REPORT OF LBM IMPLEMENTATION

Andrea Torti, Matrina Raffaelli, Marco Cioci

This report contains a brief description of the implemented files, the choice of the programming languages and some experiments and observations. All the information here can be found in the README in a more concise form, such as building and running commands, UML diagram and execution Flowchart, implementation and design choices, and also some important considerations on performances.

Here we present our implementation of the Lattice Boltzmann Method, which is a powerful computational fluid dynamics technique, in particular, with our code, it is possible to simulate in a two-dimensional lattice the lid-driven cavity problem and the Poiseuille flow problem with an obstacle. In the latter case, the drag and lift forces on the obstacle are also calculated. Optionally, a video of the simulation is generated.

Our implementation of the Lattice Boltzmann Method (LBM) is organized across multiple files, employing the following languages: C++ (parallelized with OpenMP), Python, and CUDA. Each language has been utilized for distinct purposes, aiming to leverage its specific strengths.

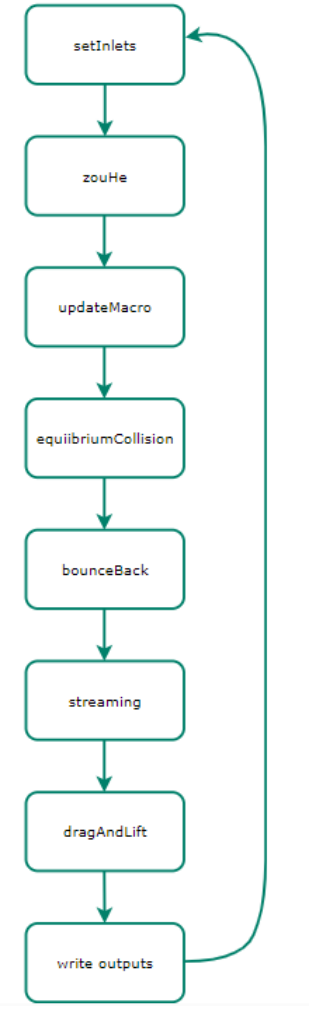
The actual simulation implementation was carried out in C++, extensively utilizing classes. Subsequently, OpenMP was employed to parallelize the execution of simulations. Python was used to generate input files for the main program, introducing obstacles such as circles and various airfoils. Lastly, CUDA was utilized to exploit the computational power of the GPU.

C++ code organizations:

The C++ code is structured across various files, leveraging the separation between source and header files:

- *'Main.cpp'*: the input arguments are managed here (the first input is the input file describing the problem to be addressed, the characteristics of the considered fluid, the duration of the simulation, and any obstacles; the second input is an integer indicating the number of plot frames). In this file, the lattice object is created, and the simulation is executed through the simulate method of the Lattice class. Additionally, if executed with the GPU flag, the CUDA code will also be executed.
- The file '*Structure.cpp*' contains the implementation of the constructor for an object of the Structure class, whose declaration is present in '*Structure.hpp*'. The Structure class is essential for the numerical discretization of the problem, as it groups information such as the size of the problem, the velocity set chosen for the simulation (typically D2Q9), and the opposite directions to these velocities. This could allow for multiple lattice representations to be used to solve the same problem and to generalize 2 and 3 dimensional problems.
- In the file '*Lattice.cpp*', the definitions of the '*Lattice.hpp*' header are implemented.
  The Lattice class is used to represent the domain considered in the simulation, containing essential information such as the type of problem being simulated, details about the fluid

under consideration, and an integer indicating the current time step. It also includes an object of the Structure class used to access velocities.
In the constructor of an object of the Lattice class an input file is read and the data needed for lattice attributes is extracted. An NDimensionalMatrix of Cells is created, and cells representing obstacles are marked as such by setting the obstacle attribute to true. Boundaries are initialized through the boundaryCalculator2d function, and fluid's initial conditions are set.
The simulate method involves successive calls to methods of the Cell class and the production of the output file (which will be later read by plotting2D.py to generate the simulation video). Writing to this file doesn't occur at every iteration but every maxIt/plotSteps iteration, aiming to generate the output file and the video faster.



Following that, implementations of the "getter" methods are provided, allowing access to various cells or private attributes of the Lattice class.

- The Cell class is defined in the file *'Cell.hpp'* and is quite detailed, containing several fundamental (public) methods for the simulation (updateMacro, equilibriumCollision, initEq, streaming, setInlets, zouHe, bounceBackObstacle, and dragAndLift), as well as getter methods. The private attributes of the Cell class are derived from numerical discretization, including the distributions f (and newF), macroscopic velocity (macroU) accounting for the overall movement of the fluid portion described by that cell, macroscopic density rho, and other information defining the cell (boundary, useful for detecting nearby obstacles;

obstacle, necessary to define whether the cell is an obstacle or not; and position, convenient to avoid recalculating the cell's position within the lattice).
In the '*Cell.cpp*' file, the constructor and other methods are implemented.
The updateMacro method aims to update rho (calculated as the sum of distributions f) and macroscopic velocity (calculated as the sum of weighted D2Q9 velocities for the distributions).
The equilibriumCollision method calculates equilibrium distributions and subsequently calculates the effect of particle collisions on the distributions. This method has a local effect, meaning that the cell's distributions are modified based on what happens within the cell itself.
The initEq method is used to set the initial equilibrium conditions, later perturbed by the lid movement or parabolic flow (applied by setInlets).
Next is the streaming method, which propagates post-collision distributions to adjacent cells (consistent with the directions described by the distributions). For this, the entire lattice is needed as input, so the cell can modify the distributions of adjacent cells using setFAtIndex in combination with getCellAtIndex.
The setInlets method is the engine of the simulation, introducing the disturbance determined by the type of problem considered. For example, in the lid-driven cavity, it sets a velocity parallel to the wall (along the x-axis) on the upper wall, while for a parabolic flow (with or without obstacles), it sets a velocity on the left side of the lattice with maximum intensity at the center and zero at the extremes (simulating Poiseuille flow).
The zouHe method applies the boundary conditions chosen for the 4 sides and 4 corners of the lattice, known as non-equilibrium bounce-back conditions. These conditions provide great accuracy and satisfy the no-slip conditions, calculating the values of unknown distributions of cells adjacent to the boundaries by enforcing the bounce-back rule for their non-equilibrium part. The Zou-He conditions for velocity are often used to simulate the movement of a solid wall or to impose a specific velocity profile on the walls. These conditions are based on the conservation of mass and momentum and can be used to set the velocity gradient or the velocity itself at the wall. However, when dealing with Poiseuille flow, we apply Zou-He pressure conditions on the right side. These conditions are commonly used when it is necessary to fix the pressure on a part of the domain.
The bounceBackObstacle method describes the boundary conditions chosen to represent the behavior of the fluid when it encounters an obstacle. Populations hitting an obstacle (solid) during propagation are reflected back from where they initially came. This ensures mass conservation, impermeability of the solid, and the no-slip condition.
Subsequently, getter methods are implemented.
Finally, the dragAndLift method is implemented, calculating the forces (drag and lift) acting on the obstacle immersed in the fluid due to the fluid movement around it.

- Finally, there is the '*Utils.cpp*' file that contains the implementation of the template class NDimensionalMatrix. This class is capable of handling matrices of variable dimensions and generic type 'T'. It includes numerous public methods such as getters, constructors, and setters, which are used to access, construct, and create cells. A very peculiar problem is addressed here, and an important discovery has been made: a vector of booleans does not behave like any other c++ vector due to compiler optimizations, so it must be treated a little differently.

OpenMP parallelization:

All these files have been implemented in C++, and their execution is sequential. However, since achieving smooth and accurate simulations requires using fine grids and performing many iterations, it is essential to parallelize the code to speed it up. For this purpose, the OpenMP language was used, leveraging its capabilities to use multiple threads on the CPU.

The choice of OpenMP was motivated by several factors, including its ease of use. Indeed, with a few instructions, a significant speedup can be achieved by simply integrating pragma directives into the C++ code. Other considerations include scalability, portability, and automatic workload management. Moreover it allows seamless execution with or without OpenMP support.

Parallelization has been implemented in the simulate method of the Lattice class, mainly using the *#pragma omp parallel* for directives. This way, the workload is automatically distributed among the available threads. To prevent variables like drag and lift from being set to 0 by all threads the *#pragma omp single* directive is used.

In the dragAndLift method of the Cell class, atomic writing is necessary to ensure that the drag and lift variables are incremented atomically by the threads, avoiding race conditions. This prevents multiple threads from writing to the same variable, ensuring the consistency of its value.

Implementation of obstacles, output generation, and drag and lift in Python:

In the endeavor to seamlessly integrate obstacles into a simulation employing the Lattice-Boltzmann Method (LBM) for the study of fluid dynamics, our choice of implementation rested upon the Python programming language. This strategic decision was motivated by the presence of pertinent libraries conducive to the implementation process, the language's innate modeling flexibility, and its inherent simplicity. Despite a marginal reduction in performance, this approach tactfully preserves the overall execution efficiency of the simulation, given that the obstacle and coordinate generation occur only once per execution.

Our initiative consists in the development of three distinct types of obstacles: a complete circle, a NACA airfoil profile, and a customizable obstacle allowing user-defined specifications.

- Complete circle (*ball.py*):

  The '*ball.py*' script represents a pragmatic implementation of coordinate generation for a circular obstacle within a Lattice-Boltzmann Method (LBM) simulation in the Python environment. This script serves to facilitate the creation of diverse obstacles, ensuring adaptability to the specific demands of the simulation.

  The '*generate_circle_coordinates'* function, a cornerstone of the script, accepts center coordinates, radius, and grid dimensions as input and yields a list of tuples depicting the circle's coordinates within the grid. An illustrative example delineates the creation of a circle with a center at (100, 50) and a radius of 12 units within a 500x100 grid. The resultant coordinates are meticulously recorded in a text file ('outputs/circle_coordinates.txt').

Extensible and adaptable, this script holds the potential for generating coordinates for various obstacle types, such as NACA airfoils or custom obstacles, thereby constituting a pivotal component for the seamless integration of obstacles into LBM simulations.

- NACA airfoil profile (*airfoil_coordinates.py*):

This Python module meticulously defines functions dedicated to the generation and manipulation of coordinates for a 4-digit NACA airfoil profile. Core functionalities encompass:

-`naca4`: Generating coordinates for the airfoil profile based on NACA specifications, encompassing the 4-digit identifier, the number of points required for profile generation, and the desired inclination angle.

- `discretize_airfoil`: Converting the airfoil profile into a binary grid representation, where 1 signifies the presence of the wing.

- `write_airfoil_to_grid`: Documenting discretized airfoil profile coordinates in a text file.

- `plot_airfoil_from_file`: Reading airfoil profile coordinates from a text file and rendering them as an image.

The usage example within the code entails the generation of coordinates for a specific airfoil profile ('2412') with an inclination angle of -15 degrees. The coordinates are meticulously inscribed in a text file and presented visually as a binary image.

- Additional Implementations and Manipulations(*airfoil_profile.py*):

The Python script '*airfoil_profile.py'* introduces an additional avenue for generating and manipulating wing profiles, utilizing the NACA profile equation. The script encapsulates diverse functions for coordinate generation, discretization, file writing, and manipulations of airfoil profile coordinates.

To visualize the obstacles we've written *'png_reader.py*' where we use the Pillow (PIL) library to identify white pixels in an image, subsequently transcribing the corresponding coordinates to a text file. A succinct depiction of the `find_white_pixels` function and the subsequent coordination documentation process is provided.

In tandem with the implementation of obstacles and the generation of coordinates, an important aspect involves the analysis and visualization of drag and lift forces within the broader context of a fluid dynamics simulation. The ensuing Python code seamlessly integrates with the C++ main code, leveraging the `matplotlib` library to graphically portray drag and lift forces. This versatile code accepts the data file path as a command-line argument and produces a graph that vividly delineates the forces' trajectories throughout the simulation.

The script reads the data file, parsing each line to extract pertinent information such as frame numbers, drag force, and lift force. These forces are then graphically depicted against the frame numbers in a unified graph, expertly utilizing the capabilities of the `matplotlib` library. The end result is an illustrative image ('outputs/lift_drag.png') that succinctly encapsulates the nuanced trends of drag and lift forces throughout the simulation.

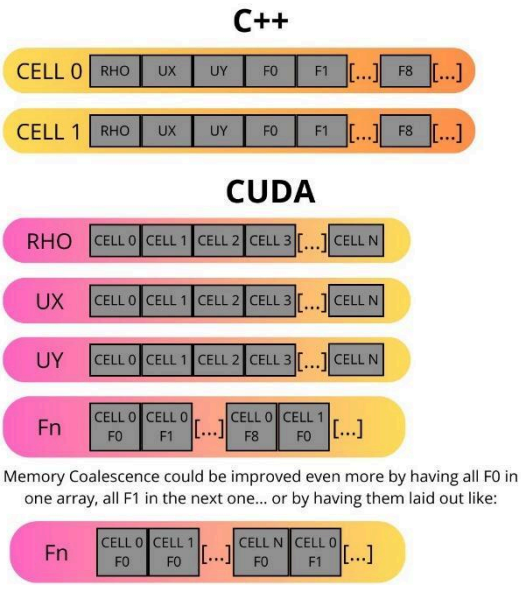
Cuda implementation:

The prevalence of object-oriented programming led us to adopt this approach over other contemporary methodologies, such as functional programming. This decision is grounded in the understanding that the primary performance gains come from the GPU. The primary objective of the CPU code is clarity and conciseness rather than aggressively optimizing for every last bit of available performance.

The key components of the gpu code are:

- Device Functions (`step1dev`):
  Implements the LBM simulation step for a specific grid point.
  Handles various boundary conditions, including lid-driven flow, parabolic profile, and Zou-He boundary conditions.
- CUDA Kernels (`step1`, `step2`, `calculateLiftAndDragKernel`):
  `step1`: Performs all the simulation steps, except the streaming.
  `step2`: Handles the streaming of fluid particles to neighboring cells.
  `calculateLiftAndDragKernel`: Computes lift and drag forces on obstacles.
- Main Simulation (`cudaCaller`):
  Allocates and initializes device memory for simulation data.
  Iteratively performs LBM steps, updating fluid properties.
  Computes lift and drag forces for problem type 2.
  Writes simulation results to files at specified intervals.

Overall, the code efficiently parallelizes the LBM simulation on the GPU, utilizing CUDA kernels and device functions to handle different aspects of the simulation. The simulation parameters, such as grid size, relaxation parameters, and problem type, are configurable.



In the GPU, we achieved better performance thanks in part to leveraging memory coalescence, as depicted in the image. The CPU code is structured as an Array of Structures, following an object-oriented paradigm, facilitating a clear code structure and a seamless implementation. On the other hand, the GPU code utilizes a Structure of Arrays data structure, optimizing for maximum performance through memory coalescence. This approach capitalizes on the GPU memory bus width, reducing the overall number of data requests when multiple threads require neighboring data.

Experiments and observations:

Simulation 1: Fluid Dynamics Around a Circular Obstacle

The proposed GIF provides a detailed visualization of the effects of the complex interaction between a fluid coming from the left wall of the grid and a parabolic velocity wave. This wave exhibits exponential growth over time, ranging from 0 to 0.2 cells per delta t. The simulation clearly reveals the adaptation of the flow to the peculiarities of the circular obstacle, implemented through the Python code section (ball.py). Placed with its center at the cell coordinates 50x50 and a radius of 12 units, the circular obstacle is inserted within a lattice of 200x100.

The stability of the execution is a crucial element in this analysis, ensuring reliable results over the 8000 temporal frames of the simulation. The visual representation of the fluid velocity magnitude, color-coded to highlight different velocity levels, provides an in-depth look at the flow behavior in the presence of the circular obstacle. Velocity peaks reach values of 0.3 cells per delta t, indicating regions of higher intensity in fluid dynamics.

Subsequently, after obtaining a complete visualization of fluid motion, graphs of the lift and drag forces acting on the obstacle were plotted. Given the arbitrary nature of multiplicative factors and measurement scales, the main focus is on the qualitative behavior of these force curves. As expected, the drag force remains relatively constant, reflecting the absence of turbulence in the flow. The lift force exhibits a linear and monotonic trend in response to the motion of the fluid coming from the left grid opening. It is noteworthy that the effects of turbulence, manifesting after delta t 2600, result in increasing oscillations, suggesting an alternation in the direction of force action.

Simulation 2: Airfoil Profile in Fluid (Airfoil 01, Reynolds 100, Inlet 0.15, Lattice 400x150)

From the analysis of a wing profile inserted into a lattice of 400x150 (Airfoil 01), generated using airfoil_generator.py with a tilt angle of 15 degrees clockwise, additional details emerge regarding the complex interaction between the fluid and the wing structure. The simulation, characterized by fluid injection with an initial maximum velocity of 0.15 cells per delta t, maintains constant execution stability for the entire duration of the process. This stability allows for an accurate analysis of the effects of lift and drag forces on the wing, providing significant insights into changes in fluid dynamics.

As anticipated, the behavior of lift and drag forces shows similarities, with both components maintaining a similar trend throughout the execution. However, a significant quantitative difference emerges, with the lift force being approximately three times higher than the drag force trend. This result can be interpreted as an advantage for a wing subjected to stress, suggesting a significant increase in lift force compared to resistance throughout the entire simulation.

Simulation 3: Oscillations in Fluid with Airfoil Profile (Airfoil 02, Reynolds 100, Inlet 0.18, Lattice 900x300)

Another experiment (Airfoil 02), replicating the wing profile but with a different inclination within a lattice of 900x300, offers an interesting comparison with the previous simulation. The wing, arranged horizontally and subjected to stress from the fluid (velocity wave with a peak at 0.18 cells per delta t), maintains weak oscillatory behavior. The analysis of lift and drag forces shows clear oscillations in both force components, suggesting greater complexity in fluid dynamics when the wing is oriented horizontally relative to the fluid.

Simulation 4: Lid Driven Cavity (Reynolds 100, Inlet 0.2, Lattice 96x96)

Finally, the simulation of a lid-driven cavity offers an overview of fluid dynamic phenomena associated with a moving wall. With a Reynolds number of 100 and an initial maximum velocity of the upper wall at 0.2 cells per delta t, the fluid reacts by generating vorticity in the northeast corner of the grid. This vorticity progressively extends towards the center of the grid, and in the complete version of the simulation, the vorticity's trajectory is visually supported by arrows indicating the direction taken by specific fluid particles.