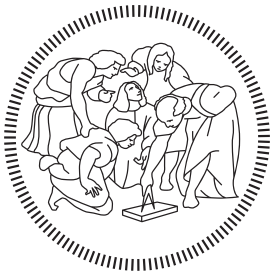


Politecnico di Milano

Lattice Boltzmann Methods

Federico Pinto
Mattia Gotti
Michele Milani

Contents

1 LBM for 2D Problems 2
1.1 Mathematical Foundation 2
1.2 2D Lid-Driven Cavity 3
Implementation Choice 3
Parallelization 4
Results 5
Validation 6
1.3 Wind-Tunnel Like Problem 7
Implementation Choice 7
Parallelization 8
Results 8

2 LBM for 3D Lid-Driven Cavity 9
2.1 Description 9
Notation and Variables 9
D3Q19 10
Collision and Streaming 10
Boundary Conditions 11
2.2 Results 12
Flow Patterns 12
Velocity Profiles 13
Isosurface Comparison 16

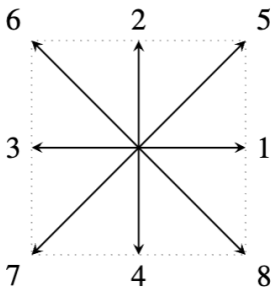
3 Conclusions 17

1 LBM for 2D Problems

1.1 Mathematical Foundation

The Lattice Boltzmann Method (LBM) solves fluid dynamics problems by modeling the microscopic behavior of fictitious particles on a lattice. Instead of solving the Navier–Stokes equations directly, LBM models the evolution of particle distribution functions $f_i(\mathbf{x}, t)$ over a discrete spatial lattice. Each function represents the density of particles at position $\mathbf{x}$, time t, moving in direction $\mathbf{e}_i$.

Discrete Velocity Set (D2Q9) In this project, we employ the D2Q9 lattice model. Each node interacts with its eight neighboring nodes plus itself through the velocity set:



$$\mathbf{e}_0 = (0,0), \quad \mathbf{e}_{1\text{–}4} = \{(\pm 1, 0), (0, \pm 1)\}, \quad \mathbf{e}_{5\text{–}8} = \{(\pm 1, \pm 1)\}$$

The associated weights are:

$$w_0 = \tfrac{4}{9}, \quad w_{1\text{–}4} = \tfrac{1}{9}, \quad w_{5\text{–}8} = \tfrac{1}{36}$$

Governing Equations The evolution of the distribution functions is governed by a discrete form of the Boltzmann equation with a BGK (single-relaxation-time) collision operator:

$$f_i^*(\mathbf{x}, t) = f_i(\mathbf{x}, t) + \frac{1}{\tau}(f_i^{eq} - f_i), \qquad f_i(\mathbf{x} + \mathbf{e}_i, t + 1) = f_i^*(\mathbf{x}, t)$$

The equilibrium distribution function f_i^{eq} is given by:

$$f_i^{eq} = w_i \rho \left[1 + 3(\mathbf{e}_i \cdot \mathbf{u}) + \frac{9}{2}(\mathbf{e}_i \cdot \mathbf{u})^2 - \frac{3}{2}|\mathbf{u}|^2\right]$$

Macroscopic Quantities and Incompressibility Fluid density and velocity are recovered from the distribution functions via:

$$\rho = \sum_i f_i, \qquad \mathbf{u} = \frac{1}{\rho} \sum_i \mathbf{e}_i f_i$$

Simulations are conducted in lattice units, with grid spacing $dx = dy = 1$ and time step $dt = 1$, yielding a lattice speed $c = 1$. The kinematic viscosity ν is related to the relaxation time τ as:

$$\nu = \frac{u_{\text{lid}} \cdot L}{Re}, \quad \tau = 3\nu + 0.5$$

To ensure incompressibility, the Mach number is kept below 0.1.

1.2 2D Lid-Driven Cavity

Implementation Choice

The code is based on the Lattice Boltzmann Method (LBM). The class `LDRIVEN` contains the principal simulation logic, including domain initialization, time evolution, and boundary conditions. The code of this particular implementation is designed to enhance computational performance, even if the coding style is not that modular and very specific for the 2D lid driven problem. We will see that the approach is different for the Wind-Tunnel like problem.

Core Components The simulation loop is driven by the `simulate()` method, which manages time stepping and output writing. The most important internal methods are:

- `feq`: computes the equilibrium distribution function based on local macroscopic quantities.
- `compute`: performs the collision and streaming steps, updating the particle distribution functions and macroscopic variables.
- `apply_boundary_conditions`: applies velocity and density conditions on domain boundaries.

Boundary Conditions Velocity boundary conditions are imposed using an equilibrium-based bounce-back approach:

- **No-slip walls** (left, right, bottom): zero velocity is imposed; density is extrapolated from adjacent fluid nodes.
- **Moving lid** (top): a constant horizontal velocity u_{lid} is applied; the vertical component is set to zero.

Memory Layout and Data Handling All fields (`velocity`, `density`, and `distribution functions`) are stored in contiguous 1D arrays for performance. Access macros (e.g., `velocity(i,j,k)`) are used to improve readability and abstract away indexing complexity. Manual memory management ensures low-level control and efficiency.

Output and Reproducibility Simulation data are periodically saved to the file `vel_data.txt` in a simple column format. A separate file `parameters.txt` logs all relevant simulation parameters to ensure result reproducibility and facilitate post-processing.

Parallelization

The compute-intensive portions of the code (streaming/collision and boundary updates) are parallelized using two different approaches: OpenMP and CUDA, to exploit computational power.

OpenMP Nested loops over the domain are collapsed to maximize parallel efficiency. To collapse them, we have used `#pragma omp parallel for collapse(n)` directives, where n is the number of `for` that have to be collapsed (usually 2).
In `LDRIVEN::apply_boundary_conditions()` since there are no nested `for` loops, we used a `#pragma omp parallel` directive which opens a parallel section. Inside the section we used `#pragma omp for` and `#pragma omp for nowait`.

CUDA CUDA parallelization is less straightforward than OpenMP parallelization, due to the explicit memory management and thread coordination required on GPUs.
We discretized the domain as a 2D grid and assigned one CUDA thread to each lattice node (i, j). The index mappings for density, velocity, and distribution functions are computed via inline **device** functions such as `idx_density`, `idx_field`, and `idx_velocity`, which convert 2D (or 3D) indices into linear memory offsets for efficient GPU access.
Three main CUDA kernels have been implemented:

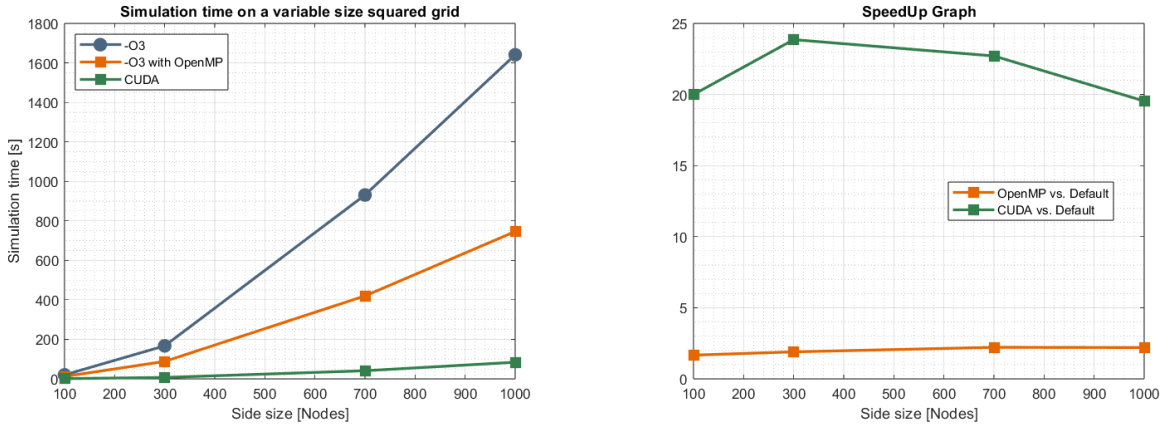
- **kernel_init:** Initializes the fields for density, velocity, and distribution functions. Each thread sets the local equilibrium values.
- **kernel_compute:** Performs the core LBM steps — collision and streaming — for each cell. The collision step is implemented using a relaxation toward equilibrium via the BGK model. Streaming is performed in a backward fashion to ensure memory coalescence and avoid race conditions.
- **kernel_apply_boundary:** Set boundary conditions on all four sides of the domain (top, bottom, left, right). The moving lid at the top boundary is set by imposing a constant velocity, while zero velocity conditions are applied on the remaining walls.

All arrays are allocated in global device memory and updated using double buffering (e.g., swapping `f` and `f2` pointers). The D2Q9 weights are stored in constant memory to optimize access. The grid is launched with a 2D block structure (`dim3 threadsPerBlock(16, 16)`), to get optimal occupancy and memory throughput.
Synchronization between kernel launches is achieved using `cudaDeviceSynchronize()` to guarantee correctness before pointer swaps and host-device memory transfers. The simulation also periodically writes velocity field back to the host for visualization and later processing by the user.

Results

The lid-driven cavity problem was simulated using three configurations: a sequential version compiled with `-O3`, a parallel version using OpenMP, and a GPU version with CUDA. Simulations remain stable up to $Re = 1000$, provided the Mach number stays low ($Ma \leq 0.1$).

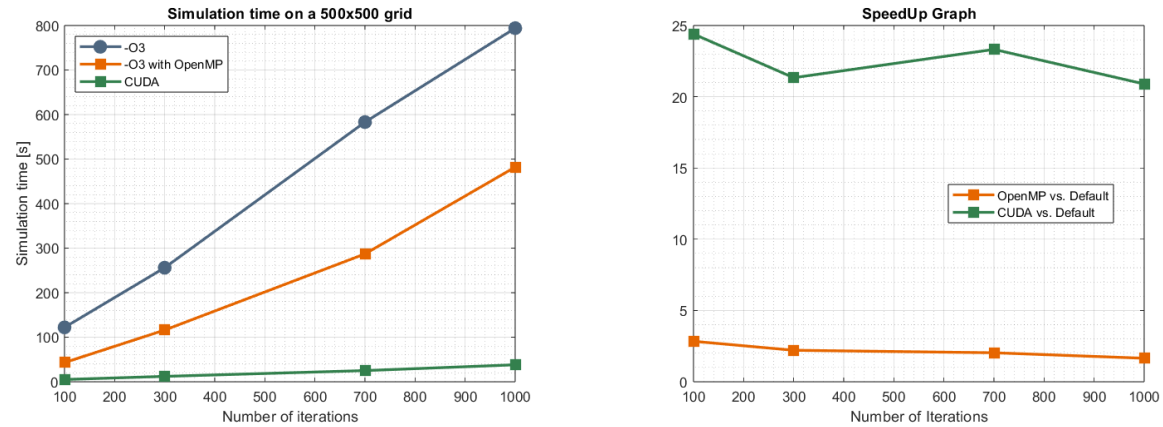
Performance: Grid Scaling Figure 1 shows how simulation time and speedup vary with increasing grid size. The CUDA version offers the best performance, achieving up to 24× speedup. OpenMP shows consistent, but more modest, gains.



(a) Simulation time (b) Speedup

Figure 1: Performance vs. grid size.

Performance: Iteration Scaling The second test (Figure 2) increases the number of iterations on a fixed 500 × 500 grid. All implementations scale linearly, but CUDA maintains its advantage. OpenMP achieves around 2.5× speedup, while CUDA exceeds 20×.



(a) Simulation time (b) Speedup

Figure 2: Performance vs. iteration count.

Validation

To validate our implementation, we compared the results of the lid-driven cavity simulation with benchmark data for Reynolds numbers $Re = 100$, 400, and 1000.
The comparison focuses on the vertical centerline velocity profile: specifically, the v_y component of the velocity along the line $x = L/2$, from the bottom ($y = 0$) to the top ($y = 1$) of the cavity.
Each plot below shows:

- Our simulation results (dashed line with markers)
- Reference values from the literature (solid line)

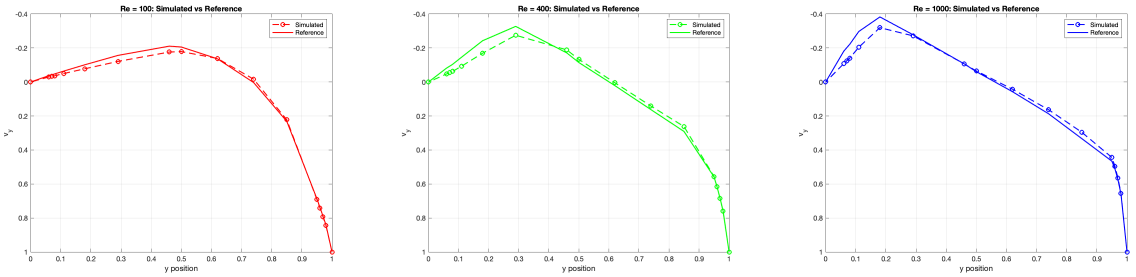


Figure 3: Vertical velocity v_y along the centerline $x = L/2$ for $Re = 100$, 400, and 1000.

For $Re = 100$, the simulation matches the reference data very closely, showing good agreement. For higher Reynolds numbers, especially $Re = 400$ and $Re = 1000$, small deviations appear, particularly near the bottom boundary. These differences are likely due to discretization effects or boundary condition approximations. Nevertheless, the simulation remains accurate and stable, confirming the validity of the method across a range of flow regimes.

1.3 Wind-Tunnel Like Problem

Another problem implemented is the Wind-Tunnel like problem, where interactions of fluid over an object can be simulated.

Implementation Choice

For this implementation the approach is totally different from the one used in the lid-driven cavity. Starting from the hands-on code of the lid-driven case, we initially attempted to insert obstacles within the domain using the same structural framework. However, this led to two main issues: the accumulation and explosion of numerical errors, and the difficulty in managing and manipulating the geometric masks used to define objects.
As a result, we decided to adopt TRT (Two Relaxation Time) as a model and more class-oriented design. The project consists of a total of 8 files, divided as follows:

- `Lattice.hpp, Lattice.cpp`
- `Node.hpp, Node.cpp`
- `Masks.hpp, Masks.cpp`
- `Auxiliary.hpp`

The architecture is based on 3 classes:

1. `template <class T> class Matrix`: contained in `Auxiliary.hpp`. In this class we define a matrix of `Node` classes which represent the domain of the simulation. It has constructor and getter operators.
2. `class Node`: declared in `Node.hpp` and developed in `Node.cpp`. Every point in the domain is defined as a `Node` object. This class has all the attributes and methods needed for the simulation: boundary conditions, equilibrium, collision, streaming, bounce-back on obstacles, and drag/lift evaluation.
3. `class Lattice`: declared in `Lattice.hpp` and developed in `Lattice.cpp`. This is the surface class of the architecture. It interacts with `run.sh` and with the user to get all the data needed for the simulation. It uses the previously mentioned classes and has a `Matrix<Node>` object that represents the domain, and a `Matrix<bool>` object that represents the obstacles in the domain. The obstacle object is a boolean matrix of dimension $NX \times NY$, with `true` indicating an obstacle cell.

All the wind-tunnel like problem runs with the `run.sh` as explained in the `Readme` of the github repository. Once compiled and runned, the program will ask you if you want to insert an object inside the fluid domain. If you, for some reason, decide to type yes[(1)] the program will ask you to choose the type of object you want to create. Here comes a nice part, since the object are created in the `Masks.cpp` file which contains all the methods to create masks and is a separate module. This means that if I would like to add a `triangular_mask` function in the future I can simply add it in `Masks.hpp` and call it inside `Lattice.cpp`. This is a modular versatile way of implementing object creation without the needs of update much of the core.

[(1)] Not obvious

Since user terminal interaction can be not worth for the reproducibility of a simulation, `Lattice.cpp` creates a file called `parameters.txt` inside the `output` folder. By using this strategy, all the parameters needed to reproduce an exact simulation are saved in a proper file. This choice has been made as a compromise between reproducibility and customizability of the simulation.

Parallelization

The parallelization here is very similar to the one done in the 2D lid driven cavity problem. We've done by writing `OpenMP` directives in certain specific points of the code. In particular, we have used:

- `#pragma omp parallel`
- `#pragma omp parallel for collapse(2)`
- `#pragma omp critical`
- `#pragma omp atomic`
- `#pragma omp single`

The main bottleneck is streaming process, where we added a `#pragma omp for collapse(2)`. Critical parts are the Lift and Drag coefficients calculations and update, where a parallel update approach leads in race conditions and diverging simulation. So, for these operations, we have used the `#pragma omp atomic` directive solving the race conditions problem. Another directive used is `#pragma omp critical`, needed in the boundary condition update inside `Lattice.cpp`. Parallelization results are shown below, in the results section.

Results

The wind-tunnel problem was simulated using a sequential implementation (`-O3`) and a parallel version with OpenMP.
Figures 4 and 5 show how the simulation time and corresponding speedup vary with increasing grid size. OpenMP demonstrates significant acceleration, with speedup rising from just under 2× to nearly 2.9× for the largest tested case.

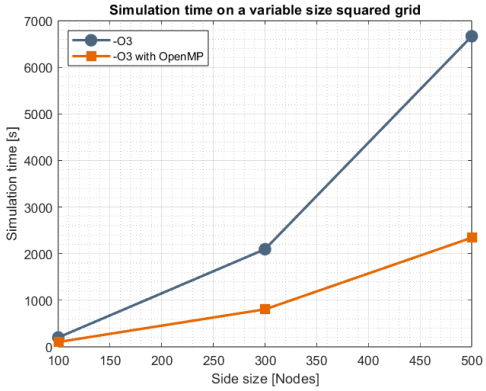


Figure 4: Simulation time

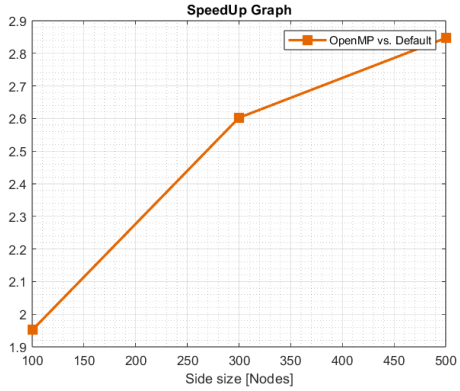


Figure 5: OpenMP speedup

2 LBM for 3D Lid-Driven Cavity

The extension of the lid–driven cavity simulation from two to three dimensions introduces substantial challenges, notably:

- **Increased computational cost:** a cubic growth in cell count ($N_x N_y N_z$) and $Q = 19$ discrete distributions per cell.
- **Memory footprint and data locality:** contiguous storage of two full distribution arrays (f and f_{new}) of size $N_x N_y N_z \times 19$ demands careful indexing.
- **Complex boundary conditions:** six moving or stationary walls require bespoke handling of incoming and outgoing populations.

2.1 Description

The transition from 2D to 3D entails:

- Cubic growth in cell count ($N_x N_y N_z$) and $Q = 19$ populations per node.
- Memory and cache-locality challenges for two large arrays of size $19 N_x N_y N_z$.
- Six walls with differing motion/velocity profiles.

Notation and Variables

Symbol	Description
N_x, N_y, N_z	number of lattice nodes in x, y, z directions
Q	number of discrete velocities (19 for D3Q19)
$f_i(\mathbf{x}, t)$	distribution function in direction i at $(\mathbf{x}, t)$
$f_i^*(\mathbf{x}, t)$	post–collision distribution
f_i^{eq}	equilibrium distribution
$\mathbf{e}_i$	discrete velocity vectors, $i = 0, \ldots, 18$
w_i	lattice weights
τ	relaxation time for symmetric mode
$\omega^+ = 1/\tau$	symmetric relaxation rate
ω^-	anti-symmetric relaxation rate
$\bar{\imath}$	index opposite to i ($\mathbf{e}_{\bar{\imath}} = -\mathbf{e}_i$)
ρ	macroscopic density
$\mathbf{u}$	macroscopic velocity
Δt	time step (set to unity)

D3Q19

The D3Q19 model defines a three-dimensional, nineteen-velocity discrete set.

- **Velocity directions:** One rest velocity ($i = 0$), six face-centered vectors along coordinate axes, and twelve edge-centered diagonals, providing second- and fourth-order isotropy conditions.

We employ the minimal description:

$$\{\mathbf{e}_i\}_{i=0}^{18} \in \{(0,0,0),\ (\pm1,0,0), (0,\pm1,0), (0,0,\pm1), (\pm1,\pm1,0), \dots\}, \quad w_i \in \left\{\tfrac{1}{3}, \tfrac{1}{18}, \tfrac{1}{36}\right\},$$

with equilibrium

$$f_i^{\mathrm{eq}} = \rho\, w_i \Big[1 + 3(\mathbf{e}_i \cdot \mathbf{u}) + \tfrac{9}{2}(\mathbf{e}_i \cdot \mathbf{u})^2 - \tfrac{3}{2}|\mathbf{u}|^2\Big].$$

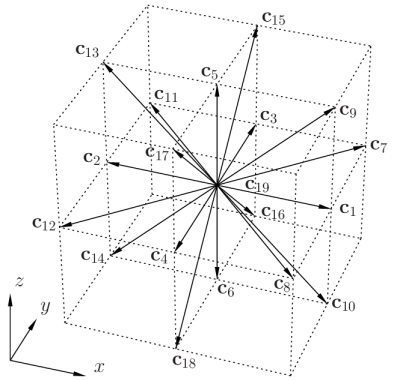


Figure 6: Vector representation of D3Q19

Collision and Streaming

To advance from t to $t + \Delta t$, we first collide populations at each node, then stream them along lattice links. We use the Two–Relaxation–Time (TRT) model to improve stability by treating even and odd modes separately.

1. Decomposition into even/odd modes:

$$f_i^{\pm} = \tfrac{1}{2}\big(f_i \pm f_{\bar{i}}\big), \qquad f_i^{\mathrm{eq},\pm} = \tfrac{1}{2}\big(f_i^{\mathrm{eq}} \pm f_{\bar{i}}^{\mathrm{eq}}\big).$$

2. Collision (relaxation):

$$f_i^{+*} = f_i^{+} - \omega^{+}\big(f_i^{+} - f_i^{\mathrm{eq},+}\big), \quad f_i^{-*} = f_i^{-} - \omega^{-}\big(f_i^{-} - f_i^{\mathrm{eq},-}\big).$$

3. Reconstruction:

$$f_i^{*} = f_i^{+*} + f_i^{-*}.$$

4. Streaming:

$$f_i(\mathbf{x} + \mathbf{e}_i\Delta t,\ t + \Delta t) = f_i^{*}(\mathbf{x}, t).$$

Boundary Conditions

We combine bounce-back for no-slip walls with Zou–He for prescribed velocities:

Stationary walls: standard halfway bounce-back

$$f_i(\mathbf{x}_w + \tfrac{1}{2}\mathbf{e}_i, t + \Delta t) = f_{\bar{i}}(\mathbf{x}_w + \tfrac{1}{2}\mathbf{e}_i, t).$$

Moving lid (Zou–He): at $y = N_y - 1$ with $\mathbf{u}_{\text{lid}} = (U, 0, 0)$, solve for unknown incoming f_i

$$\rho_w = \frac{1}{1 - u_x} \sum_{i \in \mathcal{B}} f_i, \quad f_i = f_i^{\text{eq}}(\rho_w, \mathbf{u}_{\text{lid}}) + f_{\bar{i}}^{\text{neq}}(\rho_w, \mathbf{u}_{\text{lid}}),$$

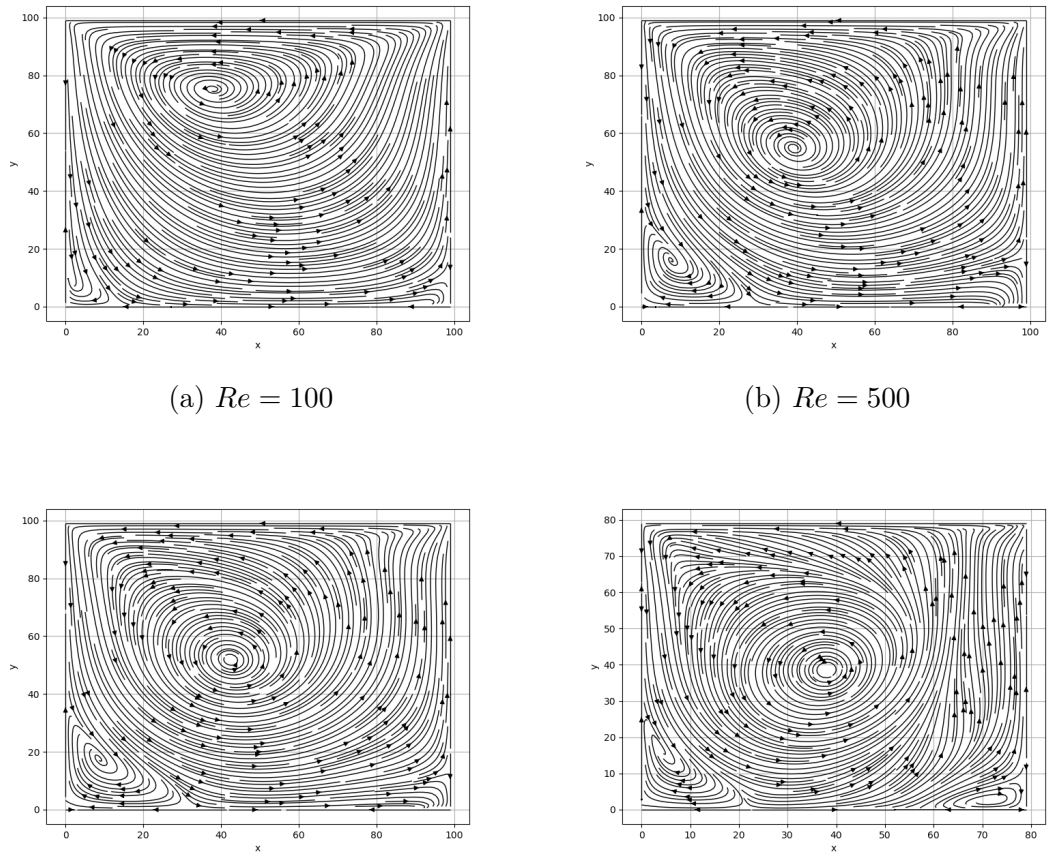
where $\mathcal{B}$ indexes known populations and $f^{\text{neq}} = f - f^{\text{eq}}$.

Indexing: $\text{idx}(x, y, z) = x + N_x\, y + N_x N_y\, z$, $\text{idxi} = 19\,\text{idx} + i$

2.2 Results

Flow Patterns

Streamlines provide a clear, instantaneous snapshot of the flow topology, revealing vortex centers, secondary eddies, and separation zones. We capture the fully developed steady state for each Reynolds number by plotting the streamline at the end of the simulation. These visualizations are essential to understand how inertial forces reshape the primary circulation and give rise to corner vortices.



(a) $Re = 100$ (b) $Re = 500$

(c) $Re = 1000$ (d) $Re = 3000$

Figure 7: Streamlines at iteration 10 000 for varying Reynolds numbers.

Streamline Comparison

As the Reynolds number increases from 100 through 3000, the primary circulation cell becomes progressively twisted and elongated toward the downstream wall, driven by stronger lid shear. Secondary vortices emerge first in the bottom-right corner at moderate Re (panel b), then appear in the bottom-left as inertia grows (panel c), and finally multiply into tertiary eddies near all corners at $Re = 3000$ (panel d). This cascade of vortical structures reflects the transition from a near-symmetric, single-cell flow at low inertia ($Re = 100$, panel a) to a highly complex, multi-vortex regime under dominant inertial forces. Such evolution underscores the critical role of Reynolds number in shaping three-dimensional recirculation patterns within the cavity.

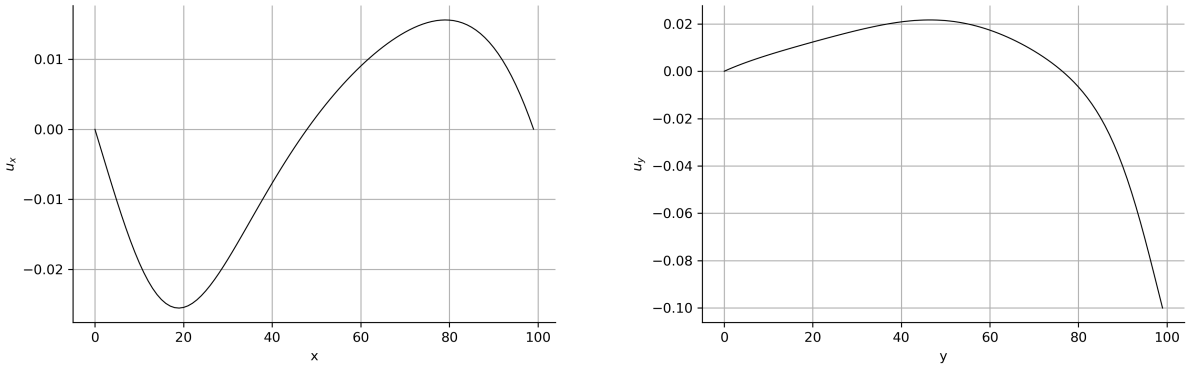
Velocity Profiles

Accurate velocity profiles along the cavity centerlines are critical to quantify boundary-layer development, recirculation strength, and overall momentum transfer. For each Reynolds number, we extract:

- Horizontal profile: $u_x(x = L/2, y, z = L/2)$
- Vertical profile: $u_y(x, y = L/2, z = L/2)$

We have tested at different Reynolds numbers to provide a broader view of the obtained results.

Note: the horizontal profile graph appears inverted compared to the validation paper (linked at the end of the section) due to flow direction conventions.

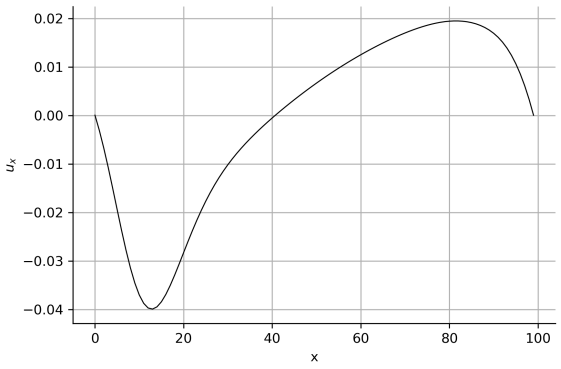


(a) Horizontal u_x at $x = L/2, z = L/2$.

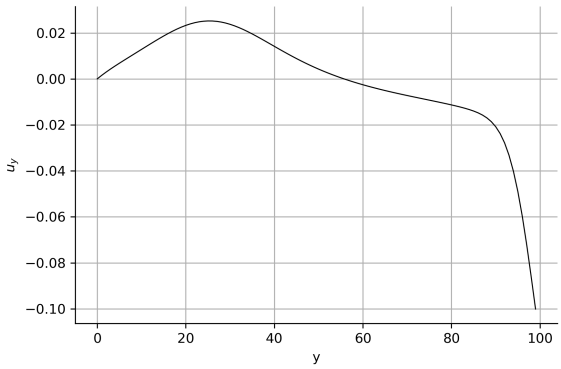
(b) Vertical u_y at $y = L/2, z = L/2$.

Figure 8: Velocity profiles for $Re = 100$.

Reynolds Number $Re = 100$ At $Re = 100$, the horizontal profile (Fig. 8a) exhibits a smooth, single-peak structure near $x \approx 80$, indicating a weak primary vortex. The vertical profile (Fig. 8b) shows a curve with maximum at $y \approx 45$, consistent with slow downward return flow.



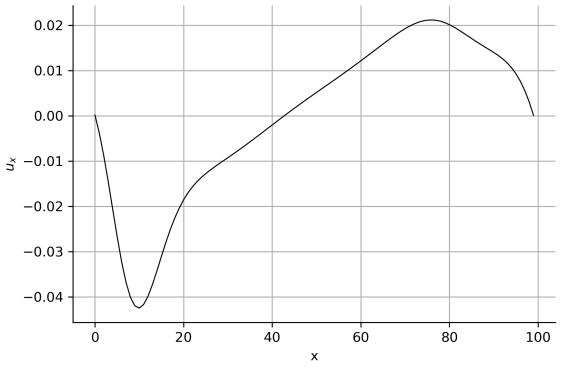
(a) Horizontal u_x at $x = L/2, z = L/2$.



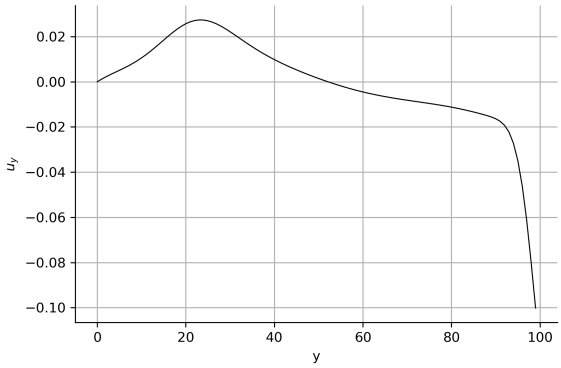
(b) Vertical u_y at $y = L/2, z = L/2$.

Figure 9: Velocity profiles for $Re = 500$.

Reynolds Number $Re = 500$ At $Re = 500$, increased Reynolds shifts the horizontal peak closer to the moving lid $x \approx 85$ and deepens the negative trough at $x \approx 15$, signaling stronger secondary vortices. The vertical profile becomes more twisted, with intensified return flow.



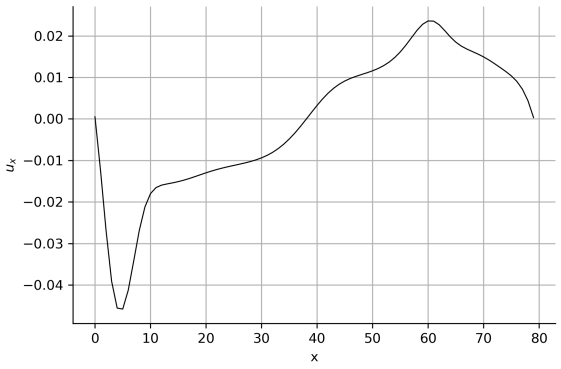
(a) Horizontal u_x at $x = L/2, z = L/2$.



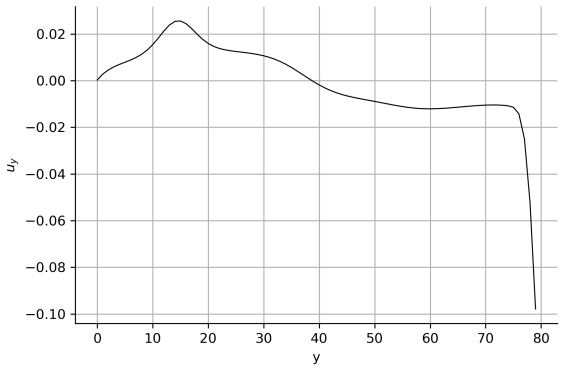
(b) Vertical u_y at $y = L/2, z = L/2$.

Figure 10: Velocity profiles for $Re = 1000$.

Reynolds Number $Re = 1000$ By $Re = 1000$, the main peak in u_x sharpens and shifts, while the negative region grows deeper—evidence of intensified corner vortices. The vertical profile shows pronounced asymmetry and larger magnitude.



(a) Horizontal u_x at $x = L/2, z = L/2$.



(b) Vertical u_y at $y = L/2, z = L/2$.

Figure 11: Velocity profiles for $Re = 3000$.

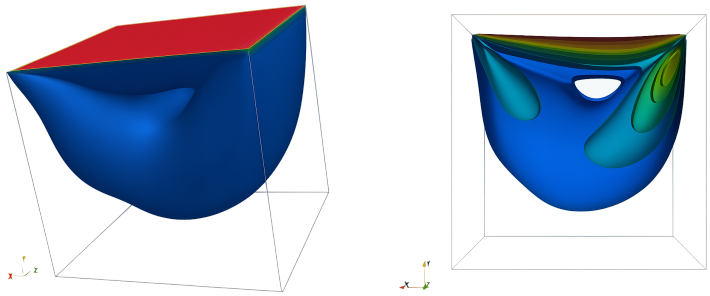
Reynolds Number $Re = 3000$ At high Reynolds ($Re = 3000$), multiple inflection points emerge in both profiles, indicating complex, multi-cell vortex breakdown and stronger secondary flows.

Cross–Reynolds Comparison Across all cases, increasing Re shifts the primary u_x peak toward the lid and amplifies return-flow minima. Similarly, u_y profiles become increasingly twisted with deeper negative excursions. These trends quantitatively capture the transition from laminar, single-vortex flow to multi-vortex regimes.

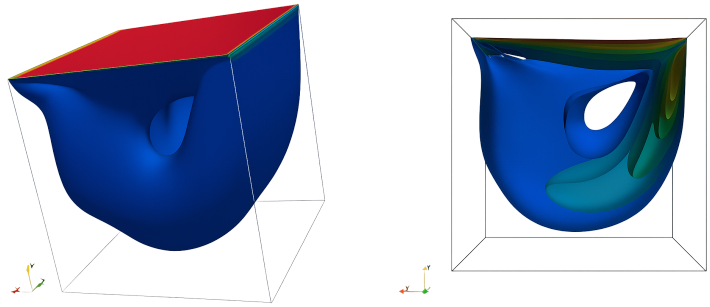
Validation All extracted profiles for $Re = 100$, 500, and 1000 have been benchmarked against lid-driven cavity data at:
`https://www.sciencedirect.com/science/article/pii/0021999187901902`

Isosurface Comparison

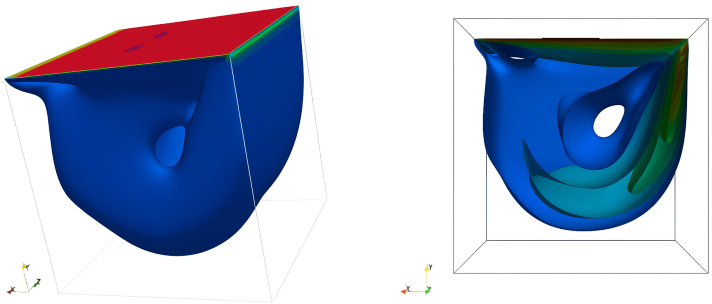
In ParaView, the velocity–magnitude field was rendered via the `Contour` filter with 10 equally spaced isovalues between the minimum and maximum values in each dataset. For sectional views, the `Slice` filter was applied at the cavity's midplanes ($x = L/2$, $y = L/2$, $z = L/2$) to extract planar cross–sections of the same variable.
Graphical comparison of these isosurfaces across Reynolds numbers provides direct insight into the three-dimensional evolution of vortical structures, highlighting changes in recirculation zones and flow separation that are not apparent from mere 2D profiles.



(a) $Re = 100$, isosurface (b) $Re = 100$, mid-plane slice



(c) $Re = 500$, isosurface (d) $Re = 500$, mid-plane slice



(e) $Re = 1000$, isosurface (f) $Re = 1000$, mid-plane slice

Figure 12: Velocity–magnitude isosurfaces (left) and corresponding mid-plane slices (right) for $Re = 100$, 500, 1000.

3 Conclusions

In the three-dimensional D3Q19 lattice–Boltzmann solver with OpenMP parallelization and VTK output for post-processing in ParaView, our approach offers several clear advantages:

- **High fidelity:** The D3Q19 stencil and two-relaxation-time model capture subtle vortex structures and maintain stability up to moderate-high Reynolds numbers.
- **Visualization:** Writing VTK files enables direct, interactive exploration in ParaView. Volume visualization, streamlines, isosurfaces, and slices can be generated without additional conversion.

However, these benefits come at a cost:

- **Large data outputs:** High-resolution 3D VTK dumps produce multi-gigabyte files, complicating storage and I/O bandwidth.
- **Computational expense:** Three-dimensional meshes of size $N_x \times N_y \times N_z$ with $Q = 19$ distributions require substantial memory and CPU time; a single run to steady state may take hours on a multi-core workstation.
- **Reynolds-number limitation:** Pushing to high Re demands finer meshes and more computational power (to simulate in a acceptable time) to preserve stability and avoid numerical divergence.

Overall, while our 3D LBM implementation successfully captures complex cavity flows up to $Re \sim 3000$, scaling to even higher Reynolds numbers or larger domains will likely require more CPU power and larger mesh to have a more precise simulation.

Note that even if it has lower dimensions, these costs also come with the 2D implementation! In particular, computational expense changes drastically from the 2D lid-driven BGK model implementation to the 2D wind-tunnel like problem with the TRT implementation. The wind-tunnel-like problem requires a big domain for a clean simulation. This happens because if the walls are too close to the object than they interfere with the simulation itself, causing oscillations in the fluid domain. Apart from that, the obtained results in both 2D and 3D configurations confirm the accuracy and flexibility of the LBM.