15418: Final Project Proposal

Owen Lu (olu), Yutai Long (yutailon)

March 26, 2026

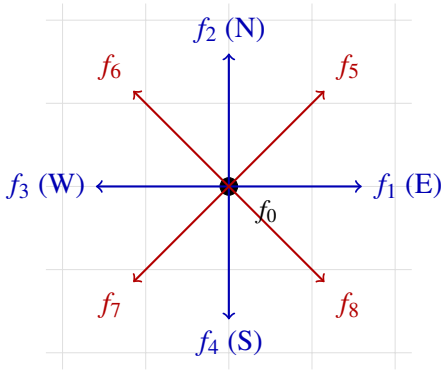
Webpage URL: https://toaster06.github.io/418-project-website/

1 Summary

We plan on implementing a GPU-accelerated 2D Lattice Boltzmann Method (LBM) fluid simulator using CUDA. LBM's two-phase structure cleanly separates a compute-bound *collision step* (no neighbor accesses) from a memory-bound *streaming step* (no computation). While the first part is embarrassingly parallel and isn't particularly interesting, the memory bottlenecks and access patterns in the second stage present opportunities for optimizations.

2 Background

Rather than directly solving the Navier-Stokes equations, LBM simulates fluid flow by tracking particle distribution functions on a grid [1, 2]. We will implement the **D2Q9** model, where each cell in the 2D grid stores 9 distribution functions f_i corresponding to 9 velocity directions shown below: stationary, N, S, E, W, NE, NW, SE, and SW.



Density and velocity are derived from these distributions:

$$\rho = \sum_{i=0}^{8} f_i, \qquad \mathbf{u} = \frac{1}{\rho} \sum_{i=0}^{8} f_i \, \mathbf{e}_i \tag{1}$$

Each timestep consists of two phases:

- **Collision:** each cell reads its 9 values, computes local density and velocity, and relaxes the distributions toward a theoretical equilibrium. This step is thus purely *local* and requires *no* neighbor communication, making it embarrassingly parallel with perfect locality.
- **Streaming:** each distribution propagates one lattice step in its velocity direction. This step involves *zero computation* and consists entirely of memory movement. At solid boundaries, distributions undergo *bounce-back* and simply reflect in the opposite direction.

This structure makes LBM fundamentally *memory-bandwidth bound*. Thus, our optimizations should focus on reducing memory traffic and improving memory access efficiency rather than reducing computation.

3 Challenge

While the collision phase is trivially parallel, the streaming phase presents several optimization challenges:

- **Data layout**: with a naive layout that stores $f_0, \ldots, f_8$ contiguously per cell, memory accesses will be *uncoalesced*. Thus, we should *interleave* the f_i's of each thread to enable coalesced accesses. Additionally, we can experiment with cache-aligned row padding to avoid cache conflicts on vertical neighbors.
- **Asymmetric access patterns**: different directions (horizontal, vertical, diagonal) have different cache and coalescing behavior: for instance, horizontal streaming enjoys good spatial locality while vertical streaming accesses cells with large strides.
- **Redundant memory traffic**: a naive implementation uses two full grids (read from old, write to new), doubling memory usage and traffic. The AA-pattern [5] eliminates the second grid with in-place streaming using alternating even & odd step logic, but it requires careful handling of boundary conditions and indexing.
- **Warp divergence**: in the collision step, obstacle / boundary cells perform a simple bounce-back while other cells perform the full collision step. Thus, we have undesirable warp divergence at these boundaries.

4 Resources

In terms of hardware, we will be using the NVIDIA GeForce RTX 2080 GPUs on the GHC machines. We will also be using their CPUs and may use the PSC machines to run experiments at larger thread counts. For OpenMP and CUDA guides, we will refer to the resources suggested in Assignments 2 and 3.

We will implement the algorithm from scratch in CUDA, using existing tutorials [3, 4] as references and for validation. In terms of the algorithm itself, we will refer to Kruger et al. [1] and Prof. Wagner's lecture notes [2].

5 Goals & Deliverables

We will start by implementing a serial CPU and naive GPU baseline (uncoalesced accesses, two-grid approach), validated against the analytical Poiseuille flow solution (which has an analytical solution). We will also implement CPU baselines using OpenMP for cross-platform speedup comparisons.

After we finish the baselines, we will address the bottlenecks identified by profiling. Our planned optimizations include representing the data in a way that allows for coalesced memory accesses, experimenting with cache-aligned row padding to reduce cache conflicts, and implementing the AA-pattern [5, 6] for in-place streaming. We also plan to explore warp-level register communication for neighbors within the same warp, and may investigate shared memory tiling if profiling reveals that neighbor accesses remain a significant bottleneck.

Beyond memory access optimizations, we were interested in investigating the tradeoff between *occupancy* and *cache efficiency*. High occupancy gives the GPU more warps to switch between during memory stalls, but since the problem is already memory-bound, additional warps may only increase cache contention without improving throughput. Thus, we suspect performance may peak below 100% occupancy, though the exact tradeoff is a question that profiling will help us answer. Throughout this process, we will revalidate correctness after each change and use Nsight profiling to guide what to prioritize next.

If time permits, we have several stretch goals in mind: extending to 3D (**D3Q19**) to test how our optimizations scale with increased memory pressure, implementing mixed FP32/FP16 precision for storage [7] to nearly double effective bandwidth, and producing a real-time flow visualization.

6 Platform Choice

We believe that LBM naturally maps to GPU hardware: the algorithm is highly-parallel as the collision phase assigns one thread per cell with no interthread communication. Moreover, the streaming phase has regular, predictable access patterns that can be fully coalesced with the right data layout, though different directions exhibit different cache locality. Since the workload is memory-bandwidth-bound, the higher memory bandwidth on the RTX 2080 is appropriate in comparison to the CPU. Moreover, CUDA provides warp-level intrinsics and control over the memory hierarchy needed for our lower-level optimizations, and Nsight provides the profiling infrastructure to measure each optimization's impact.

7 Schedule

- **Week 1 (Mar. 25 to 31):** implement a serial CPU and naive GPU baseline. Create our Poiseuille flow validator and a basic visualization tool.
- **Week 2 (Apr. 1 to 7):** extend our CPU baseline to use OpenMP and implement preliminary optimizations on the GPU, focusing on coalesced memory operations. If time permits, we may also refine our visualization tool.
- **Week 3 (Apr. 8 to 14):** implement the AA-pattern and revalidate correctness. (*Milestone report due Apr 14.*)

- **Week 4 (Apr. 15 to 21):** implement warp-level optimizations and investigate the occupancy vs. cache efficiency tradeoff.
- **Week 5 (Apr. 22 to 30):** analyze our results and run experiments (e.g. larger grids, larger thread counts) for the final report + poster. Implement stretch goals if ahead of schedule.

References

[1] T. Krüger et al., *The Lattice Boltzmann Method: Principles and Practice*, Springer, 2017.

[2] A. Wagner, *A Practical Introduction to the Lattice Boltzmann Method*, NDSU lecture notes.

[3] P. Mocz, "Create Your Own Lattice Boltzmann Simulation (With Python)," 2020.

[4] N. McDonald, "Procedural Wind and Clouds Using GPU-Accelerated Lattice Boltzmann Method," 2022.

[5] M. Wittmann et al., "Comparison of Different Propagation Steps for Lattice Boltzmann Methods," *Comput. Math. Appl.*, 65(6):924–935, 2013.

[6] P. Bailey et al., "Accelerating Lattice Boltzmann Fluid Flow Simulations Using Graphics Processors," *ICPP*, 2009.

[7] M. Lehmann et al., "Accuracy and Performance of the Lattice Boltzmann Method with 64-bit, 32-bit, and Customized 16-bit Number Formats," *Phys. Rev. E*, 106(1):015308, 2022.