Politecnico di Milano

Mathematical Engineering - Computational Science and Learning

Implementation and Parallelization of the Lattice-Boltzmann Method

Advanced Programming for
Scientific Computing
Prof. L. Formaggia

Students:
Alessandro Renna
Mattia Marzotto

Contents

List of Figures

Chapter 1

Lattice-Boltzmann Method - Introduction

The **Lattice-Boltzmann method**, or **LBM**, is a numerical method designed for incompressible viscous fluid flows, in which the fluid is considered to be composed of identical particles whose velocities are restricted to a finite set of vectors, called **lattice gas model**. In other words, the LBM uses an analogy to the kinetic theory of gases to study the behavior of fluid flows at mesoscopic scales [1, 4].

In this work, we have focused on 2D single-phase incompressible viscous fluid problems, using the **D2Q9** lattice gas model, which defines the particle velocities and weights along 9 directions inside the plane (fig. 1.1).

The fluid-dynamic problem can be solved using the Lattice-Boltzmann equation:

$$f_i(x + c_i\Delta t, t + \Delta t) = f_i(x, t) + \Omega_i(x, t), \tag{1.1}$$

where $\Omega_i(x, t)$ is the **BGK** collision operator:

$$\Omega_i(x, t) = -\frac{f_i(x, t) - f_i^{eq}(x, t)}{\tau}\Delta t. \tag{1.2}$$

To ease the computation, eq. (1.1) is divided in a **collision** and a **streaming** step (fig. 1.2), which are computed on each lattice point using the corresponding **velocity distribution functions** $f_i(x, t)$.

In the LBM, the boundary conditions are directly applied to the distribution functions. In this work, we have implemented the **Interpolated Bounce-Back** method

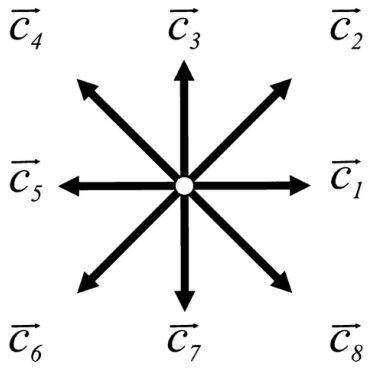

Figure 1.1: D2Q9 lattice gas model, direction from 0 to 8 (with 0 on the lattice node).

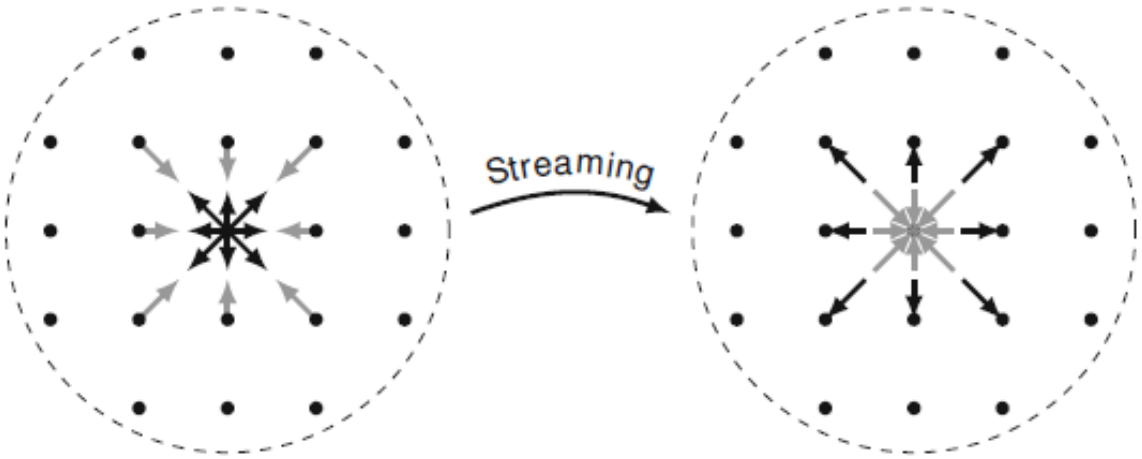

Figure 1.2: Collision and streaming steps.

(**IBB**) to handle solid walls and inlet boundaries and the **Zou-He** method for outlet conditions [4]. The IBB method includes additional information about the wall position during the bounce-back process, ensuring more precise handling of curved or inclined boundaries than other **BB** methods; however, due to the nature of LBM, the IBB can sometimes produce oscillations within the fluid. These are usually negligible but can become highly significant in the presence of outlet boundaries. To overcome this problem, we have employed the Zou He method, or Non-Equilibrium Bounce-Back, a well-known and established boundary method for working on straight boundaries.

From the velocity distribution functions, macroscopic quantities, such as mass density and velocity field (eq. (1.3)), and forces acting on the fluid or obstacles, such as drag and lift, can be computed. So, we can freely move between the mesoscopic and macroscopic scales in the LBM.

$$\rho(x,t) = \sum_i f_i(x,t), \quad \rho\mathbf{u}(x,t) = \sum_i \mathbf{c_i} f_i(x,t) \tag{1.3}$$

Chapter 2

Code Structure

In this chapter, we will discuss the structure of the code and the choices that we made during the development. The code repository can be found at [6]. The code is organized in several files and folders, employing **C++**, **Python**, **OpenMP** and **CUDA** languages.

Code folders

- **src**:
  - lattice (.cpp and .hpp);
  - node (.cpp and .hpp);
  - lattice_gpu (.cu and .cuh);
  - **python_scripts**:
    - lattice_generation_RGB.py;
    - animation.py.
  - **utils**:
    - utils (.cpp and .hpp);
    - utils_python (.cpp and .hpp).
- examples:
  - example 1;
  - example 2;
  - ...

User-made folders

- build;
- env.

General folders

- docs;
- images.

Each folder has custom-made README.md and CMakeList.txt files to link and compile the code. For some folders, the latter is used to copy all the files (**images**) or to create symbolic links (**python_scripts**) inside the corresponding folder inside **build**.

2.1 Source Code

The code is structured around two main classes: *Lattice* and *Node* (fig. 2.1), defined in **lattice.hpp** and **node.hpp** respectively. The *Lattice* class handles the geometry and simulation of a given problem, while the *Node* class contains all information regarding

a node of the lattice, such as velocity distribution functions, spatial coordinates, and node's type. The coordinates uniquely identify each node, while the type describes the behavior and the operation a node should perform. There are six types of nodes: **fluid** and **boundary** nodes, on which the movement of fluids takes place, eq. (1.1) is solved and macroscopic forces and quantities are computed, and **wall**, **obstacle**, **inlet** and **outlet** nodes, on which no computation takes place, but are necessary for the calculation of initial and boundary conditions. Boundary nodes are fluid nodes that have a wall, obstacle, inlet, or outlet node adjacent; they are defined as such since the boundary conditions are applied to them.

The code is designed to run in parallel on CPU using **OpenMP**. Every loop over the nodes of the lattice is parallelized using `#pragma omp parallel for`. Furthermore, it is also possible to run the code using a GPU parallelization, which is implemented within **lattice_gpu.cu** using the **CUDA** language.

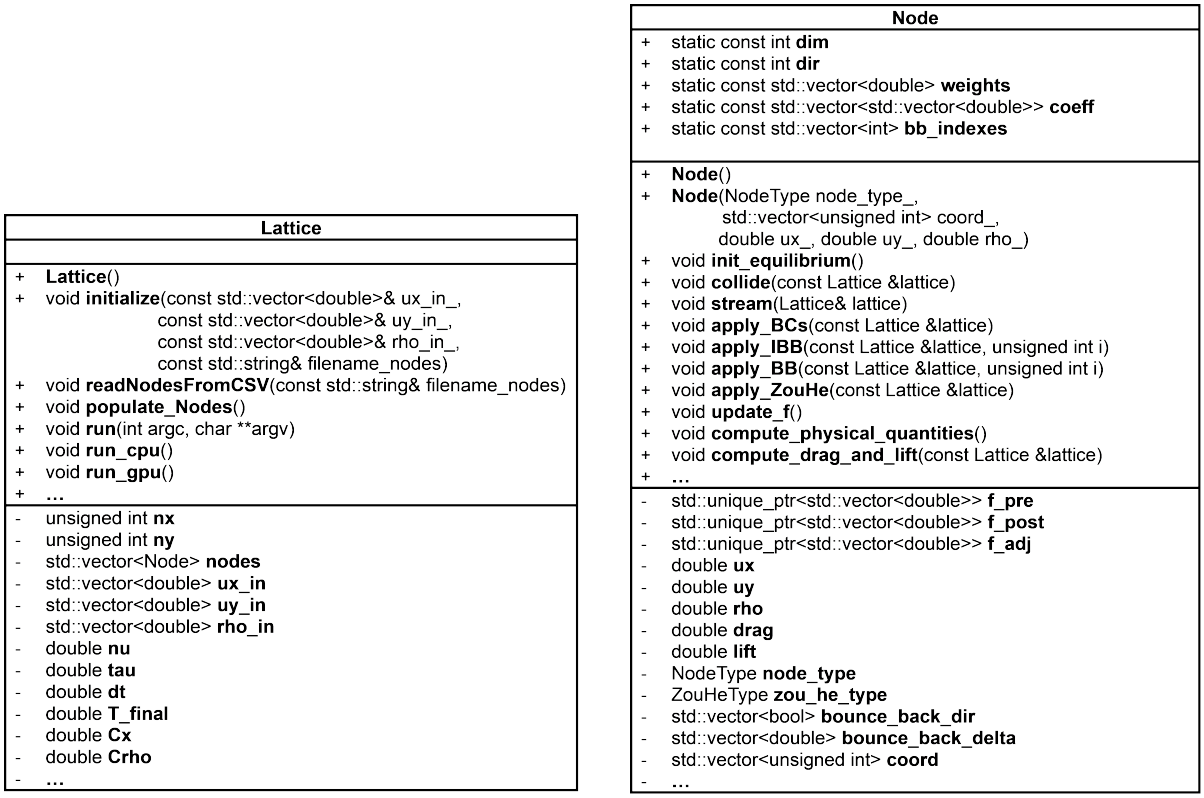

Figure 2.1: Main members and methods of **Lattice** and **Node** classes.

2.1.1 LBM execution

Figure 2.2 shows the order of operations of the **LBM**. The first step is to create and initialize the lattice and its nodes. The node's coordinates, type, and other lattice-related information are set using the output of **lattice_genaration.py** described in section 2.2, while the velocity distribution functions are set to equilibrium values based on the initial conditions. To optimize memory usage, the Lattice and Node constructors allocate and initialize their members in advance using the *resize()* function.

The second step is to call the function *Lattice.run(...)*, which checks if CUDA is installed on the user device. We have created a macro to handle this information at

compile time and improve the overall simulation flow. Then, it checks if the call of the executable contains the flag "*-gpu*":

```
./example_executable
```

or

```
./example_executable -gpu
```

the flag is ignored if the CUDA is not found. Lastly, depending on these checks, it calls either *Lattice.run_cpu()* or *Lattice.run_gpu()*, and prints a message for the user:

- CUDA not found: ***Running on CPU***;
- CUDA found, without "*-gpu*": ***CUDA enabled. Running on CPU***;
- CUDA found, with "*-gpu*": ***CUDA enabled. Running on GPU***.

These functions are the core of the solver. Following the flowchart, they behave similarly but have significant differences since they are designed with OpenMP and CUDA, respectively.

The loop over time contains two blocks parallelized over the nodes (fig. 2.2). In the first, each fluid and boundary node solves eq. (1.1) in two steps, a collision and streaming step. To manage these steps and the one in the second block more efficiently, we have introduced three types of distribution functions, which are managed through *unique pointer* to *std::vector*: *f_pre*, points to the pre-collision values, *f_post*, points to the post-collision values, and *f_adj*, points to the values streamed by the surrounding nodes into the current node.

In *Node.collide(...)*, the post-collision distribution is computed from the pre-collision one, as in eq. (1.1), and in *Node.stream(...)*, the post-collision distributions are saved in the *f_adj* of the surrounding nodes. The distributions are streamed only towards fluid and boundary nodes.

In the second block, the boundary conditions are applied, and if an obstacle is present, drag and lift are computed; then, after swapping the pre-collision and adjacent distribution pointer, the velocity field and mass density are evaluated, as in eq. (1.3).

Node.apply_BCs(...) loops over the eight directions around a node (fig. 1.1) and checks the type of each node in that direction. For obstacle, wall, and inlet nodes, it applies the Interpolated Bounce-Back method, *Node.apply_IBB(...)*, while for obstacle ones, it applies the Zou-He boundary conditions, *Node.apply_ZouHe(...)*, [2, 3, 4].

The velocity field, mass density and forces are then saved in *.txt* files, and the next iteration starts.

2.1.2 CUDA Parallelization

Due to the structure of the CUDA language, several elements need to be changed [5]. The two blocks of *run_cpu()* are now two **kernel** functions, and the functions to compute the boundary conditions are redefined as **device CUDA** functions. Moreover, all the members of the Lattice and Node objects necessary for the simulation have been converted to **C** variables and arrays. In other words, the CUDA code replicates the C++ code, where we move from an object-oriented language to a more essential C language. This introduces possible coding errors since the two codes must be updated separately while maintaining the method's consistency. However, as will be shown in chapter 3, the GPU parallelization increases the solver's power, significantly reducing the simulation time.

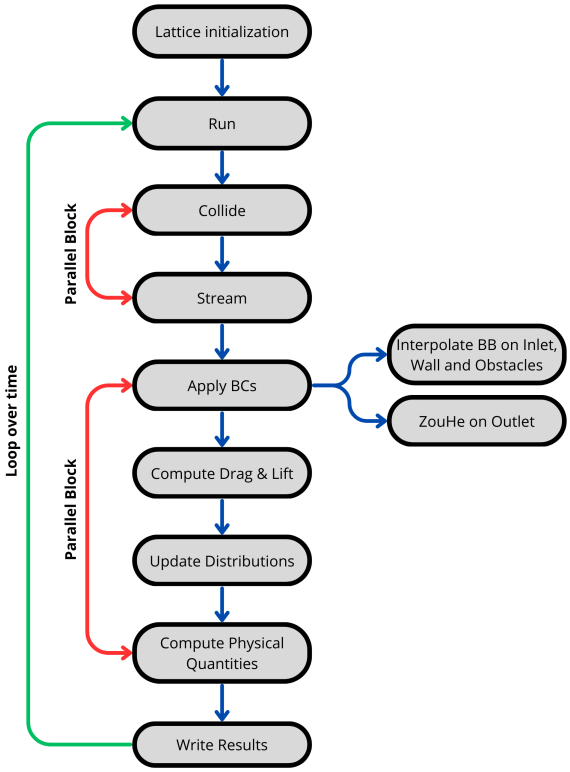

Figure 2.2: Execution flowchart.

2.2 Python Scripts

lattice_generation_RGB.py generates a *.csv* file containing the lattice information and adds new parameters needed for the simulation to *params.json*. The script is called at the beginning of an example or test. It reads the parameters entered by the user in *params.json* and, in particular, opens the image of the domain upon which the lattice grid is generated. The type of node and the **distances** between boundary and surrounding nodes (necessary for IBB) are also identified. The node's type is identified through the following color scheme (fig. 2.3):

0. black $\longrightarrow$ fluid node;
1. white $\longrightarrow$ obstacle node;
2. blue $\longrightarrow$ wall node;
3. green $\longrightarrow$ inlet node;
4. red $\longrightarrow$ outlet node.

This choice has both pros and cons. On the one hand, the solver can work with different types of geometries that can be represented in a 2D image following the above color scheme. On the other hand, these images must be very high quality since the distance among nodes is a based on a fixed number of pixels. Moreover, if the image is not designed properly, there could be many inactive nodes, such as wall nodes. Figure 2.4 shows well both pros and cons: we have a geometry that is not too simple, which represents a simplified pipe, however, the image, 3000 by 3000 pixels, has an extensive wall surface;

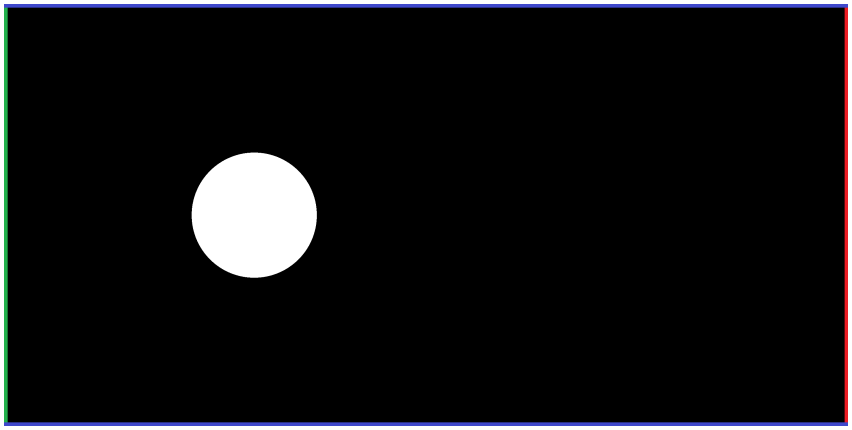

Figure 2.3: Channel image with spherical obstacle following the RGB color scheme.

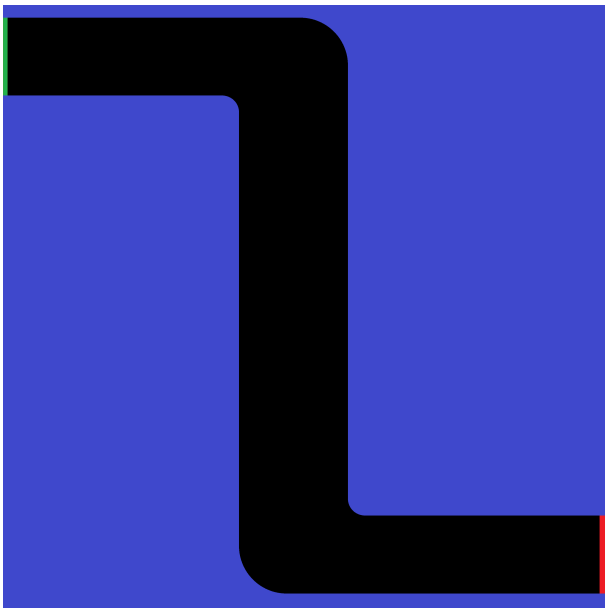

Figure 2.4: Example image of a pipe, presenting many inactive nodes.

this means that the lattice will have many wall nodes that will not participate in the simulation. A solution to this problem could be to use a data structure that efficiently stores sparse nodes, like a *std::map*.

The LBM works with its own set of units and measurement conventions [4], in our case:

$$\Delta x^* = 1, \quad \Delta t^* = 1, \quad \rho^* = 1.$$

lattice_generation_RGB.py converts the given *SI* parameters inside *params.json* into the proper *lattice units*, following the above convention. Lastly, it checks if the given values for n_x and n_y (lattice dimensions) are sufficient to create a uniform grid inside the given image; if not, they are increased to an adequate value.

animation.py is called at the end of the test to create a video of the evolution of the velocity field and mass density. If an obstacle is present inside the fluid, it also plots the drag and lift forces in time.

2.3 Utils

utils.cpp and **.hpp** contain functions used by both the source and example codes, which are not meant to be part of either the Lattice or Node class, such as operators or functions to initialize the inlet for a specific problem. **utils_python.cpp** and **.hpp** contain functions to connect C++ code and Python scripts.

2.4 User-made and general folders

env and **build** are user-made folders: **env** contains the environment to run Python code, while **build** contains libraries and executables. The **build** folder structure mirrors the structure of the repository, so the compiled code can directly refer to the files inside it, such as images, Python scripts, and parameter files.
images contains *.png* files, which are used by the various examples to run and solve different problems.
docs is created through *Doxygen* commands and contains the code documentation.

Chapter 3

Numerical Tests

In this section, we discuss the potentiality of our solver, using various tests, and describe how to create new ones from scratch. All tests were run on the same device with the following features:

- CPU: **12th Gen Intel(R) Core(TM) i5-12450H 2.50 GHz (octa-core, 12 threads)**;
- GPU: **NVIDIA GeForce RTX 2050**.

The testing phase is structured to assess the solver's performance in different regimes and geometrical configurations to evaluate the computational gains of parallel computing between a parallel CPU setup and a GPU. The tests are also designed to assess the physical feasibility of the solutions provided by the solver under various conditions.

3.1 Lid-Driven Cavity Flow ($Re = 1000$)

We first observe the case of lid-driven cavity flow. We consider a square domain with characteristic length 10 m, inlet velocity 1 m/s, density $\rho = 1\,\mathrm{kg/m^3}$ and viscosity $\nu = 1 \times 10^{-2}\,\mathrm{Nms/kg}$, using a lattice of 9×10^4 nodes. We also considered a 100 s time window, saving the data every second. With the parameters provided, 42427 iterations are needed to simulate 100 seconds. The system evolves following the theoretical results. The IBB method correctly imposes the Dirichlet boundary condition on the top wall, and a large velocity gradient is captured on the top right corner. The system evolves to a steady-state solution, producing a large vortex characteristic of this type of experiment with the given value of Re, confirming the solver's validity in this simple case.

The test was simulated on both CPU and GPU, leading to the same solution but with extremely different times (table 3.1). Simulating with the GPU, the runtime is reduced by approximately 7 times, table 3.2 shows that this value does not strictly depend on the lattice size.

	Total simulation time	Mean time per iteration
CPU	955.411 s	0.022 514 5 s
GPU	123.865 s	0.002 916 76 s
CPU/GPU	7.713	7.719

Table 3.1: Lid-Driven Cavity Flow: simulation results.

	150x150	300x300	600x600
CPU/GPU total time	7.187	7.713	6.850

Table 3.2: Lid-Driven Cavity Flow: comparison of CPU/GPU total simulation time for different lattice size.

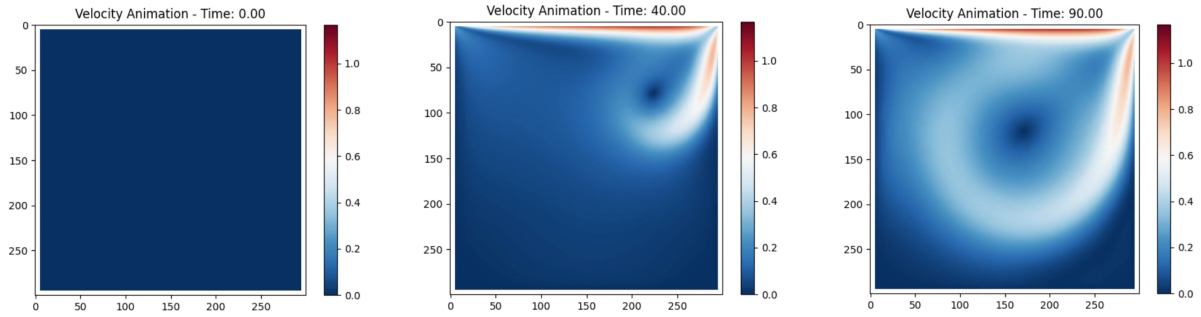

Figure 3.1: Evolution of the lid-driven cavity flow.

3.2 Channel Flow

In this second section, we simulate the flow inside a channel in the presence of a circular obstacle with a uniform velocity at the inlet. The objective of the following tests is to analyze the behavior of the solver with the new features imposed by this experiment. More precisely, we examine the imposition of boundary conditions by the IBB method to an obstacle and the Zou-He boundary conditions on the outlet. Moreover, since we include a solid object to simulate an outer flow, we can apply the methods to compute integral quantities such as lift and drag.

We will present two tests at increasing Re to capture the vortex shedding phenomenon.

3.2.1 Low-Reynolds Flow ($Re = 5000$)

We first observe the flow in the low Reynolds regime, considering a rectangular domain with characteristic length 10 m, inlet velocity 0.5 m/s, density $\rho = 1\,\mathrm{kg/m^3}$, and viscosity $\nu = 1 \times 10^{-2}\,\mathrm{Nms/kg}$, using a lattice of 9×10^4 nodes. We also considered a 100 s time window, saving the data every second.

The simulation is in line with the expected results; a symmetric low-velocity region is formed behind the circular obstacle showcasing recirculation regions. Moreover, the homogeneous outflow boundary conditions (Zou-He) correctly absorb the velocity, minimizing the creation of artificial reflective sound waves inside the domain.

	Total simulation time	Mean time per iteration
CPU	555.983 s	0.023 165 9 s
GPU	72.9739 s	0.003 037 71 s
CPU/GPU	7.619	7.626

Table 3.3: Low-Reynolds Channel Flow: simulation results

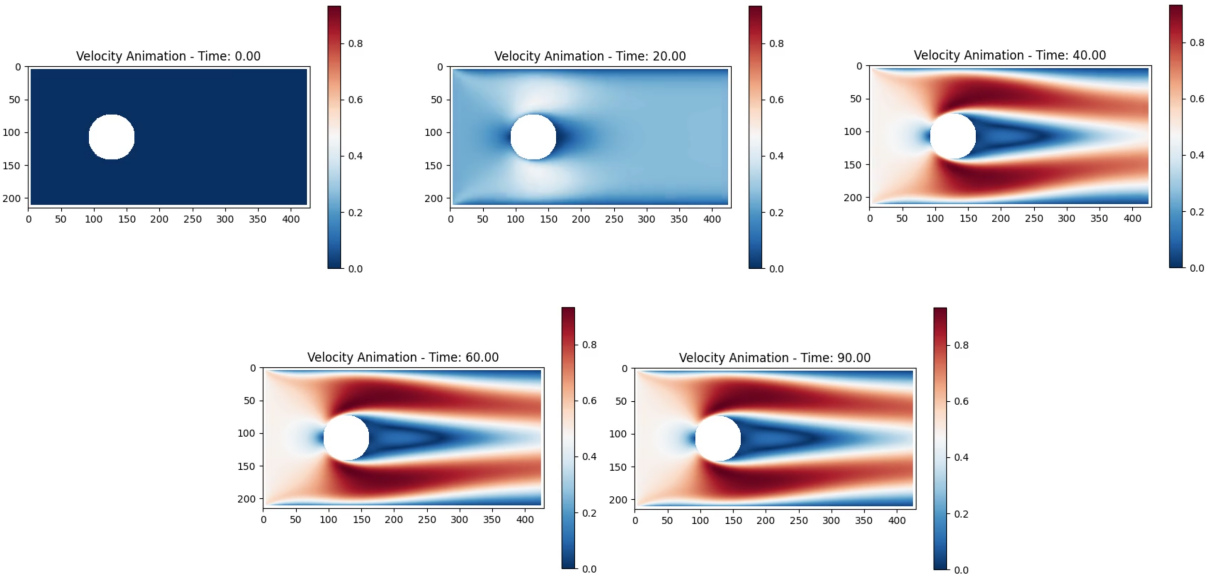

Figure 3.2: Evolution of the channel flow with spherical object at low Re.

3.2.2 High-Reynolds Flow ($Re = 10000$)

In the case of high-Reynolds flow, we modify the inlet velocity to 1 m/s, resulting in $Re = 1 \times 10^4$. We expect the simulation to reach a periodic solution depicting the vortex-shedding phenomenon.

The system progresses as expected: vortexes are generated alternately and transported correctly by the advection field. The solver seems to maintain stability even under higher numerical stress (fig. 3.3).

	Total simulation time	Mean time per iteration
CPU	1023.73 s	0.021 329 2 s
GPU	137.498 s	0.002 862 98 s
CPU/GPU	7.445	7.450

Table 3.4: High-Reynolds Channel Flow: simulation results

Comparing the two plots in fig. 3.4, we can visualize the lift and drag components of the channel simulations. We notice small oscillations in the initial time frame due to sound waves induced by the IBB condition, which reflects the wavefronts created by the interaction between the inlet and the initial condition. We also notice that with higher Re, the lift and drag forces begin to oscillate due to the vortex shedding, with the lift maintaining a mean value of zero and the mean drag value increasing with respect to the low Re case.

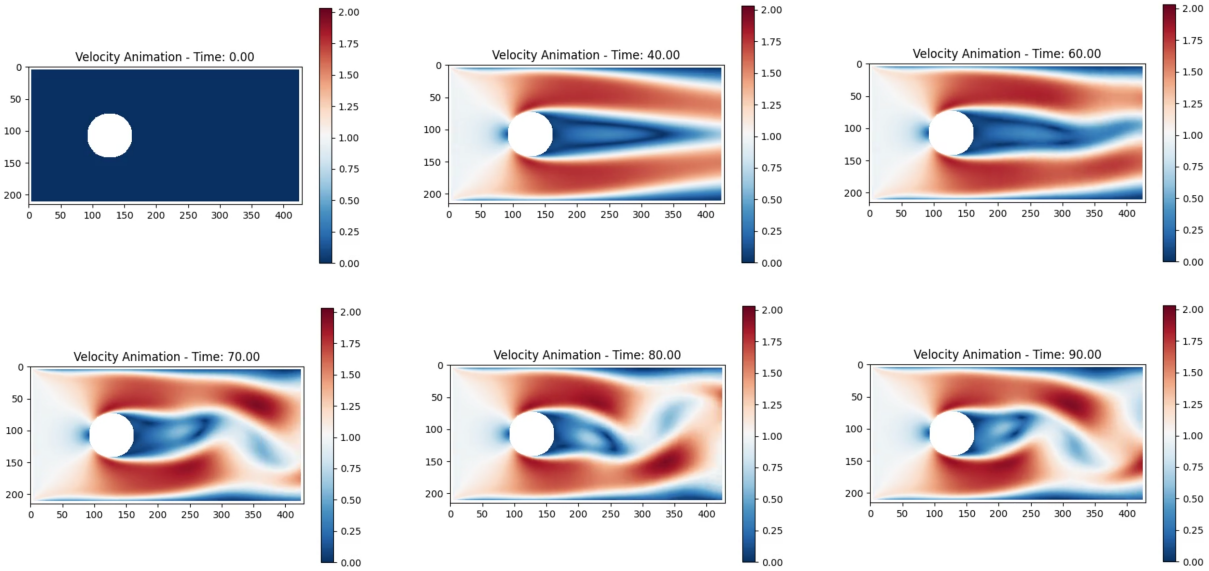

Figure 3.3: Evolution of the channel flow with spherical object at high *Re*.

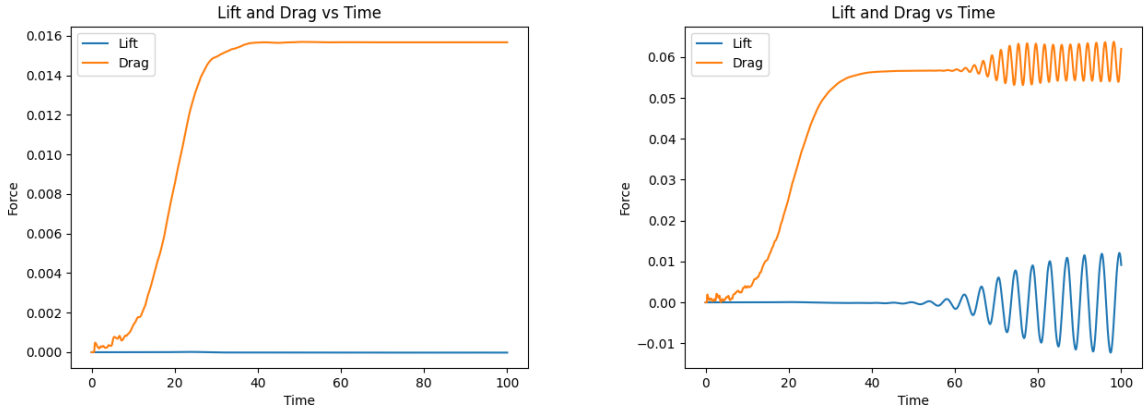

Figure 3.4: Lift and drag plots under different Reynolds conditions: fig. 3.2 on the left and fig. 3.3 on the right

Chapter 4

Conclusion and Possible Extension

The solver has proven to be highly effective and versatile, and the tests have shown its ability to adapt to different domains and flows. In particular, the parallelization via CUDA code has significantly reduced the simulation time, regardless of the lattice size.

However, given the nature of LBM and the conditions imposed on the lattice units, there are still limitations in the types of problems that can be solved. We have implemented the most common lattice units in literature as:

$$\Delta x^* = 1, \quad \Delta t^* = 1, \quad \rho^* = 1.$$

This means that the lattice values of other parameters have to be scaled accordingly. Moreover, to ensure that the numerical solution is achieved, a stability condition is imposed on the lattice velocity [4], $u^* \ll c_s$, where $c_s = 1/\sqrt{3}$ is the lattice speed of sound. For example, imposing a high-velocity value at the inlet can lead to a very small dt and many iterations, drastically increasing the execution time. Therefore, the simulation parameters must be chosen carefully to ensure a correct result in a feasible time. It may be interesting to develop the code further so that the user can choose which lattice units to use. It would also be beneficial to implement the management of mobile domains to simulate new problems and unstructured lattices to manage regions of the domain that require a more precise analysis, such as high-vorticity ones or boundary layers.

Bibliography

[1] Takaji Inamuro, Masati Yoshino, and Kosuke Suzuki. *An Introduction to the Lattice Boltzmann Method*. World Scientific, 2021.

[2] Takaji Inamuro, Masato Yoshino, and Fumimaru Ogino. A non-slip boundary condition for lattice boltzmann simulations. *Physics of Fluids*, 1995.

[3] Salvador Izquierdo and Norberto Fueyo. Characteristic nonreflecting boundary conditions for open boundaries in lattice boltzmann methods. *Physical Review E*, 78(4):046707, 2008.

[4] Timm Krüger, Halim Kusumaatmaja, Alezandr Kuzmin, Orest Shardt, Goncalo Silva, and Erlend Magnus Vigenn. *The Lattice Boltzmann Method, Principles and Practice*. Springer, 2017.

[5] Peter S. Pacheco and Matthew Malensek. *An Introduction to Parallel Programming*. Morgan Kaufmann, Cambridge, MA, 2nd edition, 2020.

[6] Alessandro Renna and Mattia Marzotto. Lbm_pacs: Lattice boltzmann methods. `https://github.com/AlRenna/LBM_PACS`. Accessed: 2025-01-10.