Chalmers University of Technology

GPU-accelerated Computational Methods using Python and CUDA

Lattice Boltzmann Method (LB2)

Author:
Christian Ispas
Jingyang Wang
Rui Cheng
Venkateswarlu Reddy Konkala

Supervisor:
Magnus Carlsson

February 16, 2026

// Abstract

The Lattice Boltzmann Method (LBM) is inherently well suited for parallel execution due to its localized collision and streaming operations. While Python-based implementations offer flexibility and rapid development, they are typically limited by CPU performance when applied to large-scale simulations. In this work, a Python LBM solver is accelerated using NVIDIA GPUs through two complementary approaches: library-based acceleration with **CuPy** and explicit kernel-based acceleration with **Numba-CUDA**.

Performance benchmarks demonstrate substantial improvements over the CPU reference implementation, achieving a maximum speedup of **212**× using CuPy and up to **1074**× using Numba-CUDA. These results highlight both the effectiveness of high-level GPU libraries for rapid acceleration and the superior performance potential of explicitly optimized CUDA kernels for compute-intensive fluid simulations.

Keywords

Lattice Boltzmann Method, GPU acceleration, CUDA, CuPy, Numba, Python, parallel computing, computational fluid dynamics

Contents

Abstract **II**

1 Introduction **2**

2 Theoretical Background **3**
2.1 The Lattice Boltzmann Approach to CFD . . . 3
2.1.1 The LBGK Collision Model . . . 4
2.1.2 Macroscopic Variables . . . 5
2.1.3 Equilibrium Distribution Function . . . 5
2.1.4 Lattice Discretization (D2Q9 and D3Q19) . . . 5
2.1.5 Enstrophy . . . 5
2.2 CPU vs GPU Approach . . . 6
2.3 Parallel Optimization Tools Used . . . 6
2.3.1 CuPy . . . 6
2.3.2 Numba . . . 7

3 Implementation **8**
3.1 CPU Implementation . . . 8
3.1.1 Algorithmic Flow . . . 8
3.2 Algorithms . . . 9
3.3 CuPy Implementation . . . 9
3.3.1 Initialization . . . 9
3.3.2 Collision and Streaming Step . . . 10
3.3.3 Key Differences from CPU Implementation . . . 10
3.4 Numba Implementation . . . 11
3.4.1 Execution Model Differences . . . 11
3.4.2 Memory Layout and Transfers . . . 11
3.4.3 Moment Computation Kernel . . . 11
3.4.4 Fused Collision and Streaming Kernel . . . 12
3.4.5 Boundary Handling . . . 13
3.4.6 Precision and Performance Considerations . . . 13
3.5 Input Model . . . 13
3.5.1 Taylor–Green Vortex . . . 13

4 Results **15**
4.1 Overview of the Performed Simulations . . . 15
4.2 Performance Analysis . . . 15

4.2.1 Execution Time Comparison 15
4.2.2 Speedup Comparison 16
4.2.3 Million Lattice Updates Per Second (MLUP/s) Comparison . 17
4.3 Accuracy and Validation of GPU Implementations 18
4.3.1 Enstrophy Evolution Comparison 18
4.3.2 Consistency Between CPU and GPU Results 20
4.4 Synopsis 21

5 Discussion **23**
5.1 Interpretation of Performance Trends 23
5.2 Framework Selection and Optimization Strategies 23
5.3 Memory-Bound Characteristics and Computational Bottlenecks . . . 23

6 Conclusion **24**

7 Outlook **25**

Bibliography **26**

1
Introduction

The **Lattice Boltzmann Method (LBM)** is a mesoscopic numerical approach for simulating fluid flows that is particularly well suited to parallel computing. Its algorithm is composed of local and repetitive operations—collision and streaming—that are performed independently at each lattice node, making LBM highly scalable on modern parallel hardware architectures.

High-level programming languages such as **Python** offer significant advantages for rapid development, readability, and flexibility when implementing complex numerical methods. However, conventional CPU-based implementations using **NumPy** often become performance-limited when applied to large three-dimensional domains or high-resolution simulations.

The primary objective of this project is to accelerate a Python-based LBM solver by offloading its computational core from the CPU to the GPU using the **NVIDIA CUDA** platform. Two complementary acceleration strategies are investigated:

1. **Library-based GPU acceleration** using **CuPy**, which enables NumPy-like array operations to be executed transparently on the GPU.
2. **Kernel-based GPU acceleration** using **Numba**, allowing explicit implementation of CUDA kernels and fine-grained control over parallel execution and memory access.

The performance of these approaches is evaluated through large-scale benchmark simulations, with particular emphasis on scalability, memory transfer overheads, and hardware utilization. All experiments are conducted on modern **NVIDIA H100 GPU architectures**, providing insight into the effectiveness of high-level versus low-level GPU programming paradigms for LBM.

2

Theoretical Background

2.1 The Lattice Boltzmann Approach to CFD

Classical computational fluid dynamics (CFD) methods are typically based on the direct numerical solution of the Navier–Stokes equations, which describe the macroscopic conservation of mass and momentum in fluid flows [1]. While highly accurate, these methods often involve the solution of coupled, nonlinear partial differential equations and require complex numerical schemes, such as pressure–velocity coupling and global Poisson solvers.

In their incompressible form, the Navier–Stokes equations consist of a kinematic constraint enforcing mass conservation and a momentum balance accounting for inertial, pressure, viscous, and body-force effects. The velocity field $\mathbf{u}(\mathbf{x}, t)$ is required to satisfy the divergence-free condition

$$\nabla \cdot \mathbf{u} = 0, \tag{2.1}$$

where $\mathbf{u}(\mathbf{x}, t)$ denotes the velocity field.
The conservation of momentum for an incompressible Newtonian fluid is given by

$$\rho \left(\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} \right) = -\nabla p + \mu \nabla^2 \mathbf{u} + \mathbf{F}, \tag{2.2}$$

In numerical implementations, the incompressible Navier–Stokes equations require the enforcement of the divergence-free constraint on the velocity field. This typically leads to the solution of a global Poisson equation for the pressure at every time step in order to couple pressure and velocity fields. The resulting elliptic problem introduces long-range data dependencies and global communication patterns, which are computationally expensive and difficult to scale efficiently. Additionally, the nonlinear convective term $(\mathbf{u} \cdot \nabla)\mathbf{u}$ introduces strong coupling between velocity components, requiring iterative solution procedures and stabilization schemes that increase computational cost and limit parallel scalability.

These characteristics can limit computational efficiency and scalability, particularly on massively parallel hardware architectures such as graphics processing units (GPUs). The Lattice Boltzmann Method (LBM) provides an alternative formulation of fluid dynamics that circumvents these challenges by avoiding the direct solution of the incompressible Navier–Stokes equations. In particular, LBM eliminates the

need for global pressure–velocity coupling and Poisson solvers by evolving particle distribution functions through purely local collision and streaming operations.
The Lattice Boltzmann Method (LBM) provides an alternative formulation of fluid dynamics that circumvents many of these challenges by avoiding the direct solution of the incompressible Navier–Stokes equations. In particular, LBM eliminates the need for global pressure–velocity coupling and Poisson solvers by evolving particle distribution functions locally in space and time. Macroscopic flow quantities such as density and velocity are recovered from local moments of these distributions, while incompressibility is satisfied approximately through a low-Mach-number formulation. The Lattice Boltzmann Method (LBM) provides an alternative formulation of fluid dynamics that circumvents many of these challenges [2].

Instead of solving the Navier–Stokes equations directly, LBM models fluid flow through the evolution of particle distribution functions on a discrete lattice. Thus the model is defined on the mesoscopic scale, which lies between the microscopic scale (molecular interactions) and the macroscopic scale (continuum mechanics) [3]. Macroscopic fluid behavior emerges naturally from local interactions and streaming processes, resulting in an algorithm composed almost entirely of local, explicit and repetitive operations. This structure makes LBM especially well suited for GPU acceleration, as it minimizes global data dependencies and enables efficient exploitation of fine-grained parallelism.

The fundamental variable in LBM is the particle distribution function, denoted as $f_i(\mathbf{x}, t)$, which represents the probability of finding a particle at position $\mathbf{x}$ and time t with a discrete velocity $\mathbf{c}_i$. The evolution of this function is governed by the discrete Lattice Boltzmann equation:

$$f_i(\mathbf{x} + \mathbf{c}_i \Delta t, t + \Delta t) - f_i(\mathbf{x}, t) = \Omega_i(f), \tag{2.3}$$

where Δt is the time step and Ω_i is the collision operator representing the local redistribution of particle populations [4].

The algorithm is typically split into two distinct steps for computational efficiency, particularly in GPU implementations [5]:

1. **Collision:** Particles interact locally at each node, relaxing towards an equilibrium state.
2. **Streaming:** Post-collision particles move to neighboring nodes according to their velocity directions $\mathbf{c}_i$.

2.1.1 The LBGK Collision Model

The most common collision operator, used in the standard implementation of this solver, is the Bhatnagar-Gross-Krook (BGK) approximation [6]. It simplifies the complex collision integral by assuming the distribution function relaxes towards a local equilibrium f_i^{eq} at a single rate τ:

$$\Omega_i(f) = -\frac{1}{\tau} \left[f_i(\mathbf{x}, t) - f_i^{eq}(\mathbf{x}, t) \right]. \tag{2.4}$$

Here, τ is the dimensionless relaxation time, which is directly linked to the kinematic viscosity ν of the fluid and the lattice speed of sound c_s:

$$\nu = c_s^2 \left(\tau - \frac{1}{2}\right) \Delta t. \tag{2.5}$$

2.1.2 Macroscopic Variables

One of the advantages of LBM is the straightforward recovery of macroscopic fluid variables from the mesoscopic moments of the particle distribution functions. The fluid density ρ and momentum density $\rho\mathbf{u}$ are calculated as:

$$\rho = \sum_i f_i, \tag{2.6}$$

$$\mathbf{u} = \frac{1}{\rho} \sum_i f_i \mathbf{c}_i. \tag{2.7}$$

2.1.3 Equilibrium Distribution Function

The equilibrium distribution function f_i^{eq} is a discretized form of the Maxwell-Boltzmann distribution. It depends on the local macroscopic density ρ and velocity $\mathbf{u}$. For low Mach number flows, it is approximated as:

$$f_i^{eq}(\rho, \mathbf{u}) = w_i \rho \left[1 + \frac{\mathbf{u} \cdot \mathbf{c}_i}{c_s^2} + \frac{(\mathbf{u} \cdot \mathbf{c}_i)^2}{2c_s^4} - \frac{|\mathbf{u}|^2}{2c_s^2}\right], \tag{2.8}$$

where w_i are lattice-specific weighting factors and c_s is the lattice speed of sound (typically $c_s = 1/\sqrt{3}$) [5].

2.1.4 Lattice Discretization (D2Q9 and D3Q19)

The domain is discretized into a lattice defined by the notation D_nQ_m, where n is the spatial dimension and m is the number of discrete velocities.
For 2D simulations, the **D2Q9** model is standard, comprising a center particle, four orthogonal neighbors, and four diagonal neighbors. For 3D simulations (as referenced in the implementation of Ellenius), the **D3Q19** model is commonly utilized to balance accuracy and computational cost [5].

2.1.5 Enstrophy

Enstrophy is a scalar quantity that measures the rotational content of a fluid flow and is defined in terms of the vorticity field [7],

$$\boldsymbol{\omega}(\mathbf{x}, t) = \nabla \times \mathbf{u}(\mathbf{x}, t). \tag{2.9}$$

The global enstrophy is

$$\mathcal{E}(t) = \int_\Omega |\boldsymbol{\omega}(\mathbf{x}, t)|^2, d\Omega, \tag{2.10}$$

where Ω is the computational domain.
For incompressible viscous flows, the dissipation rate is proportional to enstrophy via the kinematic viscosity ν:

$$\varepsilon(t) = \nu, \mathcal{E}(t). \tag{2.11}$$

In this study, the temporal evolution of enstrophy is used to validate that the implemented Lattice Boltzmann model reproduces physically consistent flow behavior.

2.2 CPU vs GPU Approach

Traditional central processing units (CPUs) are designed to deliver strong single-thread performance and low-latency execution for a wide range of tasks. They typically consist of a small number of highly optimized cores, each capable of executing complex instruction streams with advanced control logic, large caches, and sophisticated branch prediction. This architecture makes CPUs well suited for serial workloads, irregular memory access patterns, and tasks that require frequent synchronization or complex control flow.

In contrast, graphics processing units (GPUs) are optimized for throughput-oriented computing. Modern GPUs consist of thousands of simpler cores organized into streaming multiprocessors, allowing them to execute a vast number of lightweight threads concurrently. This architectural design favors data-parallel workloads in which the same operation is applied to many independent data elements. As a result, GPUs achieve significantly higher floating-point throughput and memory bandwidth compared to CPUs, albeit at the cost of increased latency and reduced flexibility in control flow.

The tradeoff between CPUs and GPUs [8] becomes particularly relevant for computational fluid dynamics methods such as the Lattice Boltzmann Method. LBM algorithms are dominated by local, repetitive operations applied independently at each lattice node, with minimal global data dependencies. While such computations can be efficiently executed on CPUs using vectorization and multi-threading, the limited number of cores constrains scalability for large problem sizes. GPUs, on the other hand, can map each lattice node to an individual thread, enabling massive parallel execution and substantially reducing overall runtime.

2.3 Parallel Optimization Tools Used

2.3.1 CuPy

CuPy provides a high-level interface for GPU acceleration by implementing a NumPy-compatible array library that executes operations on NVIDIA GPUs using CUDA [9]. Rather than primarily exposing kernel definitions, CuPy leverages pre-optimized CUDA kernels and GPU libraries such as cuBLAS, cuFFT, and cuDNN to perform

parallel computations transparently. This abstraction allows developers to accelerate numerical workloads with minimal code modification, often by replacing NumPy with CuPy.

Parallelism in CuPy is implicitly managed by the library, with GPU threads and memory hierarchies handled automatically during kernel execution. Operations are launched asynchronously, enabling efficient overlap of computation and memory transfers when properly managed.

By relying on CUDA-aware memory management and just-in-time kernel compilation, CuPy achieves high performance while maintaining portability across different NVIDIA GPU architectures. This makes it well-suited for large-scale numerical computations where ease of development and execution efficiency are both critical.

2.3.2 Numba

The primary unit of parallelization in Numba is the CUDA kernel, which represents an explicitly parallel function compiled just-in-time and executed concurrently by a large number of GPU threads. These kernels are written using Python syntax and are decorated to indicate their execution on the GPU, allowing developers to express fine-grained parallelism while remaining within the Python ecosystem. Each kernel invocation is mapped to a grid of thread blocks, with individual threads responsible for executing specific portions of the computation.

Numba utilizes the LLVM compiler toolchain to translate Python code into optimized machine instructions at runtime [10]. This just-in-time compilation enables architecture-specific optimizations, such as instruction scheduling and register allocation, while preserving portability across different NVIDIA GPU architectures. As a result, the same kernel code can be executed efficiently on multiple GPU generations without modification.

Unlike higher-level GPU abstraction libraries, Numba requires explicit management of thread indexing, memory access patterns, and kernel launch configurations. Developers must carefully define grid and block dimensions, handle global and shared memory usage, and minimize memory access latency to achieve optimal performance. This low-level control provides significant flexibility and performance potential, making Numba particularly suitable for custom algorithms, irregular data structures, and performance-critical workloads where fine-tuned GPU optimization is required.

3

Implementation

This chapter describes the implementation logic behind our GPU-parallelized Lattice Boltzmann solver.[1]

3.1 CPU Implementation

The CPU-based Lattice-Boltzmann Method leverages `NumPy` for efficient array operations, enabling rapid evaluation of lattice updates without explicit element-wise loops. The implementation adopts an object-oriented structure, where the lattice dynamics, discrete velocity stencils, and simulation workflow are modularized into separate classes. The `Lattice` class manages the evolving state variables—including particle distributions, macroscopic density, and velocity fields—while the `Stencil` class encodes the discrete velocity directions and associated weights used in collision and streaming steps. Ultimately the CPU code serves as a reference implementation and facilitates transforming the algorithm into a parallelized variant, such that it may be ported onto GPU accelerating frameworks.

3.1.1 Algorithmic Flow

The time-stepping procedure can be summarized as follows:

1. **Initialization:** Compute the equilibrium distribution function for all lattice nodes based on the initial macroscopic fields.
2. **Streaming:** Propagate the particle distribution functions along discrete lattice directions.
3. **Moment Computation:** Calculate macroscopic density and velocity from the updated populations.
4. **Collision (Relaxation):** Relax the distributions towards equilibrium using the BGK operator.
5. **Repeat:** Iterate the streaming, moment computation, and collision steps for each time step.

[1] Source code available at: `https://github.com/Chrisman2003/Lattice_boltzmann`

3.2 Algorithms

Algorithm 1 High-Level Lattice-Boltzmann Time-Stepping

1: **for** $t = 0$ to T_{end} **do**
2: Compute local density: $\rho(\mathbf{x}) = \sum_i f_i(\mathbf{x})$
3: Compute local velocity: $\mathbf{u}(\mathbf{x}) = \sum_i f_i(\mathbf{x})\mathbf{c}_i/\rho(\mathbf{x})$
4: Compute equilibrium distribution: $f_i^{\text{eq}}(\mathbf{x})$ using local ρ and $\mathbf{u}$
5: Update populations via collision: $f_i(\mathbf{x}) \leftarrow f_i(\mathbf{x}) - \omega(f_i(\mathbf{x}) - f_i^{\text{eq}}(\mathbf{x}))$
6: Stream populations along lattice directions: $f_i(\mathbf{x} + \mathbf{c}_i) \leftarrow f_i(\mathbf{x})$
7: **end for**

Algorithm 2 High-Level Collision Procedure

1: Compute local density: $\rho(\mathbf{x}) = \sum_i f_i(\mathbf{x})$
2: Compute local velocity: $\mathbf{u}(\mathbf{x}) = \sum_i f_i(\mathbf{x})\mathbf{c}_i/\rho(\mathbf{x})$
3: Compute equilibrium distribution: $f_i^{\text{eq}}(\mathbf{x})$
4: Update populations via collision: $f_i(\mathbf{x}) \leftarrow f_i(\mathbf{x}) - \omega(f_i(\mathbf{x}) - f_i^{\text{eq}}(\mathbf{x}))$

3.3 CuPy Implementation

The CuPy implementation of the Lattice-Boltzmann Method (LBM) is designed to fully leverage NVIDIA GPUs for parallel computation. While it mirrors the conceptual structure of the CPU version, key modifications allow all major arrays and operations to reside on the GPU, enabling high-performance, massively parallel execution. Unlike explicit CUDA programming, the kernels in this implementation are launched implicitly by CuPy functions such as `cp.sum`, `cp.tensordot`, and array operations. To maximize performance, all arrays are allocated once at initialization, avoiding repeated host-device transfers. The calculated results must inevitably also be copied back to the CPU at the end of the GPU computation.

3.3.1 Initialization

The equilibrium distribution function is computed fully on the GPU using parallelized operations:

- The dot product between local velocity and lattice directions, $\mathbf{c}_i \cdot \mathbf{u}(\mathbf{x})$, is evaluated using `cp.tensordot`.
- The squared velocity magnitude, $|\mathbf{u}(\mathbf{x})|^2$, is computed using `cp.sum` with broadcasting across lattice directions.
- Populations are initialized as entirely on GPU memory, without intermediate host computations.

3.3.2 Collision and Streaming Step

Listing 3.1: GPU-based initialization, collision and streaming using CuPy model

```
def __init__(self, n, stencil: Stencil):
    self.stencil = stencil
    self.stencil_c = cp.asarray(stencil.c)
    self.stencil_w = cp.asarray(stencil.w)

    self.f = cp.zeros((n + (stencil.q,)), dtype=cp.float64)
    self.u = cp.zeros((n + (stencil.d,)), dtype=cp.float64)
    self.rho = cp.ones(n, dtype=cp.float64)
    self.gamma = 1.4

def collision(self, omega):
    self.density()
    self.velocity()
    self.f -= omega * (self.f - self.feq())

def streaming(self):
    axis = tuple([i for i in range(self.stencil.d)])
    idx = (slice(None),) * self.stencil.d
    for iq in range(self.stencil.q):
        self.f[idx + (iq,)] = cp.roll(
            self.f[idx + (iq,)],
            self.stencil.c[iq],
            axis=axis
        )
```

The BGK collision operator is applied in-place on GPU arrays:

- Compute density: $\rho(\mathbf{x}) = \sum_i f_i(\mathbf{x})$ using `cp.sum`.
- Compute velocity: $\mathbf{u}(\mathbf{x}) = \sum_i f_i(\mathbf{x})\mathbf{c}_i/\rho(\mathbf{x})$ using `cp.tensordot`.
- Relax populations toward equilibrium: $f_i(\mathbf{x}) \leftarrow f_i(\mathbf{x}) - \omega(f_i - f_i^{\mathrm{eq}})$, performed elementwise across the GPU.

This contrasts with the CPU implementation, where `NumPy` arrays are manipulated on the host and loops or dot products are executed sequentially or vectorized only across CPU cores.

Population propagation is performed entirely on the GPU using `cp.roll`:

- Each lattice direction is shifted along its velocity vector $\mathbf{c}_i$ simultaneously for all nodes.
- Unlike the CPU `np.roll`, this GPU operation launches an implicit CUDA kernel for each direction, providing strong parallelism.
- No explicit Python loops over nodes are required, which eliminates significant overhead for large lattices.

3.3.3 Key Differences from CPU Implementation

- **Memory Residency:** All primary arrays are fully GPU-resident, while the CPU version resides entirely in main memory.

- **Parallelism:** Dot products, sums, and array rolls execute as CUDA kernels in parallel, replacing sequential or vectorized CPU computations.
- **Streaming Implementation:** GPU uses `cp.roll` for simultaneous lattice shifts, whereas CPU uses `np.roll` with host-based array manipulation.

3.4 Numba Implementation

The Numba implementation differs fundamentally from both the pure CPU (NumPy) and CuPy implementations in how parallelism is expressed and controlled. Rather than relying on vectorized array operations, Numba uses explicitly defined CUDA kernels to expose fine-grained parallelism at the level of individual lattice nodes.

3.4.1 Execution Model Differences

Unlike NumPy and CuPy, where array operations implicitly map to optimized backend routines, the Numba implementation requires the explicit definition of GPU kernels using `@cuda.jit`. Each kernel is executed by a grid of CUDA threads, where each thread is responsible for updating a single lattice node (i, j, k).
This approach provides:

- Explicit control over thread indexing via `cuda.grid(3)`
- Manual handling of boundary conditions
- Fusion of multiple algorithmic steps into a single kernel

3.4.2 Memory Layout and Transfers

In contrast to the CuPy implementation—where GPU memory allocation and data movement are handled implicitly through array creation and operation dispatch—the Numba implementation requires explicit host-to-device memory transfers. Arrays initialized as `NumPy` arrays on the CPU must be transferred to the GPU using `cuda.to_device` prior to kernel execution. Likewise, data produced on the GPU must be transferred back to host memory explicitly, using the `copy_to_host()` operation, which copies the contents of a device-resident array back into a `NumPy` array on the CPU.

As a result, memory movement in the Numba implementation is explicit and programmer-controlled, in contrast to the largely transparent memory management employed by CuPy.

3.4.3 Moment Computation Kernel

Macroscopic quantities are computed using a dedicated CUDA kernel:

```
@cuda.jit
def moments_kernel(...)
```

Each GPU thread computes density and velocity for exactly one lattice node by looping over all lattice directions q. This contrasts with:

- the CPU implementation, which uses Python loops and vectorized NumPy operations, and
- the CuPy implementation, which relies on `cp.sum` and `cp.tensordot`.

Here, reduction over lattice directions is performed manually within each thread, trading higher arithmetic intensity for reduced memory traffic and improved data locality.

3.4.4 Fused Collision and Streaming Kernel

Listing 3.2: Fused collision and streaming CUDA kernel for Numba model

```
@cuda.jit(fastmath=True)
def collision_and_stream_kernel(
    f_in, f_out, rho, u, c, w,
    q, omega, inv_cs2, inv_cs4,
    nx, ny, nz
):
    i, j, k = cuda.grid(3)
    if i >= nx or j >= ny or k >= nz:
        return

    ux, uy, uz = u[i, j, k]
    rho_ijk = rho[i, j, k]
    uu = ux*ux + uy*uy + uz*uz
    common = -0.5 * inv_cs2 * uu

    for iq in range(q):
        ci0, ci1, ci2 = c[iq]
        uc = ux*ci0 + uy*ci1 + uz*ci2
        feq = w[iq] * rho_ijk * (
            1.0 + inv_cs2*uc + 0.5*inv_cs4*uc*uc + common
        )

        f_star = f_in[i, j, k, iq] - omega * (f_in[i, j, k,
           iq] - feq)
        it = (i + ci0) % nx
        jt = (j + ci1) % ny
        kt = (k + ci2) % nz
        f_out[it, jt, kt, iq] = max(f_star, 1e-16)
```

A key difference in the Numba implementation is the fusion of the collision and streaming steps into a single kernel:

```
@cuda.jit
def collision_and_stream_kernel(...)
```

Each thread:

1. Computes the equilibrium distribution locally
2. Applies the BGK collision operator
3. Streams the post-collision value directly to the destination lattice node

This fused approach avoids intermediate global memory writes that are present in both the CPU and CuPy implementations, where collision and streaming are implemented as separate computational stages. In the CuPy version, although all operations reside on the GPU, collision, moment computation, and streaming are executed as distinct kernel launches, each producing intermediate arrays in global device memory.

3.4.5 Boundary Handling

Periodic boundary conditions are enforced explicitly inside the kernel by manually wrapping indices:

```
if itarget < 0: itarget += nx
elif itarget >= nx: itarget -= nx
```

This logic is handled implicitly in the CPU and CuPy implementations via `np.roll` and `cp.roll`, respectively. In Numba, this explicit handling is required due to the absence of high-level array shift operations inside CUDA kernels.

3.4.6 Precision and Performance Considerations

The Numba implementation uses `float32` precision and enables `fastmath=True` in CUDA kernels. This allows the compiler to apply aggressive floating-point optimizations, improving performance at the cost of strict IEEE compliance. In contrast:

- The CPU and CuPy implementations use `float64`,
- Numerical optimizations are handled internally by NumPy and CuPy libraries.

3.5 Input Model

3.5.1 Taylor–Green Vortex

The Taylor–Green vortex is a three-dimensional, unsteady flow configuration widely used to study the transition from ordered motion to turbulence. It is particularly suitable for evaluating the performance of parallel solvers due to its homogeneous structure and well-defined temporal evolution.

In the CPU native implementation, the vortex is initialized on a cubic $n \times n \times n$ lattice using a D3Q19 stencil. The lattice resolution is set to $n = 129$, corresponding to a computational domain with 129^3 lattice nodes. The reference Mach number is set to $\mathrm{Ma} = 0.1$ and the Reynolds number to $\mathrm{Re} = 1600$. The initial velocity field is defined analytically using trigonometric functions, producing a divergence-free flow with periodic boundary conditions in all directions. The density field is initialized uniformly, corresponding to a constant reference pressure.

The flow evolution is governed by the Mach and Reynolds numbers, which determine the characteristic velocity scale, kinematic viscosity, and relaxation time. Time is

nondimensionalized using the characteristic turnover time of the vortex, and simulation output is recorded at regular intervals to capture the decay of kinetic energy and the development of small-scale structures.

An Isentropic Vortex model is used as a validation benchmark for the CPU and GPU implementations; but it is not suitable as a performance demonstration because of its smaller, primarily two-dimensional domain and reduced computational load. The fully three-dimensional Taylor–Green vortex serves as the demanding test case for assessing both numerical accuracy and the GPU performance of the Lattice-Boltzmann implementation.

4

Results

4.1 Overview of the Performed Simulations

This chapter presents the results obtained from the GPU acceleration of the Lattice Boltzmann method originally implemented for CPU architectures. The CPU-based implementation serves as the reference solution and is compared against two GPU-based approaches developed using CuPy and Numba-CUDA.
The numerical experiments are designed to evaluate both the computational performance and the numerical accuracy of the GPU implementations. Performance is assessed through a comparison of execution times and the resulting speedup factors relative to the CPU version. To verify the correctness of the GPU-accelerated solutions, the temporal evolution of enstrophy is computed and compared across all implementations.
The Taylor–Green vortex test case is employed as a benchmark problem to validate the physical consistency of the simulations. In addition to quantitative metrics, qualitative flow features are examined through the visualization of the evolution of iso-surfaces of the Q-criterion. The results presented in this chapter provide a comprehensive assessment of the effectiveness of GPU acceleration using CuPy and Numba-CUDA for the Lattice Boltzmann method.

4.2 Performance Analysis

This section evaluates the computational performance of the GPU-accelerated Lattice Boltzmann implementations developed using CuPy and Numba-CUDA. The performance of these implementations is compared against a single-core CPU baseline and a multi-core CPU version executed using 32 cores. The GPU-accelerated simulations were performed on an NVIDIA H100 GPU. The comparison is carried out for problem sizes ranging from 10^3 to 10^7 lattice cells using identical numerical parameters to ensure a fair and consistent evaluation.

4.2.1 Execution Time Comparison

Table 4.1 presents the execution times obtained from the single-core CPU implementation, the multi-core CPU version, and the GPU-accelerated implementations using CuPy and Numba-CUDA. As the number of lattice cells increases, the execution time of the CPU-based implementations grows rapidly. The multi-core CPU version achieves a noticeable reduction in execution time compared to the single-core

CPU implementation; however, the improvement remains limited for large problem sizes.

Table 4.1: Execution time comparison between CPU, multi-core CPU, CuPy, and Numba-CUDA implementations for different numbers of lattice cells.

Number of Cells	CPU (s)	CPU Multi-core (s)	CuPy (s)	Numba-CUDA (s)
10^3	0.5	0.08	0.903	0.54
10^4	8.2	0.4	1.873	0.60
10^5	178	10	4.229	0.71
10^6	3943	116	30.505	8.7
10^7	130000	4100	613.7	121

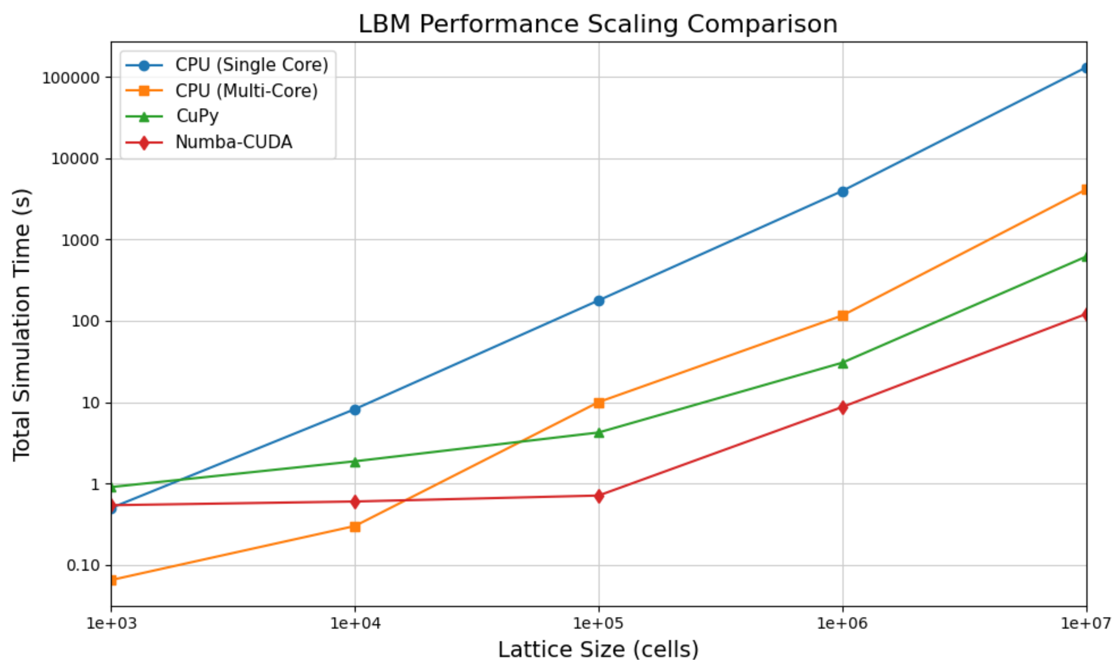

Figure 4.1: Execution time as a function of the number of lattice cells for different computational approaches.

In contrast to the CPU-based implementations, the GPU-accelerated approaches demonstrate significantly improved performance, particularly for larger problem sizes. For small workloads, the execution times of the GPU implementations are comparable to those of the CPU due to kernel launch overheads and data transfer costs. However, as the problem size increases, both CuPy and Numba-CUDA increasingly outperform the CPU-based approaches.

4.2.2 Speedup Comparison

To further quantify the performance benefits of GPU acceleration, speedup factors are computed by comparing the execution times of the GPU implementations and

CPU multi-core implementation against the single-core CPU baseline. Table 4.2 summarizes the speedup achieved.
The results show that substantial speedup is achieved when transitioning from CPU-based execution to GPU acceleration, particularly for larger problem sizes. While multi-core CPU parallelization provides moderate performance improvements, the GPU-based implementations exhibit significantly higher speedup due to their ability to exploit massive parallelism. In addition, the comparison between CuPy and Numba-CUDA highlights differences in performance characteristics arising from their respective execution models and kernel implementations.

Table 4.2: Speedup comparison for different parallel models across problem sizes.

Number of Cells	CPU/CPU-MC	CPU/CuPy	CPU/Numba-CUDA
10^3	6.25	0.53	0.92
10^4	20.5	4.37	13.66
10^5	17.8	42.18	250.70
10^6	33.99	129.27	453.21
10^7	31.7	211.82	1074.38

4.2.3 Million Lattice Updates Per Second (MLUP/s) Comparison

In addition to execution time and speedup, the computational throughput of the different implementations is evaluated using the metric of Million Lattice Updates Per Second (MLUP/s). MLUP/s is a commonly used performance measure in Lattice Boltzmann simulations, as it directly quantifies how efficiently lattice updates are performed independent of the total simulation time.
The MLUP/s is defined as

$$\text{MLUP/s} = \frac{N_{\text{cells}} \times N_{\text{timesteps}}}{10^6 \times T_{\text{exec}}}, \tag{4.1}$$

where N_{cells} is the total number of lattice cells, $N_{\text{timesteps}}$ is the number of time iterations, and T_{exec} denotes the total execution time in seconds.
Figure 4.2 presents the MLUP/s obtained for the CPU-based implementations and the GPU-accelerated implementations using CuPy and Numba-CUDA for problem sizes ranging from 10^3 to 10^7 lattice cells. For small problem sizes, the MLUP/s achieved by the CPU implementations is comparable to or higher than that of the GPU implementations, primarily due to kernel launch overheads and data transfer costs associated with GPU execution.

As the problem size increases, the MLUP/s achieved by the GPU implementations rises sharply, substantially outperforming both the single-core and multi-core CPU implementations. For the largest problem sizes considered, CuPy achieves a near order-of-magnitude improvement in MLUP/s over the multi-core CPU, while Numba-CUDA likewise nearly reaches an order-of-magnitude improvement over CuPy. This demonstrates a progressive trend in improvement similar to the speedup evaluation.

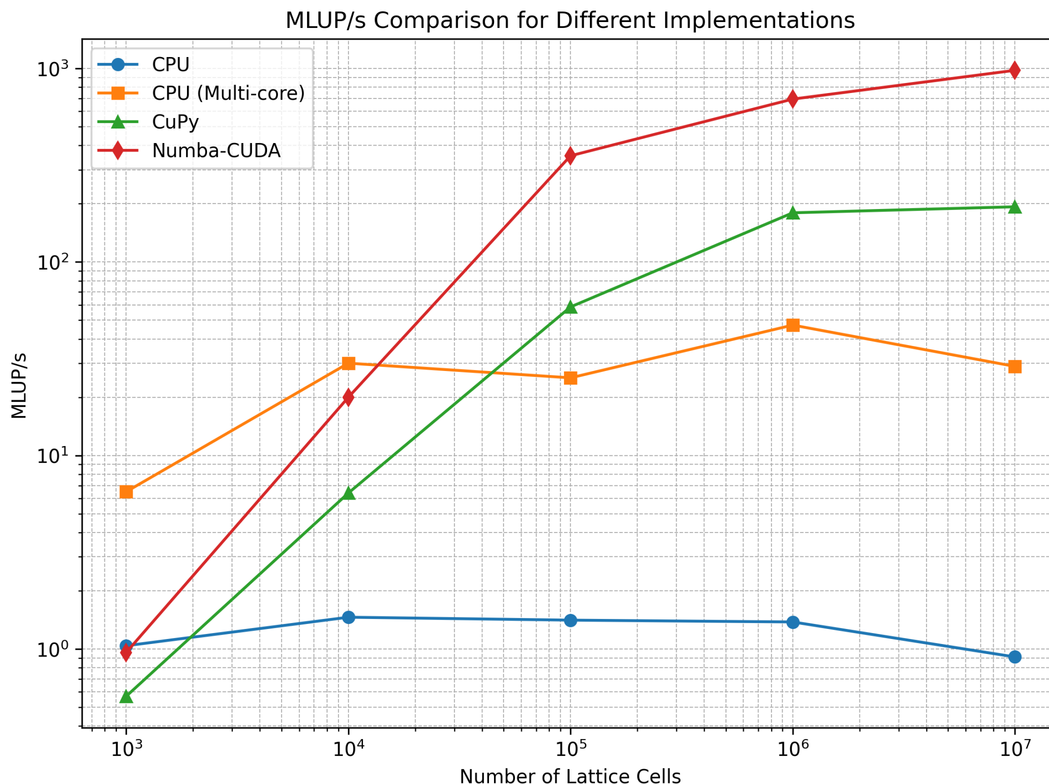

Figure 4.2: Comparison of MLUP/s for CPU, multi-core CPU, CuPy, and Numba-CUDA implementations as a function of the number of lattice cells.

4.3 Accuracy and Validation of GPU Implementations

4.3.1 Enstrophy Evolution Comparison

The temporal evolution of enstrophy is used to assess the numerical accuracy of the GPU-accelerated implementations. Enstrophy provides a global measure of vorticity in the flow and is used for verification of the Taylor–Green vortex test case.
Figure 4.3 shows the enstrophy evolution obtained using the CuPy implementation for resolutions $n = 129$, 257, and 385, while Figure 4.4 shows the corresponding results for the Numba-CUDA implementation.

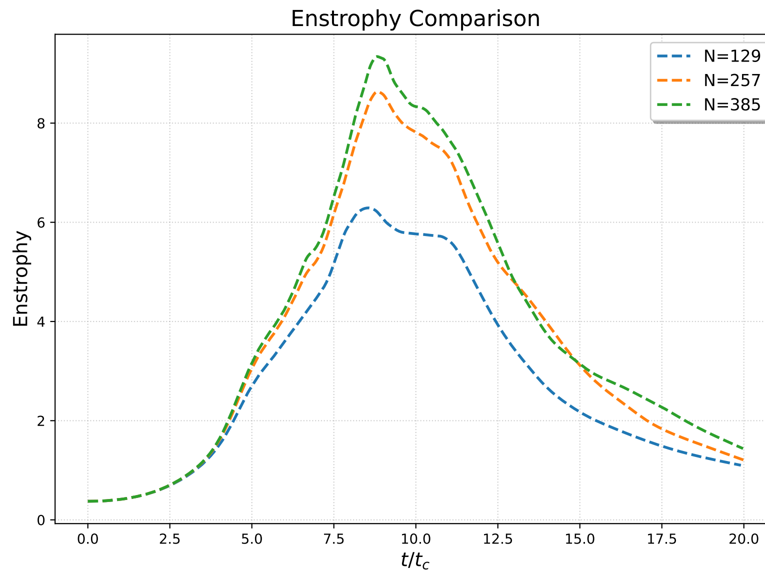

Figure 4.3: Temporal evolution of enstrophy using the **CuPy** implementation for different resolutions.

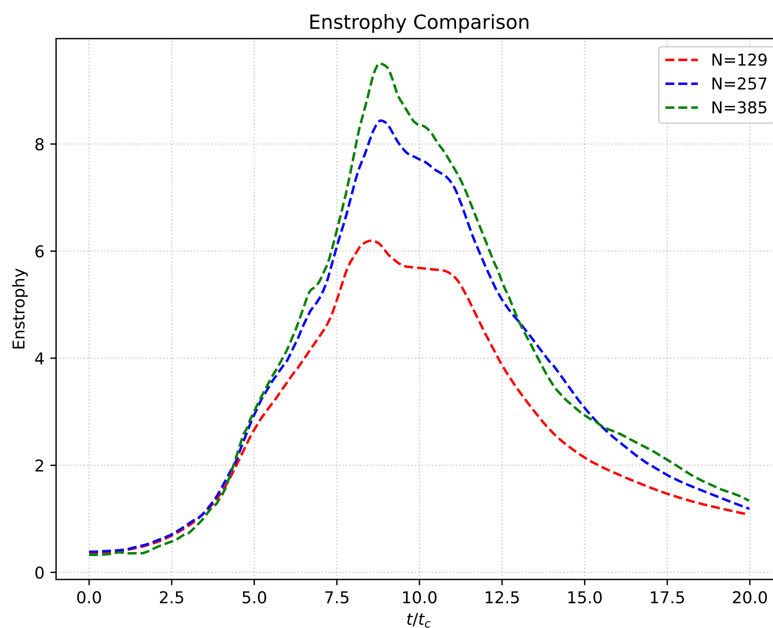

Figure 4.4: Temporal evolution of enstrophy using the **Numba-CUDA** implementation for different resolutions.

In both cases, the curves for all resolutions exhibit similar trends and overlap closely, indicating consistent numerical behavior across different grid sizes and confirming the correctness of the GPU implementations. The enstrophy evolution obtained with the GPU implementations also exhibits trends and peak values that are consistent with those reported in the CPU reference solution for the Taylor–Green vortex [7], confirming that similar global flow behavior is captured across both computational platforms.

4.3.2 Consistency Between CPU and GPU Results

To verify the physical consistency of the GPU-accelerated implementations, the final flow fields of density and velocity from the Taylor–Green vortex simulation are compared between CPU, CuPy, and Numba-CUDA implementations.
Figure 4.5 presents the final density fields for all three implementations. The results show no discernible differences in the overall structure or magnitude of the density distribution.

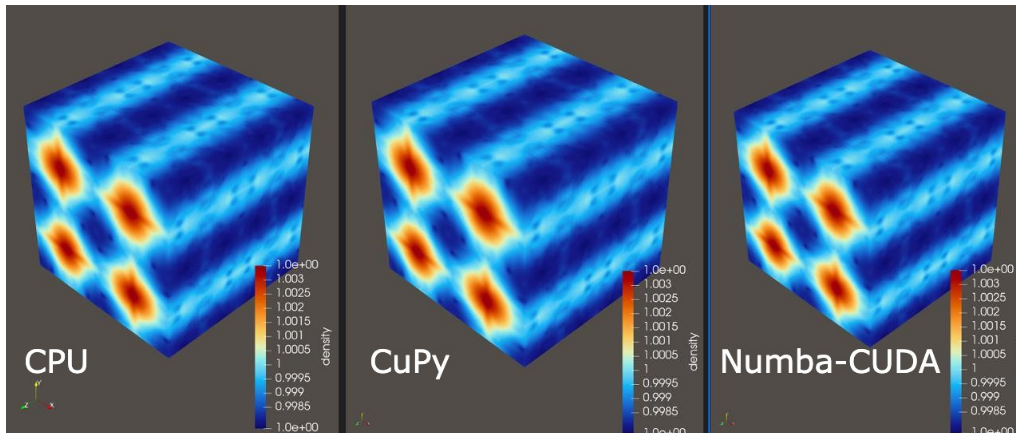

Figure 4.5: Comparison of final density fields for CPU, CuPy, and Numba-CUDA implementations.

Figure 4.6 shows the corresponding final velocity fields. The velocity distributions are visually consistent across all implementations, further confirming that the GPU-accelerated solutions reproduce the physical behavior of the original CPU implementation.

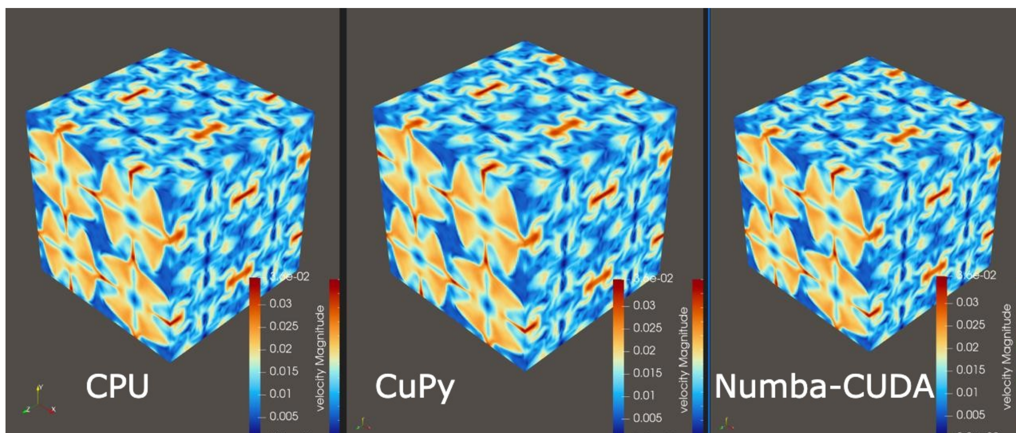

Figure 4.6: Comparison of final velocity fields for CPU, CuPy, and Numba-CUDA implementations.

Overall, the enstrophy evolution and flow field comparisons demonstrate that both CuPy and Numba-CUDA implementations produce results consistent with the CPU baseline, validating the correctness of the GPU-accelerated Lattice Boltzmann method.

This also demonstrates that the aggressive performance approach for the Numba-CUDA implementation with `float32` is fair within the scope of our validation scheme.

Quantitative L2 Error Analysis

To provide a quantitative validation of the GPU-accelerated implementations, the discrepancy between the CPU reference solution and the GPU results was evaluated using the discrete relative L_2 error norm. For a macroscopic field ϕ, the relative L_2 error is defined as

$$\varepsilon_{L_2}(\phi) = \frac{\left\| \phi^{\text{GPU}} - \phi^{\text{CPU}} \right\|_2}{\left\| \phi^{\text{CPU}} \right\|_2} = \sqrt{\frac{\sum_{i=1}^{N} \left(\phi_i^{\text{GPU}} - \phi_i^{\text{CPU}} \right)^2}{\sum_{i=1}^{N} \left(\phi_i^{\text{CPU}} \right)^2}}. \tag{4.2}$$

For the velocity field, the norm is evaluated over all velocity components simultaneously, corresponding to the full vector L_2 norm. The CPU solution is treated as the reference state. Table 4.3 summarizes the computed absolute and relative L_2 errors at the final simulation time.

Table 4.3: L_2 error comparison between CPU, CuPy, and Numba-CUDA implementations.

Field	Comparison	Absolute L_2	Relative L_2
Velocity	CPU vs CuPy	1.56×10^{-3}	8.82×10^{-5}
Velocity	CPU vs Numba-CUDA	9.57×10^{-3}	5.40×10^{-4}
Density	CPU vs CuPy	3.75×10^{-5}	2.59×10^{-8}
Density	CPU vs Numba-CUDA	5.16×10^{-4}	3.56×10^{-7}

The results demonstrate strong agreement between CPU and GPU implementations. The relative L_2 errors remain several orders of magnitude below unity, confirming that both CuPy and Numba-CUDA reproduce the reference solution with high numerical fidelity. Slightly larger deviations observed for the CUDA implementation are consistent with the use of single-precision (`float32`) arithmetic.

4.4 Synopsis

This chapter presented a comprehensive evaluation of the performance and accuracy of GPU-accelerated Lattice Boltzmann implementations.
The performance analysis demonstrated that for small problem sizes, CPU-based implementations exhibit lower execution times due to reduced overheads. However, as the problem size increases, the computational cost of CPU-based approaches grows rapidly, and GPU acceleration becomes increasingly advantageous. For the largest problem size considered, corresponding to 10^7 lattice cells, the multi-core CPU implementation achieved a speedup of approximately 31 times compared to the single-core CPU baseline.

In contrast, the GPU-based implementations delivered substantially higher performance gains. For the same problem size, the CuPy implementation achieved a speedup of approximately 211 times relative to the single-core CPU execution, while the Numba-CUDA implementation achieved a speedup of approximately 1074 times. A similar performance trend is visible for MLUP/s.

The numerical accuracy of the GPU-accelerated implementations was verified through comparisons of enstrophy evolution and flow field distributions. The enstrophy curves obtained using CuPy and Numba-CUDA for different grid resolutions exhibited strong agreement, confirming the consistency of the temporal evolution of vorticity. Finally, the final density and velocity fields produced by the GPU implementations closely matched those obtained from the CPU-based reference solution, with no observable discrepancies in the flow structures.

5
Discussion

5.1 Interpretation of Performance Trends

The observed scaling behavior highlights the interplay between computational workload and hardware characteristics. GPU acceleration demonstrates its strengths for large-scale simulations due to massive parallelism, but the relative performance for small workloads is influenced more by overheads than raw computation. This illustrates the importance of workload sizing when designing GPU-accelerated simulations and suggests that small-scale problems may not justify the overhead of GPU execution.

5.2 Framework Selection and Optimization Strategies

The differing performance of CuPy and Numba-CUDA underscores the trade-offs between high-level abstractions and low-level optimization. High-level frameworks such as CuPy can accelerate development and maintain code readability, but they may introduce hidden inefficiencies in memory access and kernel execution. In contrast, explicit kernel programming, as in Numba-CUDA, enables fine-grained control of memory patterns and instruction scheduling, yielding higher efficiency but at the cost of more complex code. Numba's developer-oriented kernel construction is what motivated us to optimize this variant as much as possible and use `float32` as it didn't produce results outside the scope of our validation scheme.
This trade-off is an important consideration for researchers balancing development speed against simulation throughput.

5.3 Memory-Bound Characteristics and Computational Bottlenecks

The simulations reveal that the Lattice Boltzmann Method is largely memory-bound rather than compute-bound. Optimizations that reduce global memory traffic, minimize intermediate arrays, or fuse computational steps can therefore provide disproportionate performance gains compared to purely arithmetic optimizations. These insights guide future efforts in kernel design and memory management for high-performance implementations.

6
Conclusion

This work investigated the acceleration of the Lattice Boltzmann Method using GPU computing frameworks, with a particular focus on CuPy and Numba-CUDA implementations. The study demonstrated that GPU-based approaches can significantly enhance computational performance for large-scale simulations, providing a clear advantage over both single-core and multi-core CPU implementations.

Beyond raw performance, the GPU implementations were shown to maintain numerical accuracy and physical consistency, as validated through enstrophy evolution and comparison of density and velocity fields. This confirms that careful implementation of GPU kernels can deliver both efficiency and reliability in scientific simulations.

The analysis further highlighted several important considerations for high-performance computing. Framework choice, memory access patterns, and problem size critically influence performance, while low-level optimizations can yield substantial improvements over high-level abstractions. Additionally, floating-point precision and algorithmic stability are essential for ensuring consistent results across different hardware platforms.

The evaluation of Million Lattice Updates Per Second (MLUP/s) demonstrates that consistent improvements of close to a factor of 10 are achieved at each stage when progressing from single-core CPU to multi-core CPU, then to CuPy, and finally to Numba-CUDA. For each step in this progression, the MLUP/s shows a near-proportional increase, reflecting the trend in observed speedup factors.

Overall, this work demonstrates that GPU acceleration represents a powerful tool for computational fluid dynamics and other memory-bound, grid-based simulations. By leveraging appropriate frameworks and optimization strategies, researchers can achieve significant reductions in simulation time without compromising accuracy, enabling more complex and higher-resolution studies that were previously computationally prohibitive.

7
Outlook

The present study demonstrates the significant potential of GPU acceleration for the Lattice Boltzmann Method. Looking forward, several avenues exist to further enhance performance, extend applicability, and improve the usability of GPU-accelerated simulations. One promising direction is the exploration of multi-GPU and distributed computing strategies. Leveraging multiple GPUs or GPU clusters could enable simulations of even larger domains or longer temporal evolutions, which are currently constrained by single-GPU memory and computational limits. Combining GPU acceleration with high-performance parallelization frameworks, such as MPI, could substantially expand the scale of achievable simulations.

A specific approach to scaling Lattice Boltzmann simulations across multiple GPUs involves the use of MPI-based Python frameworks, such as mpi4py. In principle, each MPI rank could be assigned to a separate GPU, enabling domain decomposition along one or more spatial directions and concurrent execution of kernel operations. This strategy would allow for larger simulations than are feasible on a single GPU, while maintaining the high throughput benefits of GPU acceleration.

Practical implementation of this approach with Numba-CUDA kernels on the Vera cluster is currently partially hindered by software stack compatibility issues. Specifically, the mpi4py module must be compiled against a matching MPI and GCC stack. The available numba-cuda package on Vera is provided within the foss/2025b stack, whereas the prebuilt mpi4py module is distributed within the gompi/2023a stack. Attempting to combine these modules results in conflicts between compiler versions and runtime libraries, hindering successful execution. As a result, while an MPI-GPU approach is theoretically feasible, full integration on Vera requires careful compilation of all dependencies from source and is not supported out-of-the-box, highlighting a practical limitation of the current computational infrastructure.

When scaling Lattice Boltzmann simulations from a single GPU to multiple GPUs, additional considerations arise beyond the raw computational speedup. In a distributed multi-GPU setup, global flow fields must be reconstructed from subdomain data spread across multiple MPI ranks. This data recollection introduces communication overhead that does not exist in a single-GPU simulation, meaning overall performance gains depend not only on the local GPU acceleration but also on the efficiency of inter-process communication and data aggregation. Consequently, multi-GPU performance must be evaluated with both computational and data-recollection costs in mind.

Bibliography

[1] CLMH Navier. Mémoire sur les lois du mouvement des fluides. *Mémoires de l'Académie Royale des Sciences de l'Institut de France*, 6(1823):389–440, 1823.

[2] Timm Krüger, Halim Kusumaatmaja, Alexandr Kuzmin, Orest Shardt, Goncalo Silva, and Erlend Magnus Viggen. *The lattice Boltzmann method*, volume 10. Springer, 2017.

[3] SimScale. Lattice Boltzmann method. `https://www.simscale.com/docs/simwiki/cfd-computational-fluid-dynamics/lattice-boltzmann-method-lbm/`. Accessed: 2025-12-09.

[4] Alexandr Kuzmin Orest Shardt Goncalo Silva Erlend Magnus Viggen Timm Kr-Ruger, Halim Kusumaatmaja. In *The Lattice Boltzmann Method Principles and Practice*, pages 44–46, 2017.

[5] Emil Ellénius. Lattice-boltzmann for aeronautical flows: An introduction to and evaluation of the lattice-boltzmann method. Master's thesis in applied mechanics, Chalmers University of Technology, Gothenburg, Sweden, 2024.

[6] Yue-Hong Qian, Dominique d'Humières, and Pierre Lallemand. Lattice bgk models for navier-stokes equation. *Europhysics letters*, 17(6):479, 1992.

[7] Giorgio Giangaspero, Edwin van der Weide, MH Carpenter, and K Mattsson. Case c3. 3: Taylor-green vortex. In *Case Summary for 3rd Int. Workshop on Higher-Order CFD Methods*. NASA, 2015.

[8] Jonathan Palacios and Josh Triska. A comparison of modern gpu and cpu architectures: And the common convergence of both. *Oregon State University*, 2011.

[9] ROYUD Nishino and Shohei Hido Crissman Loomis. Cupy: A numpy-compatible library for nvidia gpu calculations. *31st confernce on neural information processing systems*, 151(7), 2017.

[10] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pages 1–6, 2015.