Chalmers University of Technology

GPU-accelerated Computational Methods using Python and CUDA

Lattice Boltzmann Method

Author:
Roni Celiker
Haozhe Sun
Fengming Tong
Adam Persson

Supervisor:
Magnus Carlsson

February 13, 2025

1 Abstract

This report presents the optimization of a Lattice-Boltzmann-based Computational Fluid Dynamics (CFD) solver by transitioning from CPU to GPU acceleration. The solver, based on Emil Ellenius's master's thesis, was extended with GPU support, a key improvement identified in his work. The project was supervised by Magnus Carlsson from Saab.

Initially, replacing NumPy with CuPy provided notable performance gains. Further acceleration was achieved by implementing CUDA instead of Numba, resulting in exponential improvements, especially for large simulations.

The Lattice-Boltzmann method (LBM) offers a microscopic approach to fluid dynamics that captures microscopic characteristics as node counts increase. By using CUDA on A40 GPUs, the solver showed significant scalability, with larger simulations benefiting most from GPU parallelization. Performance comparisons confirmed that CUDA's advantages grow with problem size.

While CuPy and CUDA were tested separately, future work could explore their combined use. This project highlights how adapting open-source solvers to modern GPUs can transform CFD simulations.

2 Introduction

In recent years, the usage of graphical processing units (GPU) in programming has increased. This is because of its high performance increase in comparison to traditional central processing units (CPU) when the workload is paralizable. As the performance is substantionally increased, this entails that industry and research save time when running code. One of the more popular paralization platforms is CUDA, which utilizes NVIDIA based graphics cards, and has seen a sharp rise in popularity for its robustness and easy implementation. One of the areas that has benefited the most from paralized workloads is computational fluid dynamics (CFD), and aspecially the Lattice Boltzmann method (LBM), which this project will focus on.

2.1 Computational fluid dynamics

In mechanical engineering, computational fluid dynamics (CFD) are a branch of fluid mechanics that uses numerical analysis and algorithms to solve and analyze problems involving fluid flows with surfaces defined by boundary conditions. It is commonly used in engineering to simulate fluid behaviors in complex systems, such as in aerodynamics, weather patters or industrial processes.

As CFD methods handle fluids, they are based on solving the Navier-stokes equations, as these equations govern the motion of fluid flows under various conditions. This entails that the goal of a CFD method is to numerically solve these equations, which is often in simplified or approximated form to reduce computational calculation. It should be noted that the majority of CFD methods is based on and handle continuum mechanics. This entails that the molecular nature of the fluids is not accounted for, and the Navier-Stokes partial differential equations are solved directly to obtain the macroscopic properties of the fluid (e.g velocity, pressure etc) for simulation.

CFD methods usually solve the previously mentioned equations iteratively over a domain divided into elements. As the operations performed at each element are repetetive across the domain, this entails that the workload is well suited for paralization.

2.2 The Lattice Boltzmann Method

The Lattice Boltzmann Method (LBM) is a specific type of CFD method that, unlike traditional CFD methods, models fluid behavior from a mesoscopic perspective by simulating the movement and collision of a particle distribution function $f_i(x,t)$ and a probability density function f_{eq}. The principle is to derive the macroscopic properties of the fluid from the statistical mechanics applied from Boltzmann transport equation. This is done by making the assumption that $f_i(x,t)$ is relaxed toward f_{eq} each time step, approximating the particle density behavior and then simulating it each itteration.

2.2.1 LBM Algorithm Overview

The Lattice Boltzmann Method operates on a discrete lattice grid where each lattice node (or cell) tracks discrete velocity distribution functions. The simulation advances in discrete time steps and is typically divided into four main stages:

1. Streaming (Propagation) Step In the streaming step, the distribution functions at each lattice node are shifted (or streamed) to neighboring nodes along the discrete velocity directions. Mathematically, this corresponds to:

$$f_i(\mathbf{x} + \mathbf{c}_i \Delta t, t + \Delta t) = f_i(\mathbf{x}, t)$$

where f_i is the particle distribution function in the i-th discrete velocity direction, $\mathbf{c}_i$ is the discrete velocity vector, and Δt is the time step. This step redistributes the particle populations according to their velocities.

2. Boundary Step Following the streaming step, special treatment is applied to nodes lying on or adjacent to boundaries (e.g., walls, inlets, outlets). For example, bounce-back boundary conditions reverse the distribution functions that stream into wall nodes to enforce no-slip conditions. Inlets and outlets may apply prescribed velocity or pressure profiles. This step ensures that the correct physical behavior is enforced at the domain boundaries.

3. Collision (Relaxation) Step Once the distribution functions have been streamed and boundary conditions applied, each node undergoes a collision (or relaxation) process. The collision step drives the distribution functions toward an equilibrium distribution $f_{\text{eq},i}$, which is often derived from the Maxwell-Boltzmann distribution under the incompressible or weakly compressible flow assumption. A common example is the BGK (Bhatnagar–Gross–Krook) approximation:

$$f_i(\mathbf{x}, t + \Delta t) = f_i^*(\mathbf{x}, t) + \omega \big[f_{\text{eq},i}(\rho, \mathbf{u}) - f_i^*(\mathbf{x}, t) \big]$$

where $f_i^*(\mathbf{x}, t)$ are the post-streaming distributions, ω is the relaxation parameter (inversely related to viscosity), ρ is the fluid density, and $\mathbf{u}$ is the local velocity.

4. Macroscopic Calculation Step After collision, macroscopic flow quantities such as density ρ, velocity $\mathbf{u}$, and pressure p are computed at each lattice node by taking moments of the distribution functions:

$$\rho = \sum_i f_i, \quad \rho \mathbf{u} = \sum_i f_i \mathbf{c}_i, \quad p \approx \rho c_s^2$$

where c_s is the speed of sound in the lattice model. These macroscopic values can then be used for output, boundary conditions, or other post-processing steps.

2.2.2 Parallel Advantage of LBM

A notable strength of the LBM lies in its *locality*. Collision operations occur independently at each lattice node, while streaming only involves immediate neighbor updates. This highly

localized data dependency makes the LBM *naturally parallelizable*: thousands (or millions) of lattice nodes can be updated in parallel with minimal communication overhead. When implemented on modern GPUs (e.g., via CUDA), LBM benefits from the massive parallel processing capability, achieving significant speedups compared to CPU-based implementations. This is especially advantageous in large-scale or real-time fluid simulations.

Overall, by combining mesoscopic modeling with discrete velocity sets and localized update rules, the LBM provides a flexible and efficient framework for simulating fluid flow and related transport phenomena.

2.3 Parallelization with CUDA

The Lattice Boltzmann Method (LBM) involves repetitive calculations on a large number of small grid points, where computations at each point are largely independent. This makes LBM highly suitable for parallel processing on GPUs, which can significantly improve computational efficiency.

CUDA Implementation Strategy for LBM

- **Mapping Grid to CUDA Threads**: The physical lattice grid is mapped onto CUDA's thread structure, where each thread handles computations for one or more grid points.
- **Thread Organization**: Threads are grouped into Blocks, and multiple Blocks form a Grid. NVIDIA GPUs contain multiple Streaming Multiprocessors (SMs), each with 32 threads per processor core.
- **Memory Optimization**: Efficient data access is ensured using shared memory to minimize global memory latency.
- **Thread Synchronization**: Proper synchronization mechanisms ensure calculation accuracy.

To optimize performance, the computational domain is divided into spatially contiguous blocks along one axis. Multiple copies of the same program run concurrently, each processing its own block of data. At the end of each iteration, data at the boundaries between blocks is exchanged to ensure consistency before proceeding to the next iteration.

3 Method

In this section, 2 methods of parallization for the lbm method are implemented. The first implementation is CuPy, a high level GPU-accelerated library, as a means to optimize computational performance in Python. The second implementation is CUDA, A low level parallel computing platfrom that gives more fine-grain controll over parallization parameters.

3.1 Numpy to Cupy

CuPy is designed to mimic NumPy's API, enabling a straightforward transition from CPU-bound operations to GPU-based computation. The focus of the method was to evaluate how CuPy handles various computational tasks, highlighting both its advantages and limitations.

To implement CuPy, we replaced NumPy operations with their CuPy equivalents, allowing CuPy to handle data transfers between the CPU and GPU automatically. CuPy arrays reside on the GPU rather than the CPU, and memory allocation is managed through CUDA. Unlike native CUDA, which requires explicit thread and block management, CuPy abstracts this complexity by relying on its own optimized CUDA kernels for efficient execution. In our implementation, we exclusively used CuPy's built-in functionality without manually defining CUDA kernels or mixing native CUDA with CuPy.

A key consideration in the implementation was the synchronization of data between the CPU (host) and GPU (device). As data transfer is a potential bottleneck, special care was taken to minimize unnecessary transfers, especially in workflows that required frequent communication between the host and device.

To measure performance, a profiling tool was used to compare the execution of NumPy-based operations with their CuPy-based counterparts. We analyzed scenarios involving large arrays and simple mathematical functions, identifying cases where CuPy provided substantial speedups. We also examined bottlenecks, such as memory transfer overhead and compatibility issues with NumPy functions that lack direct CuPy equivalents.

The results and analysis provide insights into CuPy's capabilities and its potential trade-offs when integrating GPU acceleration into Python applications.

3.2 Numba to CUDA

As CUDA is a low-level programming framework, implementing it correctly required meticulous attention to detail and a deep understanding of its intricacies to ensure an accurate adaptation of the original code. The complexity of CUDA programming arises from the need to analyze the parallelization potential of functions, identify necessary variable modifications, and appropriately translate traditional CPU-based code into CUDA kernels.

To systematically convert the existing LBM implementation to CUDA, a structured methodology was employed to transform variables, input arguments, and functions into their corresponding CUDA-optimized versions.

3.2.1 Profiling and Function Selection

The first step involved profiling each segment of the code directly responsible for key LBM computations, specifically the boundary conditions, collision, equilibrium, streaming, and macroscopic calculations. This was achieved using the existing profiler within the original code, which provided execution time metrics for each segment per iteration during a simulation.

The functions were then ranked based on their execution time, with the most computationally expensive segments being prioritized for CUDA implementation to maximize performance gains.

3.2.2 Identifying Parallelization Potential

Once a segment was selected, its constituent functions were analyzed to determine their suitability for parallel execution. A function was considered a candidate for CUDA implementation if it exhibited both:

- **Independence:** The operations within the function could be executed without dependencies on previous calculations.
- **Repetitiveness:** The function performed the same type of computation multiple times, making it well-suited for SIMD (Single Instruction, Multiple Data) execution.

Functions that met these criteria were selected for CUDA conversion.

3.2.3 Adapting Input Arguments for CUDA

The selected functions' input arguments were then analyzed to ensure compatibility with CUDA, which requires data to be stored in a format optimized for efficient memory access. Since CUDA operates best with flat, contiguous memory layouts, all input arrays were converted to NumPy arrays, ensuring they adhered to the required format.

Once the data was properly structured, it was transferred to the GPU using CUDA's host-device memory management mechanisms.

3.2.4 CUDA Implementation

With the input arguments properly formatted and sent to the GPU, the selected functions were rewritten for CUDA execution. This involved:

- Implementing **host functions** to manage data transfers between the CPU and GPU.
- Developing **kernel functions** to execute parallel computations on the GPU.

To illustrate the steps taken for rewritting code for CUDA, an example LBM function will be explored. The function calculates the density at each lattice node, and its original code is shown as:

```
@nb.njit
def density(f, rho):
    rho[:] = np.sum(f, axis=0)
```

1. First, transfer input arguments to GPU memory:

```
    self.f_device = cuda.to_device(self.f)
    self.rho_device = cuda.to_device(self.rho)
```

2. Create a host function which will be called inplace, derive workload size and configure grid and block dimension

```
    def density(f_device,rho_device):
        n = f_device.shape[1]
        threads_per_block = 256
        blocks_per_grid = (n + threads_per_block - 1) //
           threads_per_block)
        density_cuda[blocks_per_grid, threads_per_block](
           f_device, rho_device,n)
```

   Here, threads per block is a constant value determined by the workload size and shape, and is a interger of 32, in this example 256.

3. Create a kernel function that is launched from the host function

```
    @cuda.jit
    def density_cuda(f, rho,n):
    j = cuda.grid(1)
    if j < n:
        sum_density = 0.0
        for i in range(f.shape[0]):
            sum_density += f[i, j]
        rho[j] = sum_density
```

   Here, each thread on the GPU is allocated to one lattice node, calculating the density by summing the 9 velocity directions in a for-loop. By deriving the workload size n (in this case, number of lattice nodes in grid) we can ensure that the thread ID j is within the bounds of n.

In the example above, a 1-dimensional host and kernel function was employed, where each thread was assigned to a single lattice node and computed the sum using a for-loop over velocity directions. However, for other functions, a 2-dimensional approach was utilized to further distribute the workload, assigning threads not only to lattice nodes but also to velocity directions or other computational parameters, thereby improving performance. This structured methodology was consistently applied throughout the project to achieve an efficient CUDA implementation.

For large grid sizes ($n > 10^6$), the number of required threads exceeded CUDA's software limitation of 65,535 blocks per grid per dimension. This constraint arises because CUDA employs 16-bit integers to store block indices. To address this, certain 2D implementations were extended to 3D, utilizing the z-axis to accommodate additional blocks. Since each additional dimension enables a multiplicative increase in available blocks, this effectively expands the limit to $65,535^3$ blocks, which is more than sufficient for the data sizes tested in this project.

3.2.5 Shared Memory Implementation

In select functions, shared memory was employed as an optimization strategy to improve intermediate computations and summation operations. This optimization was aimed at minimizing global memory access within the CUDA kernel, as frequent access to global memory can significantly degrade performance due to high latency.

The fundamental approach involves transferring input variables from global memory into a temporary, high-speed storage area within the kernel—shared memory. By doing so, instead of repeatedly fetching data from global memory, the kernel loads the necessary values into shared memory once, allowing all threads within a block to access the data much more efficiently.

Despite the potential performance benefits, shared memory optimization was applied to only a few functions and was not a primary focus in this project's methodology.

4 Results

In this section, the results and performance of the CUDA and CuPy imlementations will be shown and compared to the CPU. The metrics used to determine performance is MLUP and the total time.

4.1 Parameters and provided results

MLUPS (Million Lattice Updates Per Second) is a crucial metric for evaluating algorithmic efficiency and hardware performance, particularly in parallel computing environments. A higher MLUPS value indicates superior performance, as it represents the system's capability to process more lattice point updates per unit time.

For the benchmarking analysis presented, all tests were conducted using single-precision floating-point numbers (32-bit representation). The experiments consisted of 5000 iterations, preceded by 20 warm-up runs to ensure stable performance measurements. The simulations where done through the lid_driven_cavity_2d.py file.

In the results provided, the time taken for the pre-processing (e.g building the grid, creating and setting up streaming maps etc) were omitted from the total time taken for a single run time. This is done because of these processes not being a part of the actual LBM.

4.2 Numpy to Cupy

The results demonstrated that CuPy significantly improves performance for computationally intensive tasks, particularly those involving large arrays and matrix operations. By leveraging GPU acceleration, CuPy outperformed its CPU-bound NumPy counterpart in several test cases.

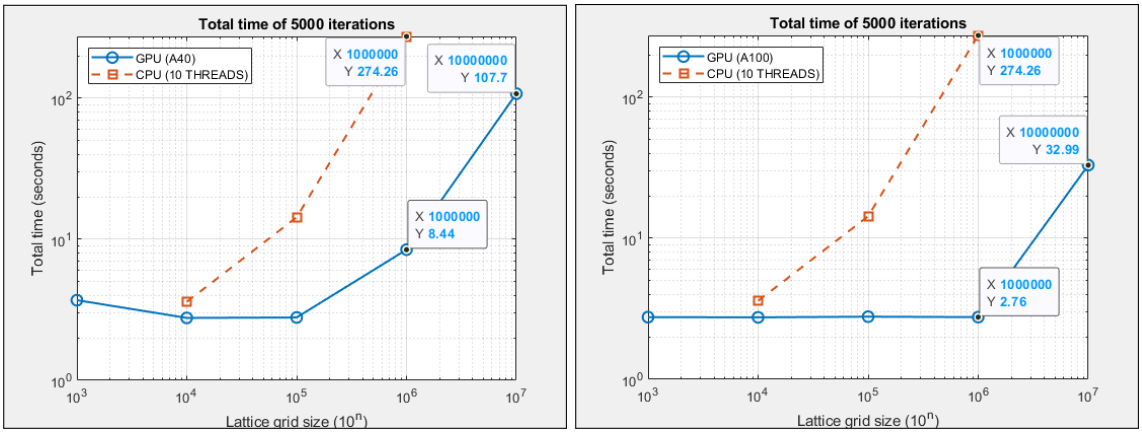
For example, when executing a large-scale matrix multiplication in the boundary.py file, the CuPy implementation showed a marked reduction in execution time compared to NumPy. Profiling data revealed that the CuPy-based code executed 15,153 function calls (15,119 primitive calls) in 5.256 seconds, whereas the NumPy implementation completed only 2,001 function calls in 6.324 seconds. Despite the increase in function calls, the GPU's parallel processing capabilities allowed CuPy to achieve faster results overall.

However, some CuPy implementations slowed down the simulation due to function incompatibilities, which led to overhead from transferring data between the host and device. Hence a complete cupy implementation was not made–meaning replacing all numpy functions with cupy functions. To fully transition from NumPy to CuPy, certain functions in the original code—particularly those of complexity O(iterative+recursive)—need to be adjusted, as they are inherently difficult to parallelize.

4.3 Numba to CUDA

4.3.1 Benchmarking

The total time taken for running the LBM simulation is shown for A40 and A100 in Figure 1a and 1b, respetively. Data points of interest are also shown in these figures.

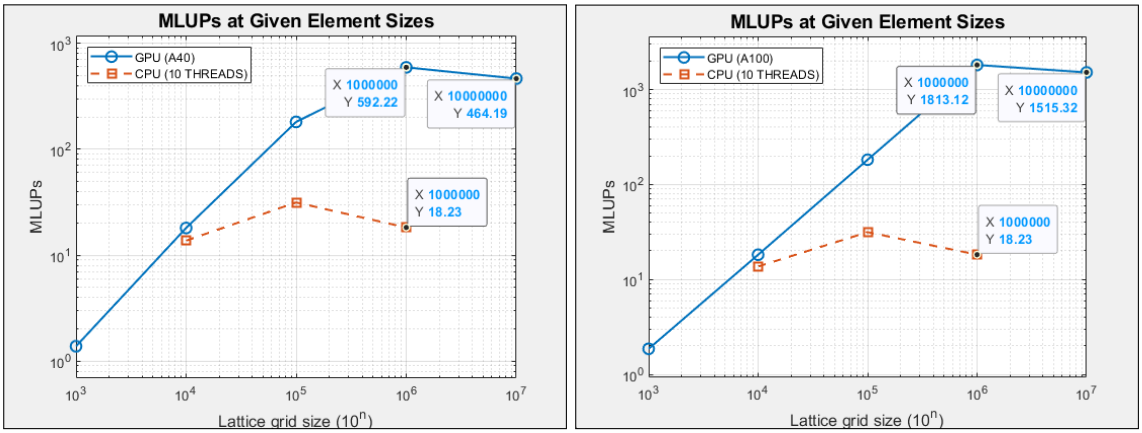


(a) Graph of time for A40 compared to cpu with 10 threads

(b) Graph of time for A100 compared to cpu with 10 threads

Figure 1: Comparison of time performance for two GPUs (A40 and A100). Data points of interest are marked

The MLUP's when running the LBM simulation is shown for A40 and A100 in Figure 2a and 2b, respectively. Data points of interest are also shown here.



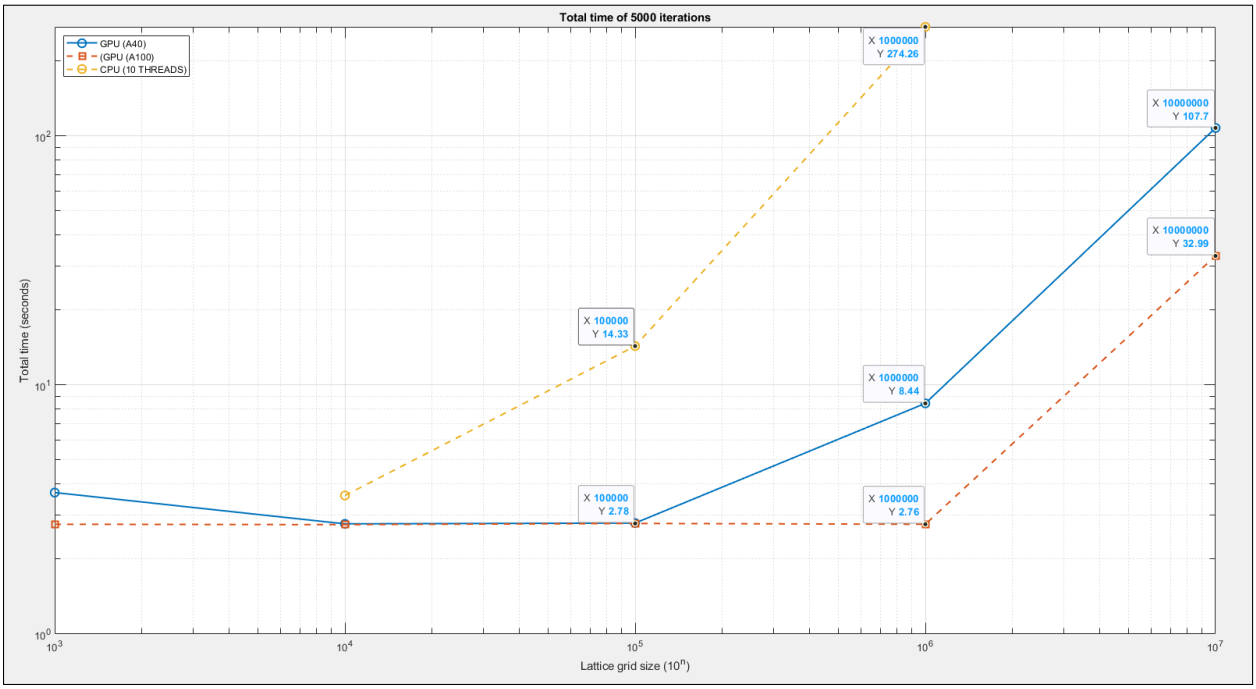
(a) Graph of MLUP's for A40 compared to cpu with 10 threads

(b) Graph of MLUP's for A100 compared to cpu with 10 threads

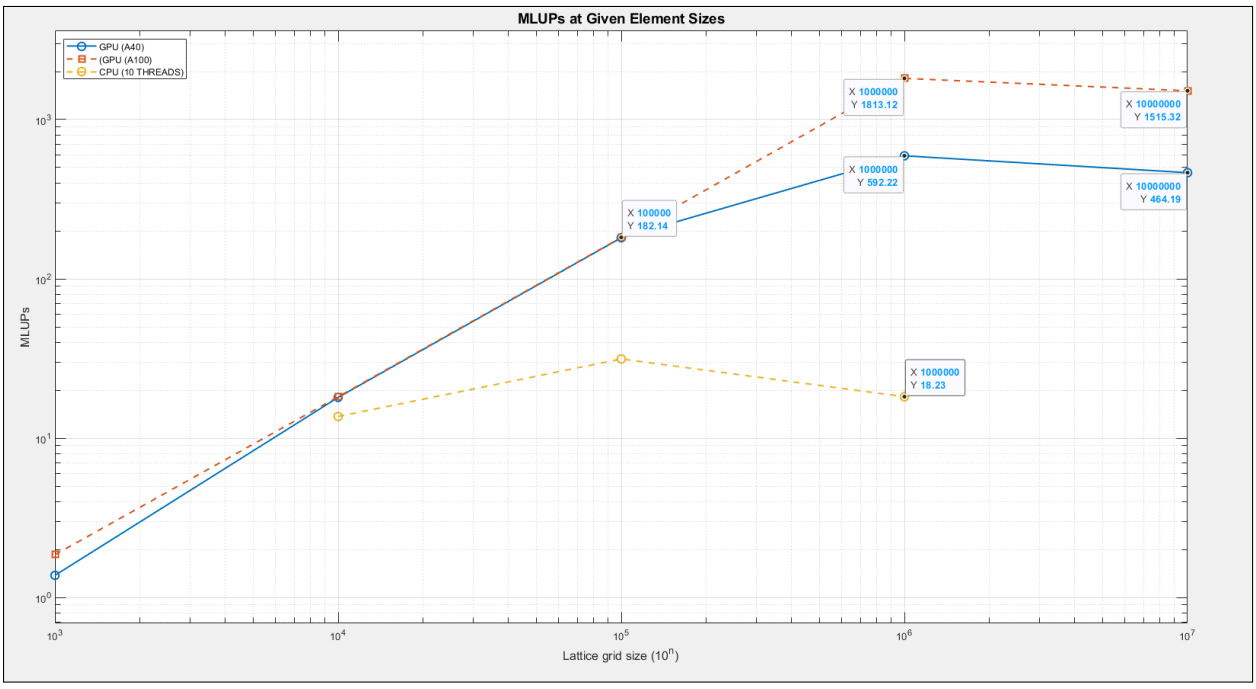
Figure 2: Comparison of MLUP's for two GPUs (A40 and A100). Data points of interest are marked

Both the parameters MLUPs and total time taken is shown in figure 3, where figure 3a shows the MLUPs of both GPUs and CPU and figure 3b shows total time for both GPUs and CPU

at the same time.



(a) Graph of MLUP's for both GPUs and CPU.



(b) Graph of MLUP's for both GPUs and CPU.

Figure 3: Comparison of MLUP's for both GPUs (A40 and A100) and CPU. Data points of interest are marked.

Table 1: MLUPS for the 2D Lid-Driven Cavity Benchmark on Different GPUs

Number of Nodes	Total time (seconds)	MLUPs on GPU
A40 48GB VRAM		
10^3	3.70	1.38
10^4	2.77	15.76
10^5	2.79	181.22
10^6	8.44	969.57
10^7	107.70	464.19
A100 40GB VRAM		
10^3	2.76	1.86
10^4	2.75	18.20
10^5	2.78	182.13
10^6	2.76	1813.12
10^7	32.99	1515.12

A table for all datapoints is provided to Table 1, where the total time and MLUPs are shown.

5 Discussion

This section will discuss the results generated from the method, and also explore potential improvements for future application.

5.1 Numpy to Cupy

Some limitations were observed. Memory transfers between the CPU and GPU posed a significant bottleneck, especially in workflows requiring frequent host-device communication. This was particularly evident in cases involving smaller datasets or functions incompatible with CuPy, where the overhead of memory transfers offset potential speedups. Additionally, some NumPy functions lacked direct CuPy equivalents or exhibited different behaviors, requiring careful code adjustments. Hence a full transition from numpy to cupy was not made.

Unlike native CUDA, CuPy abstracts thread and block allocation, automatically managing GPU resources. While this simplifies implementation, it also limits direct control over resource allocation. For advanced customization, CuPy does allow the integration of user-defined CUDA kernels, which we acknowledge as a potential avenue for further optimization

In summary, the results demonstrated that CuPy provides substantial performance improvements for appropriate use cases, such as large-scale array operations, while requiring careful management of bottlenecks and compatibility issues.

5.2 Numba to CUDA

The results indicate that the CUDA implementation of the LBM significantly outperforms the CPU-based approach for larger grid sizes, particularly when $n \geq 10^5$. However, for smaller grid sizes ($n \leq 10^4$), there is no noticeable performance advantage, and in some cases, the CPU outperforms the GPU. This outcome is expected, as the overhead associated with transferring data between the CPU and GPU becomes more pronounced for smaller grid sizes. The initialization time of CUDA kernels introduces latency, increasing the total simulation time and reducing MLUPs.

Between the GPUs A40 and A100, the results demonstrate that at lower node counts, both GPUs efficiencies are comparable, with minimal performance variations to a point of 10^5. However, when the node count reaches 10^6, the performance of the GPUs diverge, with the A100 achieving a remarkable 50-fold efficiency improvement. Theoretically, the A100 should perform worse than the A40 in float32 operations. However, when the number of nodes reaches 10^6, the A100 still demonstrates significantly superior computational capabilities compared to the A40.

5.2.1 Complexity of the Original Code

Certain segments of the original code were not parallelized due to their inherent complexity, resulting in performance bottlenecks during LBM simulations. However, these unoptimized

segments were primarily confined to preprocessing tasks rather than the core LBM functions, with the most significant bottleneck being the lattice grid construction. While this did not impact the accuracy of the results, it significantly hindered efficient testing, making simulations of very large grid sizes ($n \geq 10^8$) infeasible due to preprocessing times extending to several hours per run.

To mitigate this issue, future implementations should focus on restructuring the complex code sections for better compatibility with CUDA parallelization. A prime example is the `find_neighbors` function, which constructs the `streaming_maps` grid by identifying neighboring nodes. In its current form, this function employs a recursive depth-first search (DFS) algorithm, which is inherently difficult to parallelize due to its sequential nature and dependency on previous iterations. DFS is inherently linear and lacks repetitive steps that could be parallelized efficiently. A potential solution would be to replace it with an iterative breadth-first search (BFS) algorithm, which could better leverage parallel execution. However, such a change could disrupt existing logic, introduce bugs, and necessitate further adjustments across the codebase.

5.2.2 Utilization of CUDA Utility Tools

As described in the methodology, shared memory was employed in some functions to improve performance by reducing the frequency of global memory accesses, which are a known bottleneck in CUDA execution. By storing frequently accessed variables in shared memory, retrieval times can be minimized, potentially leading to substantial performance gains for larger grid sizes.

However, shared memory optimization was not fully implemented due to a lack of complete understanding of its behavior and limitations, compounded by time constraints. In theory, leveraging shared memory should yield significant performance improvements for large-scale simulations, where frequent access to global memory can become a dominant factor in execution time.

For the sections where shared memory was implemented, only marginal performance improvements were observed. These gains were not substantial enough to draw definitive conclusions about the effectiveness of the implementation. Further investigation is required to determine whether shared memory was utilized optimally or if additional optimizations are needed.

5.2.3 Expanding Kernel Dimensionality

Performance for larger grid sizes could be significantly enhanced by transitioning from 1D to 2D kernels, thereby increasing parallelization. This can be illustrated by examining the `density_cuda` function discussed in the methodology. In its current 1D implementation, each thread is assigned to a lattice node and iterates over all velocity directions in a loop to compute the density. By increasing the dimensionality to 2D, threads could also be allocated to individual velocity directions, eliminating the need for the for-loop and allowing each thread to perform a single velocity computation in parallel. This would increase overall parallelism and potentially improve performance.

Due to time constraints, this optimization was not applied to all functions, as the priority was to establish a functionally correct parallelized version of the LBM solver. If implemented in future iterations, special attention must be given to atomic operations, as a 2D kernel structure introduces potential contention issues when multiple threads attempt to update the same variable simultaneously. Furthermore, increasing the number of threads per kernel may reduce the number of available threads per node per iteration, which could impact overall execution efficiency. Proper optimization strategies, potentially incorporating shared memory, would be required to achieve significant performance gains while mitigating resource limitations.

5.2.4 Precision between FP32 and FP64

As mentioned in the results, the CUDA implementation was designed using single-precision floating-point numbers (FP32) instead of double precision (FP64). This decision was made to prioritize performance over numerical accuracy, as FP64 primarily benefits applications where higher precision is critical but at the cost of reduced overall performance.

Single precision (FP32) is generally more suitable for performance-oriented applications due to its higher computational throughput and lower memory usage. Modern GPUs, including the NVIDIA A100 and A40, can execute FP32 operations at up to twice the speed of FP64 on standard CUDA cores. Additionally, FP32 requires half the memory bandwidth of FP64, leading to faster memory access and improved efficiency. These advantages make FP32 the preferred choice for applications where precision is not the primary concern, particularly in simulations where numerical errors do not propagate significantly.

In the context of this project, where LBM assumes that parameters relax toward equilibrium, the impact of using FP32 regarding accuracy is minimal since small numerical errors naturally dissipate over time. However, in future implementations, if external factors such as turbulence, multiphase interactions (e.g., multiple fluids), or complex boundary conditions are introduced, numerical errors may amplify exponentially due to the chaotic nature of these phenomena. In such cases, FP64 may be necessary to ensure stability and accurately capture small-scale dynamics.

6 Conclusion

This project successfully demonstrated the potential of GPU acceleration for the Lattice Boltzmann Method (LBM) by implementing a CUDA-optimized version of an existing CPU-bound codebase. Significant performance improvements were achieved for larger lattice grids ($n \geq 10^5$), highlighting the advantages of GPU parallelism.

While the implementation proved effective for large-scale simulations, smaller grids ($n \leq 10^4$) showed limited performance gains due to GPU data transfer overheads. Additionally, certain preprocessing steps, such as recursive algorithms, remained bottlenecks, underscoring areas for future optimization. Shared memory and multi-dimensional kernels were partially explored, presenting opportunities for further performance enhancements in subsequent work.

Overall, this project underscores the feasibility of GPU-accelerated LBM simulations and provides a foundation for future scalability and optimization.