Chalmers University of Technology

# GPU Accelerated Computations Using Python and CUDA

## Accelerating Lattice Boltzmann Method (LBM) Based on CUDA

*Author:*
**Chaoyu Zheng**
**Haojie Li**
**Raj Gopalakrishna Subramani Venkatachalam**
**Yuan Wei**

*Supervisor:*
**Magnus Carlsson**

**January 23, 2025**

# Contents

# Abstract:

Traditional computational fluid dynamics (CFD) methods solve the Navier-Stokes equations using finite difference or finite volume approaches, which are computationally expensive and challenging to parallelize due to the use of implicit solvers and complex data dependencies. The Lattice Boltzmann Method (LBM) offers an alternative by simulating fluid dynamics through particle-based equations. Since grid points update independently, LBM is inherently well-suited for parallel computing. This project implements LBM for simple fluid flow simulations, leveraging Numpy for numerical efficiency with pre-computed streaming maps. While the use of Python classes simplifies the implementation, it results in slower performance. To address this, the project integrates CUDA and Numba libraries for GPU-based parallelization. GPU computations demonstrate a significant performance improvement, achieving a speedup of 3 times compared to CPU-based calculations. This highlights the effectiveness of LBM for massively parallel applications.

# 1 Introduction

In Computational Fluid Dynamics (CFD), the **Lattice Boltzmann Method (LBM)** has emerged as an efficient and highly parallelizable numerical method. Compared to traditional finite volume and finite difference methods, LBM uses a mesoscopic approach based on **distribution functions** ($f_i$) to track fluid evolution, making it particularly suitable for complex boundaries and parallel computing.

This project develops a Python-based LBM framework from scratch, featuring a basic *lid-driven cavity* (2D flow in a square cavity) example. The framework utilizes `classes` and `numba` for CPU parallel acceleration and can be further modified to use **CUDA** to offload critical computations to the GPU. This allows for better performance on large grids. This report introduces:

- Theoretical fundamentals of the Lattice Boltzmann Method,
- Comparison between GPU (CUDA) and CPU and the advantages of GPUs in LBM, and
- Implementation of GPU acceleration using CUDA in the project and potential modifications.

# 2 Lattice Boltzmann Method: Theoretical Overview

## 2.1 Basic Concept

The Lattice Boltzmann Method (LBM) is a simplified, discretized algorithm derived from the Boltzmann equation to solve the Navier-Stokes equations for fluid mechanics. Its core components include:

1. **Discrete velocity set ($\mathbf{c}_i$)**: The continuous velocity space is discretized into $Q$ directions (e.g., D2Q9, D3Q19).
2. **Distribution functions ($f_i$)**: At each lattice point, $Q$ distribution functions $f_i$ are stored, representing the amount of particles moving in each discrete direction.
3. **Collision and streaming steps**:
   - *Collision*: Using the BGK (Bhatnagar–Gross–Krook) or MRT/TRT operators, $f_i$ relaxes towards its equilibrium state $f_i^{\text{eq}}$.
   - *Streaming*: Each distribution function streams along its corresponding $\mathbf{c}_i$ direction to adjacent lattice nodes.

LBM decouples fluid dynamics into a "collision + streaming" two-step process, enabling **high parallelism** since each lattice point only needs data from a small number of neighbors.

## 2.2 Equations and Macroscopic Quantities

In the DdQq framework, each lattice point stores $f_i(\mathbf{x})(i = 1, \ldots, Q)$. The macroscopic quantities are computed as:

$$\rho = \sum_i f_i, \quad \rho\mathbf{u} = \sum_i \mathbf{c}_i f_i,$$

where $\rho$ and $\mathbf{u}$ are density and velocity, respectively. The equilibrium distribution function for the BGK model is:

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

where $w_i$ are weights and $c_s$ is the speed of sound. The BGK collision operator is given by:

$$\Omega_i = -\frac{1}{\tau}(f_i - f_i^{\text{eq}}),$$

where $\tau$ controls the viscosity.

## 2.3 CPU vs. GPU

- **CPU**: General-purpose processor specialized in handling complex control flow and diverse tasks, with a limited number of cores (usually a few to a few dozen).
- **GPU**: Graphics Processing Unit with hundreds or thousands of parallel threads, optimized for large-scale data-parallel computations (e.g., matrix operations, fluid dynamics).

LBM is highly suited for GPUs due to:

- **Local access patterns**: Streaming requires only nearest-neighbor data.
- **Uniform operations**: Each lattice node undergoes the same "collision + streaming" operations.
- **Scalability**: Larger grids (hundreds of thousands to millions of nodes) leverage GPU parallelism effectively.

# 3 Development Tools Introduction:

## 3.1 Numba:

Numba is an open-source Python library designed to accelerate numerical computations by using Just-In-Time (JIT) compilation, converting Python functions into optimized machine code at runtime. It integrates seamlessly with Numpy, enabling faster execution of array operations, and supports parallelization to utilize multi-core CPUs and GPUs. For this project, Numba was utilized to enhance the performance of the Lattice Boltzmann Method (LBM), particularly in the computationally demanding streaming and collision steps. By leveraging Numba's JIT compilation and CUDA support, the project achieved significant speedups, particularly through GPU parallelization.

### 3.2 Cupy:

CuPy is an open-source Python library designed for GPU-accelerated numerical computing, offering a drop-in replacement for NumPy, with the ability to leverage NVIDIA GPUs for massive parallelism. CuPy is widely used in machine learning, scientific computing, and deep learning tasks where high-performance computation is required.

### 3.3 Scalene:

Scalene is a high-precision Python profiler that provides deep insights into CPU, memory, and GPU usage with minimal runtime overhead. Unlike traditional profilers, Scalene distinguishes between Python and native code execution, making it particularly useful for optimizing performance-critical applications.

## 4 Implementing CUDA-Based GPU Acceleration

### 4.1 Code Profiling

Before we start parallelizing the code and optimizing computations, we need to profile the code.When profiling the performance of Python code, we use Scalene to analyze the execution time, memory usage, and CPU usage line by line. From this analysis, we identify the code segments with the highest execution time and resource consumption for further analysis and optimization. Here is a simple snippet of the code:

```
scalene samples/lid_driven_cavity_2d.py
```

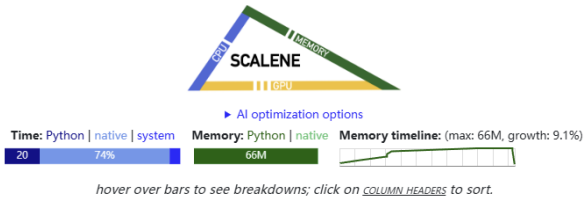


Figure 1: Main code profiling

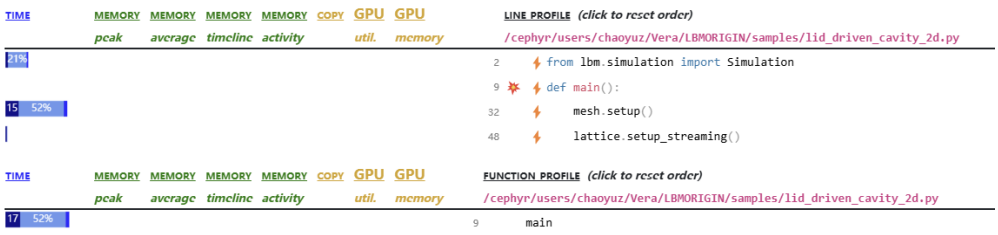


Figure 2: Main code part which takes most of the time

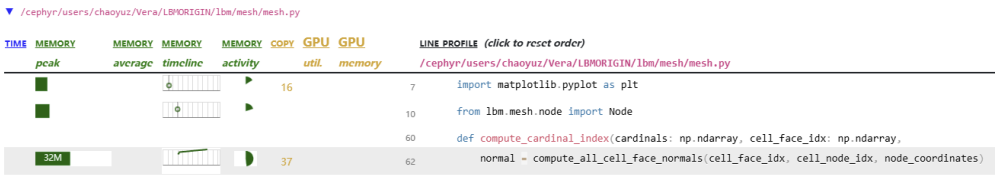


Figure 3: Main code part which takes most of the memory

The Figure above illustrates that to improve the speed of LBM computation, the primary focus should be on accelerating the creation of the LBM streaming map. We need to offload the parallelizable and vectorizable parts to the GPU for computation to achieve higher efficiency.
In the original code, node initialization is primarily achieved through two methods. In the first stage, `mesh.setup` is used to establish the connections between nodes, including `cell connectivity`and `face connectivity`, as well as the addition of nodes.
The second stage involves the establishment of the streaming map, where a Python list is used to construct and store data grid information, including boundary conditions and bulk conditions.
The establishment of relationships itself does not consume much computation time. However, due to the unique design of the GPU, we cannot directly store object-based relational data in GPU memory. This approach does not align with the GPU's design paradigm and cannot achieve efficient queries. As a result, during computation, frequent data exchanges between the host and device are required to query node relationships, especially for calculations occurring in boundary regions.
After completing the code profiling, we can then dive into the underlying code to optimize computations that consume a significant amount of execution time, assessing their potential for parallelization and vectorization.

## 4.2 Code Implementation

After analyzing the code, we identified key computational methods that can be optimized, including mesh generation and lattice relationship initialization. In the initial stages of program execution, it is crucial to focus on initializing the required data on the GPU to minimize the overhead of frequent data transfers. Fortunately, CuPy provides implicit data initialization methods, such as `cp.asarray()`, which allows direct initialization of CPU instances into GPU memory. This simplifies many unnecessary explicit operations and improves efficiency.
In subsequent code optimization, we will refactor the code by offloading fundamental computations, such as dot product calculations, equilibrium distribution function evaluations, and weighted sums, to CUDA for improved performance.
For inter-node computations, we will leverage CuPy's broadcasting and slicing methods to enhance execution efficiency. Additionally, we will utilize `cupyx.scipy.sparse` to support efficient matrix storage and operations.
In this computational phase, we need to carefully consider the handling of boundary conditions. Since sparse matrices are already used to record the relationships between node

elements, there is no need to maintain neighborhood data on the CPU. Instead, boundary conditions can be directly determined based on the position of elements within the matrix itself. Here are some examples about how we optimize the code:

```
15  def density_cupy(f,rho):
16      ff = cp.asarray(f)
17      rhoc = cp.asarray(rho)
18      rhoc[:] = cp.sum(ff, axis=0)
19      rho = cp.asnumpy(rhoc)
20
```

Figure 4: Code Example 1

The optimization method shows in Figure 4 involves explicitly converting NumPy arrays into CuPy arrays, thereby offloading the computation to GPU memory. After the computation is completed, the arrays are transferred back to the host, as shown in the figure.

The advantage of this optimization is that it requires modifications only at the implementation level without altering the top-level code logic. This minimizes code changes while improving computational performance as much as possible.

However, this approach has significant limitations. When a low-level method is frequently invoked, data transfers can consume a large portion of the computation time. To address this, we adopted a second transfer approach: using subclass inheritance.

```
class CupyLattice(Lattice):
    def __init__(self, stencil: CupyStencil, mesh:CuPyMesh):
        super().__init__(stencil,mesh)
        self._stencil = stencil
        self._mesh = mesh
        """
        print(type(self.tau))  # 期望输出 cupy.ndarray
        print(type(self.stencil.w))  # 期望输出 cupy.ndarray
        print(type(self.stencil.inv_cs_2))  # 期望输出 cupy.ndarray
        """
        self.f = cp.asarray(self.f)
        self.f0 = cp.asarray(self.f0)
        self.S = cp.asarray(self.S)
        self.feq = cp.asarray(self.feq)
        self.u = cp.asarray(self.u)
        self.g = cp.asarray(self.g)
        self.rho = cp.asarray(self.rho)

        self.S_const = cp.asarray((1.0 - 1.0 / (2.0 * self.tau)) * self.stencil.w * self.stencil.inv_cs_2)
        self.force_vector = cp.zeros(self.stencil.d, dtype=cp.float64)
        self._bulk_streaming_map = cp.empty((0, 0), dtype=cp.uint32)
        self._bound_streaming_map = cp.empty((0, 0), dtype=cp.uint32)
        self._bulk_cell_index = cp.empty(0, dtype=cp.uint32)

        self._lambda = cp.float64(0.25)
        self._tau = cp.float64(1.0)  # Relaxation time
```

Figure 5: Code example 2

```python
def equilibrium_cuda(feq, u, rho, q, w, c, inv_cs_2, inv_cs_4):
    uu = cp.sum(u**2, axis=1)  # (n,)

    # compute u and c dot, shape as (q, n)
    uc = cp.dot(c, u.T)  # (q, n)

    # compute uc sq, shape as(q, n)
    uc2 = uc**2  # (q, n)

    # equilibrium feq using vectorlization
    feq[:, :] = w[:, cp.newaxis] * rho * (
        1 + uc * inv_cs_2 + 0.5 * uc2 * inv_cs_4 - 0.5 * uu * inv_cs_2
    )
```

Figure 6: Code example 3

In this method, a subclass of the CuPy array is generated during the initialization phase. During the subclass initialization, the parent class data is explicitly converted, reducing the need for repeated data transfers and improving overall performance. In lower-level computation methods, we can directly operate on CuPy arrays, enabling efficient operations such as dot products and vectorized computations.

```python
class CupyLattice(Lattice):
    def _setup_streaming_maps_gpu(self,stencil: CupyStencil, cell_range: cp.ndarray = None,

        row = cp.asarray(grid.adjacency_matrix.row)  # 行索引
        col = cp.asarray(grid.adjacency_matrix.col)  # 列索引（邻居索引）
        #c_int = cp.asarray(stencil.c)
        c_int = stencil.c
        i_bulk = cp.uint32(0)

        # Process each cell
        for i, cell in enumerate(cells):
            # 单元格索引
            cell_index = cell.index
            neighbours = col[row == cell_index]  # 获取当前单元格所在行的所有非零列索引（即邻居）

            # 初始化邻居映射
            streaming_map = cp.full(stencil.q, cell_index, dtype=cp.uint32)

            # 检查每个方向的目标单元格是否为邻居
            target_cells = cell_index - c_int  # shape = (q,)
            neighbour_mask = cp.isin(target_cells, neighbours)

            for iq in cp.where(~neighbour_mask)[0]:  # 获取非邻居的方向索引
                bc_faces = grid.find_face(cell_index)  # 自定义方法
                if bc_faces:
                    bc = bc_priority([face.boundary.bc for face in bc_faces])
                    bc.setup_streaming_cupy(iq, cell, bc_faces, stencil)

            # 填充 bulk_streaming_map
            bulk_streaming_map[:, i_bulk] = streaming_map
            bulk_cell_index[i_bulk] = cell_index
            i_bulk += cp.uint32(1)
```

Figure 7: Code example 4

By overriding parent class methods, we aim to implement logical code execution on the GPU, such as leveraging GPU-supported sparse functions to construct the streaming map for subsequent computations and queries on the GPU.

# 5 Results:

After optimizing the computational process, we need intuitive performance metrics for comparison. "Millions of Lattice Updates per Second" (MLUPs) and computation time are key indicators for measuring performance improvements.We used a 8-core CPU processor as the baseline for LBM computation speed and MLUPs update efficiency. The performance testing was conducted on an A40 GPU computing node, where we compared the performance of CuPy optimization alone with the combined optimization of CuPy and Numba.

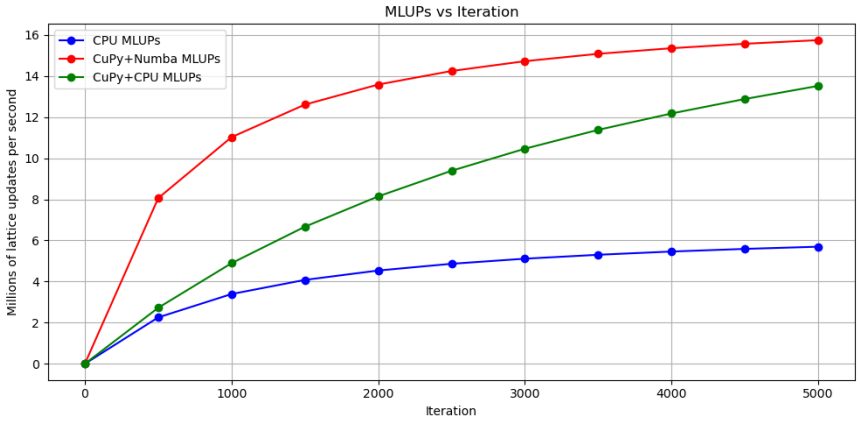


Figure 8: mlups per iteration combined 3groups

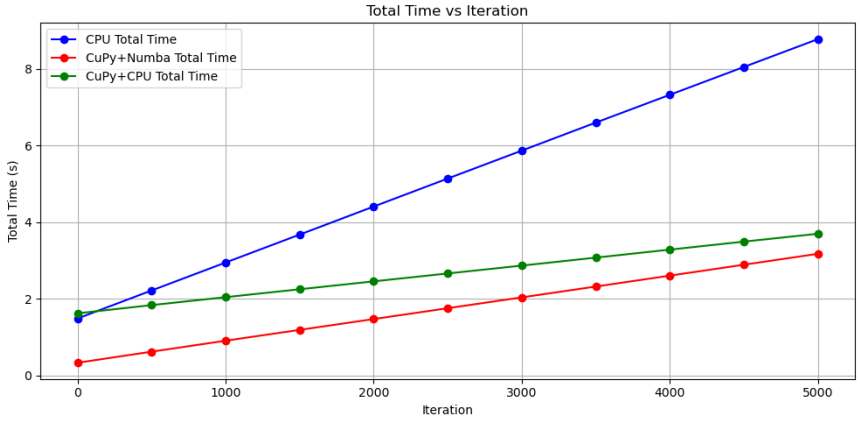


Figure 9: totaltime per iteration combined 3groups

From Figure 8 and Figure 9, we use the number of iterations as the horizontal axis and MLUPs along with total execution time as the vertical axis to visually compare the performance of CUDA-based parallelization with CPU multithreading. It is evident that the combined optimization using CuPy and Numba resulted in a significant improvement in computational efficiency and a reduction in total execution time. However, the performance

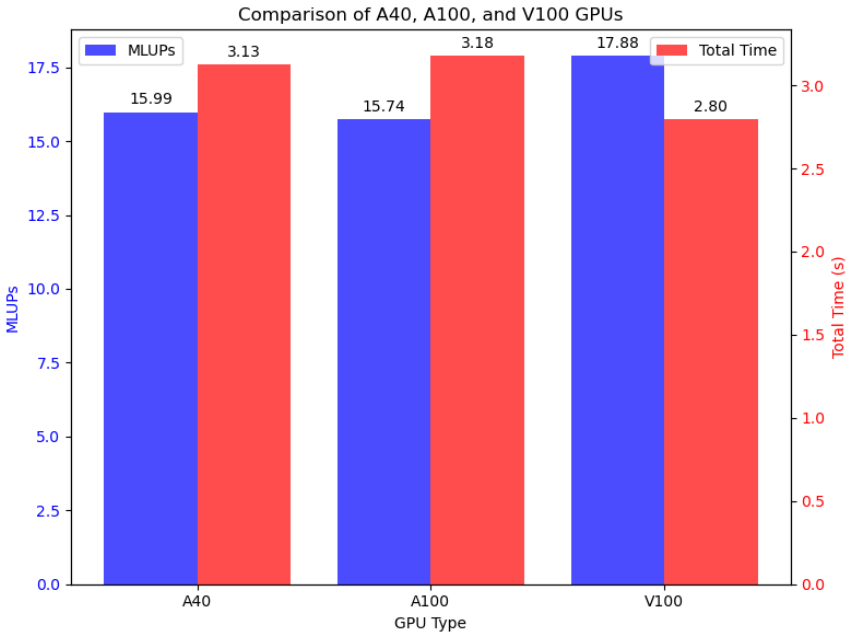


Figure 10: totaltime per iteration combined 3groups

gain compared to using CuPy optimization alone was not particularly significant.
We also compared the computational performance across three GPUs—A40, A100, and V100—resulting in the bar chart shown in Figure 10. We observed that the performance differences were not significant. However, an interesting phenomenon emerged: despite the theoretically superior performance of the A40 and A100, their computational efficiency did not surpass that of the V100.

# 6 Discussions:

While this project demonstrates the effectiveness of GPU-accelerated Lattice Boltzmann Method (LBM) simulations for simple fluid dynamics problems. Our experimental results demonstrate an improvement in computational performance.It also highlights certain limitations and areas for improvement. One key limitation lies in the use of Python, despite its accessibility and flexibility, introduces overhead compared to lower-level programming languages like C++ or CUDA, which could provide further performance gains.

Future work could address these limitations by extending the framework to incorporate more sophisticated collision models, optimizing memory management to reduce data transfer between CPU and GPU, and enabling support for multi-GPU simulations. These advancements would not only improve the computational efficiency but also enhance the physical realism of the simulations, making LBM a more versatile tool for modern computational fluid dynamics.

We also observed noteworthy phenomena worth discussing. Despite the A100 and A40 utilizing the more advanced Ampere architecture, while the V100 is based on the previous-generation Volta architecture, the V100 outperformed in certain numerical simulation appli-

cations such as LBM computations. This could be attributed to several factors:

**Specialized Matrix Computation Units (Tensor Cores)**: The V100 is highly optimized for high-performance computing (HPC), which may better align with our workload. If our code does not fully leverage the Tensor Cores or other new features of the A100, the V100 might maintain a performance edge.

**Higher Clock Frequency and core utilization**: The V100 generally operates at a higher core clock speed, making it more suitable for workloads that rely heavily on single-precision or double-precision floating-point calculations. The V100 may have achieved a higher core utilization under our workload, whereas the A100 and A40 might have encountered data transfer bottlenecks or underutilized compute cores, preventing them from fully realizing their performance potential.

Moreover, the object-oriented programming logic based on CPU multithreading acceleration may not be directly applicable to GPU high-performance computing. From a programming perspective, we want to use objects to store data and capture the relationships between them. However, simply porting this data to GPU memory for parallel computation does not align with the GPU's design principles.

In the process of continuing CuPy optimization, I discovered that CuPy supports instantiating matrices while maintaining the relationships between nodes. As a result, unnecessary relationships, such as using a string list to query neighboring nodes, are transformed into sparse matrix element operations in GPU memory. This significantly reduces frequent data transfers between the main memory and disk, improving overall performance.

# 7 Conclusion:

The main conclusion from this experiment is that CUDA is feasible for accelerating the Lattice Boltzmann Method (LBM) computation. The combination of CuPy and Numba CUDA can significantly improve computational efficiency and reduce execution time, especially for dense data like 2D matrix operations, where parallel computation outperforms the baseline computation data. However, in addressing the node relationship establishment, boundary condition creation, and other logical structures, CUDA shows clear limitations. The GPU is primarily designed for large-scale, dense computations, and its architecture is optimized for data storage, making some object-oriented programming concepts difficult to implement. We need to employ specialized techniques to store and associate indexes and data effectively.

However, this does not mean that CUDA cannot be used to solve such problems. It requires a shift in our programming approach. For example, we can use matrix-vector multiplication techniques like CSR (Compressed Sparse Row) or CSC (Compressed Sparse Column) to handle sparse matrix operations that are accessed by rows or columns. These techniques can significantly improve memory utilization and avoid unnecessary memory overhead.Due to time constraints and the fact that the initial optimizations did not achieve the desired improvement in computational efficiency, we were unable to complete the optimization for this part of the computation. However, we believe that this approach is feasible and could yield significant benefits with further refinement.

# References


[1] T. Scherlis, "Lattice-Boltzmann Algorithm Using GPU Acceleration," 2017. [Online]. Available: https://tomscherlis.com/wp-content/uploads/2017/02/Lattice-Boltzmann-Algorithm-Using-GPU-Acceleration.pdf

[2] "Lettuce: GPU-Accelerated Lattice Boltzmann Simulations in Python," [Online]. Available: https://lettuceboltzmann.readthedocs.io/en/latest/readme.html

[3] C. Gkoudesnes and R. Deiterding, "Verification and Validation of a Lattice Boltzmann Method Coupled with Complex Sub-grid Scale Turbulence Models," 2019. [Online]. Available: https://upcommons.upc.edu/bitstream/handle/2117/186779/Particles_2019-46-Verification%20and%20validation%20of.pdf

[4] "GPU-Accelerated Lattice Boltzmann Method for Direct Numerical Simulation of Turbulent Flows," [Online]. Available: https://engineering.purdue.edu/YuLab/research/products/PublishedPapers/2014PARCFD.pdf

[5] K. Jain, "Efficacy of the FDA Nozzle Benchmark and the Lattice Boltzmann Method for the Analysis of Biomedical Flows in Transitional Regime," 2020. [Online]. Available: https://arxiv.org/abs/2005.07119