Parallel GPU Implementation of the 2D Lattice-Boltzmann Method in Julia

Ryan French

Montana State University
ryanfrenchphysics@gmail.com

April 21, 2020

Abstract

An efficient approach for modeling two-dimensional, nonturbulent fluid dynamics is proposed that combines the speed of GPU-parallelized code and the power of Julia's JIT compiler. The model is derived from the Lattice-Boltzmann Method and is tightly-coupled to Julia's compact user-defined types and multiple dispatch paradigm. The Julia library executing this model, while limited, is easily extensible to models that are more demanding.

I. Introduction

Lattice-Boltzmann Method

Macroscopic computational fluid dynamics (CFD) is dominated by iterative forms of the Navier-Stokes equations, which is derived from applying classical conservation laws to fluids, along with terms describing viscosity and pressure. Methods derived from Navier-Stokes are very accurate and provide the best macroscopic solutions to a variety of problems in physics and engineering.

Bridging the gap between macroscopic and microscopic CFD methods is the Lattice-Boltzmann method (LBM), a relatively recent development forged from the method of Lattice gas automata (LGA). LGA, which is based on a d-dimensional lattice, suffered from non-Galilean invariance and unphysical pressure and viscosity relationships. In 1988, McNamara and Zanetti[1] modified LGA by neglecting particle correlations and introducing averaged distribution functions. Later, Higuera and Jimenez[2] presented a Lattice Boltzmann Equation, which featured a linearized collision operator that assumes a local near-equilibrium state[3].

Because LBM is a mesoscopic theory based around distribution functions within a small volume element, the results from this theory are easily transformed into macroscopic results (in fact, it is possible to derive the Navier-Stokes equations from Lattice-Boltzmann equations). See[4] for the full derivation. This fact presents LBM as a viable method of CFD, as opposed to current finite-difference and finite-volume methods.

Arguably one of the greatest strengths of LBM is that one can easily integrate it into distributed programming methods, because it is a theory based on nearest-neighbors instead of fields. It has historically been implemented on parallelized CPUs, using methods such as method-passing interfaces (MPIs). More recently, however, distributing the code on graphical processing units (GPUs) has become popular, due to their modern affordability. While it has commonly been claimed that GPUs achieve 2 to 3 orders of magnitude better performance over CPUs in distributed programming, these claims may fall quite short of reality, as demonstrated in[5]. However, it should be noted that GPUs are easily optimized and can indeed achieve speeds of 2.5x to 10x the speed of similar CPU setups.

Julia

Julia is a high-level, dynamically-typed, high-performance language[6][7]. Originally designed for technical and scientific applications, it is now gaining attraction in economics and statistics. The reason it is loved by its users is simple, and can be summed up by the popular phrase: "[Julia] runs like C, reads like Python."[8]

What makes Julia special is its just-in-time (JIT) compiler, which compiles code into efficient LLVM on the fly. This compiler, combined with the option for static typing, often creates programs that perform just as quickly as statically-compiled languages like C, Go, and Fortran.[9]. Julia is designed for parallel computing, and the efficienty of threading translates well into the well-supported GPU modules.

The most popular GPU programming language is CUDA (Compute Unified Device Architecture), a version of C designed for parallel computing on Nvidia GPUs[10]. Because Julia can natively use C (in fact, writing C in-line with Julia code is supported by the standard library), there exist several high-level CUDA modules which simply exist as wrappers around CUDA's base code. The relevant parallel programming modules used in the development of this project are CUDA.jl[11], CUDAnative[12], CuArrays[13], GPUArrays[14], GPUBenchmarks[15], and CUDAapi[16].

GPU Programming

A convenient way of implementing this CUDA code is to use a development kit containing an Nvidia GPU. Luckily, the Nvidia Jetson Nano[17] is a cheap board on which to run parallelized GPU programs. The Nano contains 128 Nvidia GPU cores, which can perform up to 0.5 TFLOPs. A custom Linux distribution, derived from Ubuntu, must be downloaded from Nvidia's website and flashed to a Micro-SD card, which also serves as the kit's hard drive. Included with the distribution is a broad set of libraries and APIs for optimizing code on the GPUs.

TODO: Add more information specific to the GPU and the operating system that this codebase is being developed on.

II. Theory

Equation Statements

The lattice in LBM follows a D_xQ_y convention, where x is the number of dimensions and y represents the number of particle velocities. The intended dimensionality of this paper is 2D, and the D_2Q_9 model will be used because of its popularity.

The LBE is stated as follows[18]:

$$f_i(\mathbf{r}+\mathbf{c_1}\Delta t, t+\Delta t) = f_i(\mathbf{r};t) + \Omega_{ij}\left(f_j^{(0)}(\mathbf{r};t) - f_j(\mathbf{r};t)\right) \tag{1}$$

where f_i are the distribution function and $f_i^{(0)}$ are equilibrium distribution functions along the ith direction, respectively, and Ω_{ij} is a tensor based on collision terms in the classic continuum Boltzmann equation.

Commonly implemented is the Lattice-Boltzmann Single-Relation-Time model[19]:

$$f_i(\mathbf{r}+\mathbf{c_1}\Delta t, t+\Delta t) = -\frac{1}{\tau}\left(f_i(\mathbf{r};t) - f_i^{(0)}(\mathbf{r};t)\right) \tag{2}$$

where τ is a characteristic relaxation time (time for volumetric element to reach equilibrium). Equation 2 is updated in 2 steps:

<u>Collision step</u>:

$$\tilde{f}_i(\mathbf{x},t) = f_i(\mathbf{x},t) - \frac{1}{\tau}\left(f_i(\mathbf{x},t) - f_i^{(0)}(\mathbf{x},t)\right) \tag{3}$$

<u>Streaming step</u>:

$$\tilde{f}_i(\mathbf{x}+\mathbf{c}_i\Delta t, t+\Delta t) = \tilde{f}_i(\mathbf{x},t) \tag{4}$$

where f_i and $\tilde{f}_i$ represent the pre- and post-collision states, respectively.

Macroscopic quantities such as momentum density can be obtained in the expected way, given a distribution function:

$$\rho\mathbf{u} = \sum_{i=0}^{N} f_i \mathbf{c}_i$$

where $N = 8$ for D_2Q_9 (i.e., 9 discrete velocities exist).

Discretization

The steps in equations 3 and 4 are approached in the following manner:

1. Construct a discretized lattice
2. Choose $f_i(x_0, t_0)$ for each point on the lattice
3. Stream each individual lattice point (move each point in the direction its velocity's favors)
4. Collide on each lattice point.
   (a) If two discrete fluid elements exist on the same point, use their velocities to decide subsequent velocities.
   (b) If a fluid element exists at or within a boundary, apply necessary boundary condition to change the element's velocity.
5. Repeat streaming and collision steps

An algorithm for these steps can be found below:

Algorithm 1 Lattice Boltzmann Method

```
IN: $(LAT, \nu, u_0) \leftarrow$ Lattice Pts, Viscosity, Initial Velocities

# Inverse of time constant, $\tau$:
$\Omega = \Omega(\nu)$
# Weights are known for D2Q9:
const w $\leftarrow$ weights::Array
$f_i(x_0, t_0) = f_i$(LAT, weights, $\nu, u_0$)
$\rho \leftarrow$ calc_densities()
$u_x, u_y \leftarrow$ calc_velocities()
object $\leftarrow$ gen_object()
initialize_plots()

StartLoop:
# Do one step for each point and handle object collisions
for ($pts \in$ LAT) do
    $f_i(x_i) \leftarrow f_i(x_i + e_i \cdot \delta t)$
    if ($x_i \in$ object) then
        $f_i(x_i) \leftarrow$ boundary_condition($f_i$)
    end if
end for

# Do collisions between fluid elements
# then check inlet/outlet BCs:
$\rho \leftarrow$ calc_densities()
$u_x, u_y \leftarrow$ calc_velocities()
for $pts \in$ LAT do
    $f_i(x_i) \leftarrow$ update($f_i, \Omega, u_x, u_y$)
    if ($pt \in$ border) then
        # Force flow @ initial velocity
        $f_i(x_i) \leftarrow$ force_flow($f_i$, $u_0$)
    end if
end for

# Get curl of macro velocity fields to
# calculate vorticity
curl_vals $\leftarrow$ curl($u_x, u_y$)
update_plot(curl_vals, object)
goto: StartLoop
```

III. Computational Implementations

Discretized Boltzmann Distribution

TODO: explain expansion of Boltzmann Distribution up to 4th order in thermal velocities, v. Explain discretized velocity, e = u + v (u « 1, u is flow velocity). This leads to the weights along out lattice axes.

Barriers

TODO: Custom type in Julia holds all data about each barrier and its boundary conditions in very compact, dense memory section.
Barrier generating functions, both specific (i.e., functions for circles, rectangles, NASA airfoils, etc) and generic (user-supplied data, hopefully both by function and input data file; these are being worked on right now).

Boundary Conditions

TODO: Another custom type holds information about all boundary conditions in the problem

In-Place Advection and Collision

TODO: Stream and collision steps have methods that are not in-place at the moment, but should be easily transformed into in-place (i.e., no copies of arrays are created, all data is transferred in the same contiguous block of memory). This will make it much easier to switch back and forth between CUDA arrays and Julia arrays. CUDA arrays have to be dense (continugously mapped), and Julia's instantiated arrays are also dense, but copying arrays in Julia can lead to memory fragmentation.

IV. Results

The first test of this software was to construct a two-dimensional fluid tunnel. Boundary conditions for a tunnel are simple: the inlet must maintain a constant velocity and the outlet must prevent against any bounce-back conditions. The outlet condition is maintained by applying a constant-speed condition (**TODO:** expand) in the directions that have components perpendicular to the outlet.

In i, there are images of steady-state solutions for several elementary shapes in a two-dimensional fluid tunnel. While streamlines are a decent way to visualize the flow of a fluid around an object, vorticity plots were chosen for their higher contrast in identifying how the fluid curls around a surface. For example, viewing the contrast in the right-angle plot (image 5) allows one to easily see how the fluid flowing across the top curls much more sharply than the fluid flowing across the bottom. In watching the animated plot for the right-angle, these two vortices interact quite heavily on the tail end, which causes the tail to "wiggle" back and forth. The other shapes do not exhibit such behavior because of their symmetry.

TODO: Include details on airfoils, and more specifically how they are used to determine critical Reynold's/Mach numbers (i.e., what are the upper limits on this model)? Identifying these critical values can lead to an improvement of the model. Also, FIX THE AIRFOIL PLOTS! Figure out a better way of laying them out for comparison.

V. Discussion

TODO: Discussion will mainly highlight the above critical points and how one might go about fixing them and/or improving the model. Specific shortcomings in this codebase will also be highlighted.

VI. Conclusions

TODO: Conclusion will focus on the limitations of the actual LBM model instead of the limitations made evident in this codebase. This will include a brief discussion on limits, such as:

- the expansion of the Boltzmann Distribution and the noticable effect, if any, that considering higher-order terms would have
- the effect of ignorng temperature-dependence and how this could be addressed in the actual model.

Appendix

A. Images

i. Elementary Objects in Fluid Stream

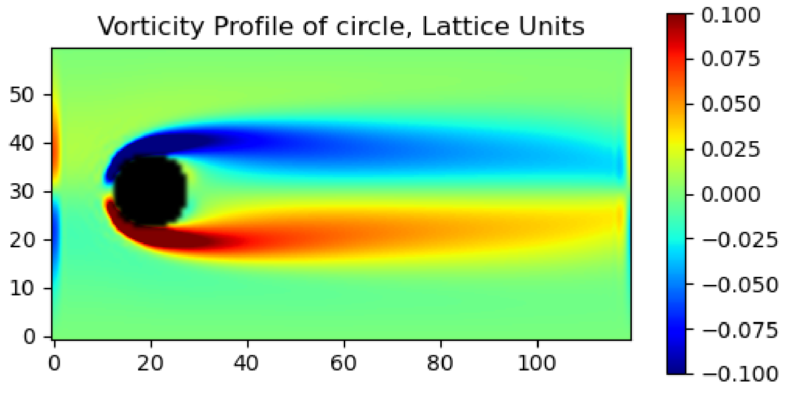


Figure 1: *Steady-state vorticity of a circle in a 2-D wind tunnel.*

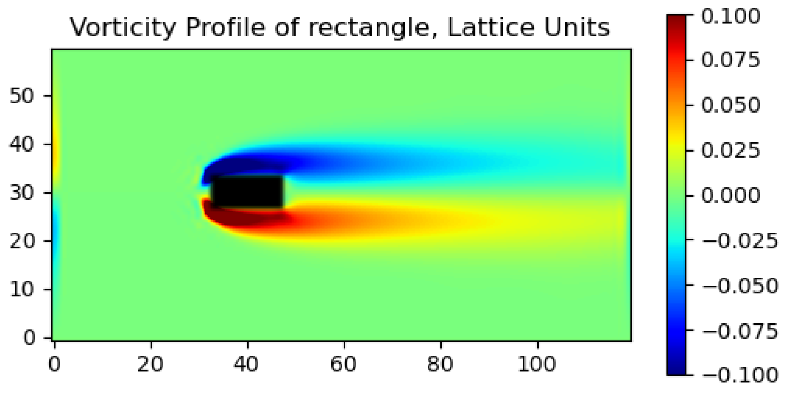


Figure 2: *Steady-state vorticity of a rectangle in a 2-D wind tunnel.*

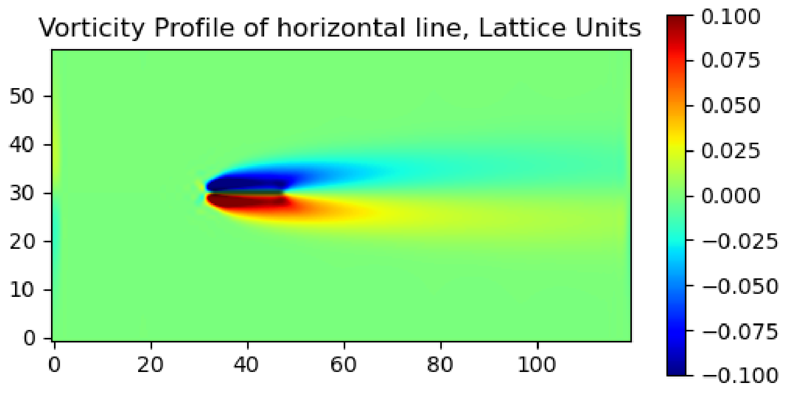


Figure 3: *Steady-state vorticity of a horizontal line in a 2-D wind tunnel.*

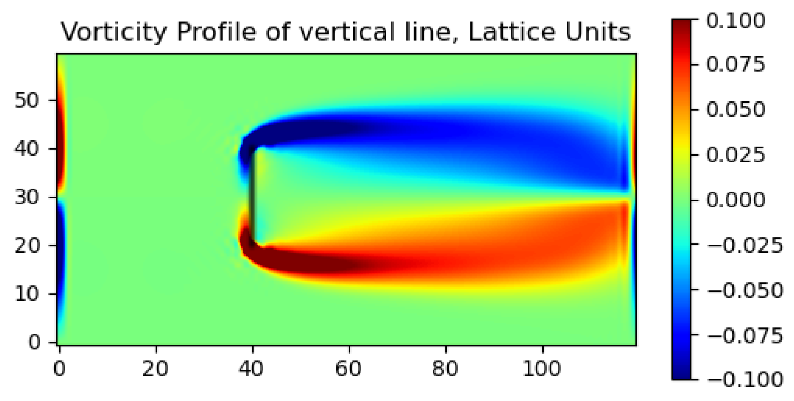


Figure 4: *Steady-state vorticity of a vertical line in a 2-D wind tunnel.*

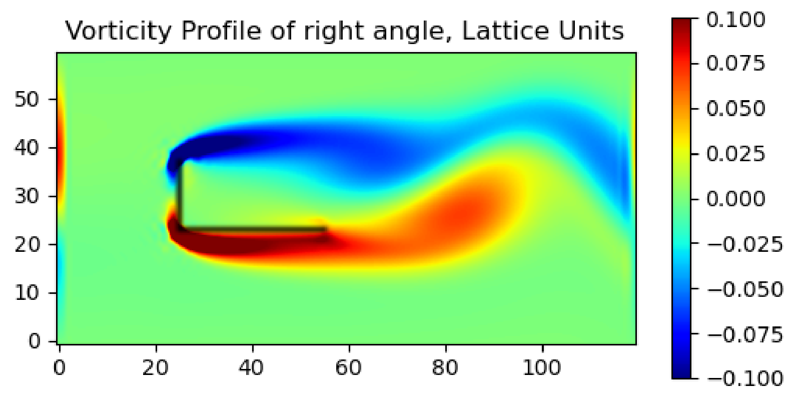
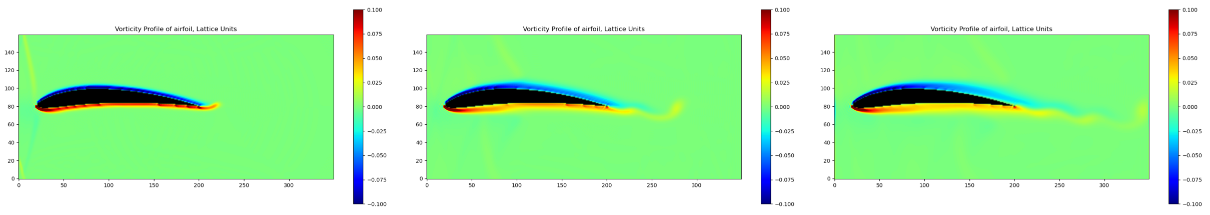


Figure 5: *Steady-state vorticity of a right-angle line in a 2-D wind tunnel.*

ii. Airfoil Wings and Stability

ii.1 Stable Flow



(a) $t \approx 0$ **(b)** $t > 0$ **(c)** $t >> 0$

Figure 6: *Time evolved vorticies of a NASA 4812 airfoil in a stable fluid flow (low velocity, high viscosity).*

ii.2 Unstable Flow

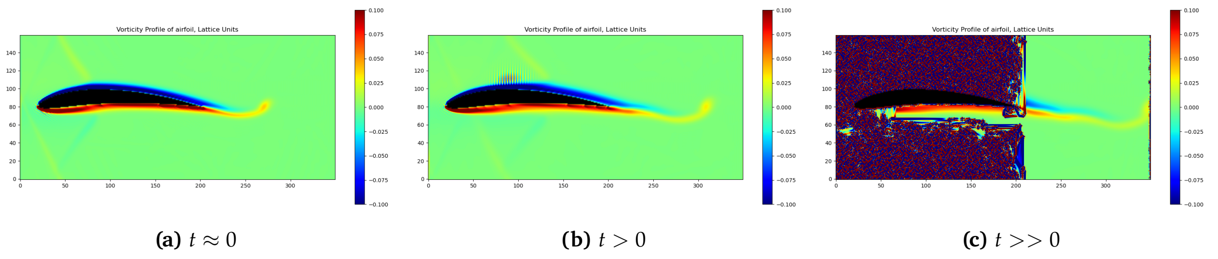


Figure 7: *Time evolved vortices of a NASA 4812 airfoil in an unstable fluid flow (high velocity, low viscosity). Notice the significant diversion beginning at the front of the wing.*

References

[1] G. McNamara, G. & Zanetti, “Use of a boltzmann equation to simulate lattice-gas automata,” *Phys. Rev. Lett.*, 1988.

[2] J. Higuera, F. & Jimenez, “Boltzmann approach to lattice gas simulations,” *Europhys. Lett.*, 1989.

[3] A. K. Permaul, D. A. & Dass, “A review on the development of lattice boltzmann computation of macro fluid flows and heat transfer,” *Alexandria Engineering Journal*, 2015.

[4] G. D. Chen, S. & Doolen, “Lattice boltzmann method for fluid flows,” *Annual Review of Fluid Mechanics*, 1998.

[5] C. K. e. a. Lee, V.W., “Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu,” *ACM SIGARCH Computer Architecture News*, 2010.

[6] “The Julia Programming Languages.”

[7] “Julia Github.”

[8] J. M. Perkel, “Julia: come for the syntax, stay for the speed,” *Nature*, 2019.

[9] “Julia Micro-Benchmarks.”

[10] Nvidia, “CUDA Zone.”

[11] “CUDA.jl Github.”

[12] “CUDAnative.jl Github.”

[13] “CuArrays.jl Github.”

[14] “GPUArrays.jl Github.”

[15] “GPUBenchmarks.jl Github.”

[16] “CUDAapi.jl Github.”

[17] Nvidia, “Jetson Nano Developer Kit.”

[18] *Lattice-Gas Cellular Automata and Lattice Boltzmann Models - An Introduction*. Spring, 2005.

[19] A. K. Perumal, D. A. & Dass, “Simulation of flow in two-sided lid-driven square cavities by the lattice boltzmann method,” *Advances in Fluid Mechanics*, 2008.