Lattice Boltzmann Method
B4M39GPU

Martin Čáp

January 3, 2019

Abstract

This paper describes the implementation of real-time Lattice Boltzmann simulation of fluids. Lattice Boltzmann Method (LBM) calculates the macroscopic averaged quantities of the Navier-Stokes equations. This is achieved by solving the discrete-velocity Boltzmann equation in a regular lattice. Both 2D and 3D C++ implementations are described in this paper. Furthermore, the solver can be run either on CPU or on Nvidia GPU using CUDA technology.

1 Introduction

Simulation of fluids is a very popular topic in computer graphics since it is a part of many interesting natural phenomena such as water, wind, smoke, clouds and many other. The methods that are used in the field of computational fluid dynamics (CFD) that try to portray these phenomena using real physical models are generally too slow for real-time rendering, especially considering that we may want to use these effects among many other systems. For example, in a video game, aside from simulating and rendering the flowing wind and river, we also need to take care of the game logic, animations, AI and much more. Thankfully, a method that can simulate these phenomena and can be run in real-time if used appropriately exists and it is called the Lattice Boltzmann method (LBM).

2 Algorithm

The main idea is that fluids can be perceived as a large number of very small particles interacting with each other, exchanging energy, colliding, i.e. fluids can be perceived as a group of their molecules on microscopic level. The LBM simplifies this model by representing these molecular particles in an equidistant grid of nodes (sometimes called cells or sites) called the lattice. Each node contains a distribution function $f_i(\vec{x}, t)$ that describes the probability that the particle in node $\vec{x}$ will move in the direction $\vec{e_i}$ (see Equation 1) or not move at all at time step t. In short, when using Navier-Stokes equations we are interested in calculating the macroscopic continuous values such as density

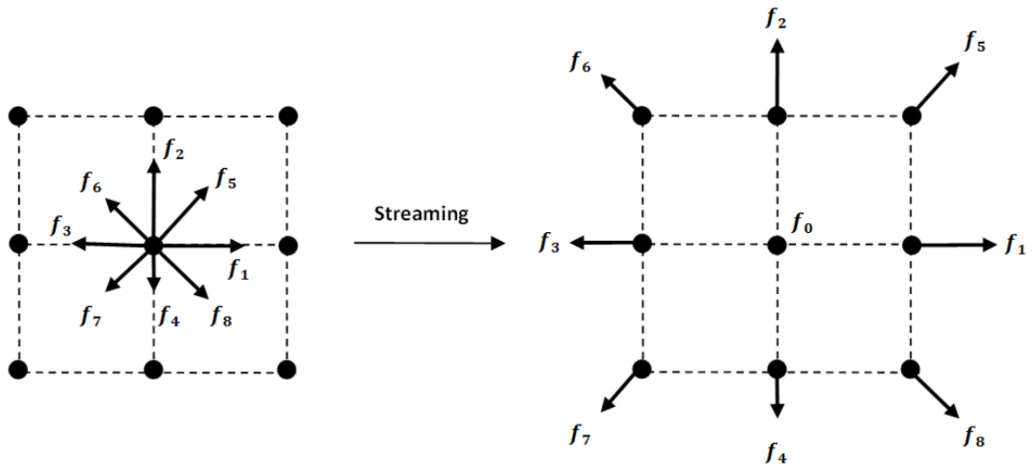


Figure 1: Streaming step for D2Q9 configuration [4].

and velocity directly, when using LBM we calculate these values from the microscopic properties of the fluid (its molecules and their chaotic movement) by establishing links between them and their continuous counterparts [1]. For the description of these properties we use the DdQq notation, where d is the number of dimensions (in our case 2 and 3) and q is the number of possible streaming directions. The most common model for 2D is D2Q9, where the streaming of particles is done in 9 directions, on axes, on diagonal, and zero vector (not moving) as shown in Figure 1. For 3D, the most common are D3Q15, D3Q19 and D3Q27. As described in [2, 3], the D3Q19 keeps the computational costs low while maintaining an isotropic lattice, hence why I chose it for this work.

The algorithm can be decomposed into multiple steps which are not dependent on the dimensionality. Let us first look at the two main steps that are essential for LBM: streaming step and collision step.

2.1 Streaming Step

The idea of the streaming step is very simple. We propagate the particle densities in the streaming directions as described by the DdQq notation. For each lattice node, we compute the updated distribution function using the values from previous frame as you can see in Figure 1 where magnitudes of vectors $\vec{f_i}$ are values of the distribution function in the streaming directions. The same principle as in Figure 1 applies in 3D, where the streaming is done in 19 directions including zero vector.

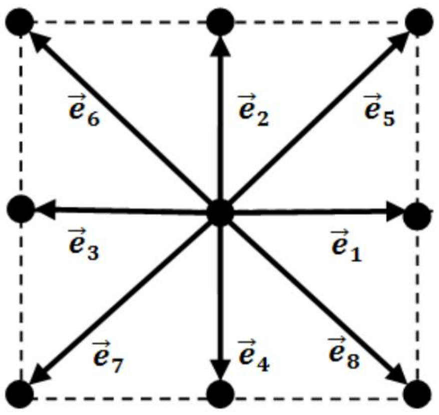
Let $\vec{u}$ be the macroscopic velocity of the particle. Vector $\vec{u}$ is simply a d dimensional velocity vector (so either 2D or 3D). As described in [4], we have a microscopic velocity vector $\vec{e}_i$ for the D2Q9 model defined as

$$\vec{e}_i = \begin{cases} (0,0) & i = 0 \\ (1,0),(0,1),(-1,0),(0,-1) & i = 1-4 \\ (1,1),(-1,1),(-1,-1),(1,-1) & i = 5-8 \end{cases} \tag{1}$$

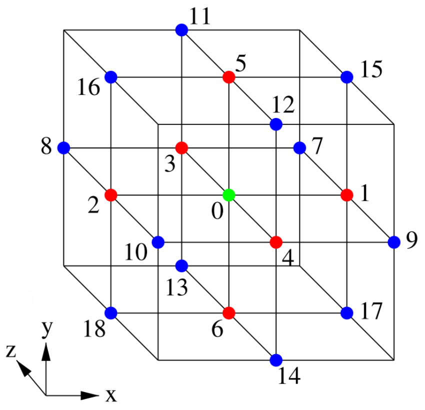
For the 3D case I have chosen to use the third ordering proposed by Woodgate et al. [2] that is described by

$$\vec{e}_i = \begin{cases} (0,0,0) & i = 0 \\ (\pm 1,0,0),(0,0,\pm 1),(0,\pm 1,0) & i = 1-6 \\ (\pm 1,0,\pm 1) & i = 7-10 \\ (0,\pm 1,\pm 1) & i = 11-14 \\ (\pm 1,\pm 1,0) & i = 15-18 \end{cases} \tag{2}$$

These orderings can be seen in Figure 2



(a) D2Q9 ordering [4].



(b) D3Q19 ordering. Third possible ordering as proposed in [2].

Figure 2: Selected orderings for the implementation.

The distribution function that we compute during the streaming step is denoted f_i^* and is important later on when the molecular particles collide during the collision step.

2.2 Collision Step

The second main step is called the collision step. Here we compute the particle collisions, hence, the particle distribution function in next frame based on the density values within each node. First, we need to compute the macroscopic density ρ which is described by the sum of distribution function values in the given node.

$$\rho(\vec{x}, t) = \sum_{i=0}^{q} f_i(\vec{x}, t) \tag{3}$$

After that, we can compute the macroscopic velocity $\vec{u}$ as

$$\vec{u}(\vec{x}, t) = \frac{1}{\rho} \sum_{i=0}^{q} c f_i \vec{e}_i \tag{4}$$

where c is the lattice velocity. Now when we have the macroscopic velocity of the examined node, we need to compute the equilibrium and update the distribution function f_i. The Bhatnagar-Gross-Krook (BGK) collision operator is used to find the equilibrium distribution f_i^{eq} for single phase flows (e.g. water, air, steam), where

$$f_i^{eq}(\vec{x}, t) = w_i \rho + \rho s_i(\vec{u}(\vec{x}, t)) \tag{5}$$

where $s_i(\vec{u})$ is defined as

$$s_i(\vec{u}) = w_i \left[3 \frac{\vec{e}_i \cdot \vec{u}}{c} + \frac{9}{2} \frac{(\vec{e}_i \cdot \vec{u})^2}{c^2} - \frac{3}{2} \frac{\vec{u} \cdot \vec{u}}{c^2} \right] \tag{6}$$

and w_i are the weights (in 2D):

$$w_i = \begin{cases} 4/9 & i = 0 \\ 1/9 & i = 1-4 \\ 1/36 & i = 5-8 \end{cases} \tag{7}$$

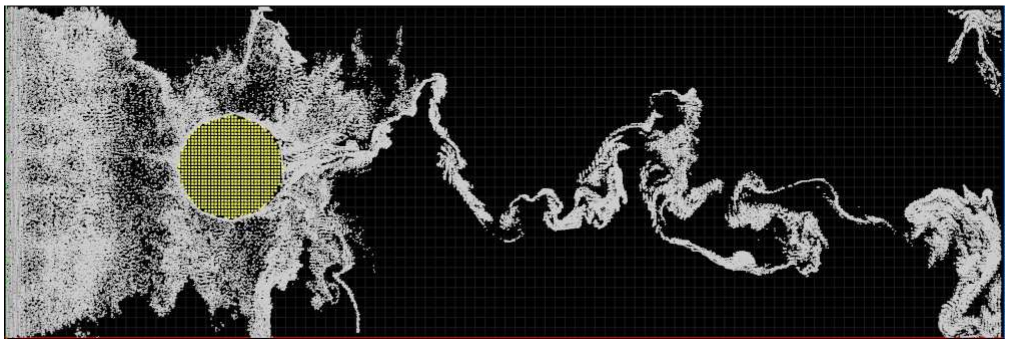
and in 3D:

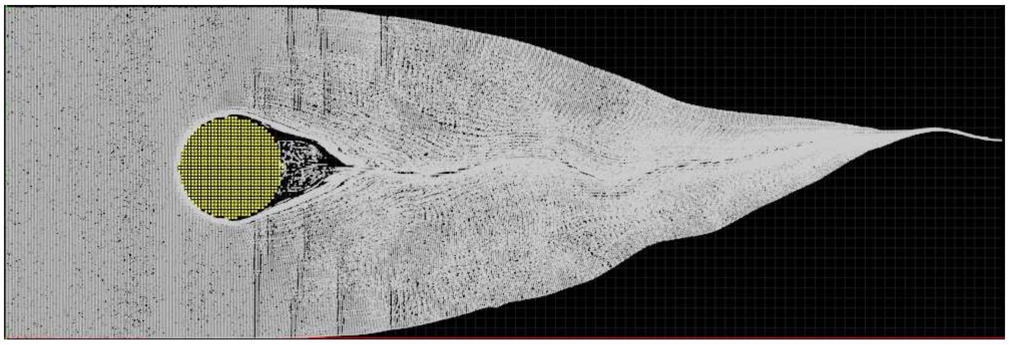
$$w_i = \begin{cases} 1/3 & i = 0 \\ 1/18 & i = 1-6 \\ 1/36 & i = 7-18 \end{cases} \tag{8}$$

Let f_i^* be the distribution function of node i that was already processed in the streaming step as mentioned earlier. The final distribution function for regular lattice node after the collision step is given by

$$f_i = f_i^* - \frac{1}{\tau}(f_i^* - f_i^{eq}) \tag{9}$$

where τ is the relaxation time which is directly related to the viscosity of the fluid. When $\tau \to 1/2$, numerical instability may arise [4].


(a) $\tau = 0.51$


(b) $\tau = 0.75$

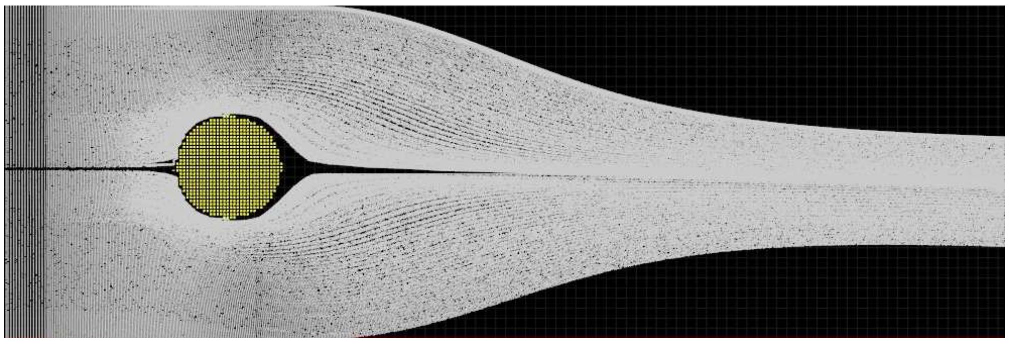

(c) $\tau = 10$

Figure 3: Flow around circular obstacle with different τ values.

2.3 Inlet Update Step

To create the flow in the simulation, we need to define an initial macroscopic velocity in selected nodes. We call these nodes inlets, since they let in the stream. This step is very similar to the collision step, only difference is that we do not compute the macroscopic density and velocity. Macroscopic density is set to 1.0 and velocity is selected as needed. Equilibrium is then established and the same computation as in collision step is done.

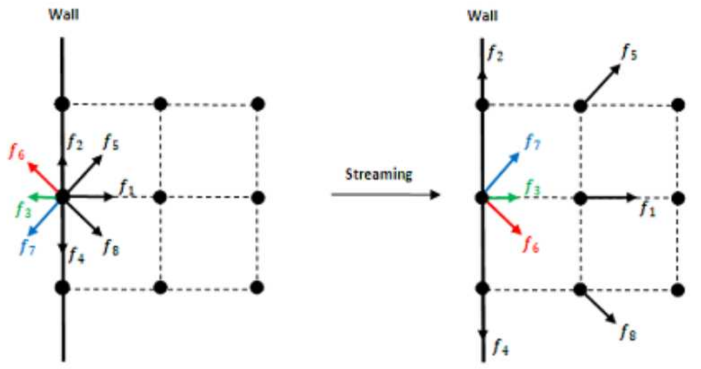


Figure 4: Full bounce back as presented in [4].

2.4 Boundary Conditions & Obstacles (Update Colliders)

The problem of boundaries and obstacles in the scene can be solved simply by using a full bounce back model as described in [3, 4]. The idea is that we simply reverse the distribution function values (left value becomes the right value, etc.) after the streaming step. The obstacles are described by a simple boolean array in 2D. In 3D, I use a simple heightmap where all nodes that lie below the heightmap are considered as obstacles. The full bounce back rule is very beneficial for the parallel algorithm since we do not need any information such as normals of the obstacle boundary other than the distribution function itself. You can see in Figure 4 a similar approach where only the incoming distribution vectors are reversed.

2.5 Particle Movement Step

Since we want to transport our particles, we need to move them according to the macroscopic velocities that are described in the lattice. Here, we find out the coordinate position of each particle and do bilinear or trilinear interpolation of the velocities that surround the particle in 2D and 3D, respectively. In other words, in 2D, we find four adjacent lattice nodes that surround the particle and interpolate their macroscopic velocities. In 3D, we find eight adjacent nodes and do the same. The final interpolated velocity is then added to the position of the particle.

If the particle reaches boundaries of the whole simulation, multiple cases can occur. For top and bottom boundaries in 2D and for left and right boundaries in 3D, we cycle the particles through as if there were no wall when the option “mirror sides” is enabled. For example, if a particle were to leave the 2D simulation area at the top of the screen, we would move it to the bottom of the screen and we would keep its horizontal coordinate x as it were. If “mirror sides” is disabled, we simply respawn the particles. At the end of the simulation area, usually called outlet, we respawn the exiting particles in desired spawn area (e.g. left edge in 2D, left wall in 3D) randomly with uniform distribution.

2.6 Main Loop

The algorithm is presented differently in many articles. The approach I've used does the algorithm steps in this order:

1. update inlets
2. streaming step
3. update colliders (obstacles)
4. collision step
5. move particles (not to be confused with streaming step)

3 CPU Implementation

The implementation on the CPU is a basic sequential implementation of the algorithm. We use two lattices, one for rendering while the other is computing the next frame. It is possible to implement indexation that supports in-place update of the lattice as proposed by Latt [5] but it is out of scope of this work. The lattice is saved in memory as a one dimensional array of Nodes where each node holds either 9 or 19 float values in 2D and 3D, respectively.

There was an attempt to parallelize the main steps of the algorithm using OpenMP library. Unfortunately, after initial tests, the usage of 8 threads produced slower results than usage of single thread since the code was not initially produced with OpenMP parallelization in mind. Due to time constraints the CPU parallelization was omitted from the final application.

The CPU implementation provides two options of respawning particles: randomly or linearly. Linear respawn means that the particles respawn "in line" on the y axis in 2D or "line by line" on the y and z axes in 3D. The value of τ can be controlled at runtime showing differences between fluids with low and high viscosity. Fluids with low viscosity create eddy currents since they have a high Reynolds number.

4 GPU Implementation

The LBM is an embarrasingly parallelizable problem. Thanks to this, the CPU and GPU implementations do not differ much. Each lattice node is managed by one thread and since we represent the lattice as one dimensional array, the indexation is also very straight forward. From the Nsight results it was observed that the biggest bottleneck (as can be seen in section 7) is the movement of particles where we access lattice nodes adjacent to the particles. Especially in the case of 3D LBM the memory access of 8 adjacent nodes is a problematic section of the algorithm. In the future, I'd like to try and solve this by using texture memory in 2D and texture array in 3D where the bilinear interpolation is done through hardware (and trilinear interpolation is then reduced to two texture taps and simple interpolation between the two values).

For the collision step we make use of shared memory cache that loads the whole block of lattice nodes. This is most effective when the size of blocks is rather small since we want to run a good amount of blocks on single streaming multiprocessor (SM). This means that we use 256 threads in 2D and 32x2 threads in 3D. This configuration of blocks was used for time measurements.

One interesting detail I'd like to mention is that it's possible to enable an option to change from CPU to GPU implementation and vice versa at runtime when using LBM 2D by defining LBM_EXPERIMENTAL and recompiling. This is highly unstable and not fully debugged but it provides a simple way to see speed differences between the two implementations at runtime. One large caveat of this is that we need to copy all particle positions back to CPU when switching from GPU to CPU implementation.

4.1 OpenGL Interoperability

The biggest bottleneck of the whole application is usually the transfer of data between CPU and GPU, especially if we need to transfer data in each frame. In the initial GPU implementation (and in the CPU implementation), the particle positions had to be uploaded to the OpenGL vertex buffer object (VBO) after each update of the lattice. With large amounts of particles, this process screeches the whole application to a halt. Since the algorithm is very nicely parallelizable, the idea is to run each simulation step on the GPU and not to transfer any data between CPU and GPU at all. This is possible thanks to the interoperability capabilities between CUDA and OpenGL where a CUDA kernel can directly manipulate data that is already buffered on the GPU using the VBO. This can be seen in the moveParticlesKernelInterop kernel function, where we set the particle positions directly. For this to work, we need a cudaGraphicsResource that we map to the VBO resource. The pointer acts as a simple glm::vec3 array in our case.

The GPU implementation provides one option that is not present in the CPU implementation which is simple color mapping of the particle velocities to viridis color map. This is done through mapping another VBO of particle colors for usage in CUDA kernels. Examples of this mapping can be seen in Figure 5 and Figure 6.

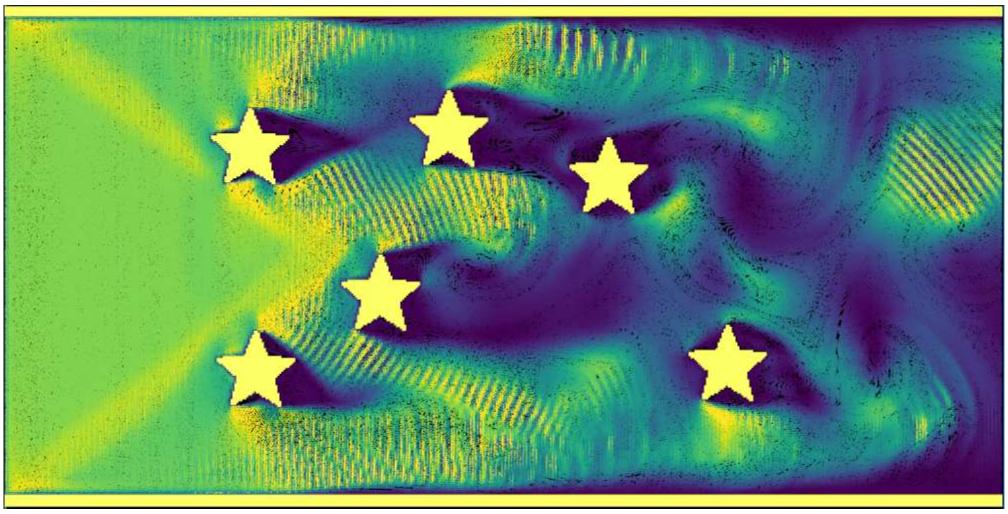


Figure 5: Visualization of particle velocities using viridis color map in 2D.

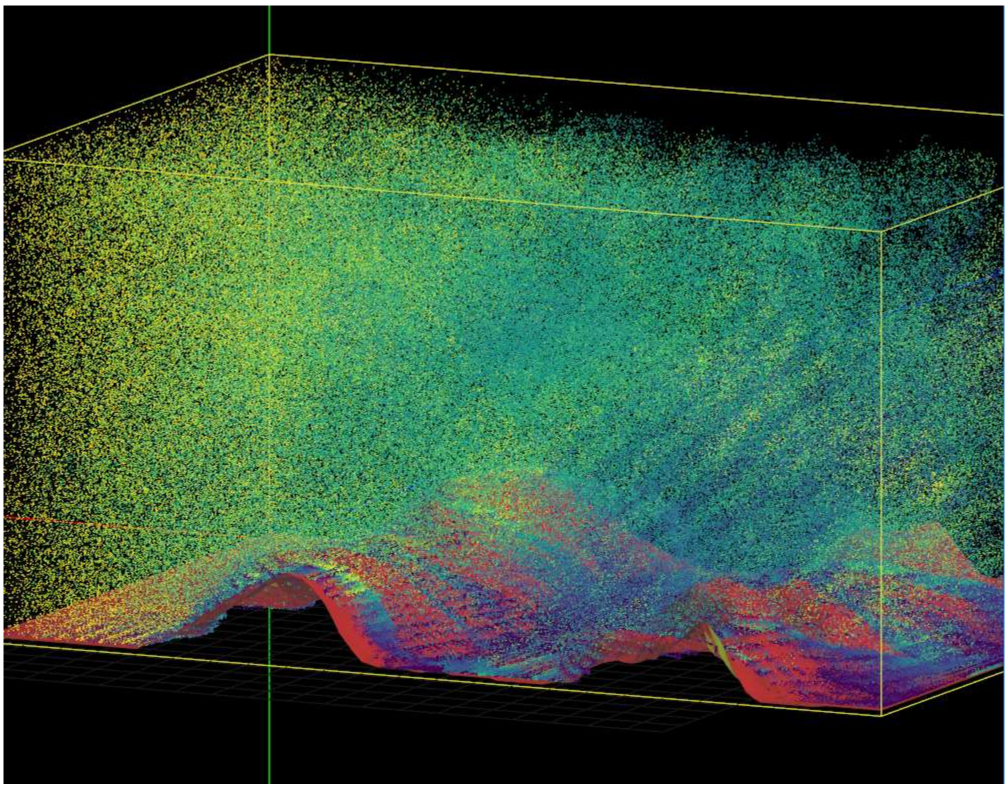


Figure 6: Visualization of particle velocities using viridis color map in 3D.

5 Application Details

5.1 Scenes

The obstacles and heightmaps are simple .ppm (ASCII) files that can be created in Gimp and loaded as scenes for the simulation. Only the red channel of images is used for scene creation. In 2D, any image pixel with intensity larger than 0 (in red channel) is considered as an obstacle. For 3D, the heightmap is generated in such manner, that pixels with intensity set to 0 are at ground level and pixels with intensity 255 (maximum range for .ppm files) are vertices that reach the top of the bounding volume of the scene.

5.2 Controls and Configuration

The application can be controlled at runtime using the provided user interface or some of the selected keyboard shortcuts. W, S, A, D keys are used to operate the camera location. Q and E are used for camera rotation in 3D. R resets the simulation and T pauses it. I, O and P are used in 3D to set front, side and top view of the camera, respectively. Some of the variables that are not modifiable at runtime can be set before running the application in the configuration file "config.ini". Command line arguments can also be used. Available options are "-h" for help, "-t" for selecting LBM type (3D or 2D), "-s" for setting scene file, "-m" for whether to measure time (true or false), "-lh" for setting lattice height (3D only), "-p" for number of particles, "-c" for whether to use CUDA or not (true or false), "-tau" for setting the value of τ, and many more for time measurements (see help for more details). It is important to note that command line arguments take precedence before the configuration value parameters (they overwrite them). This means that the user can select whether he wants to use 2D/3D simulation or whether to use CUDA or CPU implementation without recompiling the code.

Available options in the configuration file are:

- LBM_type: {2D, 3D}
- VSync: {0, 1, 2} (0 is off, 1 is 60FPS, 2 is 30FPS)
- lattice_height: (int)n (only in 3D)
- scene_filename: *.ppm
- use_CUDA: {true, false}
- block_dim_2D: 2^n, $2 \leq n \leq 10$
- block_dim_3D_x: 2^n, $2 \leq n \leq 10$
- block_dim_3D_y: 2^n, $2 \leq n \leq 10$
- draw_streamlines: {true, false} (2D CPU only)
- max_streamline_length: (int)n (2D CPU only)
- num_particles: [0, cca 50000000]

- camera_speed: (int)n
- tau: [0.5005, 10.0]
- window_width: (int)n
- window_height: (int)n
- autoplay: {true, false}
- measure_time: {true, false}
- log_measurements_to_file: {true, false}
- print_measurements_to_console: {true, false}
- avg_frame_count: (int)n

6 Testing

The application was tested on multiple scenes (over 30) during its development. Furthermore, one of the main reasons both 2D and 3D LBM implementations were kept was that the 2D version reveals errors very quickly. The macroscopic velocities (see Figure 7) and velocities of advected particles were visualized for easier debugging. Basic streamline drawing was also implemented for 2D CPU version which can be seen in Figure 8.

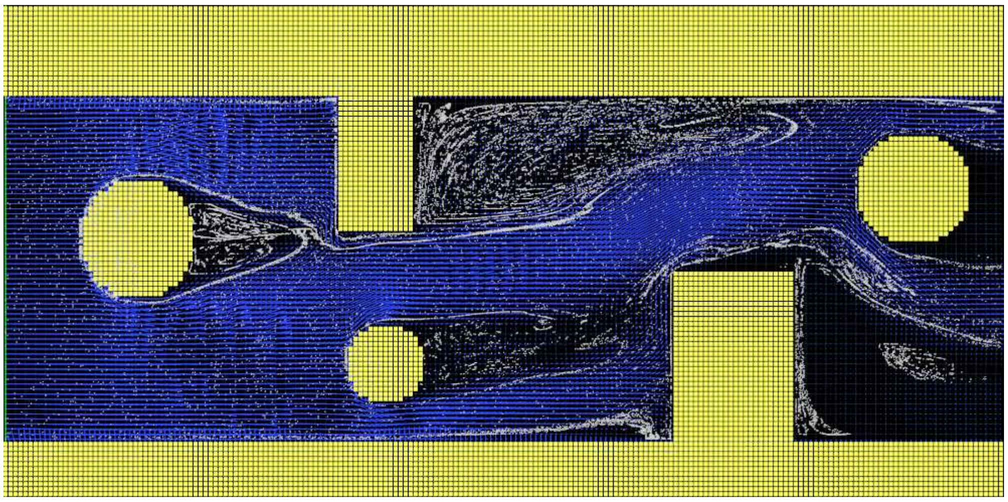


Figure 7: Visualization of macroscopic velocity vectors with $\tau = 0.75$.

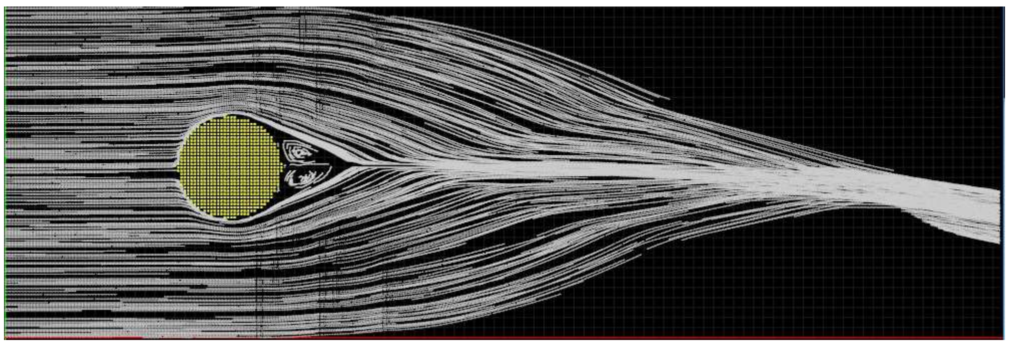


Figure 8: Streamlines drawn for 1000 particles with $\tau = 0.75$.

7 Results

The algorithm was tested on two computers: a desktop and a notebook. See Table 1 for their specifications.

	Desktop	Notebook
Operating system	MS Windows 10 Home (64-bit)	MS Windows 10 Home (64-bit)
CPU	Intel Core i7-6700K @ 4.00GHz 4.00GHz	Intel Core i7-7700HQ @ 2.80GHz 2.81GHz
GPU	Nvidia GeForce GTX 1080	Nvidia GeForce GTX 1050 Ti
Memory	32.0 GB RAM	16.0 GB RAM
Compute capability	6.1	6.1
CUDA driver version	10.0	10.0
CUDA cores	2560	768

Table 1: Specifications of computers that were used for measurements.

The measurements were initially done on some selected scenes in both 2D and 3D (tables 2 through 5). Additional measurements were performed for more comprehensive results where the effects of different lattice sizes and number of particles were considered individually. These results were obtained from a simple scene with a circular obstacle and are presented in tables 6 through 9. The times presented in tables are averages of 1000 frames for all 2D scenes up to 1 million particles, otherwise they are averages of 100 frames (3D scenes and 2D scenes with 10 million particles and more). The block configurations were, as mentioned before, 256 threads per block in 2D and 32x2 threads per block in 3D.

As the reader can observe from the presented data, the speedups for large scenes, especially in 2D, are quite significant. The largest bottleneck of the GPU implementation seems to be handling large quantities of particles and as I've suggested in previous chapters, I'd like to improve the performance there in future updates. Most significant difference between CPU and GPU versions can be seen for the 4096x4096 lattice (over 16.7 million lattice nodes in total) in 2D where the GTX 1080 provides around 85 times faster computation than the i7-6770K CPU. The GTX 1050 Ti provides around 49 times faster simulation than the i7-7700HQ CPU.

8 Conclusion & Future Work

The LBM method was implemented in 2D and 3D. The usage of GPU was crucial in speeding up the simulation to speeds that provide a real-time simulation even for fine lattices and large amounts of particles. The application is quite extensive and provides users with a lot of freedom and possibilities when it comes to testing LBM. In future, the memory accesses can be optimized for shared memory usage and the trilinear interpolation for particle movement can be rewritten to use texture arrays. Furthermore, some of the high Reynolds number methods such as the subgrid model [6] can be implemented to stabilize the computation when simulating turbulent flows.

References

[1] N. Maquignon, "Everything you need to know about the Lattice Boltzmann Method." `http://feaforall.com/creating-cfd-solver-lattice-boltzmann-method/`, 2017. [Online; accessed: 2018-12-30].

[2] M. A. Woodgate, G. N. Barakos, R. Steijl, and G. J. Pringle, "Parallel performance for a real time Lattice Boltzmann code," *Computers & Fluids*, 2018.

[3] M. Schreiber and D.-T. M. P. Neumann, *GPU based simulation and visualization of fluids with free surfaces.* PhD thesis, Diploma Thesis, Technische Universität München, 2010.

[4] Y. B. Bao and J. Meskas, "Lattice Boltzmann method for fluid simulations," *Department of Mathematics, Courant Institute of Mathematical Sciences, New York University*, 2011.

[5] J. Latt, "Technical report: How to implement your DdQq dynamics with only q variables per node (instead of 2q)," *Tufts University*, pp. 1–8, 2007.

[6] X. Wei, Y. Zhao, Z. Fan, W. Li, S. Yoakum-Stover, and A. Kaufman, "Blowing in the wind," in *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pp. 75–85, Eurographics Association, 2003.

Scene	Number of particles	CPU time [ms]	GPU time [ms]	Speedup
100x100_02 (10,000)	10k	0.92	0.49	2.51
	100k	1.80	0.85	2.11
	1m	10.92	2.93	3.72
	10m	99.33	18.24	5.45
300x100_03 (30,000)	10k	2.45	0.43	5.66
	100k	3.37	0.56	6.01
	1m	12.56	1.46	8.62
	10m	102.55	10.78	9.51
512x256_01 (131,072)	10k	11.55	0.99	11.66
	100k	12.76	1.09	11.68
	1m	24.20	2.02	11.99
	10m	142.53	11.66	12.22
	40m	535.91	41.73	12.84
2048x2048_01 (4,194,304)	10k	867.78	12.55	69.14
	100k	874.22	12.28	71.18
	1m	909.76	13.10	69.46
	10m	1224.15	19.07	64.20

Table 2: Measurements for 2D LBM on desktop computer. Values in round brackets denote total number of lattice nodes in the scene.

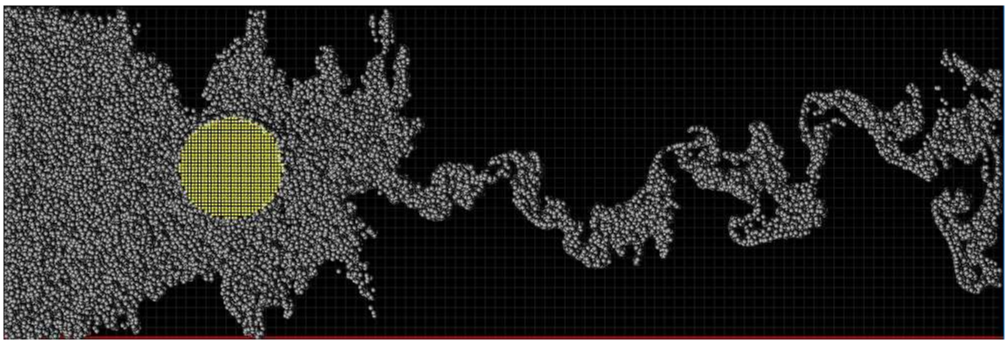


Figure 9: Another possible visualization using pointsprites.

Scene	Number of particles	CPU time [ms]	GPU time [ms]	Speedup
100x100_02 (10,000)	10k	1.41	0.56	2.51
	100k	2.51	0.98	2.56
	1m	12.85	6.96	1.85
	10m	112.12	33.79	3.32
300x100_03 (30,000)	10k	3.31	0.72	4.62
	100k	4.73	0.84	5.64
	1m	15.57	3.26	4.78
	10m	125.40	22.67	5.53
512x256_01 (131,072)	10k	14.68	1.18	12.43
	100k	15.49	1.33	11.67
	1m	29.95	4.02	7.46
	10m	156.18	23.23	6.72
	40m	588.30	88.36	6.65
2048x2048_01 (4,194,304)	10k	964.20	23.31	41.36
	100k	962.39	23.41	41.11
	1m	1000.18	24.78	40.36
	10m	1361.42	42.07	32.36

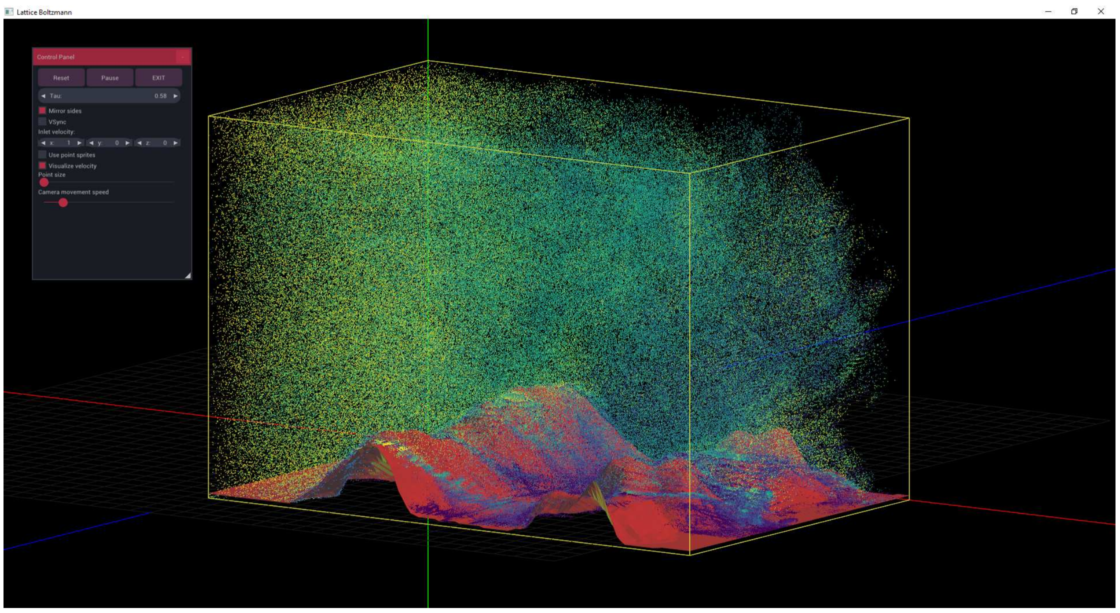
Table 3: Measurements for 2D LBM on notebook. Values in round brackets denote total number of lattice nodes in the scene.

Scene	Number of particles	CPU time [ms]	GPU time [ms]	Speedup
60x40_02 $h = 60$ (144,000)	10k	29.05	3.48	8.35
	100k	30.76	3.81	8.07
	1m	50.66	6.64	7.63
	10m	250.30	31.42	7.97
120x80_04 $h = 80$ (768,000)	10k	182.23	13.84	13.17
	100k	184.37	14.11	13.07
	1m	206.97	18.21	11.37
	10m	431.78	62.26	6.94
200x200_01 $h = 100$ (4,000,000)	10k	1361.74	62.97	21.63
	100k	1363.03	66.53	20.49
	1m	1387.35	68.12	20.37
	10m	1641.21	132.87	12.35

Table 4: Measurements for 3D LBM on desktop computer. Values in round brackets denote total number of lattice nodes in the scene and h denotes the height of the scene.

Scene	Number of particles	CPU time [ms]	GPU time [ms]	Speedup
60x40_02 $h = 60$ (144,000)	10k	35.83	4.11	8.71
	100k	38.44	4.90	7.85
	1m	62.58	11.16	5.60
	10m	293.48	70.48	4.16
120x80_04 $h = 80$ (768,000)	10k	234.06	14.43	16.22
	100k	218.49	15.07	14.50
	1m	248.78	23.56	10.58
	10m	509.63	95.51	5.34
200x200_01 $h = 100$ (4,000,000)	10k	1550.62	66.31	23.39
	100k	1531.45	66.53	23.02
	1m	1559.63	74.29	20.99
	10m	1884.39	150.51	12.52

Table 5: Measurements for 3D LBM on notebook. Values in round brackets denote total number of lattice nodes in the scene and h denotes the height of the scene.



Tau: 0.58

[✓ Mirror sides]

VSync

Inlet velocity: x: 1 y: 0 z: 0

Use point sprites

[✓ Visualize velocity]

Figure 10: Screenshot of the application with its user interface.

	Desktop			Notebook		
Lattice size	CPU [ms]	GPU [ms]	Speedup	CPU [ms]	GPU [ms]	Speedup
128x128 (2^{14})	11.15	2.65	4.21	14.00	4.97	2.82
256x256 (2^{16})	16.31	2.38	6.85	20.57	5.04	4.08
512x512 (2^{18})	36.42	2.54	14.34	44.25	6.58	6.72
1024x1024 (2^{20})	174.38	5.25	33.22	218.97	9.28	23.60
2048x2048 (2^{22})	915.33	13.95	65.62	1021.02	25.62	39.85
4096x4096 (2^{24})	3938.81	45.90	85.81	4426.68	89.91	49.23

Table 6: 2D measurements with varying lattice sizes for scene containing single circular obstacle with 1 million particles.

	Desktop			Notebook		
Lattice size	CPU [ms]	GPU [ms]	Speedup	CPU [ms]	GPU [ms]	Speedup
32x32x32 (2^{15})	28.88	4.80	6.02	35.72	10.96	3.26
32x32x64 (2^{16})	37.62	5.18	7.26	46.78	13.25	3.53
64x64x32 (2^{17})	51.12	6.29	8.13	63.29	12.27	5.16
64x64x64 (2^{18})	84.59	9.34	9.06	106.31	18.47	5.76
64x64x128 (2^{19})	164.01	13.78	11.90	199.81	18.92	10.56
128x128x64 (2^{20})	327.63	22.07	14.85	398.14	27.73	14.36
128x128x128 (2^{21})	801.16	39.24	20.42	943.62	43.10	21.89
128x128x256 (2^{22})	1652.89	70.90	23.31	1858.9	75.61	24.59

Table 7: 3D measurements with varying lattice sizes for scene containing single circular obstacle with 1 million particles.

	Desktop			Notebook		
Num. particles	CPU [ms]	GPU [ms]	Speedup	CPU [ms]	GPU [ms]	Speedup
100k	6.14	0.97	6.33	8.01	1.26	6.36
1m	16.30	2.38	6.85	20.53	5.08	4.04
10m	124.81	11.96	10.44	151.5	25.40	5.96
50m	603.67	46.87	12.88	730.54	107.03	6.83

Table 8: 2D measurements with variable number of particles for scene containing single circular obstacle with lattice dimensions 256x256 (2^{16} lattice nodes).

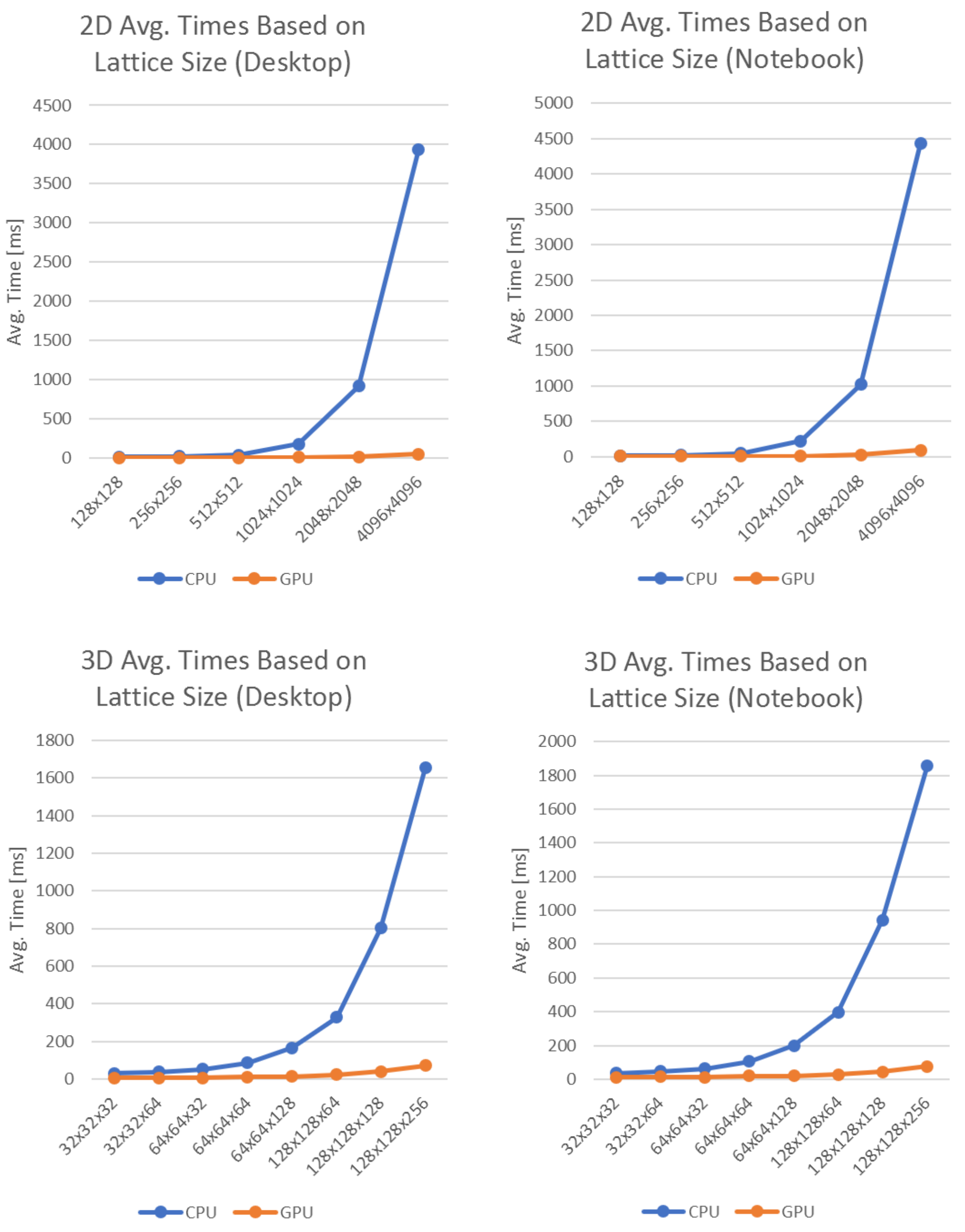


Figure 11: Graphs depicting average times from Table 6 (top) and Table 7 (bottom).

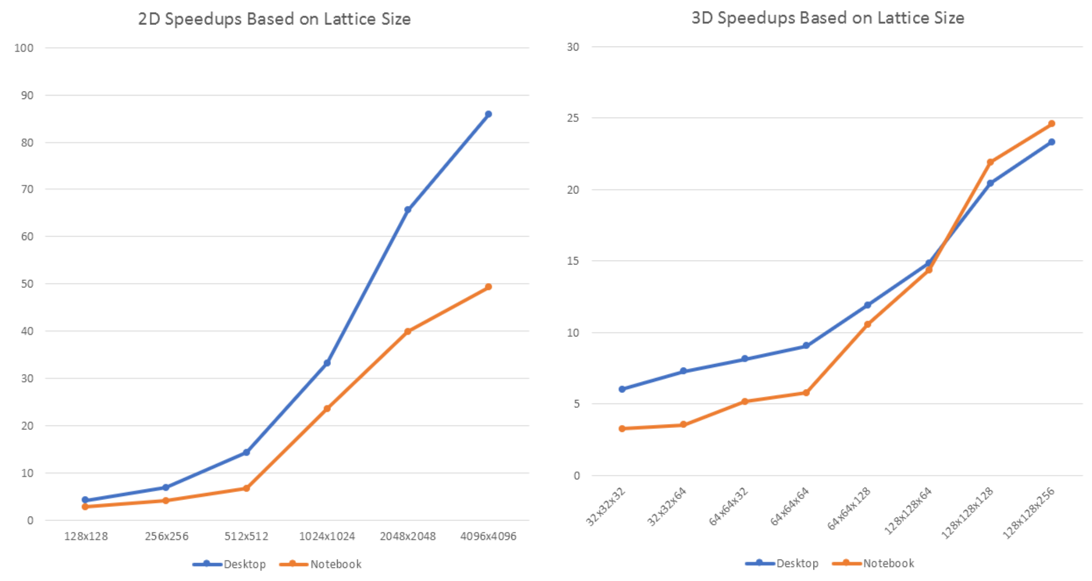


Figure 12: Graphs depicting speedups from Table 6 (left) and Table 7 (right).

	Desktop			Notebook		
Num. particles	CPU [ms]	GPU [ms]	Speedup	CPU [ms]	GPU [ms]	Speedup
100k	14.33	1.60	8.96	19.04	2.17	8.77
1m	37.92	4.92	7.71	47.47	10.83	4.38
10m	230.91	29.93	7.72	312.2	72.93	4.28
50m	1098.14	98.10	11.19	1575.36	344.06	4.58

Table 9: 3D measurements with variable number of particles for scene containing single circular obstacle with lattice dimensions 32x32x64 (2^{16} lattice nodes).