15418 Final Project Report: GPU-accelerated 2D Lattice Boltzmann Method (LBM) Simulator

Owen Lu (olu), Yutai Long (yutailon)

May 1, 2026

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

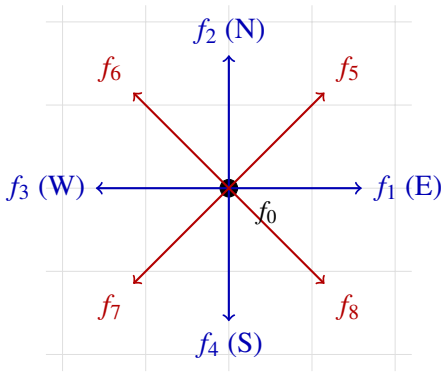
1 Summary

We implemented a GPU-accelerated 2D Lattice Boltzmann Method fluid simulator in C++ and CUDA, targeting NVIDIA RTX 2080 GPUs on the GHC machines. Starting from a serial CPU baseline, we developed several GPU implementations: a naive AoS version, a coalesced SoA version, an in-place AA-pattern version, and an AA FP16S version using shifted half-precision storage. Our final implementation reaches 8000.99 MLUPS at $N = 8192$, compared with 3007.48 MLUPS for the naive GPU implementation and roughly 18.5 MLUPS for the serial CPU baseline. The main performance gains come from optimizations that target memory coalescing and reducing memory traffic.

2 Introduction & Background

2.1 Setup

The Lattice Boltzmann Method (LBM) simulates fluid flow by evolving particle distribution functions on a grid rather than directly solving the Navier Stokes equations [1, 2]. We use the D2Q9 model, where each grid cell stores nine distribution functions f_i: stationary, N, S, E, W, NE, NW, SE, and SW. Each direction is also associated with a weight derived from the Maxwell-Boltzmann equation [1]. In our case, $w_0 = \frac{4}{9}$, $w_{1,2,3,4} = \frac{1}{9}$, and $w_{5,6,7,8} = \frac{1}{36}$.



The macroscopic density and velocity are derived from these distributions, so they do not need to be stored as separate state and are instead recomputed from the f_i when needed.

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

Each timestep consists of two phases: collision and streaming. The *collision step* is local to each cell. A thread reads the nine distributions for one cell, computes ρ and $\mathbf{u}$, and relaxes the cell toward the D2Q9 equilibrium distribution

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

Our implementation uses the BGK update

$$f_i^* = f_i + \omega \left(f_i^{eq} - f_i \right) + 3 w_i (\mathbf{e}_i \cdot \mathbf{F}), \tag{3}$$

where ω is the relaxation rate and $\mathbf{F}$ is an optional body force. Because collision only uses values from the same cell, it is embarrassingly parallel and enjoys good locality.

The *streaming step* moves each post-collision distribution one lattice cell in its corresponding direction:

$$f_i(\mathbf{x} + \mathbf{e}_i, t + 1) = f_i^*(\mathbf{x}, t). \tag{4}$$

Thus, this step performs essentially no arithmetic and is entirely memory movement. However, boundary cells need to be handled differently. At solid walls, we use bounce-back, where the distribution value is simply reflected into the *opposite* direction of the same cell:

$$f_{\bar{i}}(\mathbf{x}, t + 1) = f_i^*(\mathbf{x}, t). \tag{5}$$

For moving walls, the reflection includes an additional momentum correction from the wall velocity.

This two-phase structure makes LBM a *natural fit* for the GPU. The collision phase contains massive parallelism because each cell can be updated independently. The streaming phase is also parallel, but its performance depends heavily on memory traffic, which would benefit from the high memory bandwidth of GPUs. For this reason, our optimizations focus primarily on improving memory access efficiency rather than reducing the arithmetic in the collision formula.

2.2 Scenarios & Evaluation

We evaluate two standard 2D flow scenarios. In *Poiseuille flow*, all cells are initialized to rest equilibrium,

$$f_i(\mathbf{x}, 0) = \rho_0 w_i, \tag{6}$$

the left and right boundaries are periodic, and the top and bottom boundaries are solid no-slip walls that use bounce-back. A constant body force $\mathbf{F} = (F_x, 0)$ drives the fluid horizontally, producing a parabolic steady-state velocity distribution with the fastest flow near the channel center. We use this scenario as our main correctness test because the steady-state velocity profile has a known analytical solution. This allows us to compare the simulated horizontal velocity against the expected parabolic profile and quantify numerical error directly.

In the *lid-driven cavity* scenario, the grid is also initialized to rest equilibrium, but the domain is a closed box. The side and bottom boundaries are stationary solid walls, while the top boundary is a moving wall with horizontal velocity. Conceptually, although there is no body force, the moving lid imparts momentum to the fluid and produces a vortex inside the cavity. Unlike Poiseuille flow, this scenario is primarily useful as a more complex qualitative flow test and performance benchmark since it introduces a different boundary-condition behavior: Poiseuille uses periodic wrapping in the x direction, while cavity uses closed-wall bounce-back with a moving lid.

2.3 Performance Metric

The metric we use for evaluation is MLUPS (million lattice updates per second). One lattice update corresponds to advancing one grid cell by one timestep, including collision, streaming, and the handling of boundary-conditions. For a grid of width W, height H, and T timesteps, MLUPS is computed as

$$\text{MLUPS} = \frac{WHT}{t \cdot 10^6}, \tag{7}$$

where t is the measured runtime in seconds.

MLUPS is the standard metric for LBM because it measures the throughput of the actual simulation update and represents how quickly each implementation advances across time. FLOPS is less appropriate here because different implementations may perform different amounts of auxiliary work (e.g. extra address arithmetic, conversions, etc) while still advancing the same number of lattice cells.

Unless otherwise stated, we compute MLUPS using *kernel time* only, excluding one-time setup costs such as allocation, host-device copies, transposes, and file output. This isolates the steady-state solver performance, which is what our optimizations were all targeted towards.

3 Approach

3.1 Technologies Used

We implemented the simulator in C++ and CUDA. The CPU baseline is written in C++, and all GPU implementations use CUDA. We use Assignment 2's `CycleTimer` for timing. GPU phase timings

are measured by synchronizing after the relevant CUDA work with `cudaDeviceSynchronize()`, so the reported times include kernel execution rather than only launch overhead. For profiling, we use NVIDIA Nsight Compute to collect metrics like DRAM throughput, and L1 hit rate. These metrics were used to explain not only which implementation was faster, but why the observed speedups saturated or changed across problem sizes.

All experiments were run on the GHC machines with an NVIDIA RTX 2080 GPU, which has 46 SMs and a memory bandwidth of 448 GB/s.

3.2 Optimization Journey

N	**Naive**	**SOA**	**AA32**	**AA16**	*N*	**Naive**	**SOA**	**AA32**	**AA16**
64	817.20	951.15	**1062.72**	1016.60	**512**	2736.58	4998.62	4956.11	**7319.31**
128	1803.32	3039.23	**3430.67**	3417.46	**1024**	2941.16	5298.36	5232.63	**7761.78**
256	2415.07	6436.37	**7354.02**	7060.32	**2048**	2997.48	5368.27	5383.96	**7973.17**
					8192	3007.48	5362.82	5341.82	**8000.99**

Table 1: Throughput in MLUPS for each GPU implementation across grid sizes. The best result for each grid size is bolded.

The performance (in MLUPS) and profiling outputs for each implementation are shown in Table 1 and Table 2. We will refer to them extensively in the following subsections.

N	Version	MLUPS	DRAM	SM	L1 Hit	Odd DRAM	Odd SM	Odd L1 Hit
64	Naive	817.20	6.44%	3.83%	86.45%	—	—	—
	SOA	951.15	7.50%	4.47%	14.38%	—	—	—
	AA	1062.72	11.14%	5.23%	49.29%	5.75%	4.83%	60.63%
	AA FP16	1016.60	5.70%	5.30%	48.61%	3.33%	5.01%	65.60%
128	Naive	1803.32	13.01%	9.44%	85.90%	—	—	—
	SOA	3039.23	19.83%	13.79%	13.01%	—	—	—
	AA	3430.67	29.55%	14.47%	49.30%	18.00%	18.29%	58.78%
	AA FP16	3417.46	16.83%	17.12%	48.63%	9.29%	19.66%	63.77%
256	Naive	2415.07	22.88%	14.24%	83.28%	—	—	—
	SOA	6436.37	48.67%	31.36%	18.50%	—	—	—
	AA	7354.02	65.49%	29.33%	38.26%	41.47%	38.82%	49.81%
	AA FP16	7060.32	33.92%	35.75%	47.22%	19.02%	44.93%	65.98%
512	SOA	4998.62	79.74%	35.10%	14.13%	—	—	—
	AA	4956.11	88.31%	25.24%	23.06%	76.18%	48.50%	34.41%
	AA FP16	7319.31	72.57%	49.11%	22.79%	38.97%	60.07%	46.40%
1024	SOA	5298.36	87.29%	32.83%	13.00%	—	—	—
	AA	5232.63	89.95%	23.24%	17.42%	86.79%	48.30%	31.15%
	AA FP16	7761.78	85.74%	49.02%	12.05%	49.21%	64.11%	41.52%
2048	SOA	5368.27	88.87%	36.99%	12.55%	—	—	—
	AA	5383.96	90.73%	22.86%	12.53%	89.54%	52.99%	28.86%
	AA FP16	7973.17	89.92%	49.15%	12.45%	52.65%	65.61%	40.09%
8192	SOA	5362.82	90.23%	36.58%	12.56%	—	—	—
	AA	5341.82	90.76%	22.35%	11.68%	90.57%	52.20%	28.61%
	AA FP16	8000.99	91.06%	48.52%	9.35%	53.87%	65.56%	40.18%

Table 2: Nsight profiling metrics for each implementation across grid sizes.

3.2.1 CPU Baseline

Our CPU baseline was mainly to ensure that we have a working, naive implementation of LBM that passes our Poiseuille validator. We implemented it in C++ using the same D2Q9 update equations as the GPU versions, following standard LBM references for collision, streaming, and boundary-condition conventions [1–4]. The grid is stored in *array of structures* (AoS) format, where the nine distribution values f_i for a *single* cell are contiguous in memory. For each timestep, the CPU loops over all cells, skips boundary cells, computes density and velocity from the cell's nine distributions, applies BGK collision and optional body force, and then performs push streaming into a second grid. Solid and moving-wall bounce-back are handled from the fluid side during streaming. This version is useful both as a correctness reference and as a performance baseline, but it exposes no parallelism beyond the outer cell loop.

3.2.2 GPU: Naive

The naive GPU implementation preserves the CPU algorithm and AoS layout but assigns one CUDA thread to one grid cell. Each thread computes its corresponding location (x, y) and updates it if it is fluid. The default block size is 32×8, or 256 threads per block. This gives eight warps per block,

and since `blockDim.x` = 32, each warp covers 32 contiguous x-positions in one row of the grid. This is a natural mapping for row-major data and later becomes important for memory coalescing in the SoA layout.

The main limitation of the naive version is memory layout. In AoS, the values for one cell are contiguous, but neighboring cells are separated by nine floats for any fixed direction. Thus, when a warp processes 32 adjacent cells and all threads load the same direction i, the addresses are separated by a stride of 9 floats rather than being contiguous. This causes the warp's memory requests to touch many more memory sectors than necessary. The computation is already parallelized, but the memory access pattern prevents the GPU from using its memory bandwidth efficiently.

3.2.3 GPU: SOA

N	Version	Load	Store	Odd Load	Odd Store
1024	`Naive`	15.99	32.00	—	—
	`SOA`	2.74	4.67	—	—
	`AA`	3.70	4.00	2.73	4.67
	`AA FP16`	1.90	2.00	2.04	2.67
2048	`Naive`	16.00	32.00	—	—
	`SOA`	2.74	4.67	—	—
	`AA`	3.70	4.00	2.73	4.67
	`AA FP16`	1.90	2.00	2.04	2.67

Table 3: Memory load and store sectors per request for each implementation.

Our first optimization changes the device-side layout from AoS to *structure of arrays* (SoA). In this layout, all cells for a fixed direction are contiguous in memory: the f_i values for *every* cell are contiguous, followed by f_{i+1}. This is opposite to AoS, where the nine directions for one cell are contiguous but the same direction across neighboring cells is strided.

When a warp processes 32 adjacent x-cells and loads f_i, the 32 threads access consecutive floats, producing coalesced memory loads. The collision and streaming equations are unchanged; only the memory layout changes. Since the host-side representation remains AoS for initialization, validation, and output, we transpose from AoS to SoA after copying data to the GPU and transpose back from SoA to AoS before copying results back to the host. Our reported MLUPS use kernel time, so these transpose costs are reported separately rather than mixed into the steady-state solver throughput.

The key metric from Table 3 is *sectors per request*. The RTX 2080 GPU issues global memory at *sector* (32 bytes) granularity. Ideally, when a warp issues a memory instruction, the 32 threads access nearby and *coalesced* addresses, so the request can be served using only a small number of sectors. A lower sectors per request value therefore means that each memory request carries more useful data and less wasted bytes. At $N = 1024$ and $N = 2048$, the naive AoS kernel requires about 16 *load* sectors per request and 32 *store* sectors per request. This is the expected result of an access pattern that strides over 9 cells at a time: adjacent threads in a warp process adjacent cells, but for a fixed direction i, their addresses are separated by 9 floats. As a result, a single warp instruction is spread over many memory sectors.

As shown in Table 3, SOA reduces the load metric to 2.74 sectors per request and the store metric to 4.67 sectors per request. In the collision phase, all threads in a warp read direction i from *consecutive* cells, so the access is mostly *coalesced*. In both AoS and SoA, the reason that the store metric is slightly *higher* than the load metric is because streaming writes are less regular than collision reads: each thread writes to a neighbor, and boundary conditions may instead write to the current cell in the opposite direction. Additionally, the reported value averages over *all* reads, including to arrays that don't suffer from the uncoalesced AoS accesses.

This improvement in memory coalescing translates directly into higher MLUPS. As shown in Table 1, SOA improves over the naive GPU implementation at every grid size. The gain is modest at $N = 64$, increasing only 1.16× since the problem size is small enough such that launch overhead and boundary work matter. Once the grid is large enough to stress memory bandwidth, the benefit becomes much larger: at $N = 512$, we have a 1.83× improvement. Overall, SOA does not change the LBM arithmetic, but it makes each warp memory request *coalesced* data, allowing the GPU to use DRAM bandwidth much more efficiently.

3.2.4 GPU: AA

The issue with our AoS and SoA implementations is that it uses two grids to implement the streaming setup. Data is transferred from the source to the destination, and the two buffers are swapped after every timestep.

The AA-pattern [5, 6] keeps the SoA layout but performs streaming *in place*: rather than alternating between `grid_old` and `grid_new`, AA stores all distribution values in one grid and alternates the *interpretation* of each slot on even and odd timesteps:

- In even timesteps, slot i stores f_i. The even kernel reads the cell's own nine slots, performs collision, and writes the result into the **opposite** slot $\bar{i}$ of the same cell. This step is purely local: it has *no streaming*, no neighbor-coordinate calculation, and no boundary-condition checks beyond skipping non-fluid cells. Note that this implies that the collided values are technically stored in the "wrong" slots.

- In odd timesteps, the kernel reconstructs the streamed state by *pulling* from the backward neighbor's *opposite* slot, applies collision, and then pushes the result to the forward neighbor. This odd step is race-free because each (cell, direction) slot is read and written by exactly *one* thread under the AA interpretation.

The key point is that AA reduces memory *capacity* by half, not the memory *bandwidth* per lattice update. Importantly, each cell update *still* performs roughly 9 loads and 9 stores. This explains why AA does not produce a large speedup once memory bandwidth is saturated.

In Table 3, the AA odd kernel has almost the same sector behavior as SOA: at $N = 1024$ and $N = 2048$, SOA uses 2.74 load sectors/request and 4.67 store sectors/request, while AA odd uses about 2.73 load sectors/request and 4.67 store sectors/request. This is expected because AA odd is the step that performs the actual neighbor streaming, so its memory pattern is most similar to SOA.

The performance results in Table 1 match this interpretation. AA is faster than SOA for small grids: 1.12× at $N = 64$, 1.13× at $N = 128$, and 1.14× at $N = 256$. At these sizes, the kernels have not yet fully saturated DRAM bandwidth, so the cheaper even kernel and reduced cache pressure can

affect runtime. The reduced cache pressure comes from the fact that AA keeps only one active grid instead of separate old and new grids. Thus, although AA still performs roughly the same number of loads and stores per update, those accesses are *concentrated* within a smaller memory footprint. This makes it more likely that recently touched cache lines remain in L1 or L2 long enough to be reused by nearby warps and later directions. The profiling data in Table 2 is consistent with this: at $N = 256$, SOA has an L1 hit rate of 18.50%, while AA has 38.26% for the even kernel and 49.81% for the odd kernel. At smaller sizes the gap is even larger: at $N = 128$, SOA has a 13.01% L1 hit rate, while AA has 49.30% even and 58.78% odd.

For $N \geq 512$, however, AA and SOA are effectively tied. At $N = 1024$, SOA reaches 5298.36 MLUPS while AA reaches 5232.63 MLUPS; at $N = 2048$, SOA reaches 5368.27 MLUPS while AA reaches 5383.96 MLUPS. The profiling results in Table 2 show why: at large N, SOA and both AA kernels are already near DRAM saturation. At $N = 1024$, SOA reaches 87.29% DRAM utilization, while AA reaches 89.95% on the even kernel and 86.79% on the odd kernel. Since all of these kernels are already limited primarily by memory bandwidth, the fact that AA uses one grid instead of two does not substantially reduce the per-timestep bandwidth demand. AA is therefore most valuable as a memory-capacity optimization, enabling the simulation of larger grids on the GPU, rather than as a significant steady-state MLUPS optimization.

3.2.5 GPU: AA FP-16

The FP16 version builds on AA by storing the grid in shifted *half precision*, following the FP16S format from Lehmann et al. [7]. To avoid significant degradation in accuracy, Lehmann et al. notes that in LBM, the distribution values are usually close to their equilibrium weights w_i, and the physically meaningful part is often the small *deviation* away from that weight. If we stored raw f_i in FP16, a large fraction of these bits would be spent representing the large constant component w_i, leaving fewer significant digits for the small non-equilibrium *variation*.

Thus, we store the *deviation* from the weight $f_i^{\text{shifted}} = f_i - w_i$. This centers the stored value near zero, so FP16 precision is used on the component of the value that actually changes during the simulation.

After shifting, the value is scaled before conversion to half precision:

$$\hat{f}_i = \texttt{half}\left((f_i - w_i) \times 2^{15}\right).$$

This scaling is valid because the shifted distribution values remain in a small range in stable LBM conditions, typically within about ±2. Multiplying by 2^{15} moves these small values away from the low-exponent and denormalized region of FP16, preserving more useful significant figures during storage. On load, the value is converted back into FP32 registers as

$$f_i - w_i = \texttt{float}(\hat{f}_i) \times 2^{-15}.$$

All collision arithmetic is still performed in FP32. Thus, FP16 is used to reduce global-memory traffic, while shifting and scaling make that FP16 storage numerically more useful than storing raw distributions.

However, the MLUPS speedup is not the ideal 2×. Most notably, as shown in Table 1, AA FP16 is slightly *slower* than AA FP32 at $N = 256$, with 7060.32 MLUPS compared with 7354.02 MLUPS. At $N \geq 512$, it becomes 1.48× to 1.50× *faster*.

The reason we do not see a full 2× speedup is that halving the grid size only gives close to the full 2× improvement if memory bandwidth remains the bottleneck. That assumption holds much better for the even kernel than for the odd kernel. The profiling results in Table 2 show that at large N, the FP16 even kernel still reaches high DRAM utilization: 85.74% at $N = 1024$, 89.92% at $N = 2048$, and 91.06% at $N = 8192$. In contrast, the FP16 odd kernel reaches only 49.21%, 52.65%, and 53.87% DRAM utilization at those same sizes, while its SM utilization rises to 64.11%, 65.61%, and 65.56%. This indicates that the odd kernel is no longer purely bandwidth-bound: it is not issuing memory requests fast enough to saturate DRAM, because instruction and control-flow work have become a larger part of runtime. In particular, both kernels now have the additional overhead of converting FP16 to FP32 and vice-versa for *every* distribution value of *every* grid cell.

4 Results

Our final results and speedup are shown below in Fig. 1:

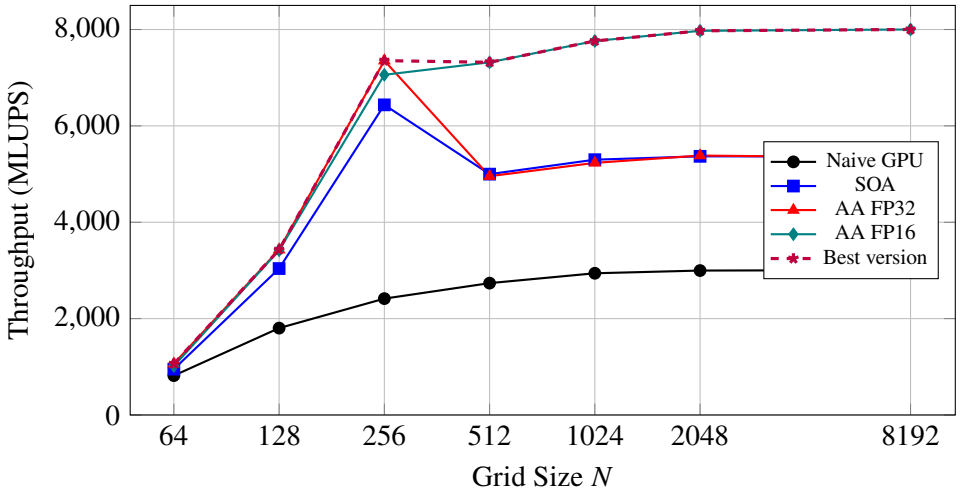


Figure 1: MLUPS of all our GPU implementations across grid sizes. The dashed line shows the best-performing implementation for each grid size.

5 Performance Analysis

5.1 Block Size Sweep

We performed a block-size sweep on the AA FP32 implementation to understand which launch conditions lead to best performance and why. Most implementations in our project use a default 32×8 block size, or 256 threads per block. This is a reasonable default because one warp is assigned

N	Rank	Block	Threads	MLUPS	Even L1	Odd L1	Even DRAM
64	1	`32x4`	128	**1160.58**	49.36%	60.17%	11.18%
	2	`64x2`	128	1159.41	49.20%	66.31%	11.12%
	3	`32x1`	32	1134.85	50.33%	59.54%	11.46%
	4	`128x1`	128	1116.69	51.00%	65.43%	10.79%
128	1	`256x1`	256	**3749.63**	51.00%	66.32%	26.73%
	2	`128x1`	128	3744.09	50.99%	66.32%	29.15%
	3	`64x2`	128	3712.41	48.77%	63.63%	29.17%
	4	`32x4`	128	3684.34	48.83%	59.07%	29.28%
256	1	`128x1`	128	**7436.16**	36.99%	60.07%	65.42%
	2	`64x2`	128	7409.94	36.68%	57.72%	65.64%
	3	`32x4`	128	7388.94	37.81%	50.44%	65.67%
	4	`64x1`	64	7374.55	37.00%	53.08%	65.62%
512	1	`256x1`	256	**5072.91**	24.17%	56.39%	86.34%
	2	`512x1`	512	5013.72	23.14%	61.04%	87.32%
	3	`256x2`	512	5011.07	31.46%	56.20%	83.97%
	4	`128x1`	128	5005.06	22.58%	52.97%	88.11%
1024	1	`256x1`	256	**5327.63**	19.46%	57.51%	89.30%
	2	`256x2`	512	5313.29	28.25%	60.86%	87.25%
	3	`512x1`	512	5293.88	21.33%	60.85%	89.39%
	4	`128x2`	256	5277.74	18.62%	54.42%	89.23%
2048	1	`128x2`	256	**5391.98**	14.27%	54.23%	90.68%
	2	`64x4`	256	5390.82	13.60%	45.62%	90.72%
	3	`128x1`	128	5390.09	14.56%	51.54%	90.79%
	4	`256x1`	256	5390.06	14.29%	55.79%	90.68%

Table 4: Top four block configurations for each grid size N. The best MLUPS value for each N is bolded.

to a contiguous group of cells, giving coalesced accesses in SoA. However, the sweep in Table 4 shows that the best AA configurations are often *wider* (high x-value) and *shallower* (low y-value).

For $N = 128$, the top two configurations are `256x1` and `128x1`. For $N = 512$ and $N = 1024$, `256x1` is the best configuration. At $N = 2048$, the top four configurations are essentially tied: `128x2`, `64x4`, `128x1`, and `256x1` all fall within about 0.04% of each other. This suggests that block shape matters most before performance is fully saturated: once the grid is large enough, several reasonable configurations reach the same DRAM bandwidth limit.

The reason why *wide* blocks perform well is the SoA row-major memory layout. For a fixed direction, adjacent x-cells are contiguous in memory. A block such as `128x1` or `256x1` places multiple warps along one row, so neighboring warps operate on adjacent horizontal memory regions. A block such as `32x4` is still coalesced within *one* warp, but *between* warps, the locations are spread across multiple rows, reducing cross-warp spatial locality.

The profiling results support this locality argument, although it should be noted in advance that this isn't the only metric that matters. At $N = 512$, increasing width from `128x1` to `256x1` raises the odd-kernel L1 hit rate from 52.97% to 56.39%, and `256x1` is the fastest configuration. At $N = 1024$, `256x1` has a higher odd-kernel L1 hit rate than `128x2` (57.51% vs. 54.42%) and is also faster. However, L1 hit rate alone does not determine performance: at $N = 1024$, `256x2` and `512x1` have

even higher odd L1 hit rates, around 60.86% and 60.85%, but both are slightly slower than `256x1`. This indicates that cache locality interacts with other factors such as occupancy, block scheduling, and other specific patterns that are hard to track in detail.

Interestingly, at $N = 64$, the best configurations are `32x4` and `64x2`, which are essentially tied at 1160.58 and 1159.41 MLUPS. Even wider blocks like `128x1` don't make much sense at this grid size because the block is wider than the row, so many lanes are inactive. This limits the benefit of horizontal locality. Even here, cache metrics alone are not sufficient: `64x2` has a higher odd L1 hit rate than `32x4` (66.31% vs. 60.17%), but their MLUPS are nearly identical. At small sizes, launch overhead, SM underfill, boundary fraction, and CTA scheduling effects can be comparable to memory-layout effects.

Moreover, block sizes that are too *large* are generally not top-tier, especially at small N. At a block size of 1024, only *one* block fits into each SM. For small grids, this can leave many of the 46 SMs idle because there are too few blocks to distribute across the GPU. Thus, we fail to fully utilize the GPU's hardware and parallelism.

At larger N, this underfill explanation becomes less important because there are enough blocks to occupy all SMs. Even then, large block sizes still perform poorly. One reason might be that a block can only retire once *all* its warps finish. Larger blocks may hold SM resources while waiting for its slowest warp. Smaller blocks give the SM more independent blocks that can retire independently, freeing resources earlier and giving the scheduler more work to mix when some warps stall.

5.2 Performance Under Different Scenarios

Version	Poiseuille MLUPS	Cavity MLUPS	% Change
`Naive`	2939.98	2981.91	+1.43%
`SOA`	5295.28	5242.89	−0.99%
`AA`	5237.69	5238.50	+0.02%
`AA FP16`	7768.72	8774.32	+12.94%

Table 5: MLUPS comparison between Poiseuille and cavity at $N = 1024$. The percentage change is computed from Poiseuille to cavity.

We compare Poiseuille flow and lid-driven cavity at $N = 1024$ in Table 5. They perform similarly on all implementations *except* for AA FP16, where cavity has a 12.94% improvement over Poiseuille.

The two scenarios use the same core collision and streaming equations, but they stress different boundary-conditions. Poiseuille uses *periodic* boundaries in the x-direction and bounce-back walls at the top and bottom implemented using modular arithmetic. Cavity is a closed box: the side and bottom walls use ordinary bounce-back (implemented using a clamp), while the top wall is a moving lid. The moving wall does introduce extra computation in cavity, but this correction applies only to the fluid row adjacent to the moving lid, which is about one row out of a 1024×1024 grid, or roughly 0.10% of cells. In contrast, Poiseuille's periodic x-coordinate handling is present in *every* fluid cell.

The reason this difference only becomes visible in FP16 is that the other kernels are dominated by memory bandwidth. In AA FP32, the odd kernel has high DRAM utilization for both scenarios:

86.88% for Poiseuille and 87.89% for cavity. At that point, extra boundary arithmetic is mostly hidden behind memory traffic, so the two scenarios perform nearly identically. In AA FP16, the memory traffic is reduced, and the odd kernel is no longer fully DRAM-bound. For Poiseuille, the FP16 odd kernel reaches only 48.78% DRAM utilization, while cavity reaches 58.63%. This suggests that Poiseuille's additional computation (modulo) prevents the kernel from issuing memory requests as frequently, exposing instruction overhead that FP32 bandwidth saturation had hidden.

Thus, the scenario comparison is consistent with the broader FP16 story. FP16 reduces memory traffic enough that non-memory overhead becomes visible. Cavity mostly avoids the global modulo path and pays only a small moving-wall cost near the lid, so it benefits more from FP16. Poiseuille, by contrast, still pays the periodic-boundary arithmetic throughout the grid, which limits the FP16 speedup.

5.3 Negative Results & Time Decomposition

N	**AA Split (s)**	**Even (s)**	**Odd (s)**
256	0.0653	0.0261	0.0392
512	0.286	0.142	0.144
1024	0.409	0.206	0.203
2048	1.57	0.781	0.786

Table 6: AA runtime and throughput (both even and odd kernels) across grid sizes. These timings were collected with per-launch synchronization, so these aren't the actual numbers used to calculate MLUPS.

We summarize our negative results below:

1. AA did not produce the large MLUPS improvement we initially expected. It halves the memory capacity, but it does not halve the memory bandwidth. Each lattice update still performs roughly 9 loads and 9 stores. This is why AA and SOA converge once N is large enough to saturate DRAM bandwidth. In Table 6, AA is faster than SOA at $N = 256$, but by $N = 512$, $N = 1024$, and $N = 2048$, the two are effectively tied. This was a useful result because it clarified the role of AA: it is primarily a memory-capacity optimization, not a MLUPS optimization.

2. Higher cache hit rate alone did not reliably predict faster performance. AA often has higher L1 hit rates than SOA, especially at small and moderate grid sizes, but this does not always translate into higher MLUPS. Similarly, in the block-size sweep, some configurations with higher odd-kernel L1 hit rates were slower than configurations with lower L1 hit rates. This shows that cache metrics must be interpreted alongside DRAM utilization, instruction overhead, and block scheduling.

3. FP16 did not achieve the ideal 2× speedup. The even kernel comes close at large sizes, which confirms that reducing the grid size helps when the kernel is bandwidth-bound. However, the odd kernel improves much less because it performs more computation through neighbor calculation, boundary checks, moving-wall handling, and FP16 to FP32 conversions. FP16

successfully reduces memory traffic, but after doing so, the odd kernel becomes limited more by instruction and control overhead. This is why the overall FP16 speedup settles around 1.5× instead of 2×.

We also considered shared-memory tiling, but the LBM streaming pattern gives little reuse for a simple shared-memory implementation. Each value streams to exactly *one* neighboring consumer. Since our strongest kernels are already bandwidth-bound and have coalesced global accesses, the added shared-memory complexity would likely increase synchronization without providing enough reuse to compensate. We therefore did not include shared memory as a final optimization.

5.4 Conclusion

Overall, our results reflect a clear optimization path. The naive GPU implementation assigns each thread to a cell but wastes memory bandwidth because AoS layout creates strided accesses. SOA fixes this by storing all cells for each direction contiguously, improving coalescing and giving up to 1.8× speedup over naive at large grid sizes. AA then reduces our memory capacity requirement by half through in-place streaming, but because it fundamentally still performs the same amount of loads and stores, it ties SOA once DRAM bandwidth is saturated. Its main value is enabling larger grids and improving small-grid behavior through a cheaper even-step kernel and reduced cache pressure.

The largest final throughput improvement comes from AA FP16. By storing shifted and scaled values in half precision while keeping collision arithmetic in FP32, AA FP16 reduces global-memory traffic and reaches about 1.5× speedup over AA FP32 for moderate and large grids. The speedup falls short of 2× because the odd kernel becomes partially compute-bound after memory traffic is reduced.

Overall, our final implementation reaches 8000.99 MLUPS at $N = 8192$, compared with 3007.48 MLUPS for the naive GPU implementation and roughly 18.5 MLUPS for the serial CPU baseline, demonstrating both the suitability of LBM for GPU parallelism and the importance of memory-aware optimizations.

6 Work Distribution

We distributed the work as follows. Both contributed to the report and poster:

- **Owen [55%]:** Naive CPU, GPU SoA, and GPU FP16S
- **Yutai [45%]:** Naive GPU, GPU AA

References

[1] T. Krüger, H. Kusumaatmaja, A. Kuzmin, O. Shardt, G. Silva, and E. M. Viggen, *The Lattice Boltzmann Method: Principles and Practice*. Springer, 2017.

[2] A. J. Wagner, *A Practical Introduction to the Lattice Boltzmann Method*. North Dakota State University lecture notes.

[3] P. Mocz, "Create Your Own Lattice Boltzmann Simulation (With Python)," 2020. Available: https://medium.com/swlh/create-your-own-lattice-boltzmann-simulation-with-python-8759e8b53b1c

[4] N. McDonald, "Simulating Wind on Procedural Terrain using GPU Accelerated Lattice Boltzmann Method," 2022. Available: https://nickmcd.me/2022/10/01/procedural-wind-and-clouds-using-gpu-accelerated-lattice-boltzmann-method/

[5] M. Wittmann, T. Zeiser, G. Hager, and G. Wellein, "Comparison of Different Propagation Steps for Lattice Boltzmann Methods," *Computers & Mathematics with Applications*, vol. 65, no. 6, pp. 924–935, 2013. doi: 10.1016/j.camwa.2012.05.002.

[6] P. Bailey, J. Myre, S. D. C. Walsh, D. J. Lilja, and M. O. Saar, "Accelerating Lattice Boltzmann Fluid Flow Simulations Using Graphics Processors," in *2009 International Conference on Parallel Processing*, pp. 550–557, 2009. doi: 10.1109/ICPP.2009.38.

[7] M. Lehmann, M. J. Krause, G. Amati, M. Sega, J. Harting, and S. Gekle, "Accuracy and Performance of the Lattice Boltzmann Method with 64-bit, 32-bit, and Customized 16-bit Number Formats," *Physical Review E*, vol. 106, no. 1, 015308, 2022. doi: 10.1103/PhysRevE.106.015308.