, -1, 0)$ $\mathbf{e}_3 = (0, 0, -1)$
$\mathbf{e}_4 = (-1, -1, -1)$ $\mathbf{e}_5 = (-1, -1, 1)$ $\mathbf{e}_6 = (-1, 1, -1)$ $\mathbf{e}_7 = (-1, 1, 1)$
$\mathbf{e}_8 = (1, 0, 0)$ $\mathbf{e}_9 = (0, 1, 0)$ $\mathbf{e}_{10} = (0, 0, 1)$
$\mathbf{e}_{11} = (1, 1, 1)$ $\mathbf{e}_{12} = (1, 1, -1)$ $\mathbf{e}_{13} = (1, -1, 1)$ $\mathbf{e}_{14} = (1, -1, -1)$

D3Q19

$c_s^2 = \frac{1}{3}$

$w_0 = \frac{1}{3}$	$w_s = \frac{1}{18}$	$w_l = \frac{1}{36}$

$\mathbf{e}_0 = (0, 0, 0)$		
$\mathbf{e}_1 = (-1, 0, 0)$	$\mathbf{e}_2 = (0, -1, 0)$	$\mathbf{e}_3 = (0, 0, -1)$
$\mathbf{e}_4 = (-1, -1, 0)$	$\mathbf{e}_5 = (-1, 1, 0)$	$\mathbf{e}_6 = (-1, 0, -1)$
$\mathbf{e}_7 = (-1, 0, 1)$	$\mathbf{e}_8 = (0, -1, -1)$	$\mathbf{e}_9 = (0, -1, 1)$
$\mathbf{e}_{10} = (1, 0, 0)$	$\mathbf{e}_{11} = (0, 1, 0)$	$\mathbf{e}_{12} = (0, 0, 1)$
$\mathbf{e}_{13} = (1, 1, 0)$	$\mathbf{e}_{14} = (1, -1, 0)$	$\mathbf{e}_{15} = (1, 0, 1)$
$\mathbf{e}_{16} = (1, 0, -1)$	$\mathbf{e}_{17} = (0, 1, 1)$	$\mathbf{e}_{18} = (0, 1, -1)$

D3Q27

$c_s^2 = \frac{1}{3}$

$w_0 = \frac{8}{27}$	$w_s = \frac{2}{27}$	$w_l = \frac{1}{216}$

$\mathbf{e}_0 = (0, 0, 0)$			
$\mathbf{e}_1 = (-1, 0, 0)$	$\mathbf{e}_2 = (0, -1, 0)$	$\mathbf{e}_3 = (0, 0, -1)$	
$\mathbf{e}_4 = (-1, -1, 0)$	$\mathbf{e}_5 = (-1, 1, 0)$	$\mathbf{e}_6 = (-1, 0, -1)$	
$\mathbf{e}_7 = (-1, 0, 1)$	$\mathbf{e}_8 = (0, -1, -1)$	$\mathbf{e}_9 = (0, -1, 1)$	
$\mathbf{e}_{10} = (1, 0, 0)$	$\mathbf{e}_{11} = (0, 1, 0)$	$\mathbf{e}_{12} = (0, 0, 1)$	$\mathbf{e}_{13} = (-1, 1, 1)$
$\mathbf{e}_{14} = (1, 0, 0)$	$\mathbf{e}_{15} = (0, 1, 0)$	$\mathbf{e}_{16} = (0, 0, 1)$	
$\mathbf{e}_{17} = (1, 1, 0)$	$\mathbf{e}_{18} = (1, -1, 0)$	$\mathbf{e}_{19} = (1, 0, 1)$	
$\mathbf{e}_{20} = (1, 0, -1)$	$\mathbf{e}_{21} = (0, 1, 1)$	$\mathbf{e}_{22} = (0, 1, -1)$	
$\mathbf{e}_{23} = (1, 1, 1)$	$\mathbf{e}_{24} = (1, 1, -1)$	$\mathbf{e}_{25} = (1, -1, 1)$	$\mathbf{e}_{26} = (1, -1, -1)$

Recent publications :

MATHEMATICS INSTITUTE OF COMPUTATIONAL SCIENCE AND ENGINEERING
Section of Mathematics
Ecole Polytechnique Fédérale
CH-1015 Lausanne

15.2010 A. MANZONI, A. QUARTERONI, G. ROZZA:
Shape optimization for viscous flows by reduced basis methods and free-form deformation

16.2010 M. J. GROTE, I. SIM:
Local nonreflecting boundary condition for time-dependent multiple scattering

17.2010 A. C. I. MALOSSI, P. J BLANCO, S. DEPARIS, A. QUARTERONI:
Algorithms for the partitioned solution of weakly coupled fluid models for cardiovascular flows

18.2010 M. DISCACCIATI, A. QUARTERONI, S. QUINODOZ:
Numerical approximation of internal discontinuity interface problems

19.2010 B. KALTENBACHER, M. KALTENBACHER, I. SIM:
Perfectly matched layer technique for the second order wave equations in time domain

20.2010 T. LASSILA, G. ROZZA:
Model reduction of semiaffinely parameterized partial differential equations by two-level affine approximation

21.2010 T. HOFER, J. RAPPAZ:
Numerical conservation schemes for convection-diffusion equations

22.2010 G. ROZZA, D.B. PHUONG HUYNH, A. MANZONI:
Reduced basis approximation and a posteriori error estimation for Stokes flows in parametrized geometries : roles of the inf-sup stability constants

01.2011 T. LASSILA, A. QUARTERONI, G. ROZZA:
A reduced basis model with parametric coupling for fluid-structure interaction problems

02.2011 A. QUARTERONI, G. ROZZA, A. MANZONI:
Certified Reduced Basis Approximation for Parametrized Partial Differential Equations and Applications.

03.2011 M. LOMBARDI, N. PAROLINI, A.QUARTERONI, G. ROZZA:
Numerical simulation of sailing boats: dynamics, FSI and shape optimization.

04.2011 P. BLANCO, P. GERVASIO, A. QUARTERONI:
Extendet variational formulation for heterogeneous partial differential equations.

05.2011 A. MANZONI, A. QUARTERONI, G. ROZZA:
Model reduction techniques for fast blood flow simulation in parametrized geometries.

06.2011 M. ASTORINO, J. BECERRA SAGREDO, A. QUARTERONI :
A modular lattice Boltzmann solver for GPU computing processors.