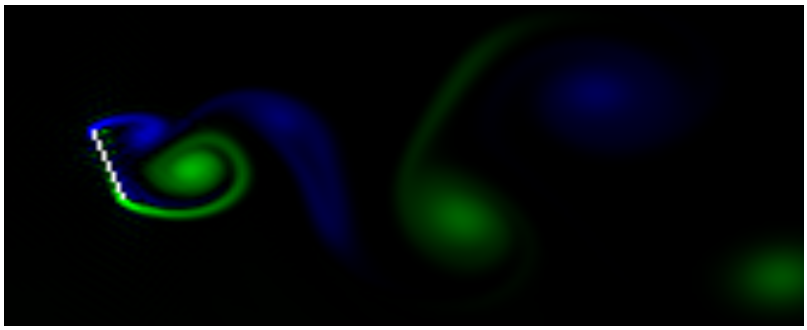


Lattice-Boltzmann Algorithm Using GPU Acceleration

Implementation Overview

Henry Friedlander & Tom Scherlis
Shady Side Academy Physics 3
19 Feb 2017

Contents

List of Figures

1 Introduction

1.1 Abstract

CUDA and OpenGL packages were used to graphically accelerate the Lattice-Boltzmann Method (LBM) Algorithms. Through the use of parallelization and efficient memory usage, considerable optimization and efficiency is possible with LBM simulation. In this simulation, we model constant 2 dimensional fluid flow through a medium, and at every lattice node the fluid's curl, probabilistic velocity vectors, and pressure are calculated and available to be plotted on the screen.

Demo: `https://www.youtube.com/watch?v=MSMGGoP24Hw`

1.2 Eulerian Viewpoint vs. Lagrangian Viewpoint

There are generally two schools for tracking each particle of fluid inside a fluid simulation: the Eulerian Viewpoint and the Lagrangian Viewpoint. The Lagrangian Viewpoint is the conventional, intuitive algorithm for keeping track of fluids inside experiments; this method tracks each individual molecule's position and velocity vectors. While this method is effective for discretizing the movement of solids, for fluids this simulation is limited by the number of nodes in the model being used. The Eulerian Viewpoint is much more effective for modeling fluids, because not only is it faster but also the simulation is not limited by the computer hardware. Rather than tracking each particle of fluid, this viewpoint observes fixed points in space and measures how the fluid density, temperature, velocity, etc. change on that point.

1.3 Maxwell-Boltzmann Distribution

The entirety of this algorithm rests on the Maxwell-Boltzmann Distribution[1] of statistical mechanics. This distribution is a probabilistic distribution which describes particle speeds in idealized gases where particles are able to move relatively freely. This is the distribution at thermal equilibrium where $\vec{v}$ is the thermal velocity, m is the mass, k is the Boltzmann constant, and T is temperature.

$$D(\vec{u}) = \frac{m}{2\pi kT} e^{-\frac{m\vec{v}^2}{2kT}} \tag{1}$$

A particle-speed probability distribution describes the likelihood of the particle reaching that speed during the next time step. Since this distribution only applies to classical ideal gases, this representation of gases is not completely accurate because of various effects (e.g. relativistic speed limits, quantum exchange interactions, and van der Waals interactions).

Rarefied gases are gases that undergo a reduction in their average density. They are similar to ideal gases, and therefore the Lattice Boltzmann Algorithm, which is a relatively computationally simple simulation technique, is accurate for approximating these gases. Lattice-Boltzmann is typically used for computer modeling of aircraft flying through the upper levels of the atmosphere because of its high accuracy and low computational cost.

[1] `http://physics.weber.edu/schroeder/javacourse/LatticeBoltzmann.pdf`

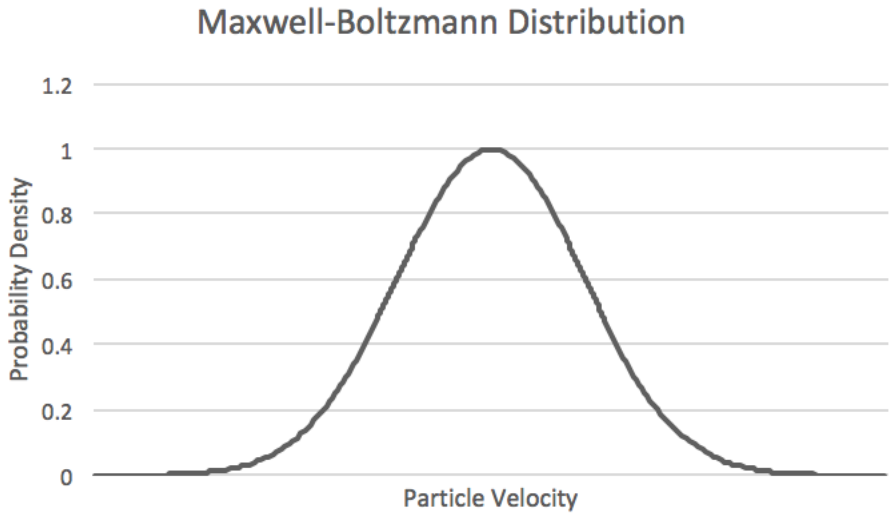


Figure 1.1: The distribution of velocities vs probabilities

As seen in Fig. 1.1, the function has a peak, which indicates the most likely velocity at which the particle would be traveling through a point in space. For example in Equation 1, which describes fluid at thermal equilibrium, the peak would be at 0 because the net velocity should be 0. In cases where the system is not in equilibrium the peak may be compressed, stretched, or shifted.

2 Lattice-Boltzmann Method

2.1 The Lattice

The Lattice-Boltzmann Method,[2] as its name suggests, operates on a lattice. A lattice is a 2d array, and in the case of a computer simulation, it corresponds to pixels on a screen. At each point of the lattice, since the Eulerian Viewpoint is being implemented, the simulation tracks the flow through that point and the likelihood that particles will move in various directions when at that point. Although particles passing through that point on the lattice in the real world are able to move in any direction, this method restricts particles to only be allowed to move in 9 directions from the point.

[2] `http://physics.weber.edu/schroeder/javacourse/LatticeBoltzmann.pdf`

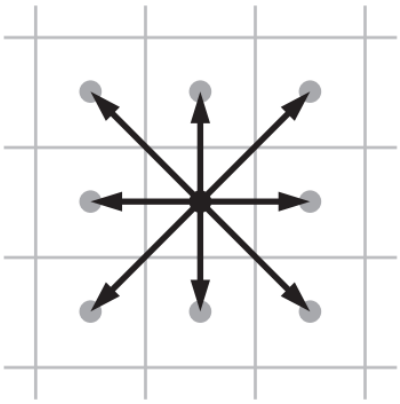


Figure 2.1: Possible directions for the number probabilities
Source: Weber State University

As seen above in Figure 2.1, there are 8 vectors emanating from the point on the lattice and one vector at the center. These vectors, defined starting from the top and proceeding clockwise, are n_n $\langle 0,1\rangle$, n_{ne} $\langle 1,1\rangle$, n_e $\langle 1,0\rangle$, n_{se} $\langle 1,-1\rangle$, n_s $\langle 0,-1\rangle$, n_{sw} $\langle -1,-1\rangle$, n_w $\langle -1,0\rangle$, and n_{nw} $\langle -1,1\rangle$. Lastly, vector $\langle 0,0\rangle$ is defined to be n_0. Since this approach discretizes the directions which a particles will be able to travel, probabilities that a particle will travel in any of these directions need to be attached to model the continuous Maxwell-Boltzmann Distribution. After double integrating the distribution over the entire range of v_x and v_y, the optimal probabilities can be found. These probabilities (or weights), based on the shape of the Maxwell-Boltzmann Distribution, can be calculated to be $w_0 = \frac{4}{9}$, $w_n = w_s = w_e = w_w = \frac{1}{9}$, and $w_{nw} = w_{ne} = w_{se} = w_{sw} = \frac{1}{36}$ (the subscript denotes the direction).

2.2 Introduction

The algorithm sets up initial conditions and iterates a collision step and a stream step. There are special cases for edges and barriers that are also evaluated throughout the stages of the algorithm.

2.3 Initial Conditions

Initially, we assumed that the fluid was equally distributed across the screen, the density of fluid at every point was arbitrarily set to be 1, and that each position the x component of each position's velocity vector, v, was set to be the flow speed according to a set of equations.

$$n_0 = \frac{4}{9}(1 - 1.5v^2)$$

$$n_n = \frac{1}{9}(1 - 1.5v^2)$$

$$n_s = \frac{1}{9}(1 - 1.5v^2)$$

$$n_e = \frac{1}{9}(1 + 3v + 3v^2)$$
$$n_w = \frac{1}{9}(1 - 3v + 3v^2)$$
$$n_{ne} = \frac{1}{9}(1 + 3v + 3v^2)$$
$$n_{se} = \frac{1}{9}(1 + 3v + 3v^2)$$
$$n_{nw} = \frac{1}{9}(1 - 3v + 3v^2)$$
$$n_{sw} = \frac{1}{9}(1 - 3v + 3v^2)$$
$$pressure = 1$$
$$v_x = v$$
$$v_y = 0$$

2.4 Collision Step

Inside the collide step, the algorithm redistributes all the probabilistic vectors pointing inward to outward. In other words, the probabilistic vectors "collide" with each other to create new vectors facing outward. Although this step of the algorithm is conceptually very straightforward, deriving these probabilistic vectors collisions proves to be very difficult.

Define ω, where ν is defined to be the viscosity of the fluid.

$$\omega = \frac{1}{3\nu + 0.5}$$

Inside the collide step, this formula derived from the Maxwell-Boltzmann distribution is used to calculate the new number densities.

$$n_{dir}^{new} = n_{dir}^{old} - \omega(n_{dir}^{eq} - n_{dir}^{old}) \tag{2}$$

In this equation, *dir* represents one of the nine directions, and n_{dir}^{eq} is defined in equation 3 below. Simple approximation of the Maxwell-Boltzmann Distribution with a Taylor series yields, where $\vec{e}$ is defined to be the macroscopic velocity and $\vec{u}$ is defined to be the thermal velocity:

$$D(\vec{v}) \approx w_i(1 + 3(\vec{e}_{dir} \cdot \vec{u}) + \frac{9}{2} \cdot (\vec{e}_{dir} \cdot \vec{u})^2 - \frac{3}{2}\vec{u}^2)$$

Using the definition of a number density ($n^{eq} = \rho D(\vec{v})$), find the number density by substituting the approximation of the probabilistic distribution.

$$n_{dir}^{eq} = \rho[w_{dir}(1 + 3(\vec{e}_{dir} \cdot \vec{u}) + \frac{9}{2} \cdot (\vec{e}_{dir} \cdot \vec{u})^2 - \frac{3}{2}\vec{u}^2)] \tag{3}$$

In the equation above ρ is defined to be the density of the fluid at the node. Finally substitute the equilibrium number pressure (the approximate concentration of particles moving on a given vector) into equation 2 to get the next number density of the next step.

2.5 Streaming Step

2.5.1 Middle Case

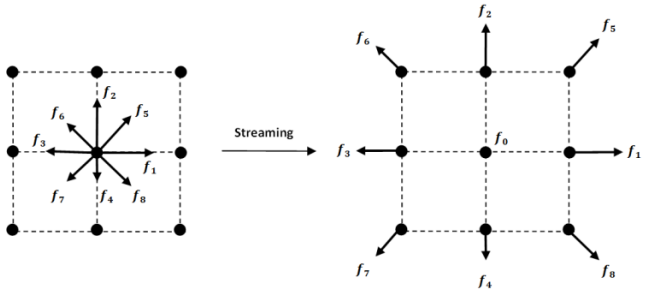


Figure 2.2: Visualization of the Middle Case
Source: Weber State University

As seen by the figure 2.2, the simulation "streams" all the probabilistic vectors to the adjacent points in their direction. The output values of this step are then fed back into the collision step of the next iteration.

2.5.2 Edge Case

If the current point being streamed has a position that is on the edge of the screen (the where either x or y equals either 0 or one less than the length of width), then set the vector in the opposite direction equal to what a vector in that direction would be initialized to. For example, if the current point is at the top of the screen, then since the vector pointing upward cannot be streamed to any point displayed on the screen, the vector pointing downwards would be initialized to the same value it was initialized in subsection 2.2 (in this case, $n_s = \frac{1}{9}(1 - 1.5v^2)$).

2.5.3 Bounce Step

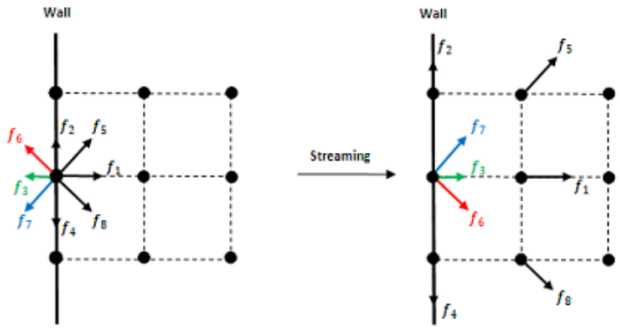


Figure 2.3: Visualization of the Bounce Case
Source: Weber State University

The bounce step handles the case where when the program streams the vectors inside an internal solid barrier cell. In this case, the simulation "bounces" the vector in the opposite direction. Figure 2.3 displays the "bouncing" process. For example, the red vector, before streaming, is facing in the northwest, but after the streaming step, it "bounces" back to, after this step, direct toward the southeast.

2.6 Storing Values for plotting

In this simulation, the user has the option to display different properties of the simulated gas: pressure, x or y component of the net probabilistic velocity vector, speed^2 (that is, the square of the magnitude of the velocity vector), or curl (defined in equation 4). To calculate ρ (the pressure) at the specific point, sum all of the number densities.

$$\rho = \sum n_{dir}$$

Calculate the x and y components of the net probabilistic velocity vector for plotting.

$$u_x = \frac{(n_e + n_{ne} + n_{se}) - (n_w + n_{nw} + n_{sw})}{\rho}$$

$$u_y = \frac{(n_n + n_{ne} + n_{nw}) - (n_s + n_{se} + n_{sw})}{\rho}$$

Curl for a given point is calculated using the values of u_x and u_y to the north, south, east, and west of the given point.

$$Curl = (u_y^{east} - u_y^{west}) - (u_x^{south} - u_x^{north}) \tag{4}$$

3 GPU Acceleration

3.1 Why Use GPU Acceleration

GPU acceleration enables the use of graphics processing hardware for computationally intensive algorithms. GPU's have thousands of compute units (cores), which make them very well suited to parallel algorithms. There is considerable overhead when running GPU accelerated code, so care must be taken to ensure that the extra time to setup CUDA acceleration actually results in a performance increase. A GPU accelerated algorithm must meet the following requirements:

1. Massively parallel
2. Order independent (CUDA threads execute asynchronously)
3. Bottle-necked by computation, not memory

Thankfully, LBM fulfills all three of these requirements and thus there is considerable opportunity for speed increases from using CUDA.

3.2 CUDA – Compute Unified Device Architecture

Cuda is Nvidia's proprietary GPU computation acceleration system. The name describes both the architecture itself as well as the software package to utilize it. CUDA is a system designed to run computationally expensive, parallel algorithms on a GPU. The system is proprietary, and thus only works on systems with CUDA-capable, Nvidia GPU's.

CUDA uses "kernels" to describe functions designed to run on the many cores of the GPU. When implementing kernels in C, the keywords `__global__` and `__device__` are used to designate GPU-side functions, as distinct from CPU-side functions. A `global` function runs on the GPU, and can be called from any device including the CPU or GPU. A `device` function runs on the GPU, but can only be called from GPU code, such as another `device` or a `global`.

Alternatively, OpenCL is another good GPU acceleration architecture; however, it is more difficult to implement and not as optimized as CUDA. CUDA requires Nvidia hardware whereas OpenCL does not.

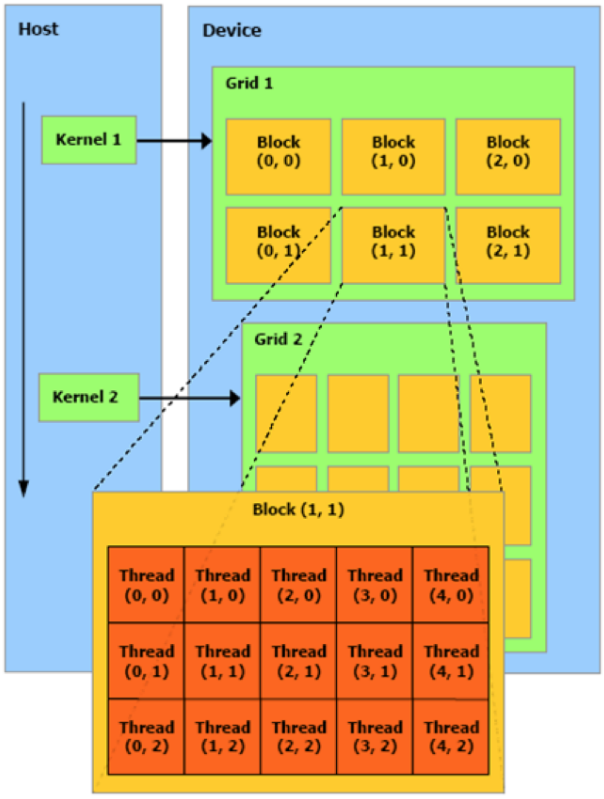


Figure 3.1: CUDA Block Diagram
Source: Colorado School of Mines

As shown in Fig. 3.1, the kernel functions are distributed across grids of threads. Each grid is divided into `Blocks`, then redivided into `Threads`. In our implementation, each core has one thread which runs the kernels for one pixel. Therefore, we have a 2d array of threads with dimensions equal to the display dimensions.

Kernel code is distributed to the threads by streaming multiprocessors (SMs). SMs queue up all kernel requests and allocate CUDA compute units for them to run on.

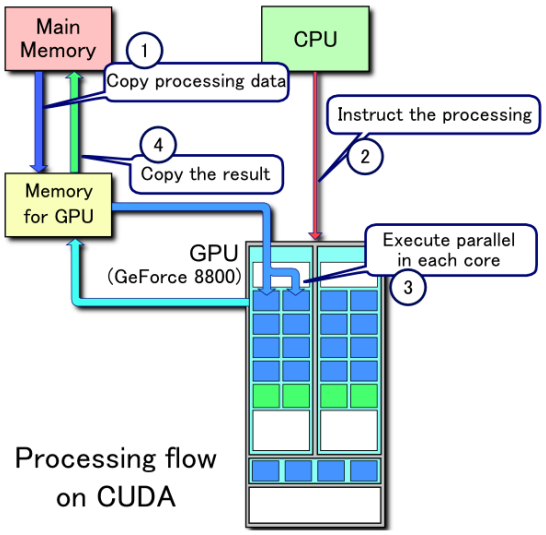


Figure 3.2: CUDA Processing Flow
Source: Wikimedia

Fig. 3.2 shows the processing flow of a GPU operated command. The diagram shows how memory is accessed and returned to the CPU in a conventional CUDA-accelerated program. In our system, however, we never send result data back to the CPU. Instead, we render it directly with OpenGL. This is more efficient than piping the data back and forth on the PCIe bus (Peripheral Component Interconnect Express bus), which connects the GPU to the CPU. Transmitting data between them is very expensive because the process is limited to the speeds of the PCIe bus, which is an order of magnitude slower than the GPU memory bandwidth (Fig. 3.3).

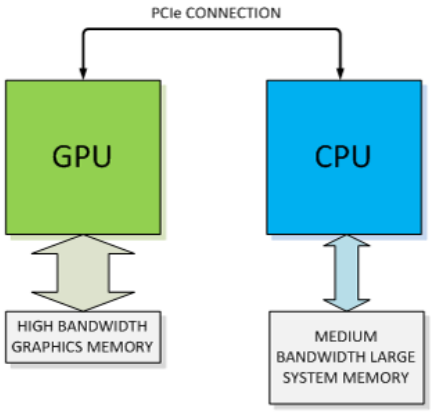


Figure 3.3: Comparison of memory bandwidths
Source: Nvidia

3.3 OpenGL – Open Graphics Library

OpenGL is a high-speed platform independent graphics library. It runs at a low abstraction level, which allows for more custom configurations at the cost of some design simplicity. We used a need-to-know approach to OpenGL, because learning all of it would be a full project on its own.

OpenGL is primarily a 3d game engine, so we used a few workarounds to render a 2d image. We create a rectangle object that is the same size as the display, map our 2d image to it as a texture, then render orthogonally to display a dynamic 2d image. One of the key advantages of using OpenGL is that the texture data exists exclusively on the GPU. By copying our image data directly into the dedicated texture buffers from the CUDA code, we remove the need to ever transmit the data to or from the CPU on the PCIe bus.

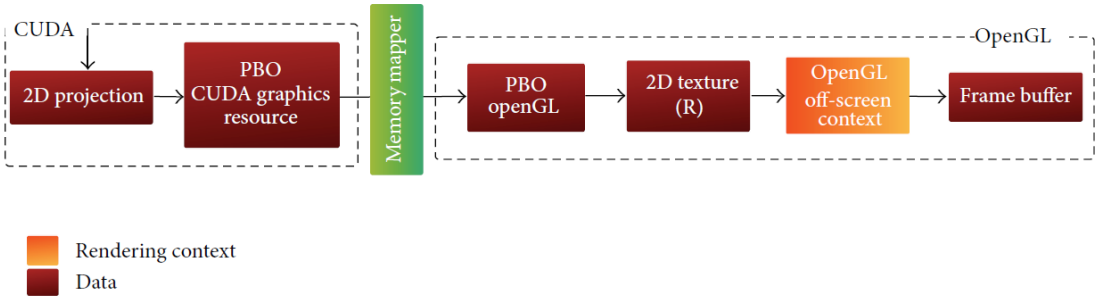


Figure 3.4: Inter-operation with CUDA and OpenGL
Source: Marwan Abdella via Researchgate.net

Fig. 3.4 shows the interaction between CUDA and OpenGL. As you can see, we write the frame data directly into a pixel buffer object (PBO), which is then mapped as a 2d texture onto a rectangle object before being rendered. We use OpenGL's double-buffering to achieve smooth refreshing without any flickering.

4 Using CUDA to implement LBM

4.1 Process Flow

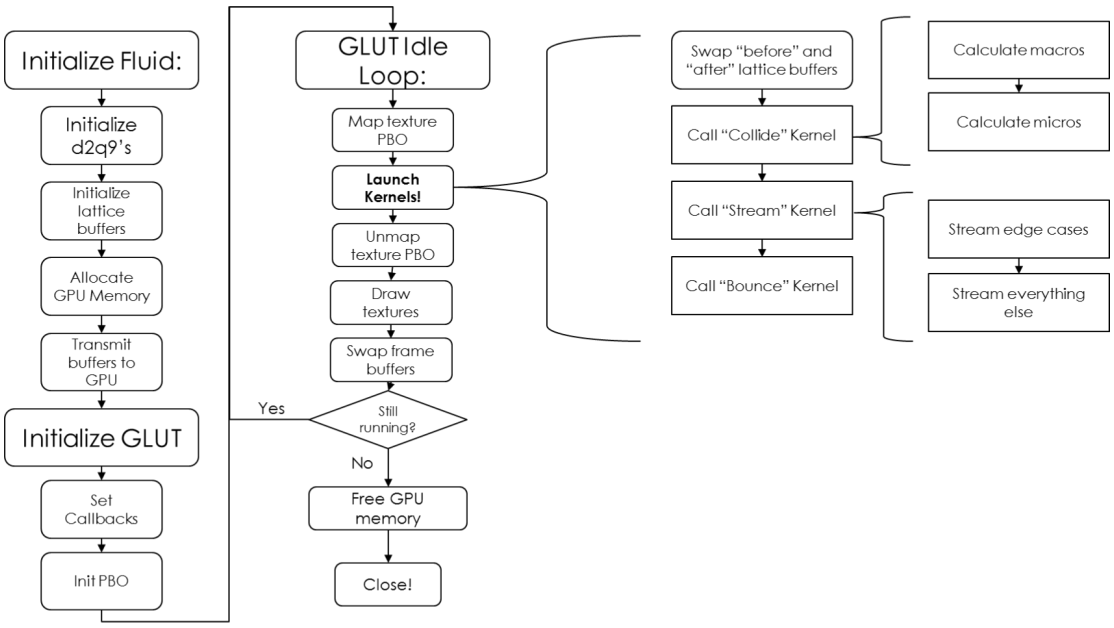


Figure 4.1: Process Flow of CUDA-Accelerated LBM

4.2 Initialization

When the program is first run, the simulation calls `InitFluid`, which initializes the initial conditions of the fluids and the variable related to the fluid. To store the number density vectors and various related to these vectors, pressure and x and y velocity, we created a struct called `lbm_node` which stored an array of the vectors and variables which we referenced throughout the document. To store the constants associated with this simulation, we created a `parameter_set` set which stored ν, ω, the dimensions of the screen, the flow speed into the screen, the color scheme, and the render mode.

4.3 Adapting the iterative LBM to asynchronous threading

Since threads are asynchronous, we are forced to make each of the threads independent of each other. This turns out to be an issue when converting from an iterative approach to a parallel one with CUDA kernels. For example, in an iterative approach, many of the steps rely on both the entirety of the previous step and specific adjacent steps to be completed in order. To fix this, we divide the algorithm of the graphics card into 3 independent kernels, collide, stream, and bounce. While this

is slower than having one kernel, it is necessary to ensure that previous steps are completed before moving forwards.

4.4 Using Array Swapping to Conserve Memory

An important restriction of parallelism is that threads are executed in a random order. Therefore, all data must be read from one array into a separate output array to ensure that the read array is not modified until all threads have executed. To save memory, we only initialize two arrays, `array1_gpu` and `array2_gpu`. Inside the `kernelLauncher` function, we construct two `lbm_node` pointers, `before` and `after`. These pointers swap between pointing to `array1` and `array2` between each kernel launch, so only the results from the previous step are saved and the input data is thrown away.

We use flat indexing for each array to represent a 2d matrix (index $[x][y]$ would be index $[x * width + y]$), which allows the simulation to directly access the value of the array rather than force the computer to access the pointer to another array. This speeds up all memory accesses throughout the program.

4.5 Collide Kernel

Since the implementation of the Collide step inside the CUDA framework is not dependent on where in the lattice the point is, transferring the code to a CUDA framework is straightforward and similar to an iterative framework. In an iterative framework, the code simply loops over the entire lattice and performs identical collisions, as outlined in §2.4, on each point. For a CUDA version, the kernel only describes one node and is called over an array of nodes. The collide kernel first calculates macroscopic variables such as ρ, u_x, and u_y before calculating the resultant microscopic variables, including the number density of the flow in each direction.

4.6 Streaming Kernel

The streaming step was significantly less trivial to implement than the collide step, because a multitude of tests needed to be completed to determine whether the node is a middle case or an edge case, and if it is an edge case then determine which type of edge case it is. In an iterative approach, many of the steps rely on both the entirety of the previous step and specific adjacent steps to be completed in a specific order. Therefore to make the kernel independent of execution order, we stream from one array into another array without ever referencing resultant data.

4.7 Bounce Kernel

Inside the bounce step, we both simulate the bounce step of the algorithm and render the data. Inside the algorithm portion of this device, we first verify that the position is a wall and complete the bounce step by reversing all fluid vectors received in the stream step. We then stream these

reflected vectors out to prepare for the subsequent collision step. Finally, the kernel runs the rendering algorithms.

4.8 Rendering in CUDA

The most computationally expensive operation when using GPU acceleration is to transmit data across the PCIe bus. To avoid this, we chose to use OpenGL for rendering. The actual process that OpenGL uses to display a 2d PBO is detailed in §3.3. The PBO is allocated on GPU memory, so writing to it from the GPU is allowed without computationally expensive calls to `cudaMemcpy`. Writing to the PBO is done at the end of the bounce step to avoid having to launch an additional kernel.

The PBO is an array of `Uchar4`'s, with the 4 `uchar`'s representing the r, g, b, and a channels. There are multiple render modes which are defined in an `enum`, including `mRho`, `mCurl`, `mSpeed`, `mUx`, and `mUy`. In the first two modes, the colors blue and green are used to differentiate between positive and negative values respectively. The mode is stored with the other parameters in the `parameter_set` struct, and it can be set from hotkeys on the keyboard.

Render Modes	
`mRho`	Pressure gradient mode
`mCurl`	Plots Curl (infinitesimal rotation of fluid)
`mSpeed`	Plots the square of the speed of the fluid
`mUx`	Plots horizontal velocity component
`mUy`	Plots vertical velocity component

An interesting phenomenon to note is that it takes slightly longer for Curl to render than the other modes. This is due to the curl algorithm relying on the velocities of neighboring nodes in addition to the current node. The difference is not noticeable, but it is visible inside the benchmarks.

5 Benchmarking and Results

5.1 Methodology

We compared our algorithm to a CPU based Java implementation of LBM written by Daniel V. Schroeder at Weber State University. The program uses separate threads for rendering and simulating, and uses Java Swing for the GUI.

For each test, we rendered densities and ran an average framerate benchmark using FRAPS for 20 seconds. The CPU version refreshes once per frame, so the refresh rate is equivalent to the frame rate. For our GPU version, however, we have the capability of having a variable number of refreshes-per-frame. We added this because OpenGL will not render at a framerate faster than the monitor's refresh rate in order to prevent screen tearing. In our case, it was capping the framerate at 60Hz. Therefore, in order to calculate refreshes-per-second we chose to measure with 10 refreshes per render to prevent the speed from being capped by OpenGL.

5.2 Benchmark Results and Analysis

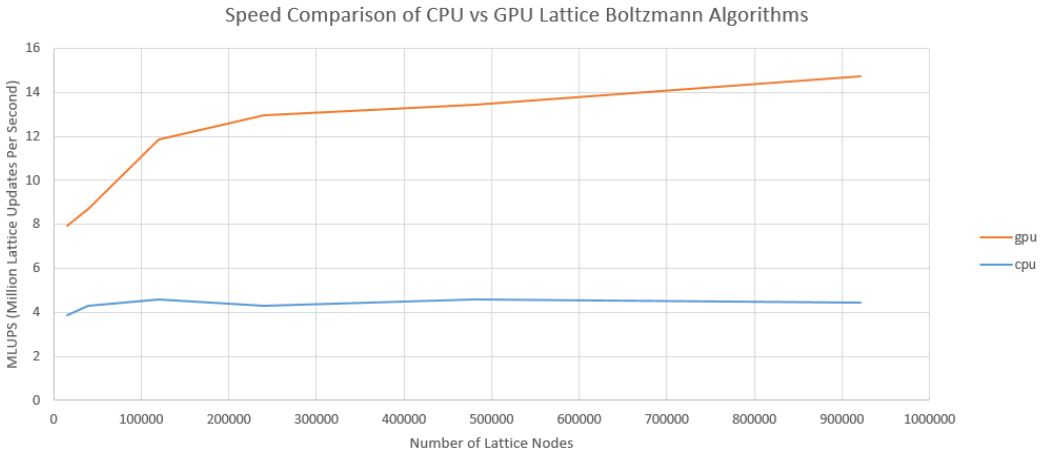


Figure 5.1: Comparison of refresh rates for CPU and GPU LBM

Fig. 5.1 shows the speed differences between the CPU and GPU implementations of the Lattice-Boltzmann Method. The speed is measured in million lattice updates per second, or MLUPS. This gives an approximation of how long each lattice update takes including all of the overhead of rendering.

Note that for the CPU implementation the MLUPS benchmark is constant relative to the number of lattice nodes. This is because increasing the number of nodes simply increases the number of iterations required for the algorithm linearly.

Alternatively, in the GPU implementation there is a slight increase in MLUPS as the number of nodes increases. This is because there is considerably more overhead in calling GPU kernels than there is with initializing a loop. The time per frame spent initializing the kernels is constant relative to the number of nodes, and that overhead takes a significant amount of time. So when the number of nodes increases, the refresh rate decreases so it spends more time running threads compared to the overhead of initializing kernels. This results in a slight MLUPS increase as the number of nodes increases.

Ultimately, our algorithm has an approximately 3.5-4x speed increase compared to the CPU method, and that difference increases as the size of the field increases.

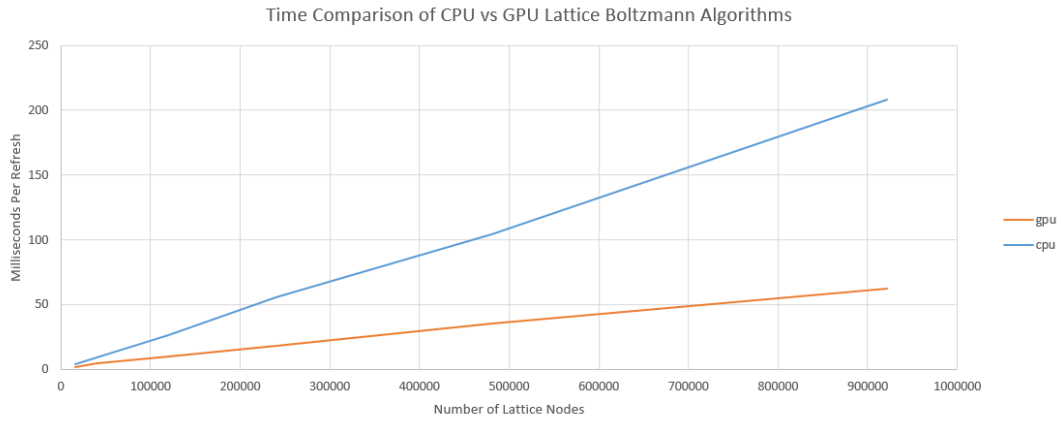
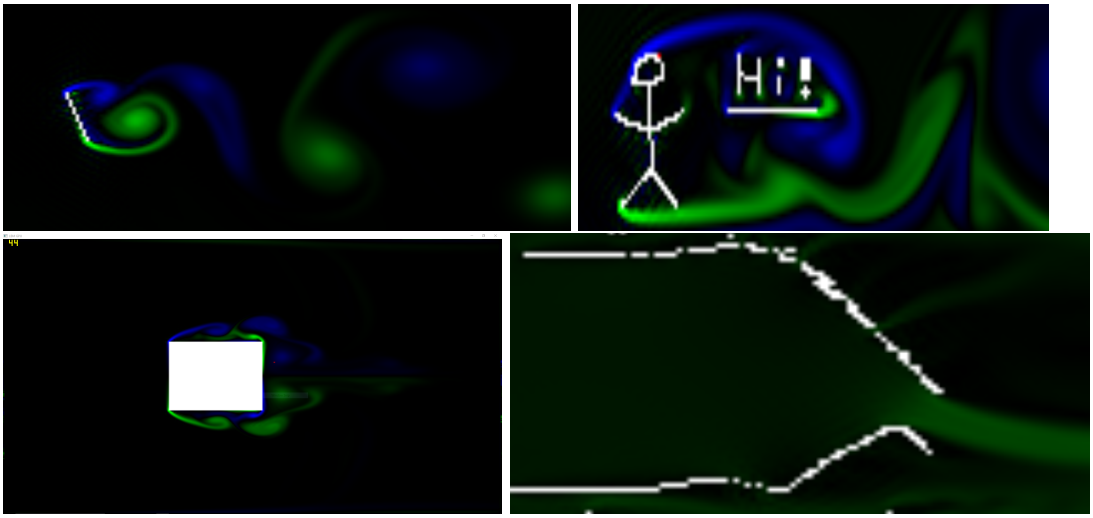


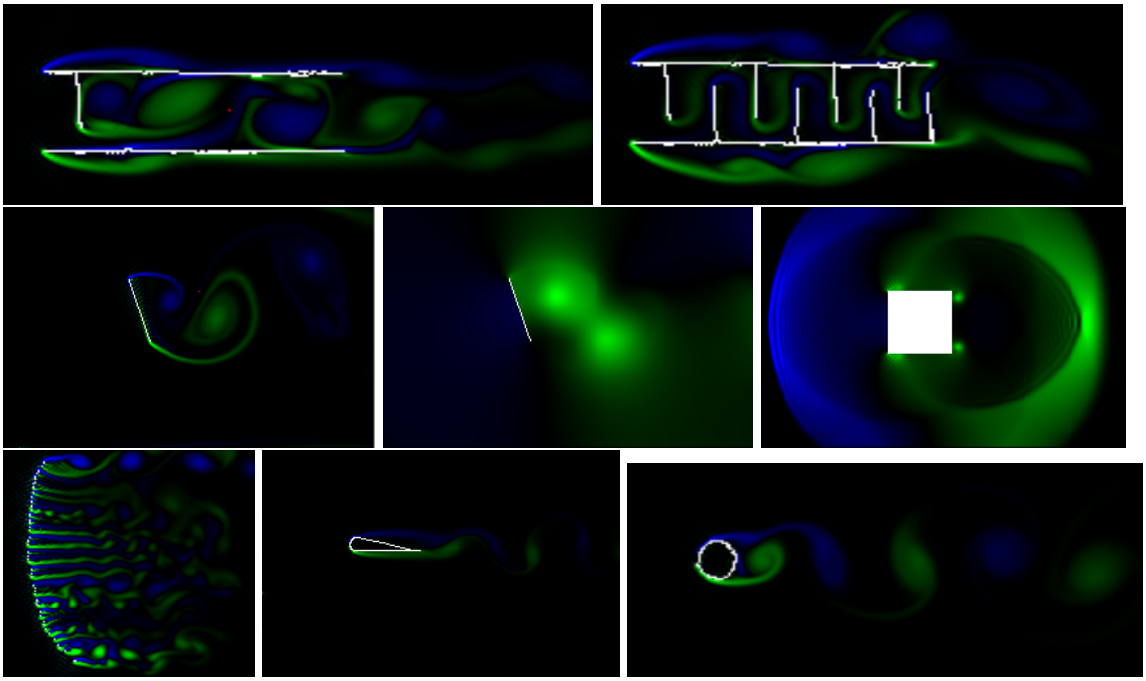
Figure 5.2: Comparison of refresh times for CPU and GPU LBM

Fig. 5.2 shows the amount of time it takes per lattice update for the CPU and GPU implementations. Both implementations grow in linear time relative to the number of nodes. The CPU update time grows much faster than the GPU update time due to the higher MLUPS rate of the GPU implementation.

One might expect the GPU implementation to increase with constant time because the nodes are calculated in parallel, however that is quite untrue. While the implementation is designed to run with massive parallelism, there is a limited number of cores on the GPU (1280 in the case of our GTX 1060). Because it cannot run more than 1280 nodes at a time, it still has to iterate a considerable number of times.

5.3 Screenshots





Demo: `https://www.youtube.com/watch?v=MSMGGoP24Hw`

6 User Interface

6.1 Running the Simulator

Running the CUDA-accelerated LBM simulator requires CUDA-capable hardware, such as a recent Nvidia graphics card. OpenGL must also be installed, but it is usually included with most graphics driver packages anyways. We used dynamic linking to enable OpenGL support, so the OpenGL.dll (dynamic linked library) file must be present in the root directory of the compiled LBM simulator.

Currently, the simulation resolution is set at compile-time, so unless you want to stick with one resolution, it is useful to build the program manually rather than downloading a windows binary. To build the program, simply load the Visual Studio project file in Visual Studio. Nvidia Nsight must be installed to build it. All dependencies besides the CUDA-specific ones are included in a portable dependencies folder inside of the project folder, so there is no need to reconfigure the project settings or install any other libraries.

6.2 Interfacing with the simulator

Interacting with the simulator is done with the mouse and keyboard. In addition, the window can be freely resized using OpenGL's built-in anti-aliasing algorithms.

LBM Controls	
Left Mouse	Draw boundaries with the mouse. (A red cursor is used to show the current mouse position on the lattice)
Right Mouse	Erase boundaries with the mouse
1-5	Set render mode(ρ, curl, speed^2, u_x, u_y)
q	Clear boundaries
w	Reset simulation
a-f	Load preset boundaries (short line, long line, diagonal line, square)
z	Set flow speed using 1-0 (requires a simulation reset afterwards)
x	Set viscosity using 1-0
c	Set number of updates per render using 1-0

7 Challenges

7.1 Edge Cases

The first difficulty when implementing LBM is the edge cases. Edge cases are defined as the nodes immediately contacting the borders of the virtual wind tunnel, including the left and right where fluid enters and exits, as well as the flanks (the top and bottom). Specific cases must be declared to handle each edge case, and errors here can ripple throughout the simulation to cause major issues elsewhere in the simulation.

7.2 Boundary Conditions

Another difficulty is the boundary conditions (i.e., what to do when the fluid is flowing into a solid wall). There are many ways to do this, but we used the bounceback algorithm. The bounceback algorithm redirects all flow towards the walls back out of them, and it enforces the no-slip conditions along each wall. One of our biggest troubles with the bounceback step was that the immediately following collide step would be using the results of bounceback that had not been streamed out of each node. That meant the pressure inside of a wall would steadily increase until data overflow occurs. We fixed this by combining streaming within the bounceback step.

7.3 Debugging

Debugging GPU kernels is a struggle. There is no step-through, and if an error flag is raised there must be dedicated error handling code to notice it. CUDA alsodoes not support breakpoints. To isolate crash points, we deleted whole blocks of code and slowly reinstated it until the offending

line is found. We set up a system of listener nodes that dump all of their data in every step into a log file in order to see exactly what is happening at a node.

8 Concluding Thoughts

In this project, we succeeded in implementing a 2 dimensional Lattice-Boltzmann method using CUDA graphics acceleration and OpenGL. We are able to display curl, u_x, u_y, and speed^2, which model constant 2 dimensional fluid flow through a medium. We learned how to use CUDA and OpenGL, and how to derive and implement the LBM algorithm. We also learned some code optimizing skills, and how to use trace-points for debugging without breakpoints (CUDA does not support breakpoints). By using computation fluid dynamics (CFD), we learned algorithm design in a highly threaded framework where you cannot rely on execution order. This was a very educational experience, and while a considerable portion of the project was taken up by the monotony of debugging, it was very rewarding to finally see our system outpace many of the other LBM implementations.

A Appendix A: Resources

First and foremost, we would like to dearly thank Mark Antosz. And then Dr. P.

Sources

`http://physics.weber.edu/schroeder/javacourse/LatticeBoltzmann.pdf`

Tutorials

`http://www.informit.com/articles/article.aspx?p=2455391`

`https://developer.nvidia.com/gpu-computing-webinars`

Pictures

`http://www.frontiersin.org/files/Articles/70265/fgene-04-00266-HTML/image_m/fgene-04-00266-g001.jpg`

`https://developer.nvidia.com/sites/default/files/akamai/cuda/images/product_logos/NV_CUDA_wider.jpg`

`https://www.opengl.org/img/opengl_logo.png`

`https://commons.wikimedia.org/wiki/File:CUDA_processing_flow_(En).PNG`

`https://devblogs.nvidia.com/parallelforall/nvlink-pascal-stacked-memory-feeding-appetite-big-data/`

`http://geco.mines.edu/tesla/cuda_tutorial_mio/`

`https://www.researchgate.net/profile/Marwan_Abdellah/publication/274900913/figure/fig2/AS:304502231060533@1449610397787/Communication-between-CUDA-and-OpenGL-on-screen-rendering-con big.png`