Fluid Dynamics with Emphasis on Visualization
By: Gilberto Júnior de Sousa Silva
Project Unit: PJE40
Supervisor: Bryan Carpenter
May 2020

Abstract

This project is a visual simulation of the Lattice Boltzmann Method (LBM) of Fluid Dynamics on the GPU using Java. Lattice Boltzmann is a method of simulating highly-complex fluid systems by solving equations of macroscopic properties. LBM allows for parallel processing, which takes advantage of devices which have multiple processing cores such as a GPU. The simulation will calculate and process 32-bit floats, and how the GPU handles these tasks. Existing code from a Python script will be converted to Java for the purpose of parallel programming on the GPU. This is done by splitting the processes into three: Macroscopic, Collision, and Streaming. The Java program will use Aparapi to execute kernels on processors with multiple threads, such as a GPU. An airfoil is used alongside with the cylinder as an obstacle and the airfoil shape changes depending on the four-digit number used. The four digit number determines the maximum camber, the distance of maximum camber from the airfoil leading edge, and the maximum thickness of airfoil.

Word Count is 9209.

Acknowledgements

Contents

List of Figures

Chapter 1

Introduction

1.1 Project Aims and Goals

This project will be about implementing Fluid Dynamics on the GPU using Java. This project's goals are to firstly convert all the code from a Python script to Java and be able to execute the code error free. The program must be able to show changes in frames in a window rather than images in a file. Secondly, the program must be able to use different objects besides a cylinder for an obstacle. The object in question is the airfoil, and the user can select between different objects to use as an obstacle. Also, the Java code must be adapted to be usable by Aparapi. This means that multi-dimensional arrays must be flattened down to one-dimensional arrays, and all double-precision floats (called "double" in Java) that would be used in a kernel must be converted to single-precision floats (called "float" in Java) as certain GPUs may not be compatible with the program if doubles are used.

1.2 Project Plan and Milestones

This project started in October 2019. At the beginning, research took place to better understand how the Lattice Boltzmann Method works, and how it compares to Lattice Gases and Navier-Stokes equations. In addition, the formulas for the NACA airfoils are also researched. The research on LBM takes place in November 2019.

The basic goal is to transform a Python script into a Java program capable of executing code on the GPU. The first deadline is converting all Python code to Java code by December 2019. The code is broken down to multiple sections, and is converted and implemented one at a time. Some lines of code may end up in different sections.

The second deadline is to create an airfoil program and the calculations to accurately draw an airfoil depending on the four-digit number used in the program. The airfoil object is then merged to the main program as an obstacle, alongside the cylinder, by January 2020.

Then, between January and March 2020, all code is adjusted to be compliant to Aparapi. To begin, all multi-dimensional arrays are converted to single-dimensional arrays. Then, double-precision floats that may be used by Aparapi are changed to single-precision floats. And finally, the kernels can be applied to the code. The macroscopic, collision, and streaming codes are run in kernels.

1.3 Project Motivations

There have been previous attempts at simulating fluid dynamics on the GPU in the past. However, the majority of fluid dynamics simulations are based on the Navier-Stokes equations, whereas with LBM there were a few attempts in the past 10 years on the GPU. There are not many LBM programs run on Java that use the GPU. Java is a programming language that is easy to use, and can run on any system without compiling it, making it flexible and multi-platform. While not as fast as C++, Java is capable of using GPU programming with the help of Aparapi.

Chapter 2

Literature Review

This literature review will cover three topics relevant to the Fluid Dynamics project, and these are: (1) LBM and NSE, (2) Previous attempts to implementing Fluid Dynamics Simulation on a Graphics Processing Unit (GPU), (3) Aparapi programming in Java. LBM and NSE will be compared with each other and will go into detail how they work. Fluid Dynamics simulations will look at GPU programming and how parallel programming is useful to GPU processing along with looking at 32-bit single floats compared to 64-bit double float precision points. Finally, how Java is used on GPUs covers OpenCL, how an API like Aparapi uses parallel programming, and how parallel programming works.

2.1 Lattice Boltzmann Method and Lattice Gases

The Lattice Boltzmann Method is a method of simulating Fluid Dynamics that is based on "microscopic models and mesoscopic kinetic equations" (Chen and Doolen 1998, p. 329-364). Cellular Automata is a grid of cells with a finite number of states with any number of dimensions. Each site updates to a new generation depending on surrounding neighborhoods' values. Game of Life is an example of Cellular Automata where the grid is in 2D and each cell has a boolean value, alive or dead, decided by interacting with the surrounding eight cells. Live cells with two or three live neighbors live, otherwise they die; and dead cells with exactly three live neighbors become live (Rendell 2002, p. 513-539).

LBM was based on the Lattice Gas Model, where it is a Cellular Automata with states at each site. The simplest model is the Hardy, de Pazzis, and Pomeaumodels (HPP) model, where it is based on a 2D grid, populated by a set of particles with varying velocities, in which they can interact with each other to four velocity states. These four velocity states are state 0, 1, 2, and 3. State 0 and 1 are the x axis, in which state 0 is negative and state 1 is positive. State 2 and 3 are the y axis, in which state 2 is negative, and state 3 is positive. Updating can be broken down to two steps: Collision, and Streaming (Frisch et al. 1986, p. 1505-1508).

2.2 Navier Stokes Equations and comparisons to LBM

Another method of fluid dynamics is using the Navier-Stokes Equations (NSE), which is about the motion of fluid substances. The formula for the compressible NSE momentum convective is:

$$\rho(\frac{\delta u}{\delta t} + u \cdot \Delta u) = -\Delta p + \mu \Delta^2 u + \frac{1}{3}\mu \Delta(\Delta \cdot u) + \rho g$$

Where ρ is fluid density, p is fluid pressure, u is fluid flow velocity, g is gravity, and Δ is fluid divergence. The formula for the incompressible NSE in convective form is:

$$\frac{\delta u}{\delta t} + u \cdot \Delta u - v\Delta^2 u = -\Delta\omega + g$$

Where u is fluid flow velocity, and Δ is fluid divergence (He and Luo 1997, p. 927-944). NSE is used for modeling weather, ocean currents, and air flow of an object. However unlike NSE, LBM allows for parallel programming due to using the two Collision and Streaming processes. Also, the LBM uses a wider range of the variable Kn, whereas NSE is limited to smaller numbers. Kn stands for Knudsen number, which is the "ratio of the molecular mean-free path, λm, to flow length scale, H" (Aidun and Clausen 2010, p. 439-472).

2.3 Fluid Dynamics on the GPU

There were previous attempts to simulate Fluid Dynamics using the GPU. One attempt in 2005 by Harris used the NSEs as opposed to LBM. It was reported that by using the Nvidia GeForce FX, the simulation ran 6x faster on the GPU compared to the CPU (Harris 2005, p. 637-665).

Another attempt was done by Pickering in 2015. This focuses on parallel programming, which allows for a process to use more than one thread for each task, on the GPUs (which has multiple cores) that are designed for streaming. It was found that the NVidia Tesla Kepler GPU (introduced in 2012) performed 20x faster than a single CPU core, and is over twice as fast as a hexadeca-core Xion server.

It was also reported that single precision floats are more efficient than double precision floats on the NVidia GPUs because it only requires half the memory bandwidth, shared space, and register file compared to double floats (Pickering et al. 2015, p. 242-253). The GPU is capable of using parallel processing in ways that surpass the CPU and this can be done by using the semi-Lagrangian method, allowing "adopting large time steps to solve NSEs with solid stability" (Wu et al. 2004, p. 139-146).

2.4 Programming Languages and APIs

Java is a general purpose programming language that only needs to be compiled once and it can be run on any machine. For General Purpose GPUs, C/C++ are the most commonly used as main programming models for developing libraries, but Java is also used for High Performance Programming (Docampo et al. 2013, p. 1398-1404).

One programming model used for General Purpose GPUs is OpenCL, a cross-platform framework for parallel programming. An easy way to make use of GPU code is to use a user-friendly high level API that makes use of OpenCL, and that is Aparapi (named after A PARallel API), allowing Java to use parallel programming on the GPU (Gupta et al. 2013, p. 1-5).

2.5 Aparapi and Kernels

Aparapi, originally developed by AMD, allows for native Java code to run directly on the GPU at runtime by converting byte code to an OpenCL kernel. A kernel is a function called to execute on a CPU or GPU, and runs on multiple threads, allowing for parallel processing. An Aparapi kernel is run as a loop, and the line of code to execute the kernel is where the loop is run. The number of iterations the loop is run is the range, and the kernel can have multiple loops running as a Global ID.

There are other APIs that use parallel programming on the GPU such as CUDA (Compute Unified Device Architecture) for NVidia cards that allows for low level programming; OpenMP for memory shared parallel processing, so that threads can access variables from shared cache or RAM; and MPI (Message Passing Interface) for message passing operations that is standardized, allowing for the code to behave the same way (Yang et al. 2011, p. 266-269).

2.6 Conclusion

In conclusion, LBM in theory allows for more threads to be used easily compared to NSE. Past attempts used NSE for Fluid Dynamics simulations with better performance on the GPU than the CPU, and how parallel programming is executed on the GPU. Aparapi allows for Java code to be executed on the GPU via OpenCL for parallel programming.

Chapter 3

Project Management & Methodologies

The development of the project will follow the Waterfall method, in which the project will start with the Requirements Specification and Analysis, then the Design Specification, Implementation, and finally the Testing and Evaluation stages.

3.1 Requirements Specification & Analysis

The Requirements Specification and Analysis will go over the Python script LBM program that is converted from, and what each section of code is for. Also, the Functional Requirements and Non-Functional Requirements for the program will be evaluated.

3.2 Design Specification

The Design Specification will go over how the Lattice Boltzmann method works by explaining how practical it is to do parallel programming on existing LBM programs; what are Lattice Gases; and explaining each step of LBM, from Macroscopic, Collision, and Streaming steps. The Design Specification will also go over how the Von Kármán Vortex Street works. Also, the way the LBM simulation is coded is also going to be discussed, such as the UI that is going to be used, along with representing velocities, and more importantly Aparapi and Kernels.

3.3 Implementation

The Implementation stage will cover the conversion from Python script to Java code, along with the explanation of relevant sections of code. The Implementation stage will also cover the NACA Airfoil programming, and implementation to LBM, along with a text-based user interface. Also, reworking the Java program to meet Aparapi criteria by flattening down arrays, and converting doubles to single-precision floating points. Finally, Aparapi code is implemented to the Java program by adding Kernels to the three processes of LBM, and reworking the code further to get the program to run correctly while utilizing the GPU.

3.4 Testing & Evaluation

The Testing and Evaluation is about the program's functionality, and what needs to be done in the future. The testing will involve the object selection and the NACA codes for the airfoil, and see if they behave in the way that was intended. Future work may involve optimization to the LBM program involving Aparapi.

Chapter 4

Requirements & Analysis

4.1 LBM Program Background

It is required by the client that the Fluid Dynamics simulation is based on the Python code from the Palabos library in Université De Genève (Palabos 2013). The python program is converted to Java for this implementation of the simulation so that parallel programming is easily achievable. The original python code is separated into five categories: Flow Definition, Lattice Constraints, Function Definition, Obstacle Setup, and Time Loop.

The Flow Definition is the variables for the number of iterations (default is 200,000), Reynold Number (default is 220.0), lattice dimensions and population, and velocity (in lattice units). It also calculates the object coordinates (cylinder in this case) using lattice dimensions and population; and relaxation parameter using velocity, Reynold number, and coordinates of object. The Lattice Constraints handles the lattice velocities as an array, the lattice weights, the unknowns in the left and right walls, and the vertical middle. The Function Definition computes two functions: the helper function for density computation; and the equilibrium distribution function. The Obstacle Setup is the creation of the object, the cylinder in this case; velocity; and frequency. And the Time Loop cycles depending on the number of iterations set by Flow Definition. The Time Loop first does the macro step by calculating the outflow condition of the right wall, the macroscopic density and velocity, the left wall for computing density for known populations and again calculated from equilibrium. The second step is the collision step, and then the streaming step. For every 100 iterations, a full frame is visualized.

A Java implementation of a similar program exists and was programmed by Jean-Luc from the same university. The program however, is more object oriented compared to the Python script. Unlike Python, Java is an object-oriented programming language. The program consists of multiple java classes, with three levels of folders at most. The first level has five java classes and two folders to the second level: tools, and collision. Level 2 has a FileIO java class in the tools folder, and the collision folder has four java classes and a folder to Level 3, called regularized. Level 3 contains 12 java classes. In addition, the data type primarily used in that program for decimal numbers is double, which are 64-bit floats. Depending on the GPU, it may be unable to run the program due to incompatibility. However, in Aparapi it will fall back to Java Thread Pool due to not supporting FP64 (Floating Point 64-bit) if the GPU couldn't handle double floats. In addition, the Java program uses multi-dimensional arrays, which is not supported by Aparapi.

4.2 Functional Requirements

A functional requirement is that the Fluid Dynamics simulation should be programmed to use a GPU. Therefore, any system that has a GPU, either integrated to the CPU, or a dedicated GPU, should be able to run the simulation, though a dedicated GPU is preferred. The GPU should have a Floating Point Unit (FPU), which handles 32-bit float single precision points, allowing decimal numbers to be calculated properly, though any GPU released in the past 25 years should be able to use floating points. 64-bit float double precision points are unnecessary as not all GPUs are capable of using doubles properly and may be slower than 32-bit floats.

Another functional requirement is that the simulation should also allow for parallel programming, meaning that tasks can use multiple threads in a single processor such as a GPU or multi-core CPU. Based on the Python script, the time loop is where the processes can be broken down to three components: Macro time, Collision time, and Stream time. These three processes can be run in parallel to each other.

Having two different objects for the obstacle is another functional requirement. The two objects will be the cylinder and the airfoil. It's important for multiple objects to be used because the LBM flow should react differently depending on the properties of the obstacle. The simulation should be able to use LBM with different obstacle shapes. The LBM simulation should be able to correctly show fluid motion on at least two different objects: a cylinder, and an airfoil. The effects shown in the simulation should render a vortex wake effect known as Kármán Vortex Street.

The final functional requirement is the user can adjust certain variables. One of them is that the user can select an object as an obstacle by typing the number listed. If it doesn't recognize the number of any available object, the cylinder will always be used by default. Another is that if Airfoil is selected, the user can enter the NACA number. It rejects all negative numbers, and 0. It only accepts 4 digit numbers not divisible by 100. 1-3 digit numbers are acceptable, but 5 or more digits are rejected. When counting digits, all 0s in the front digits are ignored, so entering "004015" will be read as "4015", which is allowed. In both cases, if the input isn't a number, the input is rejected and the user has to input a number again. This requirement allows for proper testing of the simulation to compare results of different variables. This requirement also includes validation rules to ensure values outside of range doesn't break the program.

4.3 Non-Functional Requirements

A non-functional requirement is that the simulation should be usable. To achieve this, the text used in the program should have clear instructions so that the user can enter in values in a way that the program expects without the user guessing on what they expect to input. In addition, the error messages should be very clear on what the problem is in case invalid values are entered. For example, entering text instead of a number should tell the user that they weren't using numbers, and allow them to attempt again until they put in a number. Another example is entering a number not assigned to an object, the error message should tell the user that the object entered doesn't exist, and the selected object will default to cylinder.

Another non-functional requirement is performance. The program must be able to update every 100 iterations about 30 times per second on today's hardware. Assuming that the number of iterations is left unchanged, if the simulation produces a higher frame rate, the simulation will run at higher speeds, thus finishing in less time than if the simulation is a lower frame rate.

The final non-functional requirement is reliability. The program must be able to simulate multiple objects in various conditions without the system failing. For example, entering a string on a variable that expects an integer will lead to an error. Normally, this would crash the program, but if the input was detected as non-integer before the variable is assigned, the program won't crash, and the user should be able enter a value again without restarting the program.

Chapter 5

Design

5.1 Lattice Boltzmann Method

5.1.1 Parallel Programming for Other LBM Programs

The design of the simulation is based on LBM, which allows for parallel processing. LBM models particles, in which these particles multiply and collide over a lattice mesh. To do this, the program will be derived from an existing LBM Python script, and convert it to Java. While there was a Java implementation of LBM, the problem with that particular program is that there are a lot of Java classes in three different levels, even collision has multiple classes. This can make parallel programming unnecessarily complicated. As a result, this program will only contain one main class so that parallel programming can be implemented easily.

5.1.2 Lattice Gases

Lattice Gases is a type of Cellular Automata in which each cell communicates with eight other cells around itself and updates its value (generally boolean) in the next generation. Lattice Gases are in a two dimensional grid, and the four velocity states are which axis is affected and whether it's positive or negative.

Lattice Boltzmann is a method of simulating complex fluid systems by solving equations of macroscopic properties such as population, mass, velocity, and so on. It is heavily based on Lattice Gases, except the population for velicity states are generalized in a way that instead of the values being boolean, the values are instead floating points. This is represented as an expected number of molecules in a state. In addition, the collision step is adapted to have local distributions of microscopic velocities to converge towards the equilibrium distribution.

The velocity states are represented as D2Q9, where D2 stands for two-dimensions, and Q9 meaning 9 velocity states. The LBM distribution function in discrete form closely resembles population functions from Lattice Gas Models. Assuming the LBM is 2D, the discrete function will have x and y indexes.

5.1.3 Macroscopic Step

The Macroscopic step is mainly covering Macroscopic quantities, which covers local density and velocities. The local density is the sum of expected number of particles in all

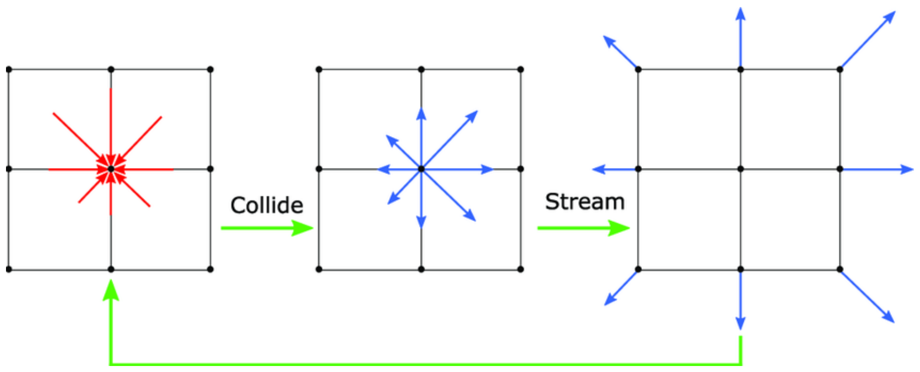


Figure 5.1: This shows the steps of LBM showing: Macroscopic, Collision, and Streaming steps (Harwood and Revell 2017, p. 38-50).

velocity states, using the formula:

$$\rho(x,y) = \sum_{i=0}^{Q-1} f_i(x,y)$$

Where ρ is the local density, Q is velocity states, and f_i is the discrete distribution function. Whereas, the local velocity is the weighted average of discrete microscopic velocities, using the formula:

$$u(x,y) = \frac{1}{\rho(x,y)} \sum_{i=0}^{Q-1} c_i f_i(x,y)$$

Where ρ is the local density, Q is velocity states, u is the local velocity, c_i is the Q vectors, and f_i is the discrete distribution function.

5.1.4 Collision Step

In the collision step, the behavior of the BGK collision operator is approximated. The collision operator uses the formula:

$$\Omega(f) = -\frac{1}{\tau} f - f^{eq}$$

Where $\Omega(f)$ is the collision operator, τ is relaxation time, and f^{eq} is assumed equilibrium distribution. The equilibrium distribution is dependent on macroscopic quantities. This is separated into three steps. The first step is to calculate flow velocity and density at the cell. The second step is to calculate equilibrium distribution. The third step is to use the calculated value to adjust $f_i(x,y)$.

The equilibrium distribution is a polynomial approximation of the “Gaussian”, which is the Maxwell distribution of molecular velocities, using the formula:

$$f^{eq}(v_x, v_y) \propto e^{-\beta}((v_x - u_x)^2 + (v_y - u_y)^2)$$

Where f^{eq} is assumed equilibrium distribution, β is inversely constant proportional to temperature, u is bulk of fluid. The generic form of the equilibrium distribution function is given by the formula:

$$f_i^{eq}(x,y) = w_i \rho(1 + \frac{u.c_i}{c_s^2} + \frac{(u.c_i)^2}{2c_s^4} - \frac{u.u}{2c_s^2})$$

Where f_i^{eq} is assumed equilibrium distribution, ρ is the local density, u is bulk of fluid, w_i and c_i are the constants that depend on choice of velocity states. After the equilibrium distribution, the update is the formula:

$$f_i^{out}(x,y) = f_i^{in}(x,y) - \frac{f_i^{in}(x,y) - f_i^{eq}(z,y)}{\tau}$$

Where $f_i^{in}(x,y)$ is the distribution pre-collision step, $f_i^{out}(x,y)$ is the distribution post-collision step, and τ is relaxation time (Kuznik et al. 2010, p. 2380-2392).

5.1.5 Streaming Step

The Streaming Step is basically updating the cells by each of their velocities for the next generation. The formula for this is:

$$f_i(x + e_i, t + 1) = f_i(x, t)$$

Where $f_i(x,t)$ is fluid density at point x at time t. As this is moving at the velocity of "e_i" for every time step, at the next time step, $t + 1$ would flow to point x+e_i.

5.2 Von Kármán Vortex Street

The simulation will show a Kármán vortex street, in which it repeats a pattern of swirls as a result of vortex shedding. This effect is seen when there is an obstacle in the flow. A cylinder is an object used to demonstrate the wake effects. This is an object commonly used as an obstacle in many other LBM simulations because the shape is simple to code. Another object to use as an obstacle is the airfoil. The airfoil is basically a fixed wing viewed from the wingtip (the side). The Kármán vortex street wake effects should easily show on a fixed wing against airflow.

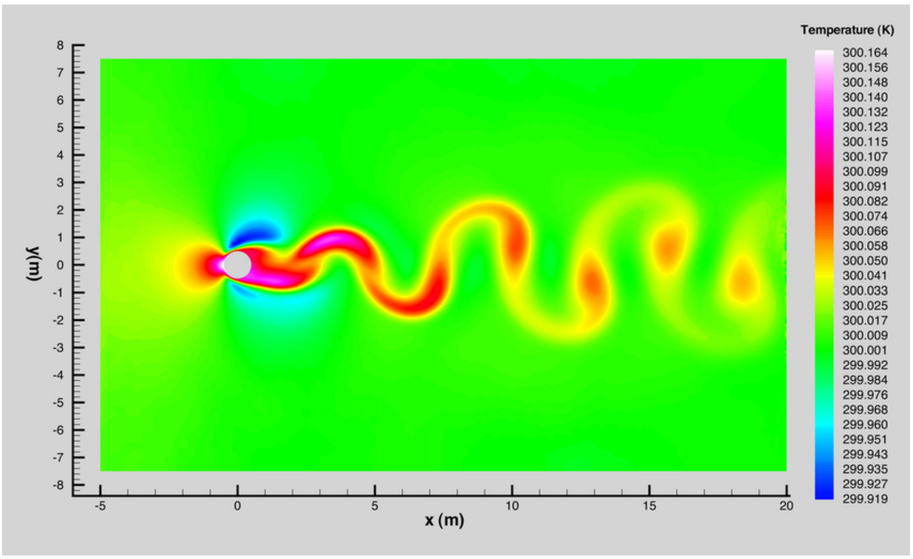


Figure 5.2: Von Kármán Vortex Street Temperature Solution. Time is 4 seconds (Berry and Martineau 2008, p. 598-610).

The vortex street will only form depending on the range of flow velocities specified by the Reynolds Number used for LBM. The fluid flow is calculated by the formula:

$$Re_L = \frac{UL}{v_0}$$

Where Re_L is the Reynold Number, U is the free stream flow speed, L is the length parameter of the channel, and v_0 is the stream kinematic viscosity of fluid.

5.3 Program Design

5.3.1 Velocity Color and Cellular Automata

In LBM, velocities are no longer boolean, and are represented as floating points. These floating points can be utilized as colors in the form of RGB. Two colors, red and blue, will be controlled by velocities, meaning that green will always be 0.0; and whenever there is more red, there would be less blue, and vice versa. As a result, the velocities will be represented as red, blue, or magenta (a color mixed with blue and red).

Everytime there is a new generation, the cells will have different velocities, therefore the RGB color values change as well. This is done in the next generation, and every following generation after that. By default, each cell is size 2, meaning it's 2x2 in pixel size.

5.3.2 User Interface

The user interface for the LBM simulation is text based. This will allow for certain variables to change, such as the NACA number for the airfoil, and the obstacle used for the simulation. Typing in the object number will allow you to select the object. If an invalid object is used, the cylinder will be used as default. Allowing text inputs will allow the user to have the freedom of choosing the object they desire.

If the user selects an airfoil, the user can type in a four digit number not divisible by 100 (anything between 1 and 9,999, as long as the number isn't divisible by 100). All other numbers will be rejected, and the user has to enter a different number. The four digit number determines the shape of the airfoil. The first digit determines maximum camber, the second digit determines distance from maximum chamber from the airfoil edge, and the last two digits determine maximum thickness of the airfoil.

5.3.3 Aparapi

Aparapi is an API for Java where Java code can be converted to an OpenCL kernel so that it can run into the GPU. The kernels can be executed by the main program on multiple threads, allowing for parallel processing. The primitive data types Aparapi supports are: boolean, byte, float, int, long, one-dimensional arrays, and short. While Aparapi does support doubles, it depends on the GPU. Some GPUs don't support double-precision floating points, therefore Aparapi would be unable to use doubles. Aparapi is unable to use char or arrays that are multi-dimensional.

Aparapi can detect what devices a system has installed, and can determine the best device to execute the program. The best device could be the GPU or the CPU. In this case, it should be the GPU that is the best device for executing code. Alternatively, a specific type of device, such as the GPU, to execute the code is possible.

Aparapi kernels in Java are basically loops. The Global ID is the equivalent to "i" in standard loops and always increments by 1, and range is the equivalent of the maximum iterations of the loop. If there are additional loops in that loop, another Global ID replaces the line where the additional loop starts, and the range number is multiplied by the max iterations of that loop. The line where the kernel is executed is where the kernel runs.

This basically means that the kernel doesn't have to be in the same line as the execute line.

Chapter 6

Implementation

6.1 Converting to Java

When converting from the Python script to a Java program, the program code is all in one file instead of being in different classes and objects. The first step is to program the definitions section of the original Python code as variables, all before calling main(). These variables are final static variables, which are initialized in the program before the objects. The lattice population and velocities of the Lattice Constraints are also before main().

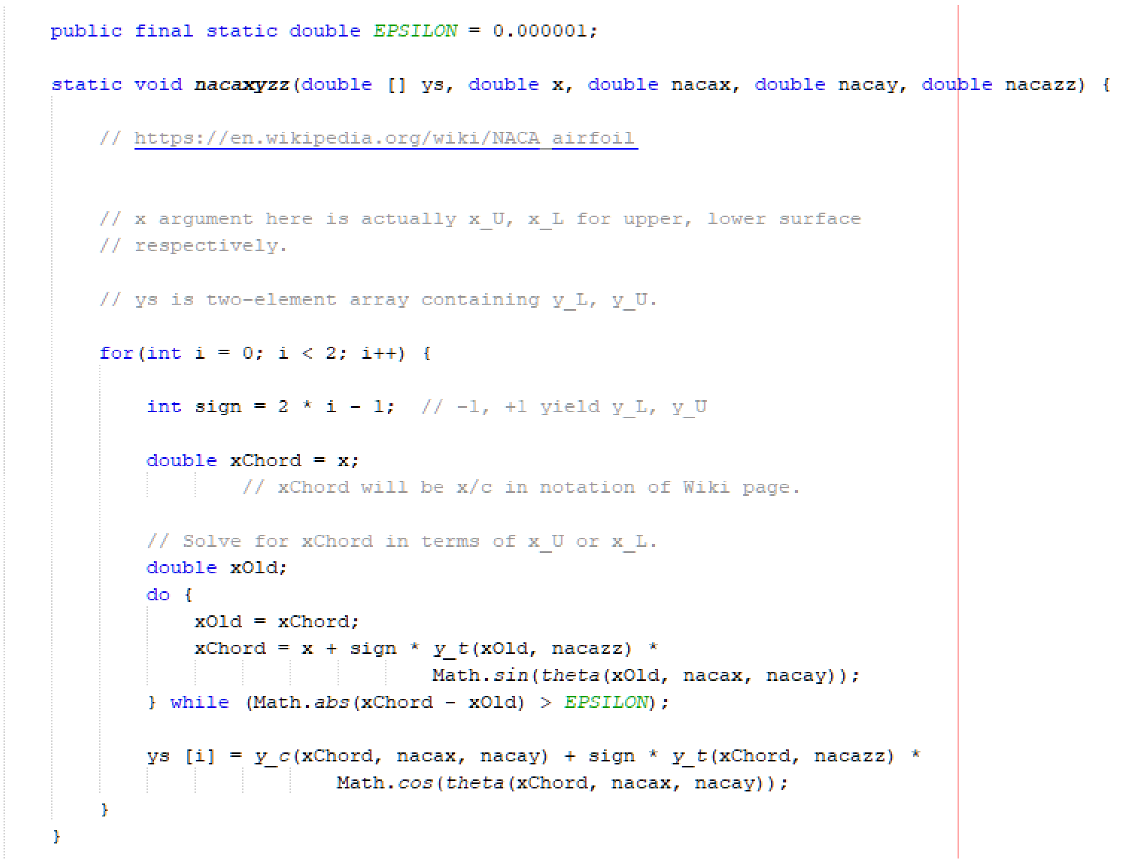
The rest of the Lattice Constraints are coded in main(). This includes vStates and noslip variables. None of the variables are static because the main() function is already a static method. The initial variables for the Macro, Collision, and Stream times are set as long and start at 0. The creation of an object (for the obstacle) is placed before Function Definitions instead of after like in Python. The object is a cylinder in this case, so a simple circle is calculated. The obstacle code is implemented later, in the collision step. The "vel", "fin", "fout", and "rho" variables come afterwards. They depend on lattice dimensions, and also call methods in the Function Definitions such as "equilibrium".

The Main Time Loop section is after Lattice Constraints, but before Function Definitions. This is where Macroscopic, Collision, and Streaming processes are coded. The macroscopic step is the calculations of the fluid flow; the collision step is the calculation of the obstacle, and the reaction to other cells and the velocities; and the streaming step is after the collision step and prepares for the next generation. Like in the python program, the three processes have their own loops, but are all inside the time loop. There are four variables, all measuring time. Time 1 is collected before the Macroscopic process; time 2 is collected after the Macroscopic process, and before the Collision process; time 3 is collected after the Collision process, and before the Streaming process; and time 4 is collected after the Streaming process. The difference in time 1 and 2 is Macro time, difference in time 2 and 3 is Collision time, and difference in time 3 and 4 is Stream time. This is to measure how long each process takes in milliseconds. The right wall outflow is calculated right before the end of each iteration. This finishes the main() method.

After the main() method, the rest of the Flow Definitions can be implemented. The main one is equilibrium, which is where the reactive flows are balanced in relation to each other. This is done as a static void, and is called when requested. The static class called "Display" is created after the equilibrium method. This is coded as a JPanel, which displays the simulation as a window. This is where each cell is assigned a color between red and blue (green is always 0) depending on velocity and relation to its surrounding cells. This is unlike the Python script where every 100 iterations are rendered images of data flow instead of showing the live simulation of the data flow in the windowed program.

6.2 NACA Airfoil

Since the conversion to Java is completed, the first thing to add to the program is to add more objects to use as obstacles. The new object is the airfoil. The airfoil is at first programmed separately from the main LBM program. The airfoil is based off the NACA-4415 airfoil, and the NACA-4415 is called as a separate function. Basically the first digit is the maximum camber in percentage, in this case 4% or 0.04; the second digit is the distance of maximum camber from the airfoil leading edge divided by 10, in this case 0.4; and the last two digits are the maximum thickness in percentage, in this case being 15% or 0.15. The EPSILON variable is a double and default value is 0.000001. These components are what is needed to make an airfoil.



```
public final static double EPSILON = 0.000001;

static void nacaxyzz(double [] ys, double x, double nacax, double nacay, double nacazz) {

    // https://en.wikipedia.org/wiki/NACA_airfoil


    // x argument here is actually x_U, x_L for upper, lower surface
    // respectively.

    // ys is two-element array containing y_L, y_U.

    for(int i = 0; i < 2; i++) {

        int sign = 2 * i - 1;  // -1, +1 yield y_L, y_U

        double xChord = x;
                // xChord will be x/c in notation of Wiki page.

        // Solve for xChord in terms of x_U or x_L.
        double xOld;
        do {
            xOld = xChord;
            xChord = x + sign * y_t(xOld, nacazz) *
                                Math.sin(theta(xOld, nacax, nacay));
        } while (Math.abs(xChord - xOld) > EPSILON);

        ys [i] = y_c(xChord, nacax, nacay) + sign * y_t(xChord, nacazz) *
                        Math.cos(theta(xChord, nacax, nacay));
    }
}
```

Figure 6.1: The first half of the main code for NACA Airfoil.

The NACA function is where the airfoil is calculated by calling the following three functions. The first is "y_t", where it calculates and returns the half thickness using the formula:

$$5t(0.2969\sqrt{x} - 0.126x - 0.3516x^2 + 0.2843x^3 - 0.1015x^4)$$

The variable t is the last two digits, in this case being 15; and x being carried from the original airfoil function.

The second function is "y_c", where it calculates and returns the mean camber line using one of the appropriate formulas depending on the value of x:

$$y_c = \frac{m}{p^2} \cdot x(2p - x), x \leq p,$$

$$y_c = \frac{m}{(1-p)^2}(1 - 2p + x(2p - x)), p \leq x,$$

The variable m is the max camber, which is the first digit (which is 4) divided by 100; and p is the distance of maximum camber, which is the second digit (which is 4) divided by 10.

The third function is “theta”, where it calculates the derivative value, using one of the appropriate formulas depending on the value of x:

$$derivative = \frac{2m}{p^2}(p-x), x \leq p,$$

$$derivative = \frac{2m}{(1-p)^2}(p-x), p \leq x,$$

The variable m is the max camber, which is the first digit (which is 4) divided by 100; and p is the distance of maximum camber, which is the second digit (which is 4) divided by 10. Derivative will then be calculated to find its arctan value, which is the value of theta.

Afterwards, the code is then modified so that the user can input any positive four digit integer not divisible by 100. Basically, only integers between 1 and 9999 (inclusive) are accepted as long as the number doesn’t divide by 100. The reason why the integer cannot be divisible by 100 is because if the last two digits are 00, the maximum thickness of the airfoil will be set to 0, and thus will not show up in the simulation. To prevent this, all numbers divisible by 100 are rejected. Any other value that fails to meet the 1-9999 criteria will also be rejected. When a value is rejected, the user has to enter another number again. This is also programmed to detect and automatically reject non-integers before being assigned to “int” variables to prevent the program from crashing.

6.3 Merging to LBM Program

When the coding is completed, the airfoil code is transferred to the main LBM program. The airfoil code (excluding the NACA function) is moved to where the cylinder code is, and the NACA function (including “y_t”, “y_c”, and “theta”) is below the display function.

However, because the cylinder is the object used for the obstacle, the airfoil would conflict with the simulation. To avoid this, an if statement is used for declaring what object is used. Object 0 is the cylinder, and object 1 is the airfoil. To determine the object used, the code will scan the input used, and the input is any integer. If an integer was entered and the integer in question is not assigned to an object, the object used will always default to the cylinder. Depending on whether the number 0 or any number not assigned to an object is entered, it should tell the user that “cylinder” is chosen in the case that the entered number is 0; otherwise it should tell the user that the desired object does not exist and will default to cylinder.

To prevent the program from potentially crashing, the scanner should be able to detect whether the inputted value is an integer or not, before the value is assigned to the variable. In the case where the value is not an integer, the program should reject the value, and have the user attempt again.

Additionally, the line ”Display display = new Display(u);” is moved from the start of Lattice Constraints to before the Time Loop. This way, the window pops up after the user successfully enters all the valid values for the program.

6.4 Flattening Multi-Dimensional Arrays

With the airfoil being added to the program, the next step would be implementing Aparapi, but there is a major problem with the code converted. Certain variables are using multi-dimensional arrays, which Aparapi does not support. What was done is convert all multi-dimensional arrays to one-dimensional arrays.

For most arrays, this is done by simply multiplying each dimension. For example, [x][y] becomes [x*y]. For arrays inside a two-dimensional loop (using i and j), other appropriate variables that affect the loop are multiplied with i, and i is added to j. Sometimes, it's not possible to simply convert to one-dimensional arrays because the variable has a number inside the array (either the left or the right side of the "=" sign). To get around this, a new variable is created as a base, such as "base_ij", and any instance of "ij" in this case has the variable add "base_ij" to the number inside the array. This applies to multiple variables with a number inside an array.

6.5 Double-Precision to Single-Precision Floating Points

After converting all multi-dimensional arrays to one-dimensional arrays, it is possible to use Aparapi. But before doing so, it is important to note that Aparapi may have issues with using double-precision floats if the GPU can't handle them. So some doubles (not necessarily all of them) have to be converted to floats for better compatibility with GPUs that are unable to use 64-bit double-precision floats.

This is done by changing "double" to "float" for all variables that are set up to be doubles. This may sound simple but in Java, changing the double to floats can cause errors in the code as doubles can't be converted to floats, even if the variable just has a number on its own. To guarantee this, all lines of code where the variable calls for a float has (float) at the beginning of a formula to make sure the variable is always going to be a 32-bit float. For example, "u [base_uij + 0] = (vel [j * 2 + 0]);" becomes "u [base_uij + 0] = (float) (vel [j * 2 + 0]);" For variables that initially have a value assigned, adding "F" at the end of the number will force it to be a 32-bit float. For example, "final static float Re = 220.0;" becomes "final static float Re = 220.0F;".

6.6 Implementing Aparapi

6.6.1 Setting up Aparapi in Java

Now, it is possible to apply Aparapi code to the program. Before being able to use Aparapi code, the Java program must be a Maven Java program. This is because Aparapi must be set as a Maven dependency to be able to execute functions. To add Aparapi to any Java project, you add to pom.xml:

```
<dependency>
    <groupId>com.aparapi</groupId>
    <artifactId>aparapi</artifactId>
    <version>1.8.0</version>
</dependency>
```

The Aparapi version used in this project is 1.8.0, which was the latest version at the time this project started (now, the latest version is 2.0.0).

6.6.2 Aparapi Kernel

The main Aparapi code to use for GPU programming is:

```
Kernel kernel = new Kernel() {
    @override
    Public void run() {
        int i = getGlobalID();
        ...
    }
};

kernel.execute(range);
```

That line of code is always used in place of loops, where the variable i becomes getGlobalID(), and "range" is how many iterations for the loop. That line of code was used at first for the Time Loop, but it caused problems for all time variables. Instead, three kernels are used within the Time Loop. These three kernels are called: kernelMacro, kernelCollision, and kernelStream; and are for Macroscopic, Collision, and Streaming processes respectively. However, just adding these lines of code, while being able to run, has a glitch where the velocities are calculated incorrectly, messing up the simulation. The only process that doesn't have this problem is kernelStream, but it ran worse than without the code, being 16 times slower in comparison.

6.6.3 Double Loop Kernels and Range

For kernelMacro, executing kernel code was changed. Because the Macroscopic process is a double-loop, "int j = getGlobalID(1)" is added below "int i = getGlobalID(0)", making the j loop unnecessary. The following lines of code are added:

Device device = Device.best() ;

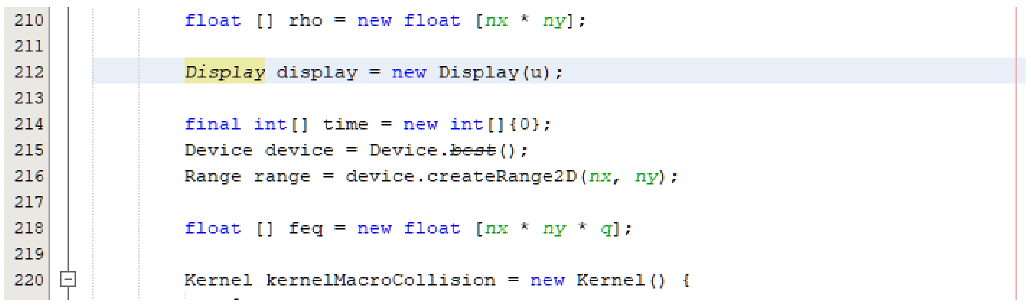
Range range = device.createRange2D(nx, ny) ;

What "Device device = Device.best();" does is to have the kernel run from the best device from the system (in this case, it should be the GPU). Because two global IDs are used, the range has to use two variables: nx and ny. This is because originally, Macroscopic was a double-loop with two different iterations for each loop. As a result, "kernelMacro.execute(nx);" is changed to "kernelMacro.execute(range);". This fixed the glitch with kernelMacro, but when these changes are applied to kernelCollision, the velocity glitch is still present.

Eventually, the three processes are moved outside of the Time Loop. They are now above the Time Loop and below the Lattice Constraints section (below "fin", and "fout"). Only the line of code that executes the kernels remain in the Time Loop. The problem with kernelCollision is gone. The changes from kernelMacro still apply to kernelCollision. kernelStream now has "int j = getGlobalId(1);" added where the loop for j begins. As a result, kernelStream is now a range like with kernelMacro and kernelCollision. The simulation speed is the same as without the Aparapi code.

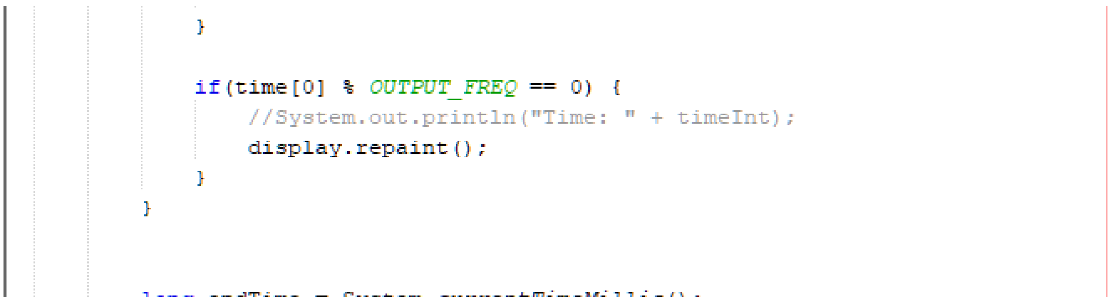
6.6.4 Explicit Buffer Handling

One possible reason why performance is not faster than the CPU is possibly because of the way Aparapi handles Buffer Handling. The problem is there are unnecessary buffer transfers from and to the GPU memory, and this can lead to a slowdown in performance.



```
        float [] rho = new float [nx * ny];

        Display display = new Display(u);

        final int[] time = new int[]{0};
        Device device = Device.~~best~~();
        Range range = device.createRange2D(nx, ny);

        float [] feq = new float [nx * ny * q];

        Kernel kernelMacroCollision = new Kernel() {
```

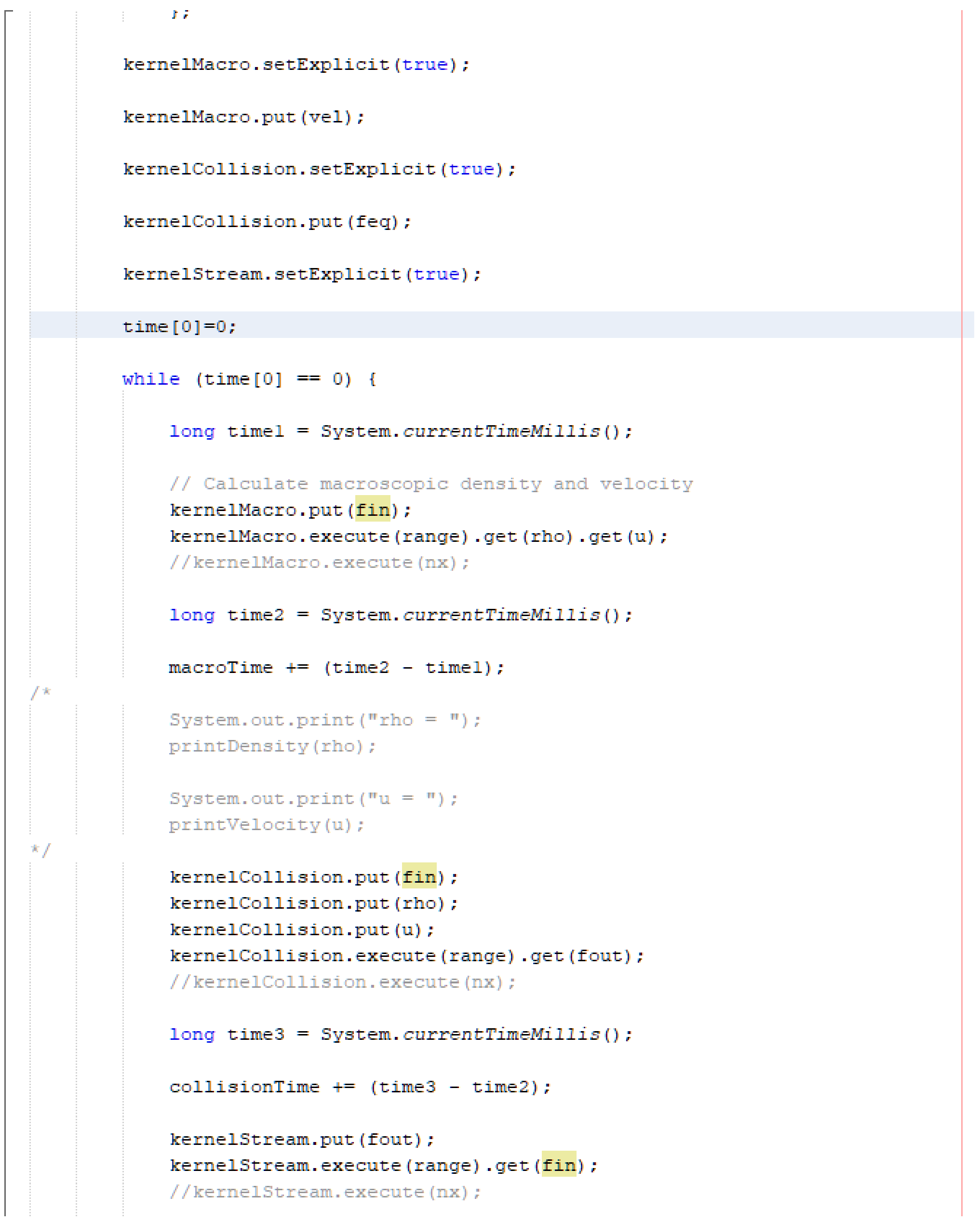
Figure 6.2: Part 1 of Buffer Handling. The variable “time” is set as an array.



```
        }

        if(time[0] % OUTPUT_FREQ == 0) {
            //System.out.println("Time: " + timeInt);
            display.repaint();
        }
    }
```

Figure 6.3: Part 2 of Buffer Handling. The revised iteration renderer.

To start with Buffer Handling, the Time Loop is changed from an if statement to a while loop. This is while time[0] == 0. All instances of “time” are replaced with “time[0]”. When this was changed, the performance was improved. This is because it no longer renders one iteration every 100 iterations, but every single iteration is rendered. This makes the simulation visually perform smoother. Note that the actual simulation rate is the same.



```
            };

        kernelMacro.setExplicit(true);

        kernelMacro.put(vel);

        kernelCollision.setExplicit(true);

        kernelCollision.put(feq);

        kernelStream.setExplicit(true);

        time[0]=0;

        while (time[0] == 0) {

            long time1 = System.currentTimeMillis();

            // Calculate macroscopic density and velocity
            kernelMacro.put(fin);
            kernelMacro.execute(range).get(rho).get(u);
            //kernelMacro.execute(nx);

            long time2 = System.currentTimeMillis();

            macroTime += (time2 - time1);
/*
            System.out.print("rho = ");
            printDensity(rho);

            System.out.print("u = ");
            printVelocity(u);
*/
            kernelCollision.put(fin);
            kernelCollision.put(rho);
            kernelCollision.put(u);
            kernelCollision.execute(range).get(fout);
            //kernelCollision.execute(nx);

            long time3 = System.currentTimeMillis();

            collisionTime += (time3 - time2);

            kernelStream.put(fout);
            kernelStream.execute(range).get(fin);
            //kernelStream.execute(nx);
```

Figure 6.4: The Buffer Transfers.

For the actual Buffer Handling, each kernel must have “setExplicit” to be set to true for Explicit Buffer Handling to work. Then for all variables used by the kernel, “put()” and “get()” calls are used. The call “put()” is for the input, and “get()” is for the output. All of the following calls are inside the Time Loop, unless stated otherwise. For the Macroscopic kernel, “vel” and “fin” are variables used for input, while the former is outside the Time Loop. The outputs are the “rho” and “u” variables. For the Collision kernel, “feq”, “fin”, “rho”, and “u” are input variables, with the first variable being outside the loop. The output variable is “fout”. Finally for the Streaming kernel, the input variable is “fout”, and the output variable is “fin”.

6.6.5 Macroscopic and Collision Kernels Merged

After testing, and finding that the Aparapi code is still slightly slower than the pre-Aparapi Java program, it was decided that the two kernels will be merged. The two kernels merged are Macroscopic and Collision processes. This is done by moving the feq variable before the Collision kernel to before the Macroscopic kernel. Then, the lines of code where the Collision kernel starts are removed, along with the “i”, “j”, and “base_ij” variables originally from the Collision kernel. This is done so that the “i”, “j”, and “base_ij” variables from the Macroscopic kernel are used instead. With that, the Macroscopic and Collision kernels are merged.

For the next step, all instances where the function calls for the Collision kernel is rewritten to use the merged Kernel. Also, the “get()” calls originally from the Collision kernel are moved to the newly merged kernel, and all duplicate calls are removed.

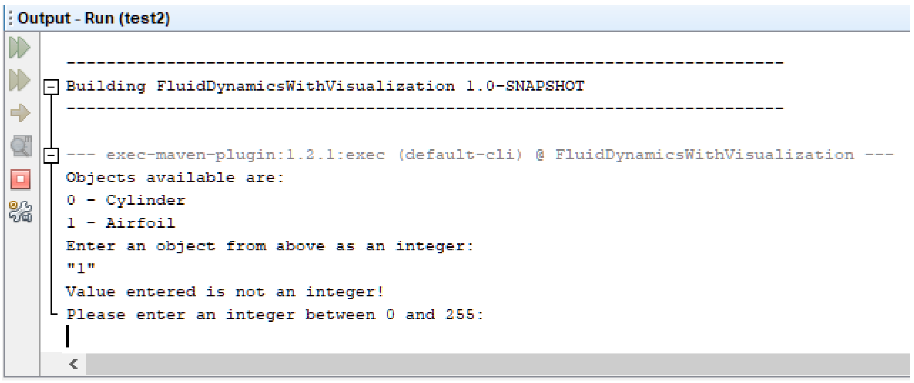
Chapter 7

Testing & Evaluation

The testing stage will cover three areas: Object Selection, Airfoil NACA Number, and Performance.

7.1 Object Selection

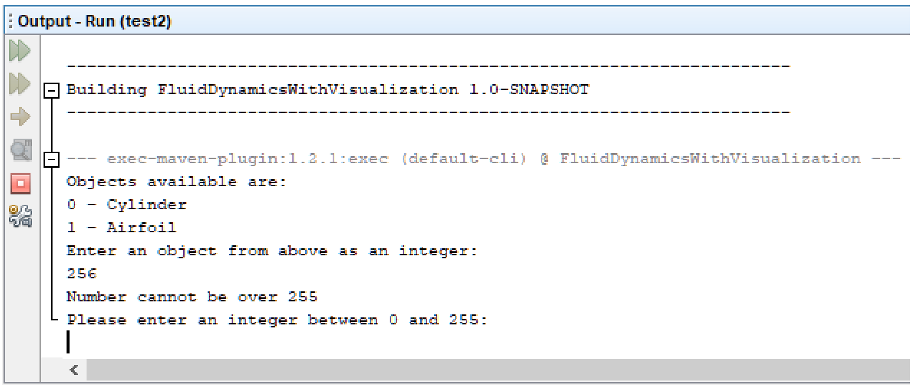
For object selection, the user inputs any value. There are six tests:



```
------------------------------------------------------------------------
Building FluidDynamicsWithVisualization 1.0-SNAPSHOT
------------------------------------------------------------------------

--- exec-maven-plugin:1.2.1:exec (default-cli) @ FluidDynamicsWithVisualization ---
Objects available are:
0 - Cylinder
1 - Airfoil
Enter an object from above as an integer:
"1"
Value entered is not an integer!
Please enter an integer between 0 and 255:
```

Figure 7.1: Object non-integer test.

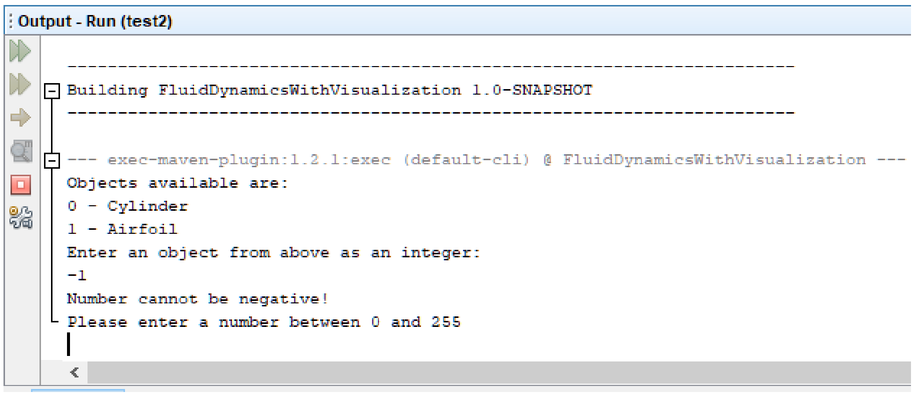
The first test will be to input anything that is not an integer. Basically, the user inputs a string or a floating point, the expected result is that the number is rejected and the program does not crash. The actual result is the number was rejected, and the user can enter another number again.



```
------------------------------------------------------------------------
Building FluidDynamicsWithVisualization 1.0-SNAPSHOT
------------------------------------------------------------------------

--- exec-maven-plugin:1.2.1:exec (default-cli) @ FluidDynamicsWithVisualization ---
Objects available are:
0 - Cylinder
1 - Airfoil
Enter an object from above as an integer:
256
Number cannot be over 255
Please enter an integer between 0 and 255:
```

Figure 7.2: Object over range test.

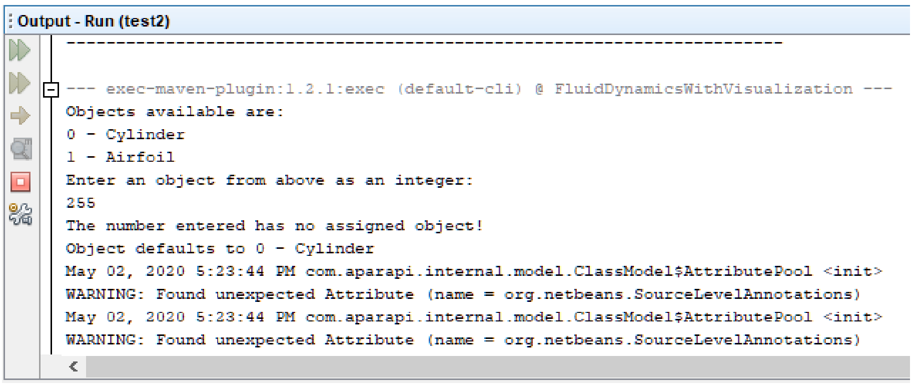
The second test will be to input 256. Since only values from 0 to 255 are accepted, 256 should be rejected. The actual result is 256 was rejected and the user can enter another number.



```
------------------------------------------------------------------------
Building FluidDynamicsWithVisualization 1.0-SNAPSHOT
------------------------------------------------------------------------

--- exec-maven-plugin:1.2.1:exec (default-cli) @ FluidDynamicsWithVisualization ---
Objects available are:
0 - Cylinder
1 - Airfoil
Enter an object from above as an integer:
-1
Number cannot be negative!
Please enter a number between 0 and 255
```

Figure 7.3: Object under range test.

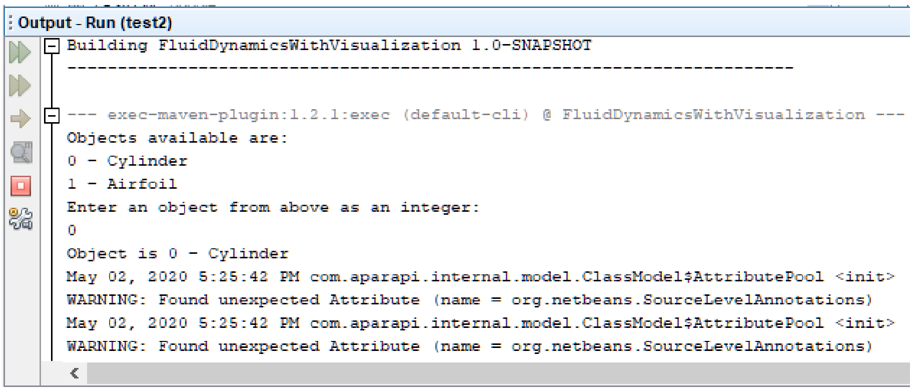
The third test will be to input -1. Since only values from 0 to 255 are accepted, -1 should be rejected because negative numbers are outside of range. The actual result is -1 was rejected and the user can enter another number.



```
Output - Run (test2)
------------------------------------------------------------------------

--- exec-maven-plugin:1.2.1:exec (default-cli) @ FluidDynamicsWithVisualization ---
Objects available are:
0 - Cylinder
1 - Airfoil
Enter an object from above as an integer:
255
The number entered has no assigned object!
Object defaults to 0 - Cylinder
May 02, 2020 5:23:44 PM com.aparapi.internal.model.ClassModel$AttributePool <init>
WARNING: Found unexpected Attribute (name = org.netbeans.SourceLevelAnnotations)
May 02, 2020 5:23:44 PM com.aparapi.internal.model.ClassModel$AttributePool <init>
WARNING: Found unexpected Attribute (name = org.netbeans.SourceLevelAnnotations)
```

Figure 7.4: Object at upper-limit test. Or Object not assigned test.

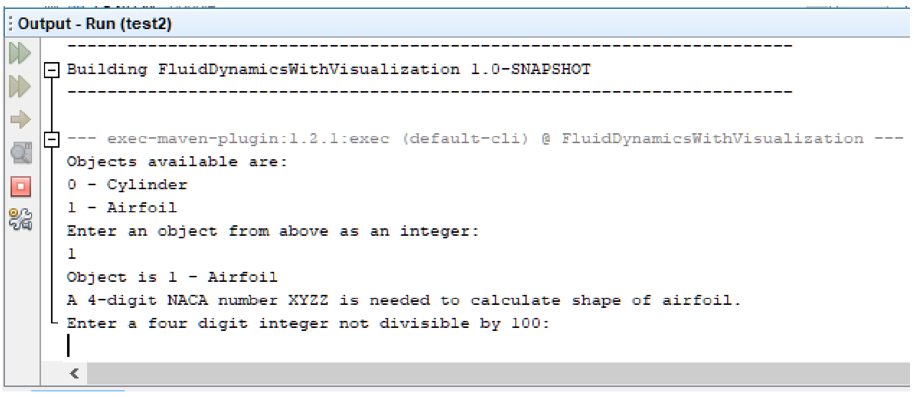
The fourth test will be to input 255. Since only values from 0 to 255 are accepted, 255 should be accepted because 255 is the upper limit. In addition, the expected result is that the object defaults to Cylinder because 255 has no assigned object, and should say that object is not found and defaults to cylinder. The actual result is 255 was accepted. Because 255 has no object assigned, it defaults to the Cylinder object. It also says that the object is not found and defaults to cylinder.



```
Output - Run (test2)
Building FluidDynamicsWithVisualization 1.0-SNAPSHOT
------------------------------------------------------------------------

--- exec-maven-plugin:1.2.1:exec (default-cli) @ FluidDynamicsWithVisualization ---
Objects available are:
0 - Cylinder
1 - Airfoil
Enter an object from above as an integer:
0
Object is 0 - Cylinder
May 02, 2020 5:25:42 PM com.aparapi.internal.model.ClassModel$AttributePool <init>
WARNING: Found unexpected Attribute (name = org.netbeans.SourceLevelAnnotations)
May 02, 2020 5:25:42 PM com.aparapi.internal.model.ClassModel$AttributePool <init>
WARNING: Found unexpected Attribute (name = org.netbeans.SourceLevelAnnotations)
```

Figure 7.5: Object at lower-limit test. Or Object 0 assigned test.

The fifth test will be to input 0. Since only values from 0 to 255 are accepted, 0 should be accepted because 0 is the lower limit. In addition, the expected result is that the object will use Cylinder because 0 is assigned to Cylinder. The actual result is 0 was accepted, and the Cylinder object is used.



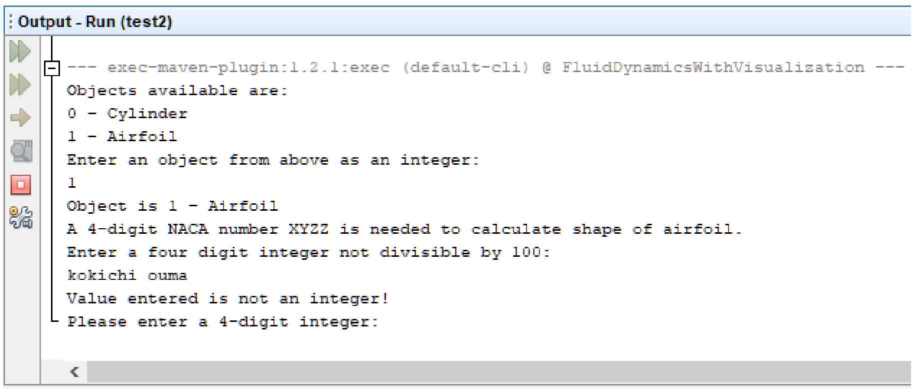
```
Output - Run (test2)
------------------------------------------------------------------------
Building FluidDynamicsWithVisualization 1.0-SNAPSHOT
------------------------------------------------------------------------

--- exec-maven-plugin:1.2.1:exec (default-cli) @ FluidDynamicsWithVisualization ---
Objects available are:
0 - Cylinder
1 - Airfoil
Enter an object from above as an integer:
1
Object is 1 - Airfoil
A 4-digit NACA number XYZZ is needed to calculate shape of airfoil.
Enter a four digit integer not divisible by 100:
```

Figure 7.6: Object 1 assigned test.

The final test will be to input 1. Since only values from 0 to 255 are accepted, 1 should be accepted because 1 is inside the range. In addition, the expected result is that the object will use Airfoil because 1 is assigned to Airfoil. The actual result is 1 was accepted, and the Airfoil object is used.

7.2 NACA Number

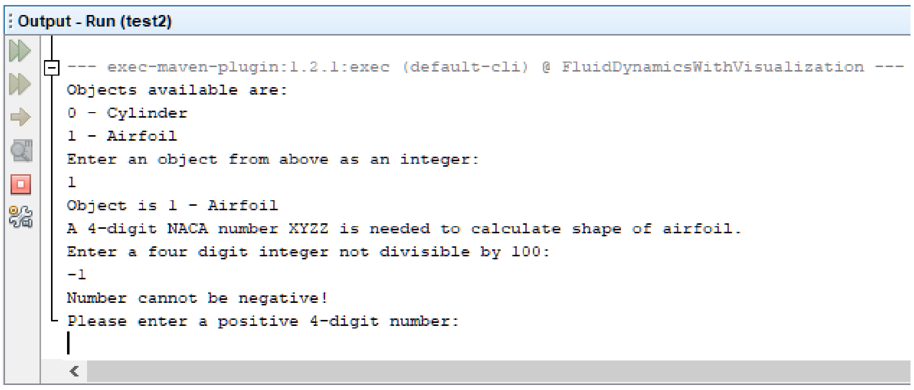
For NACA Number, the user inputs any integer between 1 and 9999 as long as the last two digits are not 00. There are eight tests:



```
Output - Run (test2)
--- exec-maven-plugin:1.2.1:exec (default-cli) @ FluidDynamicsWithVisualization ---
Objects available are:
0 - Cylinder
1 - Airfoil
Enter an object from above as an integer:
1
Object is 1 - Airfoil
A 4-digit NACA number XYZZ is needed to calculate shape of airfoil.
Enter a four digit integer not divisible by 100:
kokichi ouma
Value entered is not an integer!
Please enter a 4-digit integer:
```

Figure 7.7: Airfoil non-integer test.

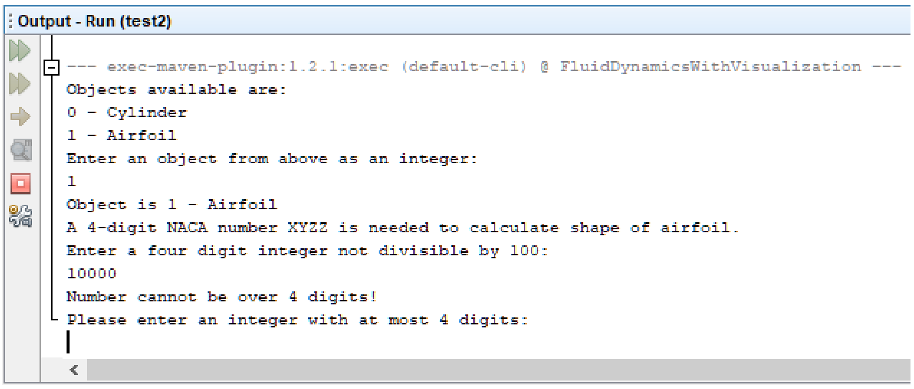
The first test will be to input anything that is not an integer. Basically, the user inputs a string or a floating point, the expected result is that the number is rejected and the program does not crash. The actual result is the number was rejected, and the user can enter another number again.



```
--- exec-maven-plugin:1.2.1:exec (default-cli) @ FluidDynamicsWithVisualization ---
Objects available are:
0 - Cylinder
1 - Airfoil
Enter an object from above as an integer:
1
Object is 1 - Airfoil
A 4-digit NACA number XYZZ is needed to calculate shape of airfoil.
Enter a four digit integer not divisible by 100:
-1
Number cannot be negative!
Please enter a positive 4-digit number:
```

Figure 7.8: Airfoil negative integer test.

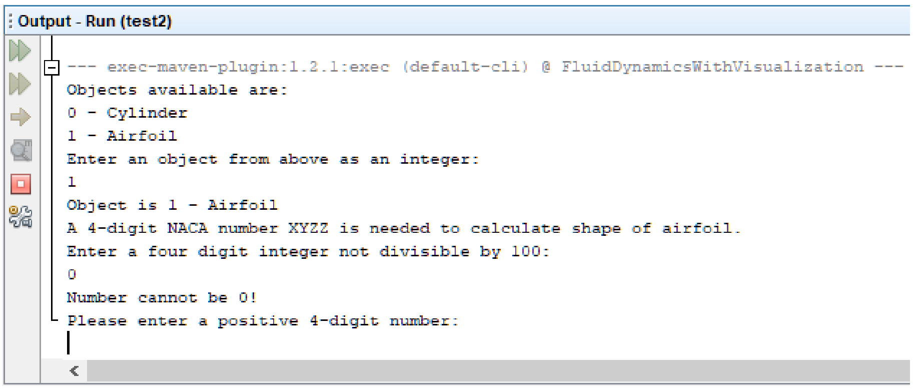
The second test will be to input a negative number, such as -1. The expected result is that negative numbers will be rejected, and the user will be notified that they entered a negative number. The actual result is -1 was rejected, the user was told a negative number was entered, and the user can enter another number. NOTE: Negative numbers divisible by 100 are not reported as divisible by 100.



```
--- exec-maven-plugin:1.2.1:exec (default-cli) @ FluidDynamicsWithVisualization ---
Objects available are:
0 - Cylinder
1 - Airfoil
Enter an object from above as an integer:
1
Object is 1 - Airfoil
A 4-digit NACA number XYZZ is needed to calculate shape of airfoil.
Enter a four digit integer not divisible by 100:
10000
Number cannot be over 4 digits!
Please enter an integer with at most 4 digits:
```

Figure 7.9: Airfoil 5-digit number test. Or Airfoil over limit test.

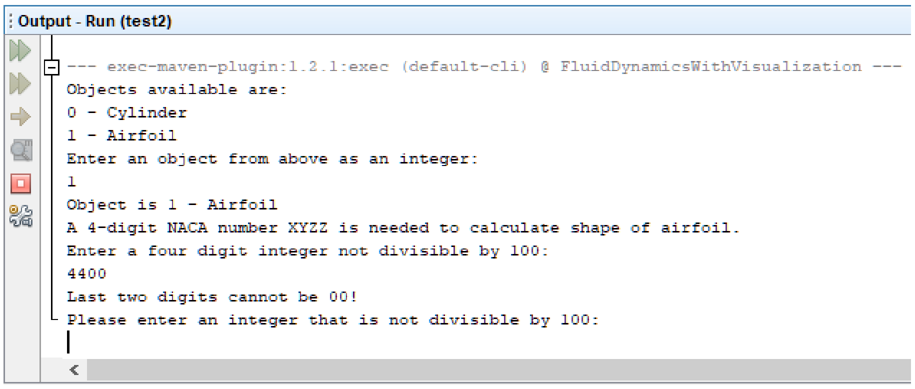
The third test will be to input any 5-digit number, such as 10000. The expected result is that all numbers with 5 digits will be rejected, and the user will be notified that they entered a number with more than four digits. The actual result is 10000 was rejected, the user was told a number over four digits was entered, and the user can enter another number. NOTE: 10000 is divisible by 100, but this is ignored.



```
--- exec-maven-plugin:1.2.1:exec (default-cli) @ FluidDynamicsWithVisualization ---
Objects available are:
0 - Cylinder
1 - Airfoil
Enter an object from above as an integer:
1
Object is 1 - Airfoil
A 4-digit NACA number XYZZ is needed to calculate shape of airfoil.
Enter a four digit integer not divisible by 100:
0
Number cannot be 0!
Please enter a positive 4-digit number:
```

Figure 7.10: Airfoil number divisible by 100 test.

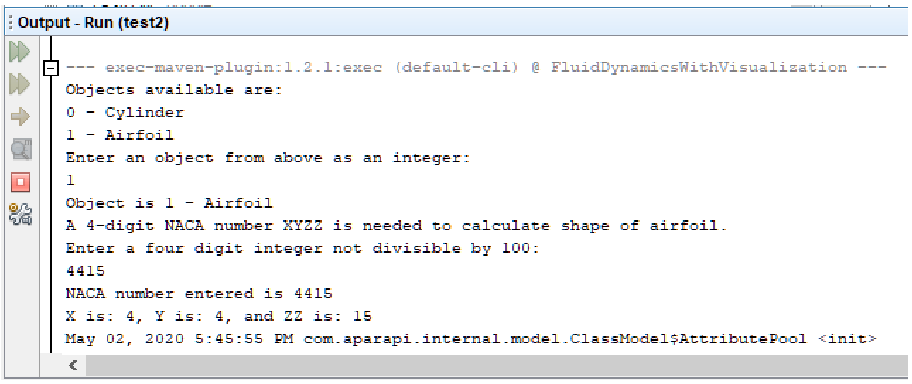
The fourth test will be to input a 4-digit number, but the number is divisible by 100, such as 4400. The expected result is that the number is rejected because any number with the last two digits being 00 would lead to the airfoil thickness being 0, so it would be rejected. The actual result is 4400 is rejected, the user is told the number is divisible by 100, and the user can enter another number.



```
--- exec-maven-plugin:1.2.1:exec (default-cli) @ FluidDynamicsWithVisualization ---
Objects available are:
0 - Cylinder
1 - Airfoil
Enter an object from above as an integer:
1
Object is 1 - Airfoil
A 4-digit NACA number XYZZ is needed to calculate shape of airfoil.
Enter a four digit integer not divisible by 100:
4400
Last two digits cannot be 00!
Please enter an integer that is not divisible by 100:
```

Figure 7.11: Airfoil 0 test. Or Airfoil under limit test.

The fifth test will be to input 0. The expected result is that the number is rejected because 0 is outside the range. The actual result is 0 is rejected, the user is told they entered 0, and the user can enter another number. NOTE: 0 is divisible by 100, but this is ignored.



```
--- exec-maven-plugin:1.2.1:exec (default-cli) @ FluidDynamicsWithVisualization ---
Objects available are:
0 - Cylinder
1 - Airfoil
Enter an object from above as an integer:
1
Object is 1 - Airfoil
A 4-digit NACA number XYZZ is needed to calculate shape of airfoil.
Enter a four digit integer not divisible by 100:
4415
NACA number entered is 4415
X is: 4, Y is: 4, and ZZ is: 15
May 02, 2020 5:45:55 PM com.aparapi.internal.model.ClassModel$AttributePool <init>
```

Figure 7.12: Airfoil 4415 test.

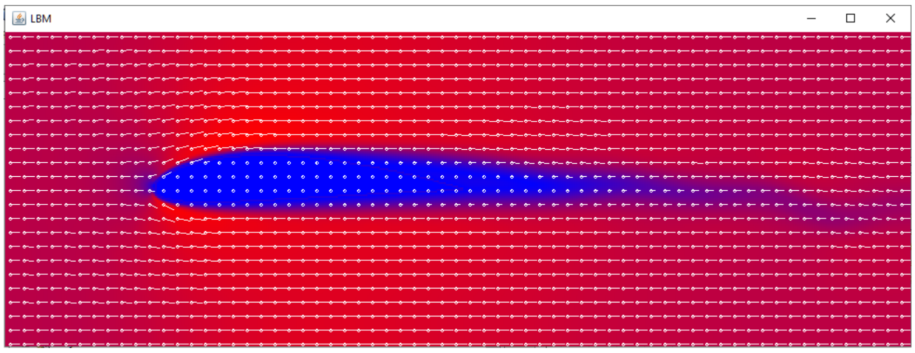
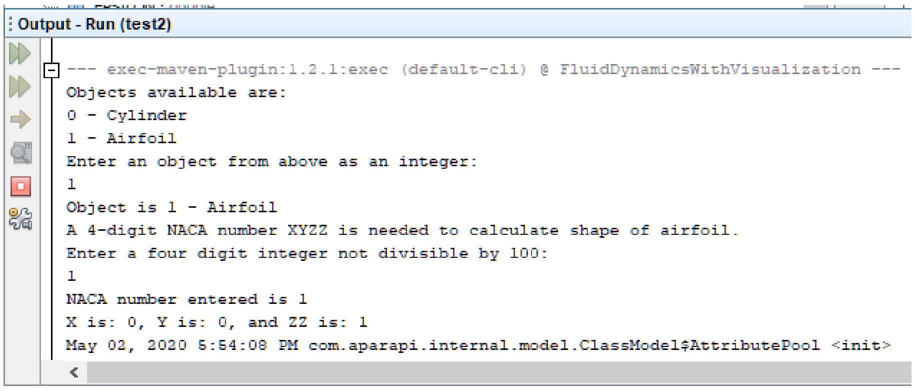


Figure 7.13: Airfoil 4415 simulation.

The sixth test will be to input 4415. The expected result is that the number is accepted because 4415 is inside the range and is not divisible by 100. Also, X is 4, Y is 4, and ZZ is 15. The actual result is 4415 is accepted, X is 4, Y is 4, and ZZ is 15.



```
--- exec-maven-plugin:1.2.1:exec (default-cli) @ FluidDynamicsWithVisualization ---
Objects available are:
0 - Cylinder
1 - Airfoil
Enter an object from above as an integer:
1
Object is 1 - Airfoil
A 4-digit NACA number XYZZ is needed to calculate shape of airfoil.
Enter a four digit integer not divisible by 100:
1
NACA number entered is 1
X is: 0, Y is: 0, and ZZ is: 1
May 02, 2020 5:54:08 PM com.aparapi.internal.model.ClassModel$AttributePool <init>
```

Figure 7.14: Airfoil at lower limit test. Or Airfoil 1 test.

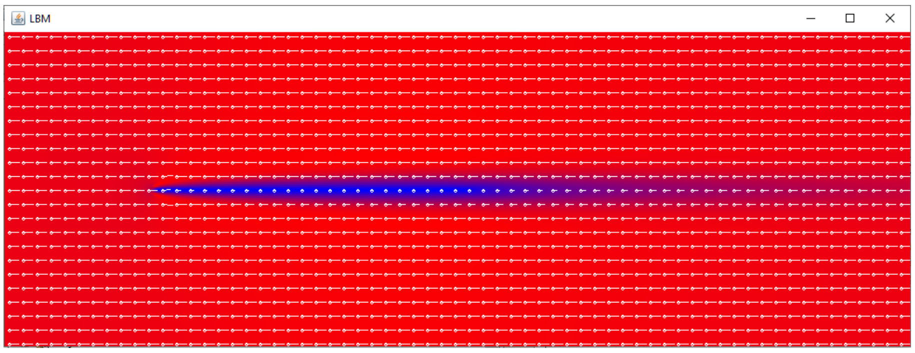
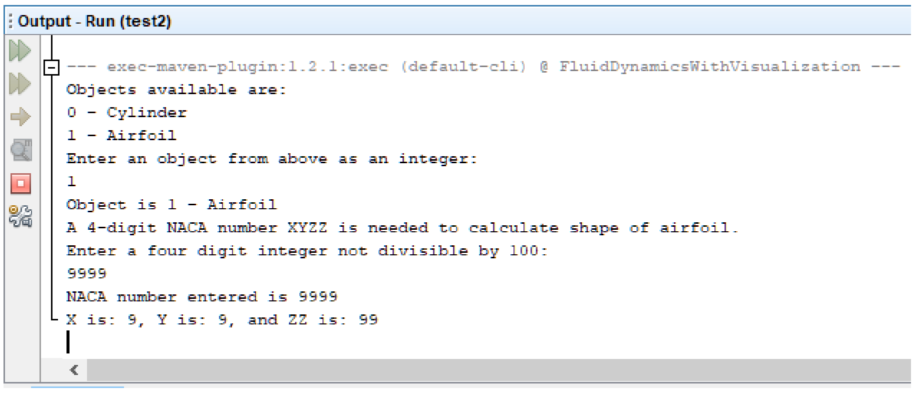


Figure 7.15: Airfoil 1 simulation.

The seventh test will be to input 1. It doesn't matter how many 0s are entered before 0, as long as the value is 1, the expected result is that the number is accepted because 1 is at the lower range and is not divisible by 100. Also, X is 0, Y is 0, and ZZ is 01. The actual result is 1 is accepted, X is 0, Y is 0, and ZZ is 01.



```
--- exec-maven-plugin:1.2.1:exec (default-cli) @ FluidDynamicsWithVisualization ---
Objects available are:
0 - Cylinder
1 - Airfoil
Enter an object from above as an integer:
1
Object is 1 - Airfoil
A 4-digit NACA number XYZZ is needed to calculate shape of airfoil.
Enter a four digit integer not divisible by 100:
9999
NACA number entered is 9999
X is: 9, Y is: 9, and ZZ is: 99
```

Figure 7.16: Airfoil at upper limit test. Or Airfoil 9999 test.

The final test will be to input 9999. It doesn't matter how many 0s are entered before 0, as long as the value is 9999, the expected result is that the number is accepted because 9999 is at the higher range and is not divisible by 100. Also, X is 9, Y is 9, and ZZ is 99. The actual result is 9999 is accepted, X is 9, Y is 9, and ZZ is 99. However, the simulation did not run unlike the previous two tests. After multiple tests with other values, the simulation not loading error only occurs if ZZ is set to any value above 48 whenever X and Y are both set to 9 each. Also whenever x and y are any other values, the program loads normally. However, the risk of a glitch occurring increases whenever the value exceeds 48. The glitch is where the velocities go unstable and thus making the simulation inoperable. All values ending in numbers higher than 58 will experience the glitch. While values ending between 49 and 58 don't experience problems, there is no guarantee these values will have any stability for longer run times.

7.3 Performance

For the Performance, the total time taken can be measured between three versions of the program. The first version is the Java program without any Aparapi code used. The second version is the Java program with Aparapi code, and with the Macroscopic and Collision kernels separate. The third version is the Java program with Aparapi code, but the Macroscopic and Collision kernels merged. In all tests, the Cylinder object is used. In the first three sets of tests, the Max Iteration is set to 2,000. In the second three sets of tests, the Max Iteration is set to 10,000. In the final three sets of tests, the Max Iteration is set to 50,000. The test will measure the time to complete the program in milliseconds. Each test will run 5 times per version, meaning in total 45 tests.

7.3.1 No Aparapi Test 2,000

The No Aparapi Test 2,000 results are:
First test: 7705ms.
Second test: 7309ms.
Third test: 7411ms.
Fourth test: 7351ms.
Fifth test: 7299ms.
Average time: 7415ms.

7.3.2 Aparapi Three Processes Test 2,000

The Aparapi Three Processes Test 2,000 (Macroscopic and Collision kernels not merged) results are:
First test: 15580ms.
Second test: 17409ms.
Third test: 14448ms.
Fourth test: 15255ms.
Fifth test: 14390ms.
Average time: 15416.4ms.

7.3.3 Aparapi Two Processes Test 2,000

The Aparapi Two Processes Test 2,000 (Macroscopic and Collision kernels merged) results are:
First test: 13164ms.
Second test: 12891ms.
Third test: 12974ms.
Fourth test: 12860ms.
Fifth test: 13576ms.
Average test: 13093ms.

7.3.4 No Aparapi Test 10,000

The No Aparapi Test 10,000 results are:
First test: 43813ms.
Second test: 43841ms.
Third test: 44247ms.
Fourth test: 45222ms.
Fifth test: 44170ms.
Average time: 44258.6ms.

7.3.5 Aparapi Three Processes Test 10,000

The Aparapi Three Processes Test 10,000 (Macroscopic and Collision kernels not merged) results are:
First test: 54479ms.
Second test: 55289ms.
Third test: 56731ms.
Fourth test: 53476ms.
Fifth test: 53825ms.
Average time: 54760ms.

7.3.6 Aparapi Two Processes Test 10,000

The Aparapi Two Processes Test 10,000 (Macroscopic and Collision kernels merged) results are:
First test: 47787ms.
Second test: 48047ms.
Third test: 48258ms.
Fourth test: 47989ms.
Fifth test: 47760ms.
Average test: 47968.2ms.

7.3.7 No Aparapi Test 50,000

The No Aparapi Test 50,000 results are:
First test: 256084ms.
Second test: 254087ms.
Third test: 245869ms.
Fourth test: 253839ms.
Fifth test: 249382ms.
Average time: 251852.2ms.

7.3.8 Aparapi Three Processes Test 50,000

The Aparapi Three Processes Test 50,000 (Macroscopic and Collision kernels not merged) results are:
First test: 253529ms.
Second test: 253734ms.
Third test: 253932ms.
Fourth test: 253026ms.
Fifth test: 252955ms.
Average time: 253435.2ms.

7.3.9 Aparapi Two Processes Test 50,000

The Aparapi Two Processes Test 50,000 (Macroscopic and Collision kernels merged) results are:
First test: 235552ms.
Second test: 229372ms.
Third test: 232900ms.
Fourth test: 227846ms.

Fifth test: 222979ms.
Average test: 229729.8ms.

7.3.10 Conclusion

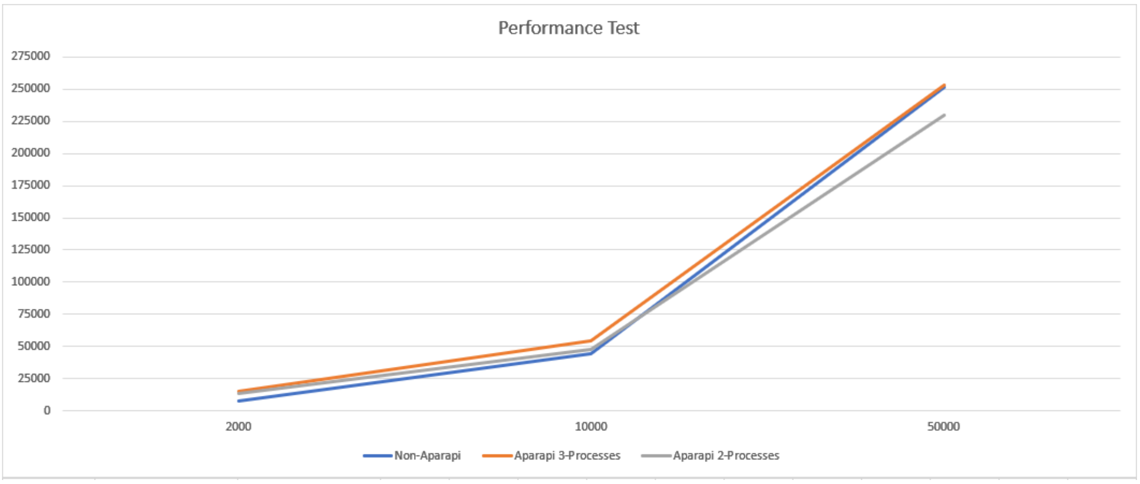


Figure 7.17: Line graph showing Performance results. The x-axis is time in milliseconds, and the y-axis is number of iterations.

The results show that with short time iteration (2,000), the Aparapi program with unmerged processes took twice the time to complete than the non-Aparapi program, while the merged processes Aparapi program is 2 seconds faster than the unmerged processes Aparapi program on average.

With medium time iteration (10,000), the Aparapi code with three processes is about 10 seconds slower than the original non-Aparapi code. However, the Aparapi code with the merged processes is about 7 seconds faster than the program with unmerged processes on average.

However, with longer time iteration (50,000), the two-process Aparapi code is 22 seconds faster than the non-Aparapi code on average, and while the three-process Aparapi code is still slower than the non-Aparapi code, the simulation is now 1.5 seconds slower on average compared to 10 seconds previously.

Chapter 8

Conclusion

The developed program has met all of the functional requirements. The Python script has been successfully converted into a Java program and successfully runs. The Fluid Dynamics simulation can visually simulate LBM on the GPU. The airfoil has been implemented to the LBM program and can handle any 4-digit positive integer not divisible by 100. The code was adapted to use Aparapi kernels. The program is easy to use. However, the program with Aparapi code (with two processes) is slower than the non-Aparapi program when run for a shorter duration, but faster when run for a longer duration.

8.1 Program Implementation

To start, the original Python script for the LBM program is successfully converted to Java. The Flow Definition section of the code is straight forward to port to Java since the variables are just defined or using simple calculations. After that, several lines of code were implemented in a different format from Python. Variables using arrays are generally the ones that are reworked the most. Additionally due to the way Java works, certain sections of code are in an area different from where it was originally.

Secondly, the NACA Airfoil is coded and is correctly calculated based on the 4-digit number used. Numbers divisible by 100 are rejected because these numbers always have the airfoil thickness set to 0. The airfoil is successfully implemented into the LBM program, and mostly runs successfully. However, the main limitation to the airfoil simulation is that any value that ends between 49 and 99 may experience a bug where the velocities lose stability and render the program unusable. However, while values ending between 49 and 58 are somewhat safe (as long as the first 2 digits are not 99), these values are not recommended to be used for longer run times.

Thirdly, the user can select any object they desire as the obstacle for the simulation. None of the values the user enters for the object selection crashes the program, even if a non-integer is inputted. When the user enters a value not assigned to an object, the obstacle will always default to cylinder. For both the Airfoil and the Object selection, the instructions for the user interface is clear, and should be very easy to use without any major training.

Additionally, the program has been adapted to use Aparapi by flattening down multi-dimensional arrays to one-dimensional arrays, and by using single precision floats instead of doubles. The Aparapi kernels were implemented successfully, and the code is executed to the GPU. The Aparapi program is also optimized for performance by introducing explicit buffer handling, and merging the Macroscopic and Collision kernels together. However, the performance gains compared to the non-Aparapi program are only minor.

8.2 Program Performance

It was expected that executing the program in the GPU would be faster than the CPU because the GPU has hundreds of cores. In this case however, the GPU code is slightly slower than the CPU code for approximately the first minute. However, this wasn't the case at first. The CPU code ran faster at first, then started to slow down after 1,200 iterations. In comparison between 2,000 and 10,000 iterations, the program ran about 19% slower; and when compared to 50,000 iterations, it's 36% slower.

In the case of Aparapi code before the Macroscopic and Collision kernels are merged, the program ran twice as slow as the non-GPU program at 2,000 iterations, but unlike the non-Aparapi program, the Aparapi program ran faster instead of slower. When comparing 2,000 and 10,000 iterations, the Aparapi merged-processes program runs about 43% faster; and comparing 2,000 and 50,000 iterations, the Aparapi program runs about 52% faster.

For Aparapi code after the Macroscopic and Collision kernels are merged, the program also ran twice as slow as the non-Aparapi program, but ran 18% faster than the Aparapi code with three processes. When comparing 2,000 and 10,000 iterations, the two-process Aparapi program ran 36% faster; and when comparing 2,000 and 50,000 iterations, the Aparapi program ran 42% faster.

Future work would be to look at why the Aparapi code runs faster over time, while the CPU program runs slower over time. Also for why the CPU runs faster than the GPU for the first 1,200 iterations, then started to slow down afterwards while the GPU is gaining speed.

Bibliography

Aidun, C. K. and Clausen, J. R., 2010. Lattice-boltzmann method for complex flows. annual review of fluid mechanics. *Annual review of fluid mechanics*, 42, 439–472.

Berry, R. A. and Martineau, R. C., 2008. Examination of the pcice method in the nearly incompressible, as well as strictly incompressible, limits. *Journal of Power and Energy Systems*, 2 (2), 598–610.

Chen, S. and Doolen, G. D., 1998. Lattice boltzmann method for fluid flows. *Annual review of fluid mechanics*, 30 (1), 329–364.

Docampo, J., Ramos, S., Taboada, G. L., Exposito, R. R., Tourino, J. and Doallo, R., 2013. Evaluation of java for general purpose gpu computing. *In 2013 27th International Conference on Advanced Information Networking and Applications Workshops*, 1398–1404.

Frisch, U., Hasslacher, B. and Pomeau, Y., 1986. Lattice-gas automata for the navier-stokes equation. *Physical review letters*, 56 (14), 1505.

Gupta, K. G., Agrawal, N. and Maity, S. K., 2013. Performance analysis between aparapi (a parallel api) and java by implementing sobel edge detection algorithm. *In 2013 National Conference on Parallel Computing Technologies*, 1–5.

Harris, M. J., 2005. Fast fluid dynamics simulation on the gpu. *SIGGRAPH Courses*, 220 (10.1145), 1198555–1198790.

Harwood, A. R. and Revell, A. J., 2017. Parallelisation of an interactive lattice-boltzmann method on an android-powered mobile device. *Advances in Engineering Software*, 104, 38–50.

He, X. and Luo, L. S., 1997. Lattice boltzmann model for the incompressible navier–stokes equation. *Journal of statistical Physics*, 88 (3-4), 927–944.

Kuznik, F., Obrecht, C., Rusaouen, G. and Roux, J. J., 2010. Lbm based flow simulation using gpu computing processor. *Computers & Mathematics with Applications*, 59 (7), 2380–2392.

Marié, S., Ricot, D. and Sagaut, P., 2009. Comparison between lattice boltzmann method and navier–stokes high order schemes for computational aeroacoustics. *Journal of Computational Physics*, 228 (4), 1056–1070.

Palabos, P., 2013. Lattice boltzmann. Retrieved from https://palabos.unige.ch/lattice-boltzmann/lattice-boltzmann-sample-codes-various-other-programming-languages/ [Accessed: 2020-05-05].

Pickering, B. P., Jackson, C. W., Scogland, T. R., Feng, W. C. and Roy, C. J., 2015. Directive-based gpu programming for computational fluid dynamics. *Computers & Fluids*, 114, 242–253.

Rendell, P., 2002. Turing universality of the game of life. *In Collision-based computing*, 513–539.

Wu, E., Liu, Y. and Liu, X., 2004. An improved study of real-time fluid simulation on gpu. *Computer Animation and Virtual Worlds*, 15 (3-4), 139–146.

Yang, C. T., Huang, C. L. and Lin, C. F., 2011. Hybrid cuda, openmp, and mpi parallel programming on multicore gpu clusters. *Computer Physics Communications*, 182 (1), 266–269.

Appendices

This is the original Python script used to convert to Java.

```
from numpy import *; from numpy.linalg import *
import matplotlib.pyplot as plt; from matplotlib import cm
###### Flow definition #########################################################
maxIter = 2000 # Total number of time iterations.
Re      = 220.0  # Reynolds number.
nx = 520; ny = 180; ly=ny-1.0; q = 9 # Lattice dimensions and populations.
cx = nx/4; cy=ny/2; r=ny/9;          # Coordinates of the cylinder.
uLB     = 0.04                       # Velocity in lattice units.
nulb    = uLB*r/Re; omega = 1.0 / (3.*nulb+0.5); # Relaxation parameter.

###### Lattice Constants #######################################################
c = array([(x,y) for x in [0,-1,1] for y in [0,-1,1]]) # Lattice velocities.
t = 1./36. * ones(q)                                   # Lattice weights.
t[asarray([norm(ci)<1.1 for ci in c])] = 1./9.; t[0] = 4./9.
noslip = [c.tolist().index((-c[i]).tolist()) for i in range(q)]
i1 = arange(q)[asarray([ci[0]<0  for ci in c])] # Unknown on right wall.
i2 = arange(q)[asarray([ci[0]==0 for ci in c])] # Vertical middle.
i3 = arange(q)[asarray([ci[0]>0  for ci in c])] # Unknown on left wall.

###### Function Definitions ####################################################
sumpop = lambda fin: sum(fin,axis=0) # Helper function for density computation.
def equilibrium(rho,u):              # Equilibrium distribution function.
    cu   = 3.0 * dot(c,u.transpose(1,0,2))
    usqr = 3./2.*(u[0]**2+u[1]**2)
    feq = zeros((q,nx,ny))
    for i in range(q): feq[i,:,:] = rho*t[i]*(1.+cu[i]+0.5*cu[i]**2-usqr)
    return feq

###### Setup: cylindrical obstacle and velocity inlet with perturbation ########
obstacle = fromfunction(lambda x,y: (x-cx)**2+(y-cy)**2<r**2, (nx,ny))
vel = fromfunction(lambda d,x,y: (1-d)*uLB*(1.0+1e-4*sin(y/ly*2*pi)),(2,nx,ny))
feq = equilibrium(1.0,vel); fin = feq.copy()

###### Main time loop ##########################################################
for time in range(maxIter):
    fin[i1,-1,:] = fin[i1,-2,:] # Right wall: outflow condition.
    rho = sumpop(fin)           # Calculate macroscopic density and velocity.
    u = dot(c.transpose(), fin.transpose((1,0,2)))/rho

    u[:,0,:] =vel[:,0,:] # Left wall: compute density from known populations.
    rho[0,:] = 1./(1.-u[0,0,:]) * (sumpop(fin[i2,0,:])+2.*sumpop(fin[i1,0,:]))

    feq = equilibrium(rho,u) # Left wall: Zou/He boundary condition.
    fin[i3,0,:] = fin[i1,0,:] + feq[i3,0,:] - fin[i1,0,:]
    fout = fin - omega * (fin - feq)  # Collision step.
    for i in range(q): fout[i,obstacle] = fin[noslip[i],obstacle]
    for i in range(q): # Streaming step.
        fin[i,:,:] = roll(roll(fout[i,:,:],c[i,0],axis=0),c[i,1],axis=1)
```

```
if (time%100==0): # Visualization
    plt.clf(); plt.imshow(sqrt(u[0]**2+u[1]**2).transpose(),cmap=cm.Reds)
    plt.savefig("vel."+str(time/100).zfill(4)+".png")
```