INTERACTIVE PARALLEL FLUID SOLVER USING THE LATTICE-BOLTZMANN METHOD AND CUDA

Jorge Mario G. Mazo

EAFIT UNIVERSITY
School of Engineering
Department of Informatics and Systems
MEDELLÍN

2010

INTERACTIVE PARALLEL FLUID SOLVER USING THE LATTICE-BOLTZMANN METHOD AND CUDA

Graduation project for the degree of:
Computer Scientist

Assessor
Prof. Dr. Manuel Julio García

EAFIT UNIVERSITY
School of Engineering
Department of Mechanical Engineering
MEDELLÍN
2010

“Well, it’s 1 a.m. Better go home and spend some quality time with the kids.”
Homer Simpson

To my mother for all her support and patience during all these years. Also to my brother, my master and best friend. And to my sister, that although she is away, she has been closer than anyone else. They know that without them, this journey could not have been completed.

Abstract

Particle methods have been gaining momentum in the Computational Fluid Dynamics world, for their computational simplicity and inherent parallelism. This has lead the field to jump to new hardware technologies that exploit the parallelism in a whole new way.
This project is an interactive parallel implementation of the Lattice Boltzmann Method that runs on a hyper parallel architecture as a modern GPU and uses CUDA technology for its implementation. First the mathematical concepts of the method are introduced, then the implementation is done using the SIMD paradigm, to later on attack the visualization and interactivity simple way.

Acknowledgements

There were many people involved on the development of this project and I would like to thank the following people: my advisor, Prof. Manuel J. Garcia, for returning me the faith in my degree. Also the fine crew of the Applied Mechanics Research Laboratory, Juan (abuelo) Duque, Santiago (kid) Orrego, Santiago (muerzo) Giraldo and Doriam for making the lab a fun place to work and learn. My friends Andres (chapa) Chaparro y Gustavo (tavo) Betancur, the university would have been such a boring place without those two guys.
Last but not least to my girlfriend Luisa (snake) Machado, grandma, all my aunts, my cousins and my nephew Juan Manuel, thank you all for your constant support.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

The mankind nature to try to understand what it is around, has driven humans to study the environment we are immersed in... *fluids*. We are surrounded by fluids, water and air, and until recently we knew little about the behavior of those and the impact they have in our lives.

1.1 Mechanics of Fluids

Fluid mechanics is a filed that study the substances that continually flow under an applied shear stress and the forces on them. It can be divided into fluid statics, the study of fluids at rest, and fluid dynamics, the study of fluids in motion [wik10c]. Fluid dynamics, is an active field of research with many unsolved or partly solved problems. This makes the field, an interesting subject, to get involved in the scientific world.

1.1.1 Historical review

With only a rudimentary knowledge of fluid flow humans built wells, channels, pumping devices etc. With the exception of the works of Archimedes (287 – 212 B. C) on the principles of buoyancy, little knowledge of the ancient times is present on modern fluid mechanics. After the fall of the Roman Empire there is no record of advances in fluid mechanics until Leonardo da Vinci (1425–1519). He unleashed a new era in hydraulic engineering, he studied the flow of turbulent currents, the flight of the birds and the forces involved, although his work was impressive it was more artistic than

science.
After da Vinci hydraulics gained momentum with great scientific contributions from Galileo, Torriceli, Marioti, Pascal, *Bernoulli, Euler* and d'Alembert. All their works were magnificent but there was always discrepancies between the theory and the practice, this discrepancy lead to the born of two schools of thought that still exist today, the mathematica field of fluid mechanics know as *hydrodynamics* and the practical filed known as *hydraulics*.
Near the middle of the 1800s Navier and Stokes succeeded modifying the general equations for ideal fluid motion to fit in the viscous fluid, narrowing breach between hydraulics and hydrodynamics. Towards the end of that century four advances were crucial to the development of the fluid mechanics science, and there were: (1)Theoretical and experimental work by *Reynolds*, (2)Dimensional analysis by Rayleigh, (3)Use of model by Froude, Reynolds et. al and (4)Experimental and theoretical progress in aeronautics by Lanchester, Lilienthal, kutta, Joukwsky, Betz and Prandtl. The most important contribution was made by Prandtl in 1904 when he introduced the boundary layer, linking the ideal and real fluid motion for fluids with small viscosity [SWV96]. This contribution by Prandtl laid the ground of modern fluid mechanics.

1.1.2 Navier–Stokes and the fluid motion

Fluid motion can be seen in many different forms, from simple flows such as laminar flows in a pipe, to very complex flows including high degrees of turbulence. Many flows have been analyzed experimentally, however nowadays for more complex fluid flows is more convenient to develop a numerical approach capable of simulating many different flows, not yet analyzed experimentally.

In the following sections of this chapter a review of the equation of fluid motion will be presented. A set of *classical* numerical methods, to solve such equation will be presented, to later on move to small introduction to the relatively new particle methods and its implementation in new parallel hardware architectures.

Fluid motion of an incompressible fluid is governed by the continuity equation

$$\nabla \cdot \mathbf{u} = 0 \tag{1.1}$$

and the Navier–Stokes equation.

$$\rho\left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u}\right) = -\nabla p + \mu\nabla^2\mathbf{u} + \mathbf{f} \tag{1.2}$$

where ρ is density, $\mathbf{u}$ is velocity, p is pressure and μ is the kinematic viscosity. The Navier–Stokes equation is a non–linear second order partial differential equation with not known analytical solution except for a small number of simplified cases.
With the advent of computer technology there has been attempts to approximate the solution of the Navier–Stokes equation using numerical analysis to simulate fluid flows.

1.2 Numerical methods

The functionality or efficiency of a technical system is always determined by some properties. A deep knowledge of these properties is needed in order to properly *modify* or *optimize* these technical system. In engineering and particular to fluid mechanics, some of these properties are:

- flow velocities.
- temperature.
- pressure
- drag or lift forces

A list with few examples of these properties related to fluid mechanics is given in Table 1.1.

Property	**Functionality**
Aerodynamics of vehicles	Fuel consumption
Pressure drop in vacuum cleaners	Sucking performance
Pollutants in exhaust	Environmental burden

Table 1.1: Examples of correlation of properties with functionality and efficiency of technical systems

For engineering, the study of these properties are important for the redevelopment and enhancement of products or processes. The knowledge gained from the study of important variables of fluid motion can help engineers to:

- improve efficiency
- improve of durability
- reduce pollution
- save raw material
- have a better understanding of a process

In the industrial world cost reduction might be the key term related to the list mentioned above but for scientists some times is more important to have a better understanding of the process subject of study.

The importance of numerical methods arises from checking the available methods to gain knowledge on the properties of complex systems. The list of these methods is very limited and be expressed as follows:

- Theoretical methods.
- Experimental investigations.
- Numerical simulations.

Theoretical methods: analytical consideration of the equations of complex problem is not usually viable, the equations governing realistic physical phenomena are so complex (usually non–linear partial differential equations) that they are not solvable analytically for most of realistic applications.

Experimental investigations: aim to to obtain the required system information by the experimental tests. In many cases this can be rather problematic: the measures of the object are too big or too small making difficult to measure variables, the process elapses for a very small fraction of time to takes very long time to end, object of study cannot be accessed (a galaxy), the experiment is too risky, and the most important aspect for the industry is that experiments are time consuming and *expensive*.

Numerical simulations: in–between theoretical methods and experimentation, in recent years numerical simulation has become a well established scientific and industrial discipline. Here physical phenomena is studied by means of numerical methods on computers. The advantages of these compared to experimentation are:

- Faster results at lower costs.
- Parameter variations on the computer are usually easy.
- More comprehensive information about variables of the system.

An important prerequisite for exploiting the usefulness of numerical simulations are of course *computers*. The booming in the computer industry has for sure helped the take–off of the numerical simulations, but numerical simulations are not and will never be a full replacement for experimental investigations and in general experimental validation should be accomplished to verify the accuracy of the numerical simulation.

The development of numerical methods is nothing new, Gauß and Euler where already working on numerical methods but due to the large number of computations required to approximate an equation no real benefit was extracted until the computer era. All the advances in computers like faster processors each year, better memory schemes, etc. have helped the mankind to really exploit the numerical simulations, however it was not just only the development in computers that helped the field to succeed but mayor improvements in numerical algorithms and better measurement tools. One can see that improvements in numerical simulation in the future are going to be fantastic, emerging new technologies like hyper–parallel processors (GPUs just to mention one), and new fast adaptive numerical methods will allow us to simulate more complex systems than ever before.
Based on this assumption one can presume that the demand for specialist in numerical simulation will be increasing in the near future, therefore the need to train and research on these topics.

Many aspects are involved during a numerical simulation. One mayor problem that arises in numerical modeling is the continuous analysis – usually engineering problems are differential or integral equations derived from continuum mechanics – continuum solutions are not viable due the fact the computational resources are

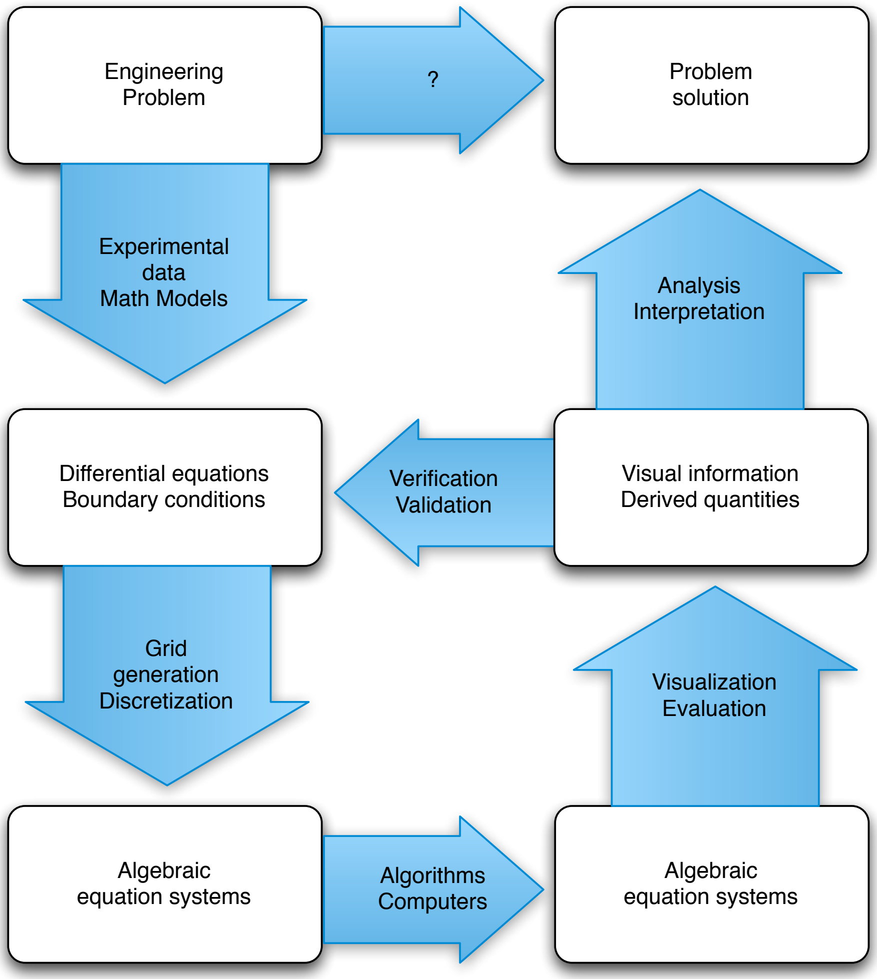


Figure 1.1: Numerical simulation procedure of engineering problems (Schäfer).

limited and a discrete approach has to be taken. The unknown quantities have to expressed by a finite number of values, this process is know as *discretization* [Sch06] and it has two main tasks:

- Discretization of the problem domain.
- Discretization of the equations.

The discretization of a continuous problem consists in the division of the continuous domain in space and time into smaller sub–domains know as *elements*. The description of a topology discretized by elements is know as *mesh*. Figure 1.2 shows the computer generated mesh of a war airplane.
The discretization of the equations is a core aspect of the numerical methods, this

is because problems modeled with differential equations are expressed in terms of continuum variables in space and time.

There are several numerical techniques, to approximate the differential equations, and these are continuously improving due to the larger research in the area. Some of these techniques would be briefly reviewed in Section 1.4.

An overview of setting up a numerical simulation can be seen on in Figure 1.1. Starting with an engineering problem that is described either by experimental data or differential equations and boundary conditions. This model is discretized into a set of algebraic equations suitable for solving with computers. The solution is accomplished by different techniques and the data is analyzed to obtain the problem solution.

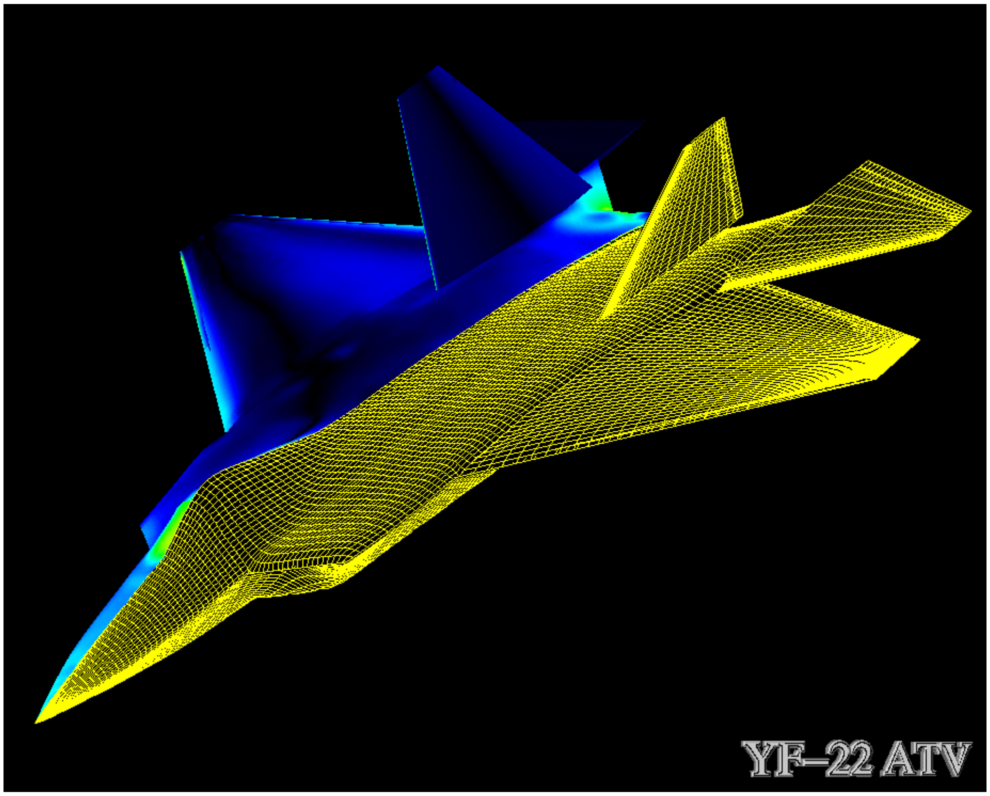


Figure 1.2: Mesh of an F–22 airplane(NASA).

1.3 Computational Fluid Dynamics

Computational Fluid Dynamics or CFD is the analysis of systems involving fluid flow, heat transfer and associated phenomena such as chemical reactions by means of computer–based simulations[VM95]. The technique is very powerful and has a lot of industrial and scientific applications such as:

- aerodynamics
- hydrodynamics
- turbo–machinery
- chemical process
- meteorology
- biomedical engineering

Since the 1960s the aerospace industry has been leading the way in the CFD arena integrating the methodology to the all its manufacturing chain. The CFD has lagged behind other numerical simulation techniques due to the tremendous complexity of the fluid flows. But the introduction of cheap high–performance computing solutions and easier programming models, leaded CFD to enter the industrial community in the 1990s.

1.3.1 How does CFD works?

Pre–Processor

The pre–processor consist of the input of data parameters of a flow problem to a CFD program, this input is described in a form that the solver can understand. The activities at this stage include: meshing, setting fluid properties, choosing the solver and specifying boundary conditions.

Solver

The solver is where the numerical methods kick–in. After you have a model of a physical system, it's necessary to compute the solution. Usually, the model is so

large and complex that numerical methods are needed to approximate the solution, rather than attempting to get an “exact” solution.
Numerical solvers have a basic characteristics, and these have to efficient not just numerically but also computationally.
Many streams of numerical methods for CFD exist today, the classic approach using elements and the new trends using *particles*.

Post–processor

After the problem has been solved, the results are usually sets of data that cannot be easily interpreted by humans. The post–processor is a *visualization* tool that presents the output of the solver in such a way that can be easily interpreted and analyzed by people. Some of the possibilities that post–processors give, are: vector plots, particle tracers, viewing manipulations, etc. Also post–processing offers the possibility to operate the solver data into new variables of interest such as energy, vorticity, etc.

1.3.2 Problem solving with CFD

When solving fluid flow problems, we have to be aware that the physics are complex and that the solution is going to be at best as good as the physics and at worst as its operator. The user of CFD must have skills in a number of areas prior to setting up, running, and evaluating a solution. On CFD many assumptions have to be made according to the physics of the problem, and importation decisions regarding the modeling of the problem have to be taken. These decisions can determine the result of the problem, sometimes yielding improper results.
Also a good understanding about the numerical method is also crucial, key mathematical concepts such as **convergence**, **consistency** and **stability** have to be always present for the CFD operator.

1.4 Numerical methods for CFD

Many methods have being created to study the fluid flows. The *classical* ones, the ones that have received more research and the trust of the industry are: the finite difference method, finite elements method, spectral method and finite volume method. These methods will be explained in more detail below. Although they differ, the basic requirements for numerical simulations are the same for all methods, and these(requirements) are shown on Figure 1.3.

1.4.1 Finite difference method (FD)

The FD methods describe the unknown function at the points of grid co–ordinate lines. Truncated Taylor series expansions are often used to generate finite difference approximations of the derivatives of the unknown functions in terms of point samples of the functions at each grid point and its immediate neighbors.

1.4.2 Finite elements method (FEM)

The FEM approximate solutions of partial differential equations (PDE) as well as of integral equations. The solution approach is based either on eliminating the differential equation completely (steady state problems), or rendering the PDE into an approximating system of ordinary differential equations, which are then numerically integrated using standard techniques such as Euler's method, Runge-Kutta, etc[wik10b].

1.4.3 Spectral method (SM)

The SM approximate the unknowns by means of truncated Fourier series or series of Chebyshev polynomials. Unlike finite element method or the finite difference method the approximations are not local but valid throughout the entire computational domain. The weighted residuals concept is also used to minimize the error similar to the finite element method.

1.4.4 Finite volume method

The FVM was originally developed as a special finite difference formulation. The method consists in three steps: the formal integration of governing equations over all the elements, converting the integral equations into a system of algebraic equations and then solving those system via an iterative method. The first step distinguishes the FVM from the other techniques because the resulting statements express the *exact* conservation of relevant properties for each element[VM95].

In general all these methods involve the solution of large systems of linear equations, making parallel computing practically a must for realistic applications.

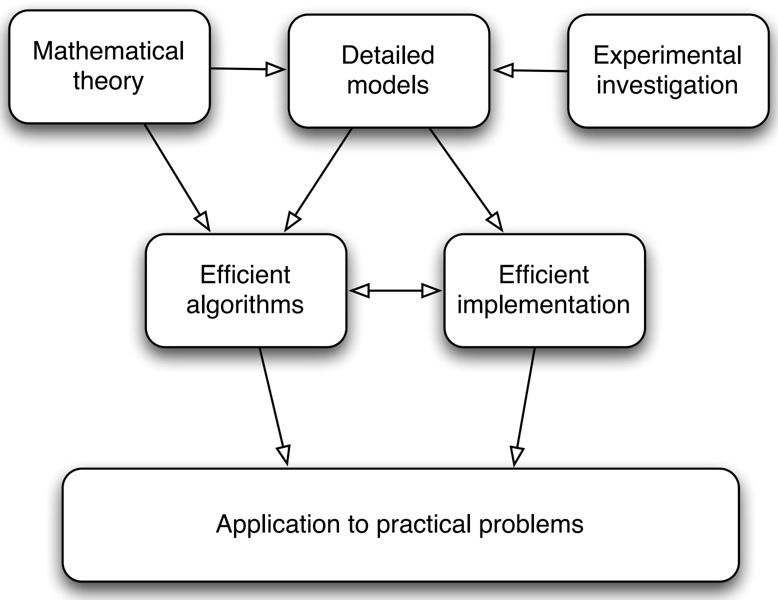


Figure 1.3: Requirements and interdependences for the numerical simulation of practical engineering problems(Schäfer).

1.5 Particle methods “the new trend”

During the last few decades a new trend has been gaining popularity among the CFD community. This trend know as *particle methods*. Particle methods have become one of the most useful and widespread tools for approximating solutions of partial differential equations in a variety of fields. In these methods, a solution of a given equation is represented by a collection of particles, located in points x_i and carrying masses w_i. Equations of evolution in time are then written to describe the dynamics of the location of the particles and their weights.
Many particle methods exist, for example Smoothed Particle Hydrodynamics a mesh–free method, Molecular Dynamics, Lattice–Boltzmann, etc.
These “new” methods rely on the simulation of the motion of interacting particles

carrying physical information [eth10], this approach is deceivingly simple, yet powerful and natural, method for exploring physical systems as diverse as planetary dark matter and proteins, unsteady separated flows, and plasmas.

One big advantage of the Particle Methods is the inherent parallelism. This is due that in most cases each particle or packet of particles can be computed independently. This makes convenient the use of massively parallel architectures such as Graphic Processing Units (GPUs) to implement these numerical methods. The speed–up achieved but such architectures can be impressive in most application as shown in the Figure 1.4. The figure shows how the raw power of GPUs has helped to get dramatical improvements in performance, but it also shows how the efficiency, has been dropping which means that hardware architectures are not been exploited to their fullest.

The Lattice Boltzmann Method was chosen for this work, because its highly parallel capabilities, also the method uses a rectangular fixed grid, and Fix Grid FEM is a mayor area of research in the laboratory where this work is being done. This will allow future ideas of coupling between the two methods for the analysis of complex fluid–structure interactions.

This work presents a novel approach to interactive fluid simulations. A reliable and parallel–friendly method was necessary to achieve speeds fast enough to allow the user to interact with the simulation. The user would be able to move objects inside a fluid domain and it will get physically accurate feed back of different fluid variables as pressure, speeds and vorticity in almost real time, without the need of a post processing tool.

To get near real time speed a fast computing architecture is also needed, and most NVIDIA GPUs provide enough power, to supply the demanding needs of interactive simulations.

Finally, it has to be noted, that the arena of the parallel architectures is changing fast. The cell architecture (the one used by the Sony Play Station 3) is becoming every day more appealing for the implementation of scientific applications because of its high powered parallel processors and its low cost [KPP06].

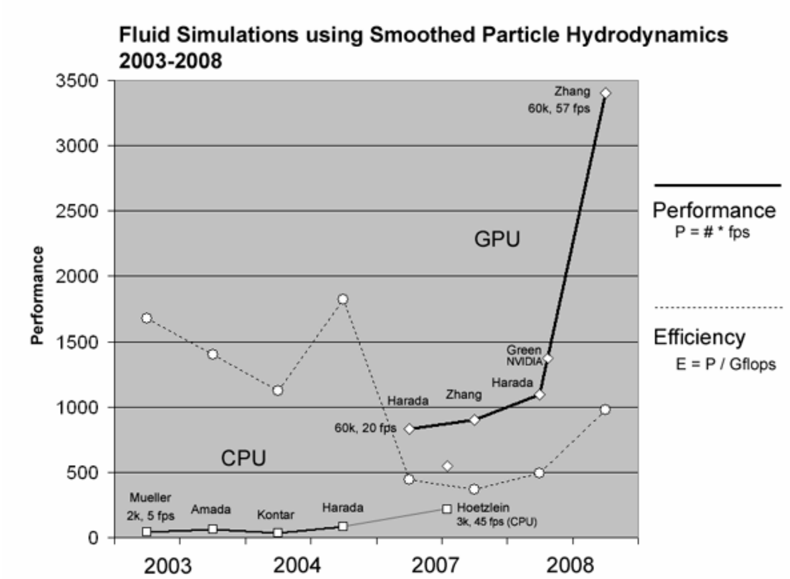


Figure 1.4: GPU performance improvements over CPU(RAMA HOETZLEIN).

1.5.1 Document structure

The first part of the document is an introduction and theoretical view of the method subject of study. This first chapter was an introduction and presentation of the work realized in this project. Then on chapter 2, a review of the cellular automata is made as an introduction to the Lattice Boltzmann Method. Chapter 3 is a description of the Lattice Boltzmann Method and some others aspects particular to this project and the method itself.

The second part of the document consist in the computational aspects of the work, which are GPU computing and implementation of the method covered in chapter 4, and visualization and interactivity covered in chapter 5.

Finally the numerical results and benchmarks are described in chapter 6, and conclusions and future work on chapter 7.

Chapter 2

Lattice gas automata

2.1 Background

A cellular automaton (CA) is a discrete model studied in mathematics, physics, microstructure modeling and other sciences. It consists of a regular grid of finite cells, each one has a finite number of states, such as 1 and 0. For each cell, a set of cells (usually including the cell itself), known as neighborhood is defined relative to the specified cell.
An initial state $t = 0$ is selected by assigning a state for each cell. A new generation is created after setting $t = t + \delta t$, according to some fixed rule, usually, a mathematical function [wik10a]. This determines the new state of each cell in terms of the current state of the cell and the states of the cells in its neighborhood.

2.2 Lattice gas automata

The lattice gas automata (LGA) also know as the lattice gas cellular automata (LGCA) is a cellular automata known as the precursor of the Lattice Boltzmann Method [Suc01]. Here with the purpose of introducing the Lattice Boltzmann Method is presented a small review of the LGA methods.

The purpose of LGAs is to simulate the behavior and interaction of many single particles in an ideal gas [Vig09]. Unfortunately these are based on boolean mathematics which are unfamiliar for most people.

A cellular automata consist of a lattice geometry where the intersection points can take a finite number states. The model evolves in discrete time steps. The state of each corner is determined by its own state and the state of its neighbors at the previous time step. A gas is modeled as set of spheres that travel over the lattice. The collision between spheres is modeled by a set of elastic collision rules that ensure conservation of mass m and momentum $\vec{p}$. The LGA, as molecular dynamics, works on a microscopic (lattice) level, but macroscopic quantities as density ρ and velocity u can be recovered from this level making possible the study of fluids at a macroscopic level.

2.3 The HPP model

The HPP model was proposed by Hardy, Pomeau, and de Pazzisin in two landmark publications in 1973 [HPdP73a, HPdP73b]. In the HPP model the lattice is squared so each node has four neighbors. The particles can have four possible velocities determined by $e_i = \sin[\frac{\pi}{2}(i-1)]i + \cos[\frac{\pi}{2}(i-1)]j$ with $i = 1..4$ as seen on figure 2.1.

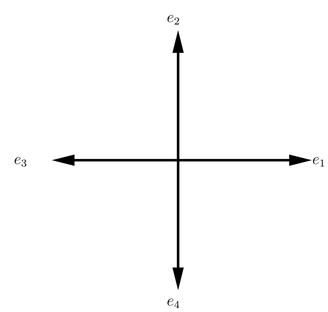


Figure 2.1: HPP lattice scheme

Each time step the particles are moved towards the e_i direction. When two or more particles arrive to the same node during a time step a collision occurs. To meet the conservation of mass and momentum principles, the number of particles and the total velocity must be the same before and after the collision. When two particles collide they bounce into a direction perpendicular to their original direction as seen on figure 2.2. This preserves momentum as the sum of the velocities of the two particles is zero in both configurations. When three of four particles arrive to a same

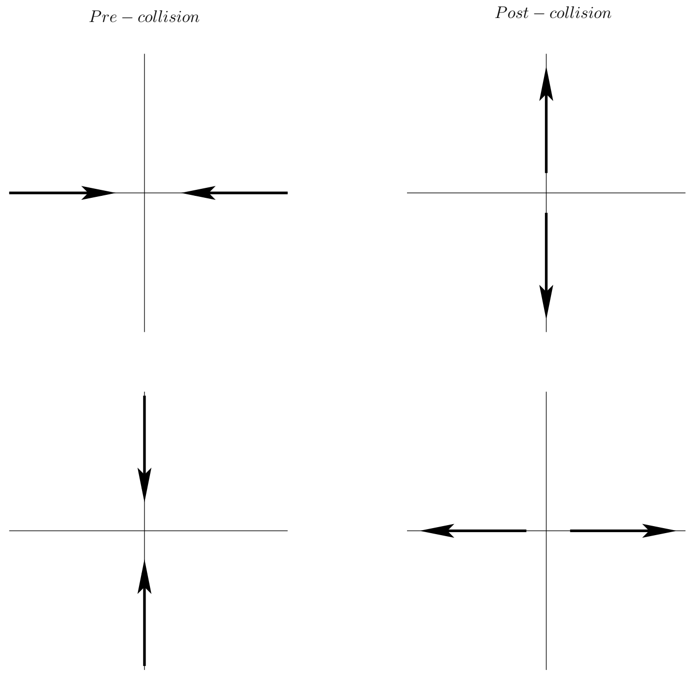


Figure 2.2: HPP collision rules

node the only configuration that guarantees conservation of mass and momentum is the same configuration before the collision therefore the state is not changed (and cannot be changed). As a result in the method behaves as there was no collision. The collisions are deterministic, so each collision has one and only one possible result. For this reason, this model has a property called *time reversal invariance*, that means that the model can be run in reverse to recover any earlier state.

The numerical density can be easily calculated, as the sum of all particle on a node at any given time.

$$\rho = \sum_i n_i \tag{2.1}$$

Where n_i is the boolean occupation number $(0$ or $1)$ expressing the number of particles at a node with velocity e_i. In this same way the velocity u can be easily calculated.

$$u = \frac{\sum_i e_i n_i}{\rho} \tag{2.2}$$

From Equation 2.2 the total momentum at each node can be deduced using $\vec{p} = \rho u$.

2.3.1 Advantages and disadvantages

The HPP has some notable advantages and weakness. One big advantage is the boolean nature of the model, which makes its implementation using *short ints* extremely fast.

Another advantage is the inherit parallel nature of the model, as each node can be computed independently only using the information of the particles.
The *time reversal invariance* feature is one of the greatest advantages as not many CFD models let you "travel" back in time.

Although this good advantages the HPP has one mayor flaw: it fails to achieve rotational invariance. This means that vortices using the method would look squared [Bui97], because of this the method was abandoned for simulation of fluid flows. Many other advantages and weaknesses exist in the method, but they are out of the scope of this review.

2.4 The FHP model

In 1986 Frisch, Hasslacher and Pomeau introduced the FHP model, moving from a square lattice to a hexagonal lattice, this added rotational invariance to the model which leads to the proper recovery of the Navier–Stokes equations[FHP86].

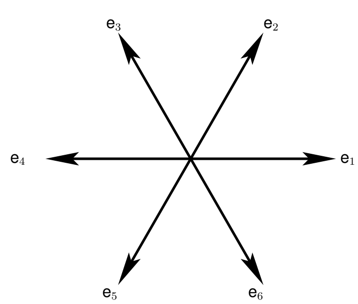


Figure 2.3: FHP hexagonal grid

The lattice velocity vectors are defined by $e_i = \cos[\frac{\pi}{3} - \frac{\pi}{6}]i + \sin[\frac{\pi}{3} - \frac{\pi}{6}]j$ as shown in figure 2.3. The hexagonal grid allows that for a heads–on collision there is two probable paths random chosen for the bounce as seen on figure 2.4, this conserves mass and momentum in the same way as expressed in section 2.3, but the introduction of a probability makes the model *stochastic*, this fact makes impossible the *time reversal invariance*.

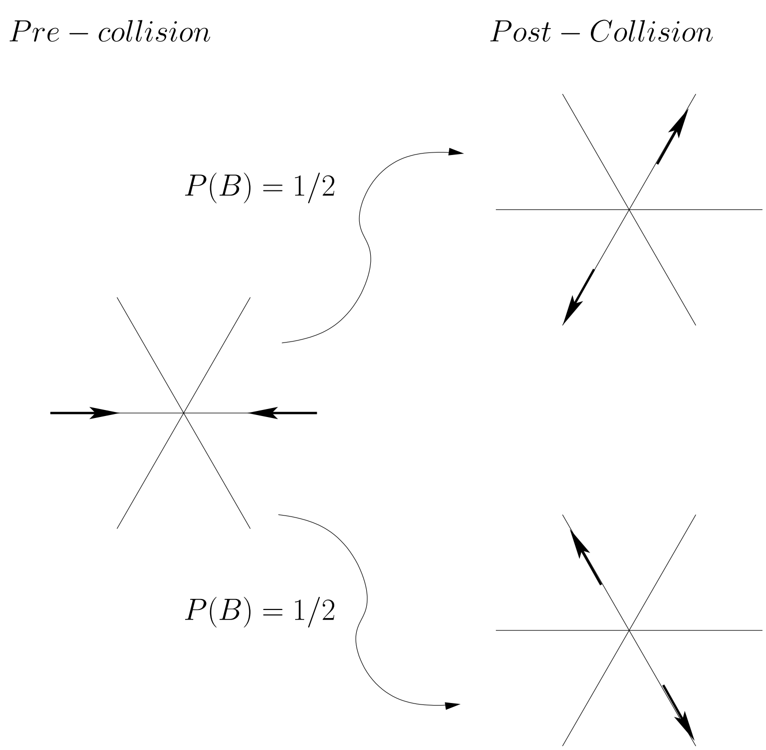


Figure 2.4: FHP collision probability

The FHP model also has a resolution for a three–particle collision this means that collision bounces back each particle back the way it came. This is shown in Figure 2.5, for all other collisions of four or more particles, no change should be computed[FHP86]. There are too many collisions allowed by this model, and the way to calculate can be found in [Bui97].

Many variations to the FHP model exist, these introduce new collision rules and resting particles. Interested readers should check into [Bui97] for a greater coverage of the different models of cellular automata.

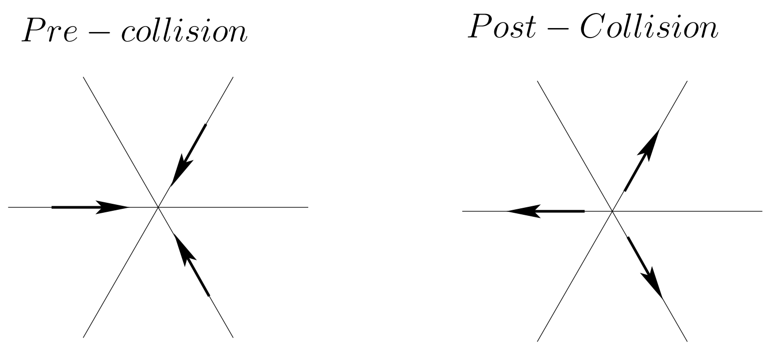


Figure 2.5: FHP three particle head on collision rule

2.4.1 Advantages and disadvantages

The FHP model is essentially the HPP model with a change of the lattice geometry and extended collision rules set, therefore in inherits most of the advantages that HPP has.

Due to the stochastic nature of the model the *time reversal invariance* is lost.

One big advantage over HPP is the introduction of the rotational invariance that permits to recover the incompressible Navier–Stokes equations.

One mayor disadvantage is the hexagonal mesh, such mesh elements are very computer–unfriendly[ST05], and hexagonal meshers are scarce, reason why the model did not take–off in the CFD world.

Chapter 3

Lattice Boltzmann Method

In 1998 McNamara and Zanetti proposed a fix to boolean nature of the Lattice Gas Automata[MZ88]. They removed the Boolean occupation number n_i with an ensemble average

$$f_i = \langle n_i \rangle$$

so f_i is a number between 0 and 1. With this modification the evolution equation of a lattice gas automata became the *lattice boltzmann evolution equation*.

$$f(\vec{x} + \vec{e}, t + 1) - f(\vec{x}, t) = \Omega(\vec{x}, t) \tag{3.1}$$

McNamara and Zanetti proposed a way to simulate the ensemble average directly in the numerical method instead of it being a theoretical quantity in an analytic derivation to prove the method's conformance to the Navier–Stokes equation. The Equation 3.1 represents the main equation of study for the Lattice Boltzmann Methods.

Since the method is no longer keeping track of single particles, it is are not longer in the microscopic scale. It has moved to a *mesoscopic* scale, which means that one is now tracking averaged packets of particles.

It is mentioned here that from now on, vectors $\vec{x}, \vec{p}$ and $\vec{e}$ are going to be just x, v, e, for the sake of notation clarity.

3.1 Boltzmann equation

A familiar equation in the world of *statistical mechanics* the Boltzmann equation represents the particle density in the position range $x + dx$ and the momentum range $p + dp$[1] at time t

$$f(x, p, t)dxdp$$

If there are no collisions, then at time $t + dt$ the position of particles starting at x is going to be $x + dx$ and the new momentum is going to be $p + dp$. This allows to calculate the momenta and positions of particles for $t + dt$ if we know these values at time t:

$$f(x + dx, p + dp, t + dt)dxdp = f(x, p, t)dxdp \tag{3.2}$$

The Equation 3.2 is know in the Lattice Boltzmann Method world as the *streaming step*.

However there are collisions that result in particles at (x, p) not arriving at $(x + dx, p + dp)$ and particles that arriving at $(x + dx, p + dp)$ but did not come from (x, p). It is set $\Gamma^- dxdpdt$ equal to the number of molecules that do not arrive to the expected phase space due to collisions during time dt. In the same way $\Gamma^+ dxdpdt$ is set to the number of particles that start somewhere different to (x, p) and end at $(x+dx, p+dp)$. If this notion is added to equation 3.2 the following term is obtained:

$$f(x + dx, p + dp, t + dt)dxdp = f(x, p, t)dxdp + [\Gamma^+ - \Gamma^-]dxdpdt \tag{3.3}$$

The first order terms of a Taylor series expansion of the left hand side of the Equation 3.3 give the Boltzmann equation:

$$u \cdot \nabla_x f + F \cdot \nabla_p f + \frac{\partial f}{\partial t} = \Gamma^+ - \Gamma^- \tag{3.4}$$

Where ∇_x is $(\frac{\partial}{\partial x}, \frac{\partial}{\partial x_\beta}, ...)$, ∇_x is $(\frac{\partial}{\partial p}, \frac{\partial}{\partial p_\beta}, ...)$, F is an external force and f is the distribution function. It is to note that although Equation 3.4 was derived using an ideal gas as reference it can be derived for an arbitrary chemical component [ST05].

In its complete form Equation 3.4 is a nonlinear integral differential equation and

[1] In statistical mechanics, $(\vec{x}, \vec{p})$ are said to be coordinates in phase space [Vig09]

it is particularly complicated to solve. With the Lattice Botlzmann methods an approximation to the Equation 3.3 is found with the particle perspective. This solution contains the *collide* and *stream* notions, central to the Lattice Boltzmann Method and avoid one to solve the rather complicated Equation 3.4.

3.2 LBM framework

The lattice Boltzmann models simplify Boltzmann's original concept by reducing the number of possible particle spatial positions and momenta from a continuum to a few possible results, and similarly the time is discretized to *time steps*. Particle positions are now confined to the nodes on the lattice, variations of momenta are reduced to 8 directions and 3 magnitudes, mass is reduced to a single particle mass. Figure 3.1 shows the cartesian lattice and velocities e_i where $a = 0, ..., 8$, e_0 represents particles at rest.

Not every lattice is appropriate for the Lattice Boltzmann Method. There are several conditions that have to be met to give the lattice sufficient isotropic behavior and allow a full recovery of the Navier–Stokes Equation [Suc01]. Some schemes have gained popularity due to easy implementation, some of these are shown in Figure 3.2 for the 2D and 3D cases.

Figure 3.3 shows the model known as the D2Q9. It is 2 dimensional (D2) and contains 9 velocities (Q9). This model has became the *defacto* standard for $2D$ domains. Since this work comprehends a $2D$ domain the D2Q9 lattice was selected and it is used throughout this document. From now on when referring to a lattice, it is understood as the model mentioned above.

To comply with the isotropy condition, the 2DQ9 model must have the magnitude of the vector e_i and set of weights ω_i for each velocity. The velocity magnitude is given by

$$e_i = \begin{cases} 0 & \text{for } i = 0 \\ 1 & \text{for } i = 1,2,3,4 \\ \sqrt{2} & \text{for } i = 5,6,7,8 \end{cases}$$

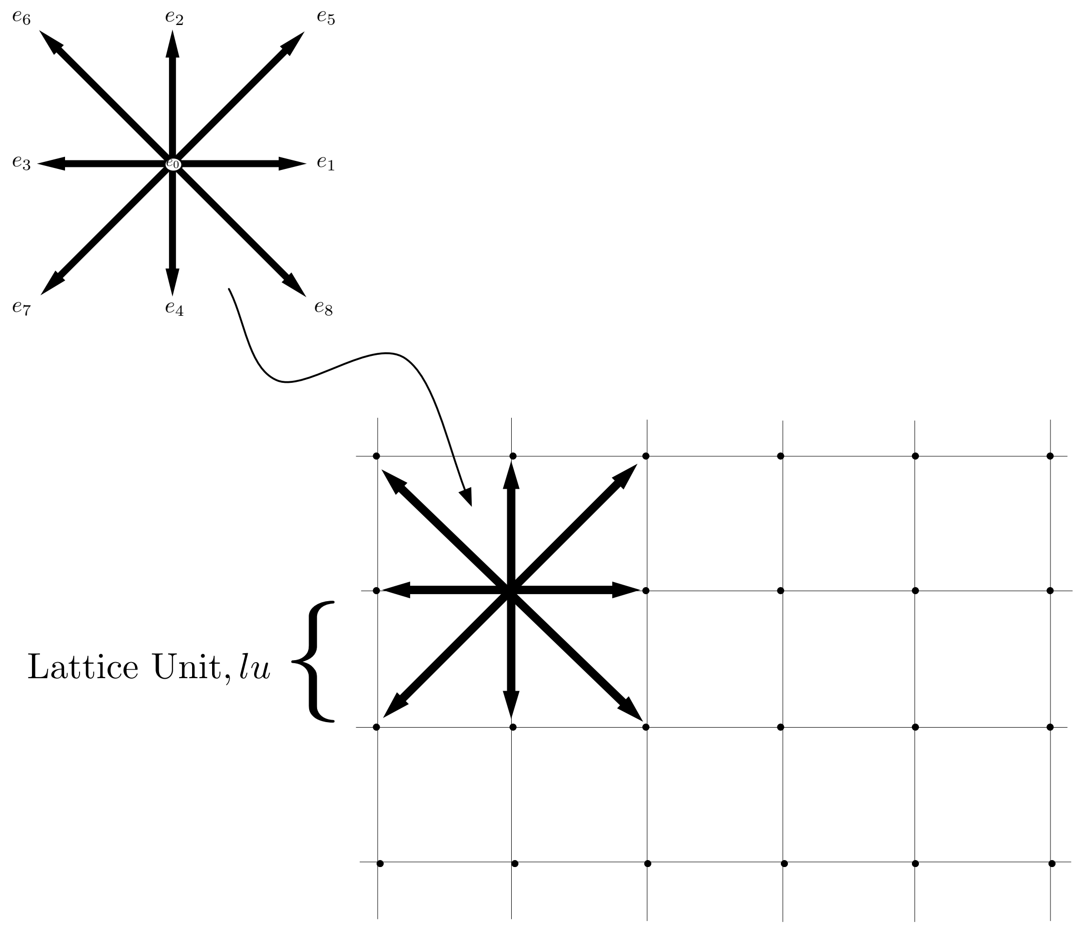


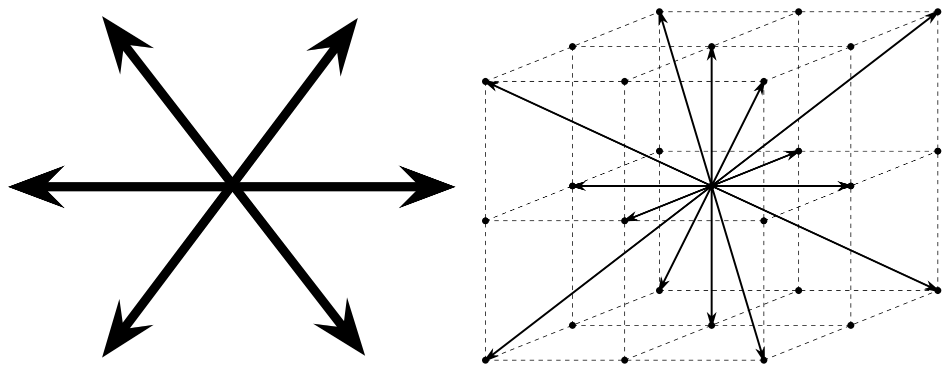
Figure 3.1: LBM cartesian grid and vectors of the D2Q9 lattice.

and the weights corresponding to the model implemented in this work are

$$\omega_i = \begin{cases} 4/9 & \text{for } i = 0 \\ 1/9 & \text{for } i = 1, 2, 3, 4 \\ 1/36 & \text{for } i = 5, 6, 7, 8 \end{cases}$$

3.2.1 Macroscopic variables

Since the method uses *statistical mechanics* which is a great mathematical tool for dealing with large populations, of particles in this case, the frequencies can be considered as direction specific fluid densities [ST05]. This the fluid density can be



(a) D2Q7 hexagonal lattice (b) D3Q15 lattice

Figure 3.2: Two common lattice schemes for the Lattice Boltzmann Method

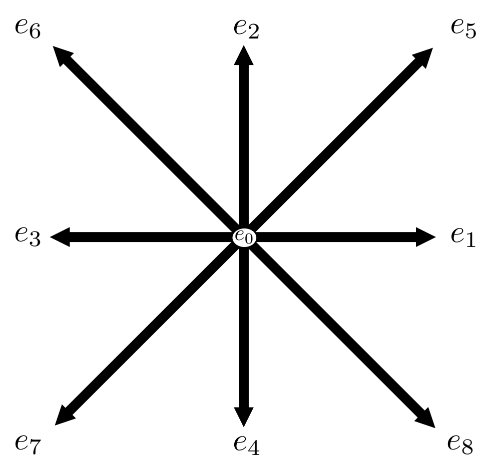


Figure 3.3: Vectors of the D2Q9 lattice.

calculated using the following relationship:

$$\rho = \sum_{i=0}^{8} f_i \tag{3.5}$$

and the macroscopic velocity u, is an average of the discrete velocities e_i, weighted by the directional densities:

$$u = \frac{1}{\rho} \sum_{i=0}^{8} f_i e_i \tag{3.6}$$

These two simple equation allows us to jump from the discrete microscopic velocities of the method to a continuum of macroscopic velocities that represent the fluid's motion.

3.2.2 The equilibrium distribution function

The equilibrium distribution is used to simulate the collision between fluid particles. It can be derived from the Maxwell–Boltzmann velocity distribution from statistical mechanics:

$$f_i^{eq}(x) = \omega_i \rho(x) \left[1 + 3\frac{e_i \cdot u}{c^2} + \frac{9(e_i \cdot u)^2}{2c^4} - \frac{3u^2}{2c^2}\right] \tag{3.7}$$

It is proven that the equilibrium distribution of Equation 3.7 conserves mass and momentum [Vig09]. Equation 3.7 is not the only equilibrium function available. This function can be modified to simulate different fluids like plasmas, non–newtonian fluids, etc.

3.2.3 The BGK model

So far the collision operator Ω has not been discussed. The collision operator was proposed by Qian, d'Humieres, and Lallemand as a simplified collision operator similar to the one proposed for the Boltzmann equation by Bhatnagar, Gross, and Krook in 1954 [Vig09]. The BGK collision operator is given by

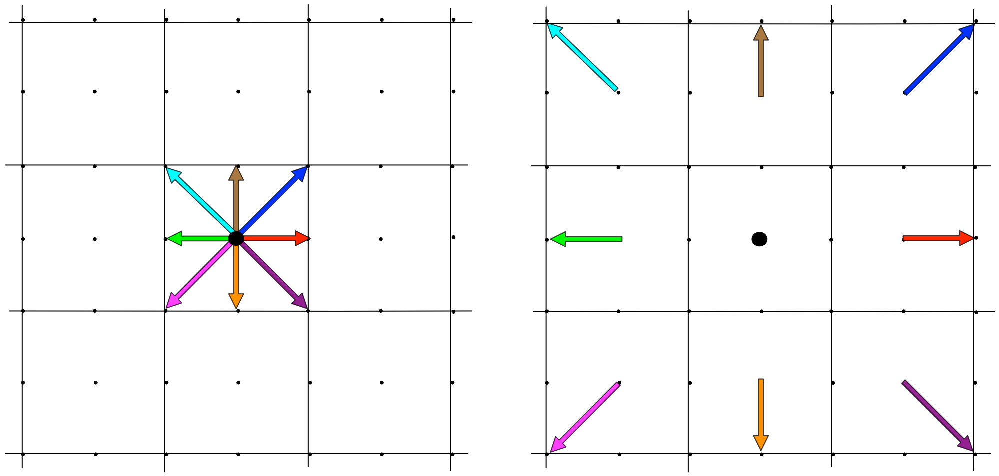
$$\Omega_i = -\frac{1}{\tau}\left[f_i - f_i^{eq}\right] \tag{3.8}$$

where τ is a free parameter known as the *relaxation time*, and f_i^{eq} is the equilibrium distribution function of particles. The collision operator represents the relaxation time of the distribution function f_i towards the equilibrium f_i^{eq}. It is proven that this collision operator conserves mass and momentum [Suc01].

If we replace the BGK collision operator in Equation 3.1 the Lattice Boltzmann BGK model is obtained

$$f_i(x + e_i, t + 1) = \left(1 - \frac{1}{\tau}\right) f_i(x, t) + \frac{1}{\tau} f_i^{eq}(x, t) \tag{3.9}$$

It has to be mentioned that the value of τ, – the time relaxation parameter –, cannot be chosen arbitrary because as $\tau \to 0.5$ numerical instabilities arise in the method [ST05].



(a) Discrete direction–specific densities for time t (b) Discrete direction–specific densities for time $t + dt$

Figure 3.4: LBM streaming step

3.2.4 Streaming

In the streaming step, the direction–specific densities are moved one grid–node away in the direction they are pointing. This process can be seen on figure 3.4.

3.2.5 Numerical kinematic viscosity

For the BGK D2Q9 model the numerical viscosity, which is not related to the physical viscosity, but to the relaxation parameter, is given by:

$$\nu_{lb} = \frac{1}{3}\left(\tau - \frac{1}{2}\right) \tag{3.10}$$

Note that as mentioned on section 3.2.3 numerical instabilities arise when τ approaches $1/2$. A value of $\tau = 1$ is the safest [ST05] to keep the method numerical stable.

3.3 Boundary conditions

The Lattice Boltzmann Method has a rich set of boundary conditions, for many different situations. All these boundary conditions vary in stability and convergence.

In this work we are going to use periodic boundaries, bounce–back, and Zou/He for the flux and pressure boundary conditions. These boundary schemes are simple to implement and they are stable [LCM+08] in the range of the work comprehended here.

3.3.1 Periodic

With the periodic boundary conditions the system is closed by the edges and they are treated as they are attached to the the opposite edges of the domain. These boundaries as useful to simulate infinite domains, or finite repetitive (periodic) domains.

The boundary is easy implemented, the direction–specific densities pointing to outside of the domain are *streamed* to the corresponding directions in the other side of the domain as seen on Figure 3.5.

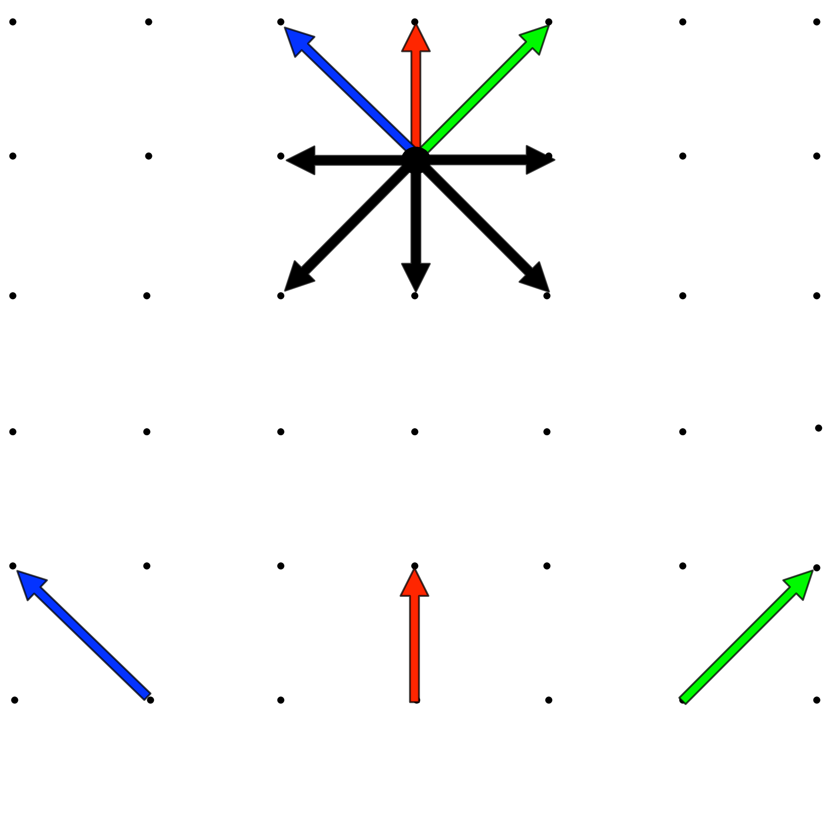


Figure 3.5: Stream step for a periodic boundary

3.3.2 Bounce–back

As mentioned above the bounce–back boundaries are particularly simple and they have played a mayor role in making the Lattice Boltzmann Method popular for modeling complex geometries. The easiness of this boundary is that you just have to set a particular node as a solid boundary and nothing else has to be done. For example the a node marked with a 1 means that is a fluid and a node marked with a 0 is considered a solid boundary. No special treatment is necessary direction–wise or whatsoever. This makes the programming rather simple and makes the method suitable to complex geometries such as the found in porous media.
Here it is mentioned that many bounce–back schemes exists, but we are going to concentrate in the mid–plane full bounce–back. In this mode the solid wall (dark colored on the image) is between two lattice nodes as seen Figure 3.6.

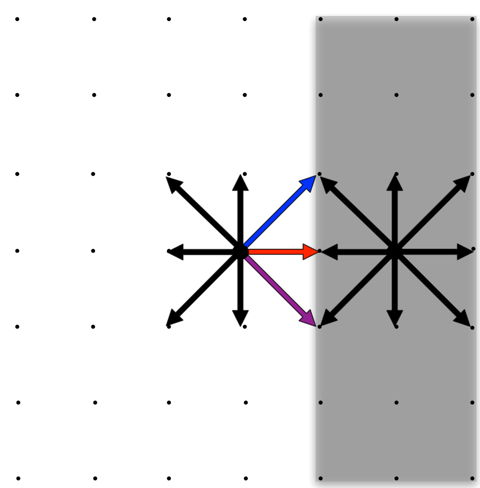


Figure 3.6: Pre–Stream step for a bounce–back boundary for a time t

After the direction–specific densities are streamed they are “absorbed” by they solid temporarily as outlined on Figure 3.7. Then the direction–specific densities are reflected to the opposite direction. This process is shown in Figure 3.8. Finally in the next time step the direction–specific densities are streamed back into the fluid domain as seen in Figure 3.9.

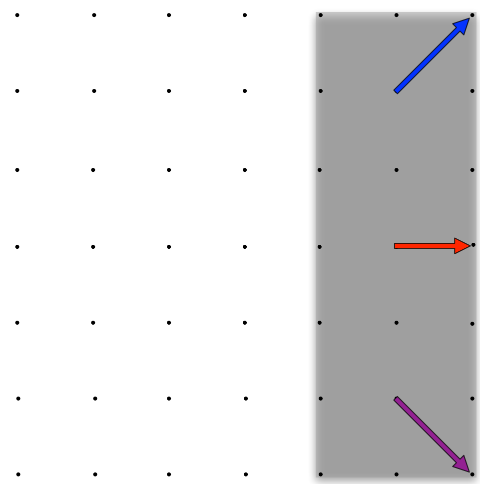


Figure 3.7: Post–Stream step for a bounce–back boundary for a time t

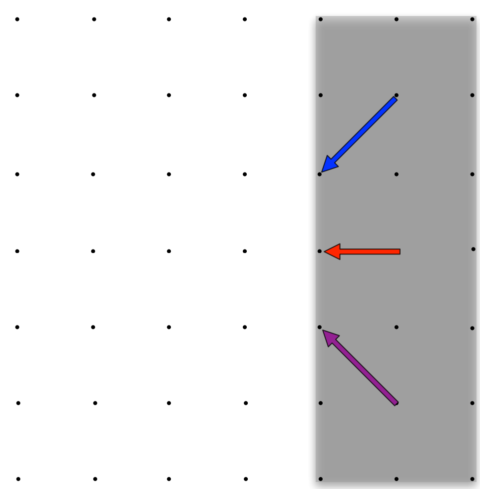


Figure 3.8: Bounce–back of direction–specific densities

3.3.3 Von Neumann

Von Neumann or flux boundary conditions set the flow speed at a boundaries. A velocity vector (both components) are specified at the node from which density/pressure is computed.

Not just the density/pressure has to be computed, also unknown direction–specific densities appear at the boundaries and these have to be calculated properly to conserve mass and momentum. Lets consider the inlet of a channel, at the first node on the east side, after the streaming step, some direction–specific densities are un-

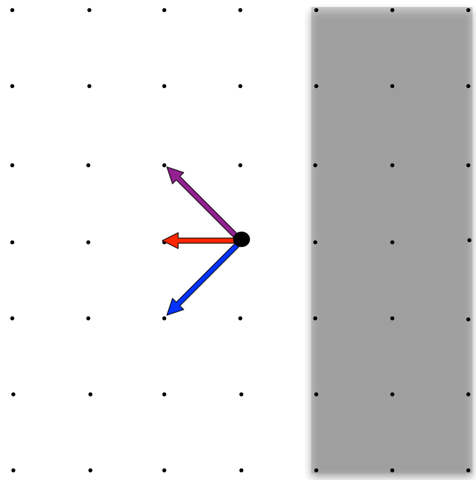


Figure 3.9: Post–Stream step for a bounce–back boundary for a time $t + dt$

known (shown dashed) as seen on Figure 3.10.

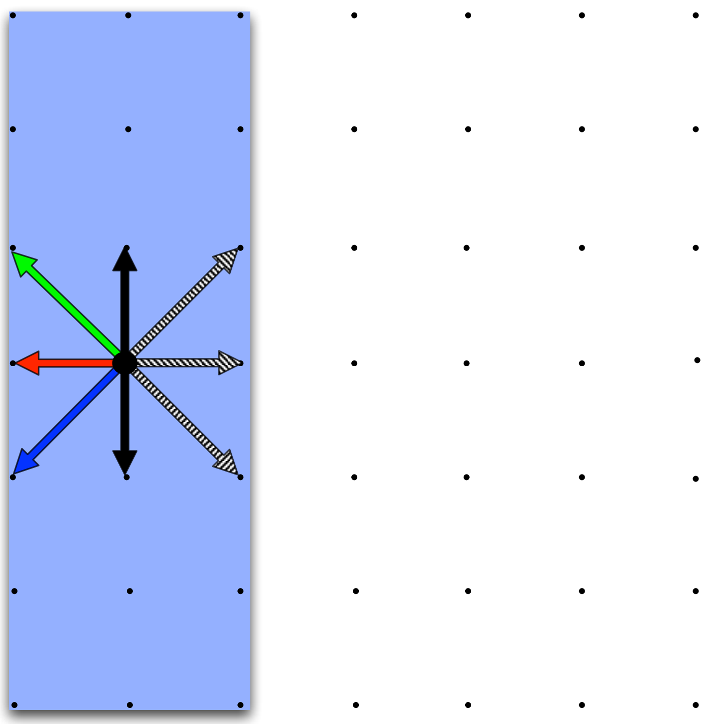


Figure 3.10: Post–Stream boundary node showing unknown direction–specific densities

Using the know velocity set at the boundary and using Equations 3.5, 3.6 and 3.7 and assuming that direction–specific densities parallel to the boundary are equal to

0, one can find the unknown direction–specific densities.

Let's suppose that one want set a von Neumann boundary condition on the North lid. According to Figure 3.3, the direction–specific densities that have to be solved are f_7, f_4 and f_8. The condition is a vertical velocity towards the south

$$u_0 = [0, v_0] \tag{3.11}$$

Using Equation 3.6, one gets two equations, one for each component of the velocity vector

$$0 = f_1 - f_3 + f_5 - f_6 - f_7 + f_8 \tag{3.12}$$

and

$$\rho v_0 = f_2 - f_4 + f_5 + f_6 - f_7 - f_8 \tag{3.13}$$

As proposed by [ZH97], one can assume that the bounce–back is preserved in the direction normal to the boundary, so

$$f_2 - f_2^{eq} = f_4 - f_4^{eq} \tag{3.14}$$

This is a system of four equations and four unknowns, and it can be solved as follows. Equations 3.5 and 3.13 have the unknown direction–specific densities f_7, f_4 and f_8, so they can be rewritten with those variables on the left hand side.

$$f_4 + f_7 + f_8 = \rho - f_0 - f_1 - f_2 - f_3 - f_5 - f_6 \tag{3.15}$$

$$f_4 + f_7 + f_8 = f_2 + f_5 + f_6 - \rho v_0 \tag{3.16}$$

They the right hand sides are equated

$$\rho - f_0 - f_1 - f_2 - f_3 - f_4 - f_5 - f_6 = f_2 + f_5 + f_6 - \rho v_0 \tag{3.17}$$

and solve for ρ

$$\rho = \frac{f_0 + f_1 + f_3 + 2(f_2 + f_5 + f_6)}{1 + v_0} \tag{3.18}$$

Now, from Equation 3.14 one can solve for f_4. As this equation contains the equilibrium terms. Direction–specific densities of Equation 3.14 have to be substituted on Equation 3.7, yielding

$$f_4 = f_2 - \frac{2}{3}\rho v_0 \tag{3.19}$$

Substituting Equations 3.12 and 3.19 into the Equation 3.13 it can be solved for f_7. Equation 3.19 is used to replace f_4 and Equation 3.12 is used to replace f_8.

$$f_7 = f_5 + \frac{1}{2}(f_1 - f_3) - \frac{1}{6}\rho v_0 \tag{3.20}$$

to solve for f_8, the last step can be repeated, except that Equation 3.12 is nows used to substitute for f_7

$$f_8 = f_6 - \frac{1}{2}(f_1 - f_3) - \frac{1}{6}\rho v_0 \tag{3.21}$$

Now all the direction–specific densities are known, and the system configured properly to start a simulation.

3.3.4 Dirichlet

These boundary conditions constrain the pressure/density at the boundaries of the domain. The solution of these boundaries is very similar to the von Neumann ones. A density is specified at the node and with this information the speed is computed, and the remaining unknown direction–specific densities. Note that specifying density is equivalent to specifying pressure because they are related via an Equation of State of an ideal gas [Suc01]. For the D2Q9 the relationship is given by

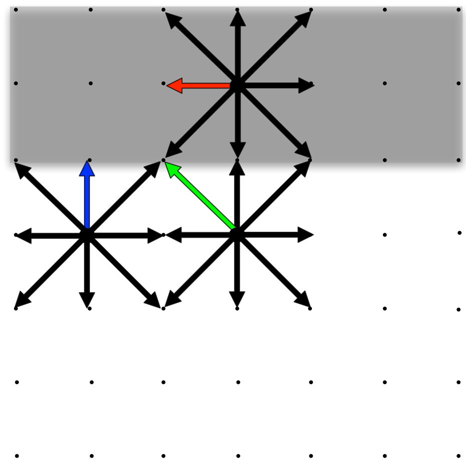
$$p = \frac{\rho}{3} \tag{3.22}$$

The computation of the pressure boundary conditions, is very similar to the flux ones. Now instead of having a velocity, a density is specified at the node and the velocity vector is unknown. With this known density, the same process can be followed to solve for the unknown velocity and unknown direction–specific densities.

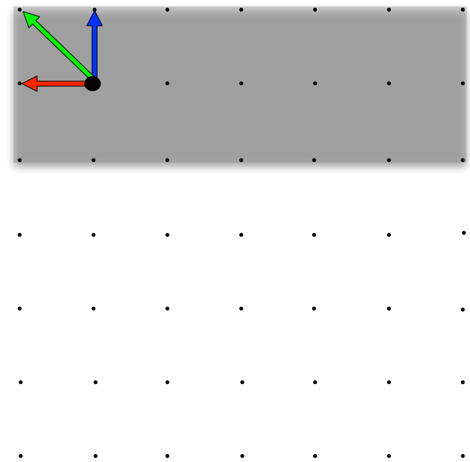
3.3.5 Corner nodes

With the model presented on Sections 3.3.4 and 3.3.3 serious problems arise at the corner nodes.

Let's take for example the North–West node of a 2D channel. After streaming only 3



(a) Pre–streaming of direction–specific densities



(b) Post–streaming of direction–specific densities

Figure 3.11: Corner node problem of boundary condition

direction–specific densities are known as seen on Figure 5.3(b), and we don't have enough data to use Equations 3.5, 3.6 and 3.7 to solve for the unknown densities.

Zou and He proposed a solution for this situation using the off–equilibrium densities. The idea is to *bounce* the know densities as seen on Figure 5.4. After the stream two densities are still unknown (show dashed), but now Equations 3.5, 3.6 and 3.7 can be used to find the two missing densities[ZH97].
It is quite obvious that these two densities after the streaming step do not aport anything to the fluid (they both are pointing towards outside the domain) but their value is needed to properly calculate ρ, and this is going to be needed if we want o satisfy the conservation of mass.

This approach is different for velocity and pressure boundary conditions. It is fully developed for the pressure boundary conditions, for the velocity boundaries, some assumptions have to be made in order to retrieve all the values, this assumption introduces "noise", that make the boundary prone to instabilities [ZH97].

3.3.6 Moving boundary

One of the novelty of this work is the ability to move objects inside the fluid interactively. Moving objects inside the fluid is a complex task since is prone to break the continuity of the model, this problem is solved by computing the new direction–

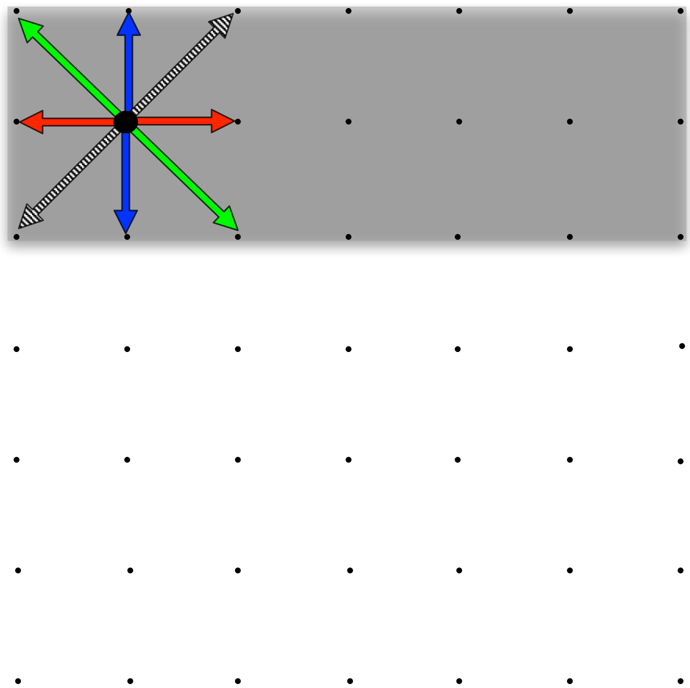


Figure 3.12: Post–Stream boundary node showing bounced direction–specific densities

specific densities each time the object is moved. Also a new moment has to be computed, because the movement of the object adds a momentum to the particles towards the direction it is moved [Lad94].

$$f'_i = f_i - 2\rho\frac{w_i(\vec{c}_i \cdot \vec{u})}{c_s^2} \tag{3.23}$$

Where f'_i are the bounced direction–specific densities f_i, and $\vec{u}$ is the velocity of the moving obstacle inside the fluid. One key issue emerges of "when" to move the obstacle. In this work, the motion of the object is done based on the speed of the object. With this speed, it can be calculated the number of iterations needed to achieve one dx, and since this is a discrete system, the object will not be moved until all the iterations needed to move the object one dx have been calculated.

3.4 Units conversion

Lattice Boltzmann Method simulations represent the physics of actual macroscopic systems, but the method works at a microscopic level. This introduces the need to convert physical macroscopic units to microscopic units or *lattice units*.

There are two approaches to this conversion: the first one consist in converting from physical units straight to lattice units, and a more popular one that uses and intermediate dimensionless system to do the conversion.
There are several reasons why taking a intermediate step is preferred. Discrete variables chosen, using this path, are directly related to important numerical parameters of the method, which have an impact on the accuracy and the stability of a simulation of the system [Lat08]. Also one may be in a situation in which there is no physical system to refer, like when you are benchmarking an article.

For these reasons more attention is going to be put into the second approach and a full developed example is shown in Appendix A.

3.4.1 Direct conversion

In this method the lattice units are related to physical units through the time step Δt and the node spacing Δx. The subscript is used to identify physical units is $_p$ and $_{lb}$ to identify Lattice Boltzmann Method units. The methodology described here is presented in Sukop and Thorne's book [ST05].

The time step is given by the equation

$$\Delta t = \frac{\nu_p}{c_{s,p}^2(\tau - 1/2)} \tag{3.24}$$

Where ν is the kinematic viscosity, $c_{s,p}$ is the speed of the sound, and τ the only free parameter is the relaxation time.
In the same way the space step is given by the equation:

$$\Delta x = \frac{\nu_p}{c_{s,lb} c_{s,p}(\tau - 1/2)} \tag{3.25}$$

The only missing parameter is $c_{s,lb}$ which can be found with the relation

$$c_{s,p} = c_{s,lb}\frac{\Delta x}{\Delta t} \tag{3.26}$$

The Equation 3.26 can be used to convert any desired speed from physical units to lattice units.

Similary for the pressure the resulting equation is

$$p_p = p_{0,p} \frac{\rho_{lb}}{\rho_{0,lb}} \tag{3.27}$$

Where p_0 is the reference pressure and $\rho_{0,lb}$ is the reference density. One can use the Equation 3.22 to relate the density and pressure to find the unknown parameters of this equation.

3.4.2 Dimensionless conversion

The dimensionless approach is described by Lätt [Lat08]. In this approach a physical system is converted to a dimensionless system denoted by the subscript $_d$, and then converted lattice system. Taking dimensionless path is required for instance when analyzing lattice Boltzmann accuracy [Vig09].

For example one can use characteristic length l_0 and time t_0 and use them as a base, to convert units to a dimensionless system. Take the physical time t_p. This is given in a dimensionless notation by

$$t_d = \frac{t_p}{t_{0,p}} \tag{3.28}$$

and a length l can be described as

$$l_d = \frac{l_p}{l_{0,p}} \tag{3.29}$$

In the same way, a unit conversion is introduced for other variables, based on a dimensional analysis. Take for example the physical velocity u_p:

$$u_p = \frac{l_{0,p}}{t_{0,p}} u_d \tag{3.30}$$

In the dimensionless system, the characteristic length and time of the system are both normalized to 1. Then the dimensionless system is discretized into a grid with Nx nodes used to resolve its characteristic length. $Niter$ time steps are used to resolve the system's characteristic time. Space and time are then divided into intervals δx and δt respectively.

$$\delta x = \frac{1}{Nx} \tag{3.31}$$

$$\delta t = \frac{1}{Niter} \tag{3.32}$$

And with this simple definition other variables, such as velocity and viscosity, are easily converted between dimensionless system and lattice system through dimensional analysis:

$$u_d = \frac{\delta x}{\delta t} u_{lb} \tag{3.33}$$

$$\nu_d = \frac{\delta x^2}{\delta t} \nu_{lb} \tag{3.34}$$

Although this a process looks complicated, it will be clarified by the example described on Appendix A.

3.5 Algorithm summary

A summary of the algorithm implemented in this work is presented here. One can notice the simplicity of the algorithm, which is one of the advantages of the method. Also it is noticeable, that for each time step the same algorithm is applied to each lattice node what makes the method fully data parallel and therefore convenient to a GPU implementation.

Algorithm 1 Lattice Boltzmann Algorithm

while (TRUE) **do**

  Calculate macroscopic variables: $\rightarrow \rho = \sum_{i=0}^{8} f_i$ and $u = \frac{1}{\rho}\sum_{i=0}^{8} f_i e_i$

  Stream: $\rightarrow f(\vec{x}+\vec{e}, t+1) - f(\vec{x}, t) = \Omega(\vec{x}, t)$

  Move obstacle: $\rightarrow f_i' = f_i - 2\rho w_i(\vec{c}_i \cdot \vec{v})/c_s^2$

  Apply boundary conditions

  Collide: $\rightarrow f_i^{eq}(x) = \omega_i \rho(x)\left[1 + \frac{3(e_i \cdot u)}{c^2} + \frac{9(e_i \cdot u)^2}{2c^4} - \frac{3u^2}{2c^2}\right]$

end while

Chapter 4

CUDA implementation

4.1 Introduction

Computers based on a single central processing unit (CPU), saw rapid performance increases and cost reductions for more than two decades. During this performance drive, software developers relied on advances on hardware technology to increase the speed of their applications. This trend, however, has slowed down since 2003, due to the power consumption and overheating issues of CPU cores[KmWH10]. Since then almost all CPUs have switched to multi–core models, where multiple processing units are placed onto a single chip to increase performance keeping power consumption and heat at a lower levels. This switch had a big impact on the software developer community due to the fact, that optimizations on performance where no longer handled by hardware, and had to be addressed by programmers during the coding of the software.

Traditionally software applications where written as sequential programs, these programs will only run on one of the processing cores, and will not achieve performance increases on modern multi–core CPUs. Now programs have to be written as parallel programs. Programs in which multiple threads of execution cooperate to achieve the functionality faster.

The parallel programming is nothing new. The scientific community has been developing parallel programs for decades. These programs run on large scale expensive computers (clusters). Due to cost only few applications where allowed to run on those computers, making parallel software development rare and scarce.

4.2 GPUs as super computers

Since 2003 a kind of multi–core processors called Graphics Processing Units (GPU), have lead the race for computing performance. This performance gap is shown in Figure 4.1

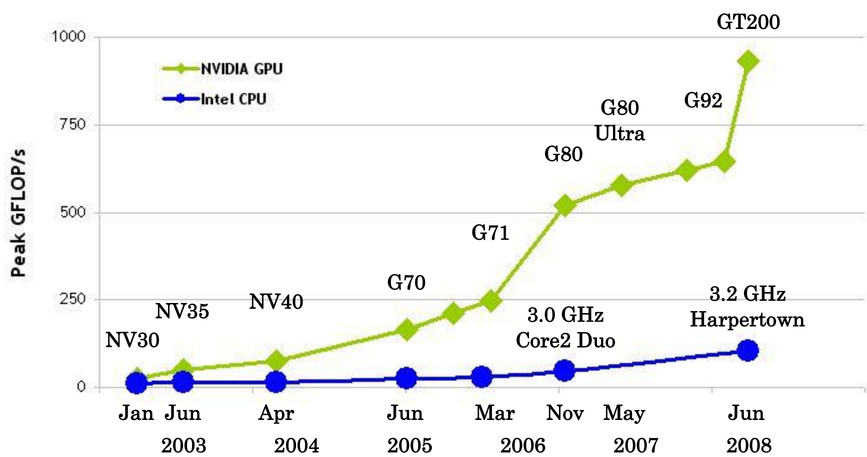


Figure 4.1: Performance gap between GPUs and CPUs, taken from [NVI10]

Such a large performance gap has *motivated* developers to move the computationally intensive parts of a program to GPU for execution. These computationally intensive parts of the program is where the parallel programming focuses, because where more work is done, there is more chances of performance increases due to parallelization techniques.

The reason for such a large performance difference between the GPU and CPU, is that the GPU is specialized for compute–intensive, highly–parallel computations, exactly what graphics rendering is about, and therefore designed in such a way that more transistors are devoted to computing. In the other hand, CPUs are optimized for sequential code and I/O. For these reasons CPUs need more advanced control logic, and cache. This difference can be see on Figure 4.2.

Not only the amounts in transistors makes a difference, memory bandwidth is also an important issue. GPUs have been operating at approximately 10x the bandwidth

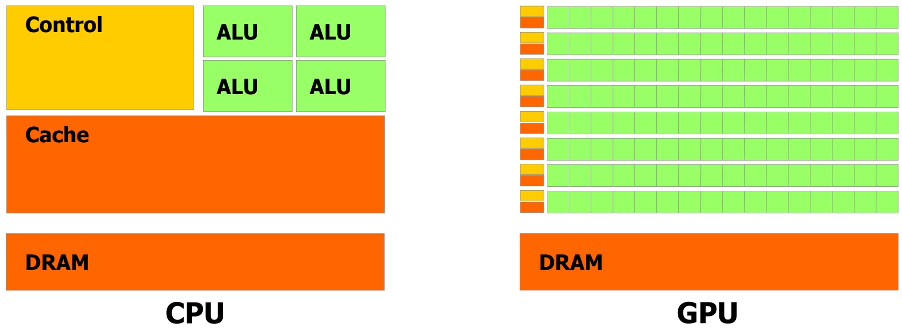


Figure 4.2: Design difference between GPUs and CPUs, taken from [NVI10]

of contemporaneously available CPUs. Recent GPUs chips can achieve speeds of 100GB/s compared to the 20GB/s of modern CPU as seen on Figure 4.3.

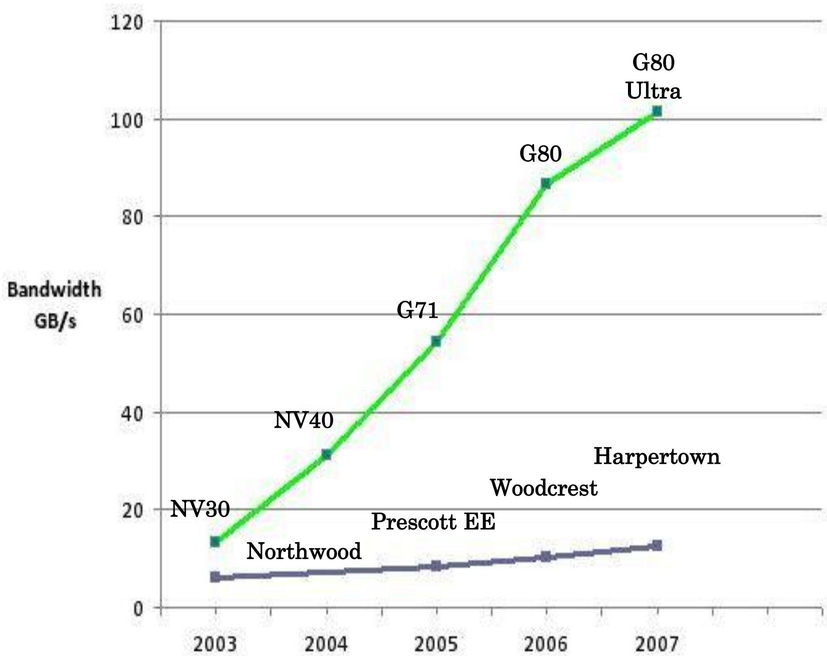


Figure 4.3: Memory bandwidth for the CPU and GPU, taken from [NVI10]

The design philosophy of the GPU is forced by the ever growing video game industry, and its need of to perform a massive number of calculations per video frame in advanced video games.

4.3 Architecture of a GPU

Modern GPUs are organized into sets of highly threaded Streaming Multiprocessors (SMs). A pair of SMs form a building block as seen in Figure 4.4. Each SM has 8 Streaming Processors (SPs). Each SP has a multiply–add (MAD) unit, and an additional multiply (MUL) unit.

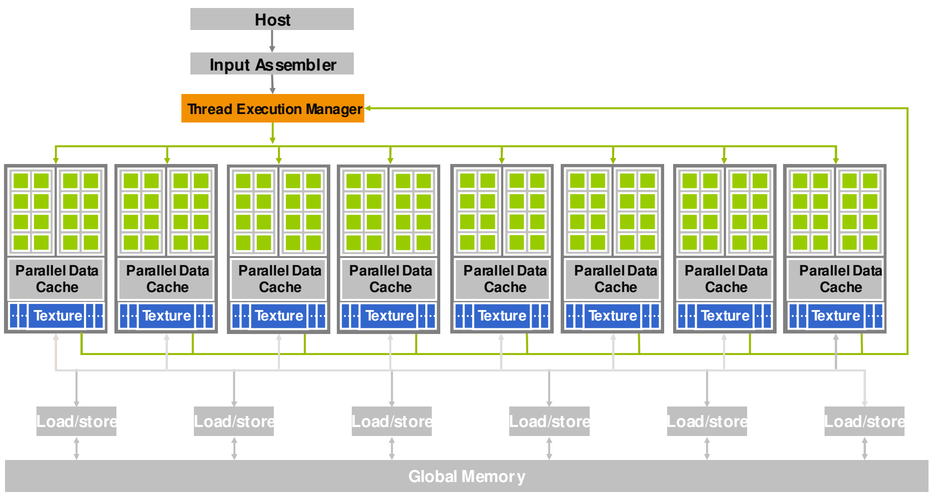


Figure 4.4: Modern GPU architecture, taken from [KmWH10]

This kind of architecture design, comes attached with a programming paradigm. The programming model used by NVIDIA is named SIMT (Single Instruction Multiple Thread) which is akin to the more standard SIMD (Single Instruction Multiple Data). An important difference is that SIMD vector organizations expose the SIMD width to the software, where SIMT instructions specify the execution logic of a single thread. This way, SIMT enables programmers to write thread–level parallel code for independent, scalar threads, as well as data parallel code for cooperative threads.

4.4 The need for speed

One may ask. Why is it needed more speed?. There a many *typical* applications that run fast enough on single or dual core CPUs. But the world of scientific computing can always benefit from more computing power. This enable user to run larger and more complex simulations, more variables could be studied at the same time, results for medical exams can be immediately available, etc.

Also technologies like HDTV or 3DTV will see improvements in definition and features thanks to parallel computer architectures and software. Also the mobil arena, now becoming ubiquitous, it is starting to be powered by parallel architectures, putting parallel applications in the hand of billions people. For this reasons and much more, is needed the careful study of parallel programming and development of parallel application.

4.5 CUDA

NVIDIA's Compute Unified Device Architecture is a parallel computing architecture developed to allow developer to exploit the full processing power of the modern GPUs created by the company. CUDA gives developers access to the native instruction set and memory of the parallel computational elements in CUDA capable GPUs. This provides tools to program GPUs, that resemble the tools for programming CPUs, like compilers, programming languages, debugging tools, etc.

4.5.1 Programming model

To a CUDA programmer the system consists of two main components, the *host* or the CPU and one or more *devices* or the GPUs. In scientific applications there are sections a rich amount of data parallelism, a property where many arithmetic operations can be performed on data structures in simultaneous way.
For these reason CUDA provides a heterogenous architecture, where the serial parts of a program can be run on the host, and the parallel parts can be moved to the device to improve performance as show in Figure 4.4.
Parallel parts of the code are known are *kernels* in CUDA terms. A kernel consists of a grid of threads, that match the SIMT paradigm, that NVIDIA adopted to permit the full usage of data parallelism.

4.5.2 Threads

As seen in Section 4.5.1, launching a CUDA kernel creates a grid threads that all execute the kernel function. This way each, individual thread executes the kernel function when this is launched. Since all threads in grid execute the same kernel

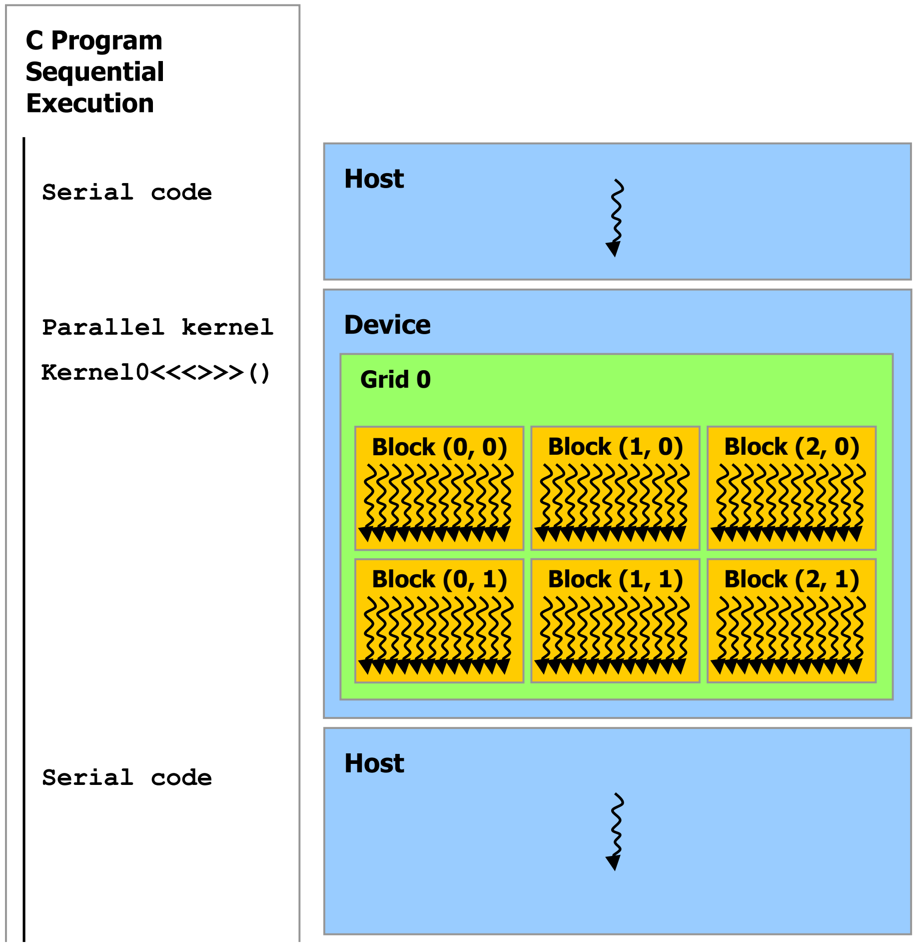


Figure 4.5: Heterogenous programming model, taken from [NVI10]

function, they need an effective way to distinguish from each other. For this reason, NVIDIA has create a unique set of coordinates to solve this problem. The threads are organized in a two–level hierarchy using unique coordinates called blockId and threadId, assigned to them by the CUDA runtime system, as it is shown in Figure 4.6.

The grid size can be any number between 1 and 65536, organized as a one–dimensional or a two–dimensional array[1]. The block size can be between 1 and 512, organized as a one–dimensional, two–dimensional or three–dimensional array. This is organization is intended to match common programming styles and to facilitate a mapping to a 3D physical world.

One big advantage of the Grid–Block–Thread organization, is the transparent

[1]The just released NVIDIA fermi architecture supports three–dimensional grids.

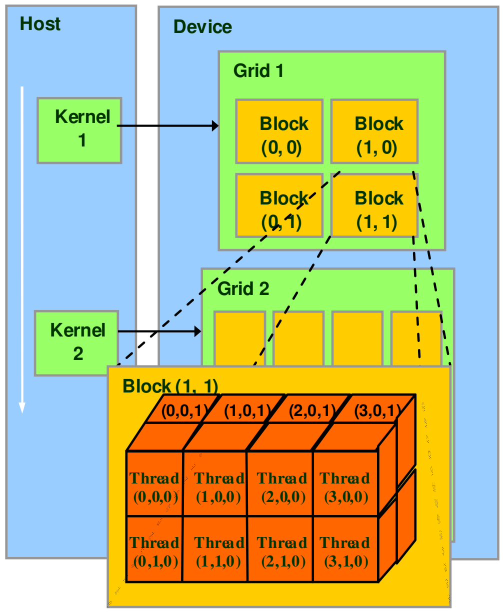


Figure 4.6: CUDA thread organization, taken from [NVI10]

scalability if provides. This is, a program written in CUDA, does not have to be modified to take advantage of more power devices. Figure 4.7 shows how a CUDA program runs faster when using a more powerful device, without any code modification.

4.5.3 Memory layout

On Section 4.5.2 it was shown, that much better performance can be achieved using multiple threads, but not all performance of GPU architecture comes from a multiple–threaded program. Memory management is also an important factor to get better performance.

Memory access to global GPU memory can be a mayor bottleneck, this is due to the fact that global memory is a Dynamic Random Access Memory (DRAM) which tends to have long access latencies. One can easily run into a situation where traffic congestion in the global memory access paths prevent threads from doing any work. For this reason CUDA provides set of memory types that can filter out a majority of

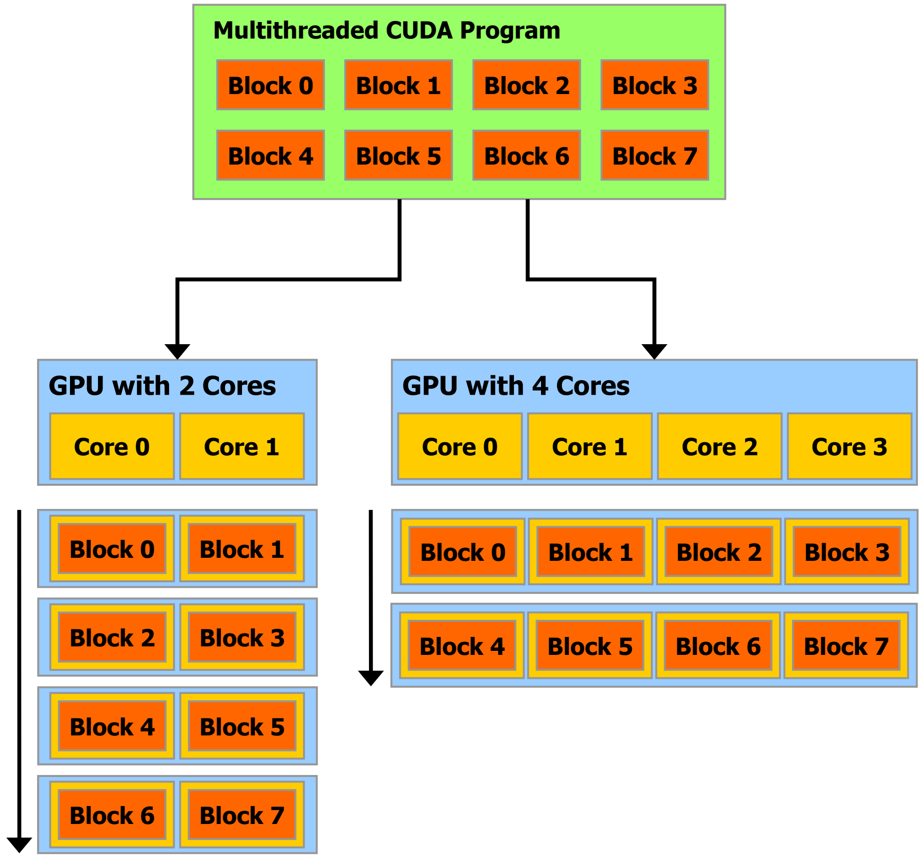


Figure 4.7: Transparent thread scalability, taken from [NVI10]

data requests to the global memory.

Figure 4.8 the CUDA devices memories. At the bottom are located the Global Memory and the Constant Memory, these can be accessed by the host, and are the way to transfer data from the host to device. These memories are large, but tend to be slow. The scope of these memories are the entire kernel, this mean that every thread created by the kernel have access to the this memories.
On top of the figure, one can see Shared Memory and Registers. Shared memory is a very fast kind of memory, which can be seen only by threads in the same block. This memory is very limited 16KB[2], and it is used to reduce access to the global memory, increasing the overall performance of the program. Registers can be accessed by individual threads to store the variables needed to accomplish a task.

[2] Up to 48KB for the new fermi architecture

Textures units are sort of hardware interface, built on top of global memory. They do caching and filtering in the same way as in graphics, and compared to “raw” global memory do not impose rather tight restrictions on access patterns, required for best performance.

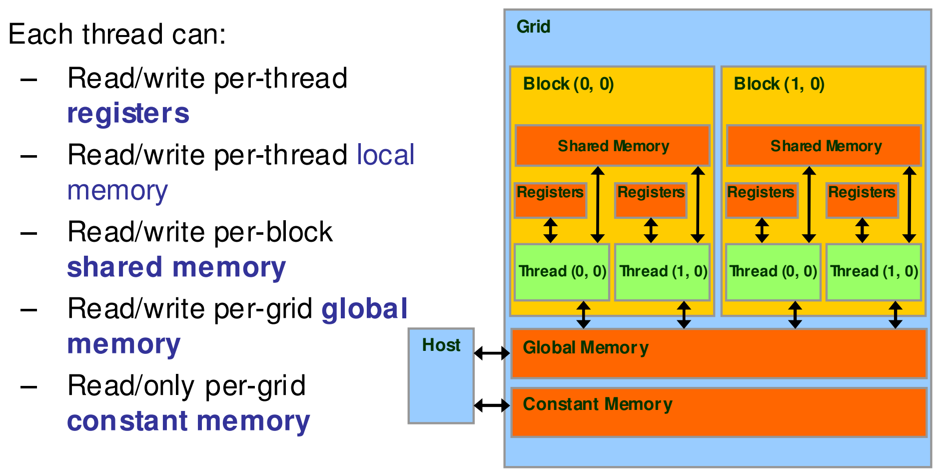


Figure 4.8: CUDA memory hierarchy, taken from [KmWH10]

One important limiting aspect of performance is known as *register pressure*. The pressure comes from the limited number of register available to the GPU, which is 128000. If you take for example the device GeForce 8800 GTX, which has 16 Streaming Multiprocessors, each SM has 8000 registers, and each SM can handle 768 threads as a maximum. In order to achieve the maximal thread capacity, each thread can use only 8000/768 = 10 registers. So if one wants to use more registers (lets say 11 threads for this example), the number of threads will be reduced, and this reduction is done at the block level. For example if each block has 256 threads, the next lower number from 768 is 768-256 = 512 threads. This is a reduction of 1/3 of the active threads working on task.

4.6 LBM on the GPU

It was said on Section 1.5 and 3.5, that one of the reasons why the Lattice Boltzmann Method was chosen for this work, was its parallel *friendliness*. Figure 4.9 shows an outline of the algorithm. The Lattice Boltzmann Method runs the algorithm in each

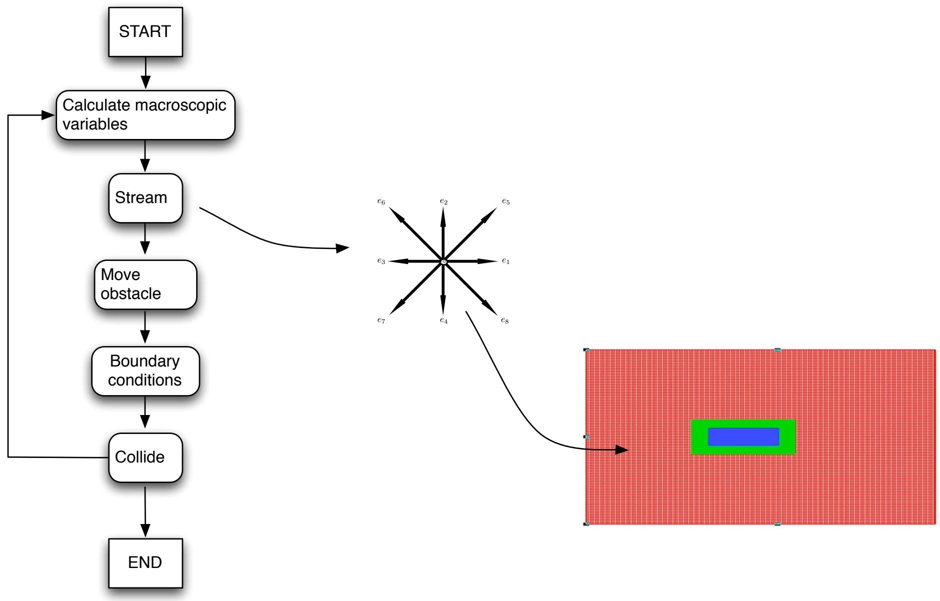


Figure 4.9: LBM runs the same algorithm over each node of grid

node independently, so it does not need to implement a complex communication scheme, which makes the method very parallel. This is a "perfect" match for the SIMD paradigm that is used to program GPUs, and that's why a GPU seems like a logical architecture to implement the method.

In this work, a simple but fast implementation of the Lattice Boltzmann Method was made using CUDA. Figure 4.10 shows the implementation done for this work. It uses a CUDA kernel for each mayor function of the method, this way it is possible to launch a thread for each node of the grid. To increase speed the direction–specific densities were stored in textures for the streaming step. Textures allow very fast fetches and since the streaming process involves many reads, it was clear that performance could be gained using textures for this kernel. Also textures where used to store the macroscopic variables. This allows an easy and fast interoperability with the OpenGL subsystem, which gives the program the ability to do fast rendering of the variables computed by the method and this way achieve a decent rate of Frames Per Second.

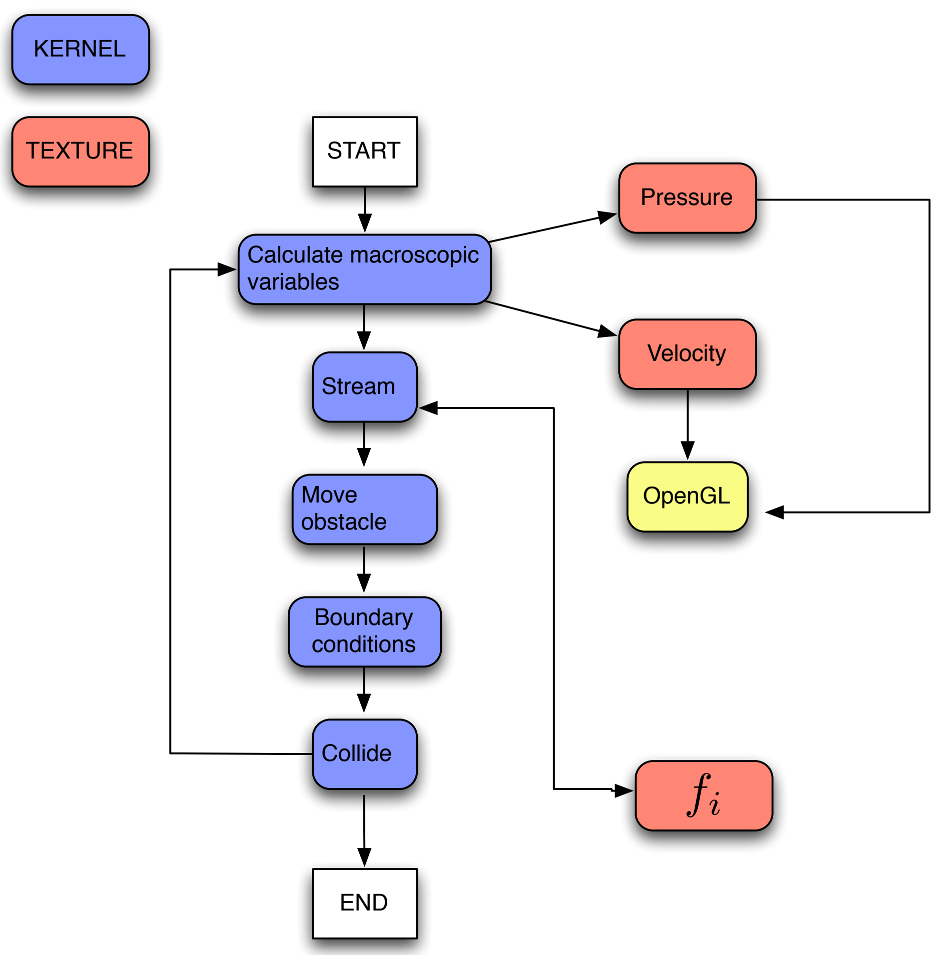


Figure 4.10: CUDA implementation in this work

Chapter 5

Visualization and interactivity

Not so long ago, computers could hardly draw a character on the screen. Since then much has changed. Once graphics systems evolved to be capable of generating sophisticated images in real time, engineers and researches began to use them as tool for simulations [Ang09].

5.1 Basic concepts

5.1.1 Visualization

In general visualization is any technique for creating images, diagrams, or animations to communicate a message. More specifically and related to this work is the *Scientific Visualization*. Scientific visualization is the transformation, selection or representation of data from simulations or experiments, with an implicit or explicit geometric structure, to allow the exploration, analysis and understanding of the data. Traditional areas of Scientific Visualization are flow visualization, medical visualization, astrophysical visualization and chemical visualization [wik10d].

5.1.2 Interactivity

One of the most important advances in computer technology was enabling users to interact with computer displays. Interaction is a process that involves at least two participants to complete another process[Sva00]. In the context of human–computer interaction, the human is interacting with the computer. An artifact is cataloged as interactive if it allows the user to have any level of interaction. Interactivity denotes the quality of the interactive aspects of an artifact (levels and forms of interaction) [Duq07].

The basic paradigm of in Human–Computer interaction is: the user sees an image on the displays, he/she reacts to this image by means of an interactive device, such as the mouse, then the image changes in response to the input, and later the user reacts to this change, and so on.

One important aspect of the interactive simulation is a *rapid response to human input*, Experiments have shown that a delay of more than 20 ms between when input is provided and the computer reaction is updated is noticeable by most people [wik10d]. So is desirable for an interactive visualization to provide a rendering based on human input within this time frame.
The term interactive framerate is good metric to measure how interactive is a visualization. A framerate of 50 frames per second (fps) is considered good enough for an interactive visualization, while 5 fps would be considered too low for what a human expects.

5.1.3 OpenGL

OpenGL (Open Graphics Library) is a standard specification defining a multi–language, multi–platform API for writing applications that produce 2D and 3D computer graphics. The API consists in a set of function calls which can be used to draw complex 3D scenes from simple *primitives* like: points, lines, rectangles, etc [Ang09].

The basic model of OpenGL is a *black box*, a term that engineers use to denote a system in which its properties are described only by its inputs and outputs. There is no need to know how it operates internally, to use it. It is similar to a car, one does not need to know how the engine works to drive it.
The main reason for this design are:

- Hide complexities of interfacing with different 3D accelerators by presenting a single, uniform interface.
- Hide differing capabilities of hardware platforms.

OpenGL operates accepting primitives such as points, lines and polygons, and converting them into pixels. This is done by the *graphics pipeline* known as the OpenGL state machine. Most OpenGL commands either issue primitives to the graphics pipeline, or configure how the pipeline processes these primitives.

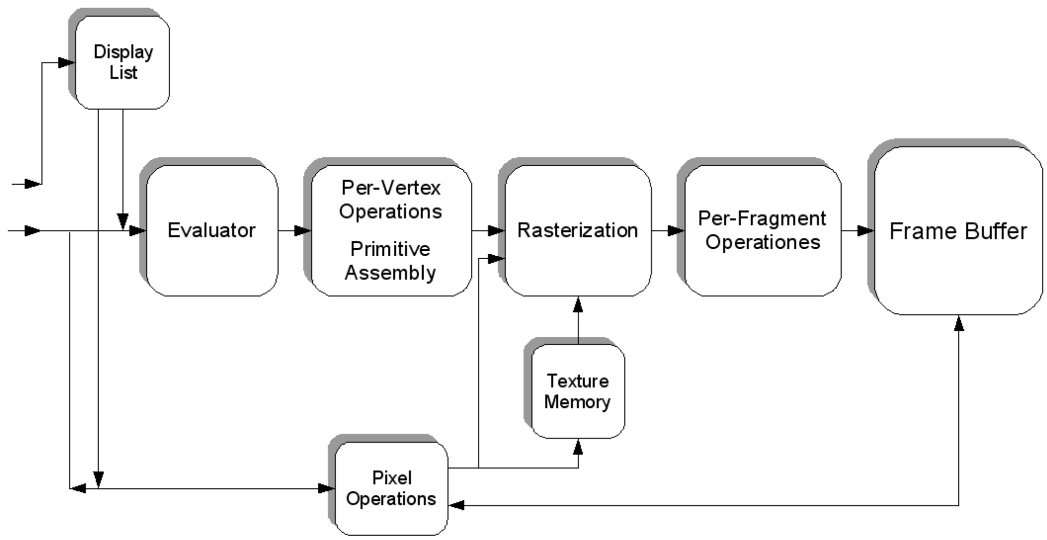


Figure 5.1: Simplified version of the Graphics Pipeline Process, taken from [wik10e]

Figure 5.1 shows the process in the graphics pipeline, which could be: first an evaluation, if necessary, of the polynomial functions which define certain inputs, like *NURBS* surfaces, approximating curves or the surface geometry. Then, vertex operations, transforming and lighting the polynomial functions depending on their material properties, and also clipping non visible parts of the scene in order to produce the viewing volume. After this, rasterization or conversion of the previous information into pixels. Later the polygons are represented by the appropriate color by means of interpolation algorithms. Fragment operations, like updating values depending on incoming and previously stored depth values, or color combinations. Finally, fragments are inserted into the Frame buffer, which is a video output device that drives a video display from a memory buffer containing a complete frame of data.

5.2 Interactive tool

An interactive visualization tool was created to utilize the LBM solver implemented in this work. This tool is a proof of concept and it is not intended for use in production environments. The objective of the tool is to allow the user to interact with the solver by means of moving an object with a common input device as the mouse, inside the the fluid. The future aiming of this application to allow the interactive analysis of fluid–structure phenomena. This could could be potentially be exploited by interactive design software applications.

The tool was built using OpenGL. Pixel Buffer Objects are used to store the macroscopic variables which are velocity magnitude and pressure coming from the solver, this increases the rendering performance and allows easy integration with CUDA.

5.2.1 Example

The tool plots the velocity magnitude using the *jet color map*, which is a color gamut from blue to red, representing lower values with blue, and higher values on red. Figure 5.2 shows the fluid flow around an obstacle which is enclosed by to walls at the top and bottom. The fluid flows from right to left, this set up produces the Poiseuille flow observable in the Figure. It also can be observed the von Karman street after the fluid hits the obstacle.

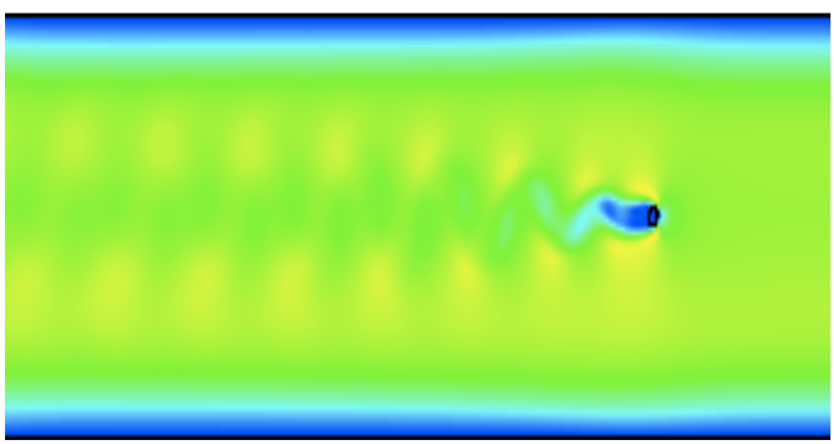
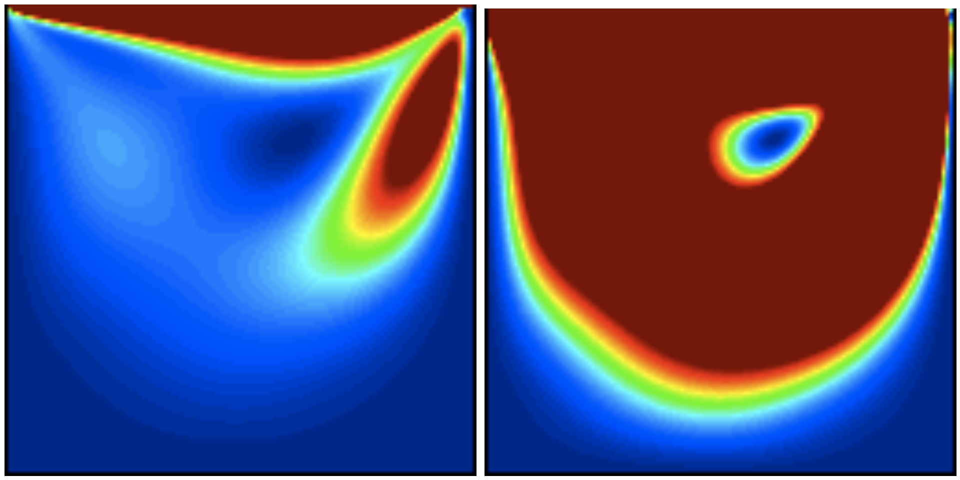


Figure 5.2: Von Karman street



(a) Cavity flow, low color saturation   (b) Cavity flow, high color saturation

Figure 5.3: Color saturation of a cavity flow, at the same time t

5.2.2 Color scale

A color is assigned to different ranges of a scalar field. However due to the transient nature of the simulation the scalar range change constantly, an some times the wrong color scale can led to a lose of details of the simulation, which can make its interpretation harder. To solve this, it was necessary to create a way to auto–scale and saturate the color map, to allow the user check for details not visible on a different color scale. Figure 5.3 shows and image of a cavity flow generated with this application. The image shows an under saturated and an over saturated velocity field at the same time step.

The application also has the ability to plot the pressure field. The same jet color map is used. Figure 5.4 shows the pressure field obtained for a cavity flow simulation. As for the velocity field, color saturation is also available to the pressure field.

Finally the application has the capacity to display the number Millions of Lattice Updates Per Second (MLUPS), the MLUPS are calculated by the following relation

$$MLUPS = \frac{\text{\# of Nodes} \times 10^{-6}}{dt} \tag{5.1}$$

Where dt is the clock time required to do one iteration. The MLUPS have become an standard metric when benchmarking LBM codes. Figure 5.5 shows the MLUPS achieved by the simulation of a cavity flow. The text is rendered over the active filed (velocity or pressure).

Figure 5.4: Cavity flow pressure field

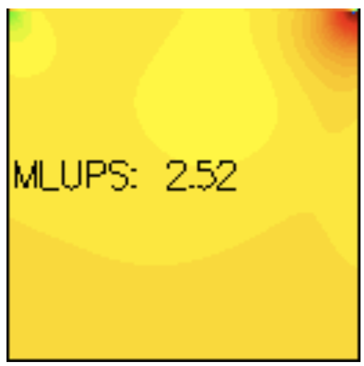


Figure 5.5: MLUPS displayed

5.2.3 Commands

The interactive tool by default loads a channel case, with fluid flow moving from right to left. The top and bottom boundaries are periodic to simulate an infinite domain. The tool allows the user the draw and obstacle and select it, to be later moved to a new position. The inlet velocity it is set in a configuration file and should be set in Lattice Boltzmann units. The application uses the keyboard and mouse as input, the command list is a follows:

p pressure

u velocity

i mlups information

s auto scale color map, to maximum and minimum bounds

r restart simulation

mouse/left click draw obstacle

mouse/right click pick obstacle to move

mouse/center click obstacle destination

Chapter 6

Numerical Experiments

Several tests were accomplished to validate this work. Performance tests where made on Mac Pro with an Intel Xeon 2.8Ghz processor with 8GB DDR2 of RAM. The GPU tests where made on the same machine running on a NVIDIA Geforce 8800 GT. This chapter presents the results from this tests and some conclusions about the results obtained.

6.1 Numerical results

6.1.1 2–D Poiseuille flow

The fully developed laminar flow in a channel is a typical case to examine the accuracy of CFD solver, due to the existence of an analytical solution. The Reynolds number is defined as $Re = u_0(2L)/\nu$ in a channel of height $2L$, and u_0 is the maximum velocity. The analytical solution of Poiseuille flow is given by

$$u_{exact} = u_0 \left(1 - \frac{y^2}{L^2}\right) \tag{6.1}$$

Here, the flow is driven by pressure gradient $\Delta P = 2\rho\nu u_0/L^2$ applied at the inlet and the oulet of channel.

Figure 6.1 shows the predicted results in comparison with the analytic solution. It can be seen that the solver and the implemented boundary conditions can produce results with minimal error, and highly accurate for this kind of flow. The Table 6.1 presents the data from which Figure 6.1 was obtained. In this table **A** stands for

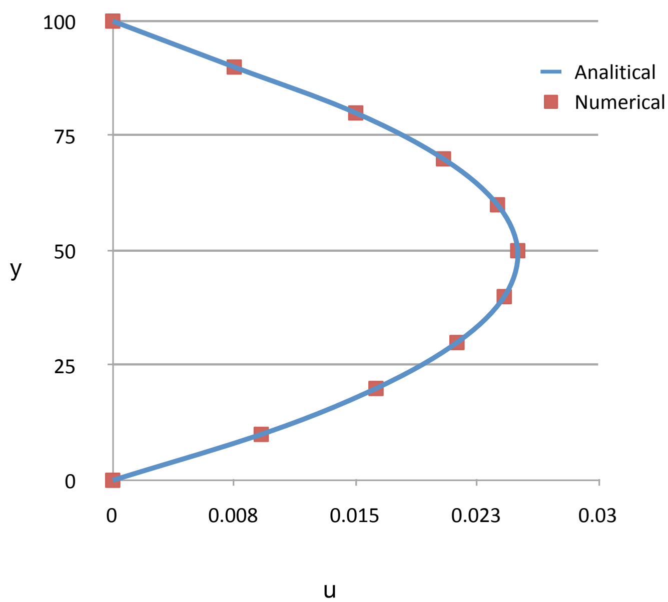


Figure 6.1: Predicted velocity profile of Poiseuille flow

analytical value and **N** for numerical values obtained.

6.1.2 Other common flows

The solver was able to reproduce other common flows like the cavity flow and vortices of von Karman. Although there is not known analytical solution for this kind flows Figure 5.3 and Figure 5.2 show that these two flows were generated according to a “general‘” agreement of the CFD community.

6.2 Computational results

The computational results presented here, are a comparison of performance between a CPU implementation running on a single core and an implementation running on GPU. The GPU implementation is a parallel version of the algorithm specially tuned for the architecture being used. The *Acceleration* achieved by the GPU implementation for this comparison is calculated by

$$Acceleration = \frac{S_p}{S_s} \tag{6.2}$$

Poiseuille flow data table				
L	**A**	**N**	**A−N**	**Error %**
0	0.0	0.0	0.0	0
10	0.00916	0.00908	0.00007	1.83
20	0.01624	0.01618	0.00005	1.34
30	0.02124	0.02116	0.00007	1.34
40	0.02415	0.02409	0.00006	1.25
50	0.02498	0.02498	0.00000	1.01
60	0.02374	0.02370	0.00003	1.14
70	0.02040	0.02032	0.00008	1.40
80	0.01499	0.01491	0.00008	1.53
90	0.00749	0.00742	0.00007	2.00
100	0.0	0.0	0.0	0

Table 6.1: Poiseuille flow error committed by the solver

Where S_p is the parallel speed in MLUPs and S_s is serial speed.

The Table 6.2 shows a comparison of the GPU and CPU. The Poiseuille flow was selected to realize all the following tests. This test, compares the acceleration of the different implementation solving different mesh sizes, with 20000, 80000, 180000 nodes. The results are provided in Millions of Lattice Updates Per Second (MLUPS). It can be seen that great acceleration is achieved by the algorithm run on the GPU, compared to its CPU counterpart.

GPU vs CPU comparison table			
# of nodes	**CPU**	**GPU**	**Acceleration**
20000	3.4	119	35X
80000	3.2	445	142X
180000	3.1	940	303X

Table 6.2: Speed in MLUPs achieved by the GPU and CPU

It is noticed that the CPU shows a degradation in performance at 180000, while the GPU shows an almost linear performance increase as it is depicted in Figure 6.2. This can be explained by the perfect match between the LBM algorithm and SIMT model.

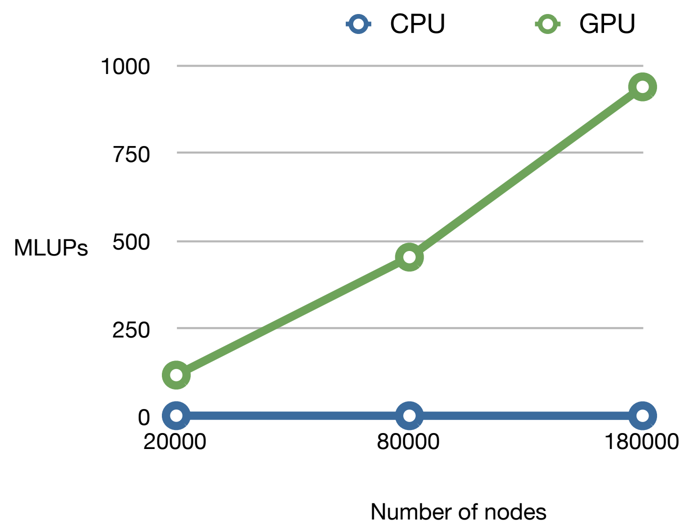


Figure 6.2: Acceleration gained using GPU

This next test is more related to the tweaking of the GPU implementation of the algorithm. For this reason, and since the comparison against the CPU implementation was already established, the CPU results wont be analyzed here.

This test consists in comparing the acceleration achieved by the GPU implementation solving different mesh resolutions, related to the variation of the block size. As seen on Section 4.5.2, the block size is the GPU parameter that sets the maximum number of threads going to be executed by the GPU.

GPU vs GPU comparison table			
# of nodes	**64**	**128**	**256**
20000	117	118	119
80000	451	455	455
180000	940	941	940

Table 6.3: Speed in MLUPs achieved by different block sizes

Table 6.3 shows the results of this test. It can be seen, that the acceleration gained is marginal. Much more investigation has to be made in this area to clarify this event. It was established during this test, that any block size greater than 256 would max out the GPU, and one of the GPU kernels would not launch, quitting on error.

Chapter 7

Conclusions

We have seen that the Lattice Boltzmann Method gives a behavior consistent with the Navier–Stokes equation. This makes the method a good replacement (in some situations) for the traditional methods as FEM and FVM. The advantages of the method are, easy implementation and high parallelism that can be achieved by the method.

As seen on this work, the GPUs can offer noticeable speed ups for problems that fit well into the SIMD paradigm. It was also noticed that the transition of the CPU code to GPU code is very simple using CUDA, and can be taken as a serious approach for the development of CFD applications.

7.1 Future Work

This work can be improved in many ways. Only the simplest model was implemented, and in 2D, this work could be easily extended to solve fluid problems in 3D. Also the simplest and most popular collision operator (BGK) was used, this operator has several drawbacks, as it only handles low Reynolds (up to 150). The collision operator can be replaced by a Multiple Relaxation Time (MRT) operator. This model supports Reynolds numbers bigger than BGK in up to 2 orders of magnitude and improves the general stability of the method. Also this work could be used as a base for coupled problems like fluid–structure interaction.

Finally, this model does not support curved geometries, all it does is a stair–cased approximation. This can be corrected using one of the many models for curved boundaries.

The GPU implementation can be optimized even more, the use of shared memory might improve performance for the streaming step.
OpenCL a standard technology and is a new direct competence to CUDA which is proprietary to NVIDIA. OpenCL works not only on GPUs but also on CPUs, FPGAs, etc. This technology offers the benefit of transparent scalability from CPUs to GPUs with no changes on the code. This make the technology attractive, because the need to write a CPU version to compare, is waived.

Bibliography

[Ang09] Edward Angel. *Interactive Computer Graphics*. Pearson, fifth edition, 2009.

[Bui97] James Maxwell Buick. *Lattice Boltzmann Methods in Interfacial Wave Modeling*. PhD thesis, University of Edinburgh, 1997.

[Duq07] Juan Fernando Duque. *INTERACTIVE CFD SIMULATIONS*. Universidad EAFIT, 2007.

[eth10] Particle methods – cselab [online]. 4 2010. Available from: http://www.cse-lab.ethz.ch/index.php?&option=com_content&view=article&id=315&catid=39 [cited 2010 April 25].

[FHP86] U. Frisch, B. Hasslacher, and Y. Pomeau. Lattice-gas automata for the navier- stokes equation. *Physical Review Letters*, 1986.

[HPdP73a] J. Hardy, Y. Pomeau, and O. de Pazzis. Time evolution of a two-dimensional classical lattice system. *Physical Review Letters*, 1973.

[HPdP73b] J. Hardy, Y. Pomeau, and O. de Pazzis. Time evolution of a two-dimensional model system. i. invariant states and time correlation functions. *ournal of Mathematical Physics*, 1973.

[KmWH10] David B. Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010.

[KPP06] Michael Kistler, Michael Perrone, and Fabrizio Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Computer Society*, 2006.

[Lad94] A. J. C. Ladd. Numerical simulations of particulate suspensions via a discretized boltzmann equation. part i. theoretical foundation. *J. Fluid Mech*, 1994.

[Lat08] Jonas Latt. *Choice of units in lattice Boltzmann simulations*. LB-Method.org, April 2008.

[LCM+08] Jonas Latt, Bastien Chopard, Orestis Malaspinas, Michel Deville, and Andreas Michler. Straight velocity boundaries in the lattice boltzmann method. *Phys. Rev. E*, 77:056703, 2008.

[MZ88] G. McNamara and G. Zanetti. Use of the boltzmann equation to simulate lattice–gas automata. *Physical Review Letters*, 61(20):2332–2335, 1988.

[NVI10] NVIDIA. Nvidia cuda programming guide. www.nvidia.com, 2 2010.

[Sch06] M. Schäfer. *Computational Engineering – Introduction to Numerical Methods*. Springer, 2006.

[ST05] Michael Sukop and Daniel Thorne. *Lattice Boltzmann Modeling An Introduction for Geoscientists and Engineers*. Springer, 2005.

[Suc01] Sauro Succi. *The Lattice Boltzmann Equation For Fluid Dynamics and Beyond*. Oxford University Press, 2001.

[Sva00] Dag Svanæs. *Understanding interactivity steps to a phenomenology of human–computer interaction*. PhD thesis, NTNU, 2000.

[SWV96] Robert L. Street, Gary Z. Watters, and John K. Vennard. *Elementary Fluid Mechanics*. Wiley, 7 edition, 1996.

[Vig09] Erlend Magnus Viggen. The lattice boltzmann method with applications in acoustics. Master's thesis, NTNU, 2009.

[VM95] H. K. Versteeg and Malalasekera. *An introduction to Computational Fluid Dynamics*. Longman Scientific and Technical, 1995.

[wik10a] Cellular automaton - wikipedia [online]. 4 2010. Available from: http://en.wikipedia.org/wiki/Cellular_automaton [cited 2010 April 24].

[wik10b] Finite element method- wikipedia [online]. 4 2010. Available from: http://en.wikipedia.org/wiki/Finite_element [cited 2010 April 20].

[wik10c] Fluid mechanics - wikipedia [online]. 4 2010. Available from: http://en.wikipedia.org/wiki/Fluid_mechanics [cited 2010 April 20].

[wik10d] Interactive visualization - wikipedia [online]. 4 2010. Available from: http://en.wikipedia.org/wiki/Interactive_Visualization [cited 2010 April 24].

[wik10e] Opengl - wikipedia [online]. 4 2010. Available from: http://en.wikipedia.org/wiki/OpenGL [cited 2010 April 27].

[ZH97] Qisu Zou and Xiaoyi He. On pressure and velocity boundary conditions for the lattice boltzmann bgk model. *Physics of Fluids*, 1997.

Appendix A

Units Example

There is always confusion about the units conversion for the newcomer to the Lattice Boltzmann Method . For the this reason, this section presents an example of how to relate the lattice units with physical units. This section will use the subscript $_p$ to identify physical units and $_{lb}$ to identify Lattice Boltzmann Method units.

The example is called *channel case*. The channel case consist on a simple 2D domain with solid walls in the top and bottom, and a incompressible fluid that flows from left to right. The channel has a speed boundary condition at the inlet (left), and pressure boundary at the outlet (right). Also there is an object in the channel that is going to set the proper *Reynolds* number of the simulation.

A.1 Channel case

Here the process to convert between the different units related to the simulation is described. As many authors recommend different approaches to convert some parameters of the simulation, a couple of them are presented, and the differences are explained.

Simulation parameters:

- $u_{xp} = 2cm/s$
- $p_p = 0.5Pa$
- $\nu_p = 1cm^2/sec$

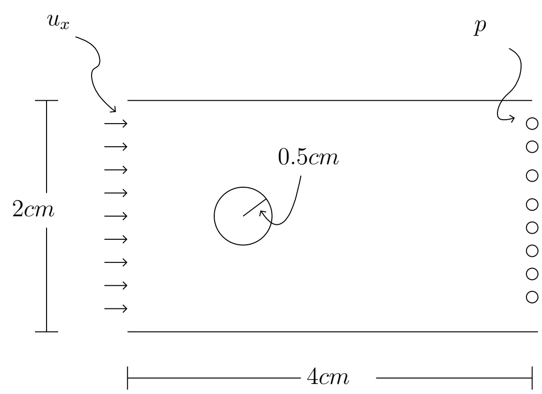


Figure A.1: Sample Channel

- $L = 4cm$
- $H = 2cm$

As a dimensionless number, the *Reynolds*, can be used to convert from a system to the other one,

$$\mathrm{Re} = \frac{\rho u L}{\mu} = \frac{uL}{\nu} \tag{A.1}$$

where L is the characteristic length.

A.1.1 Initial approach

The first step is to calculate the Reynolds number. This rises a new problem. Which L is the proper one to be chosen?. Since we are going to study the flow around a cylinder the proper L would be $2r$ (the length of the object of study).

1. Replacing in Equation A.1 we get $\mathrm{Re} = 0.4$
2. Now we have to choose are grid size, I will use a 400×200 $(Nx \times Ny)$grid. One has to keep in mind that the Lattice Boltzmann Method uses squared lattices and the proper ratio has to be chosen to make sure that $\delta x_p = \delta y_p$.
3. With the grid size we can calculate δx_p as $\delta x_p = L/Nx$. This way $\delta x_p = 0.01cm$ is obtained.

4. The next thing we need to calculate is the numerical viscosity (ν_{lb}), but for this calculation a fixed relaxation time (τ) is need. I will use $\tau = 1$. Using the equation $\nu_{lb} = (\tau - 1/2)/3$ we get $\nu_{lb} = 0.1666$

5. With $\delta x = 0.01$, ν_{lb} and ν_p, it is possible to calculate the time δt_p using the equation $\delta t_p = (\nu_{lb}/\nu_p)\delta x^2$ we get $\delta t_p = 1.666 \times 10^{-5} sec$

6. Now we calculate the number of iterations to achieve a δt_p using the relation $Niter = 1/\delta t_p$ this way we know that we need $Niter = 60000$ to simulate $1sec$.

7. To convert the u_p for the inlet boundary condition using the formula $u_{lb} = u_p(\delta t_p/\delta x_p)$ So our inlet boundary condition has to be implemented using $u_{lb} = 8.33 \times 10^{-4}$. Always keep in mind that the numerical Mach number has to be smaller than 0.3 or the method wont converge to Navier–Stokes [Suc01] so keep $\mathrm{M}a << 0.3$ using the formula $\mathrm{M}a_{lb} = u_{lb}/c_s$.

8. Finally for the oulet boundary condition ρ_{lb} has to be calculated[1]. Its been established that $\rho_{lb} = 1 + (1/c_s^2)p_p\delta t_p^2/\delta x_p^2$ with $c_s^2 = 1/3$ we get a $\rho_{lb} = 1.0$

A.1.2 Calculating τ

It is also possible determine the relaxation time τ using Reynolds number to calculate the numerical viscosity $\nu_{lb} = u_{lb}L_{lb}/Re$ and the equation $\tau = 1/2 + \nu_{lb}/c_2^2$ although this is valid it is dangerous since τ could get close to $1/2$ which makes the method unstable.

A.1.3 Different approach to calculate δt

Since in the Lattice Boltzmann Method there is no easy way to calculate δt it is recommended to set $\delta t \sim \delta x^2$ to keep the method numerically stable [Lat08].

A.1.4 Calculating Nx

If we have the numerical values for a simulation, the size of the mesh can be found or *optimized*.

[1]Note that as this is an incompressible fluid ρ_p does not have to be calculated, but ρ_{lb} is needed for the method to do its computations

Lets suppose that for the simulation above we have the Reynolds number we want to simulate, but with do not know the adequate size of the mesh to achieve proper results. Again the Reynolds number is used to get the missing value, we know ν_{lb}, u_{lb} and Re, then using the Equation A.1 we get $L = \mathrm{Re}\nu_{lb}/u_{lb}$ replacing we get $L = 80$, which means that a mesh of 160×80 would have been enough for the simulation.

Appendix B

Fast 2D point in polygon

A new "fast" 2D point in polygon algorithm was introduced during this work. This algorithm is very simple, it uses only 1 cross product and depends of a δy value that can be preset according to what is needed.
The objective of this algorithm is find a single point inside a polygon to use it a seed for a flood-fill.
The algorithm was designed to work on discrete schemes, so it is assumed that minimum δy is going to be big enought so is never collinear with the created vector $\vec{B}$. Although picking the right δy can tweak the algorithm to make it work on continuum schemes.

The figure B.1 shows the basic sketch of the algorithm. A vertex $\hat{V}$ is chosen arbitrary and always using the right-hand rule, the mid point between $\hat{V}$ and $\hat{G}$ is found (p_0), after this a new point is created using δy^+ (p_1) a cross product is calculated to determine if the direction of the resulting vector goes out of the plane or goes into the plane. If it goes outside the plane δy^+ is inside the polygon, if it goes into the plane then p_1 (δy^+) is inside the polygon. The algorithm is shown in pseudocode below:

As seen on the algorithm each instruction is executed only once keeping the algorithm $O(1)$. The calculation of the cross product is straight forward for 2D and is also $O(1)$ which produces a fast simple way to check if a point is inside a polygon.

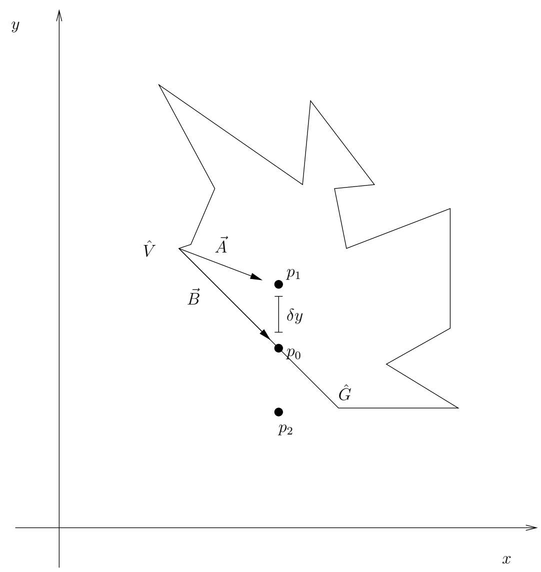


Figure B.1: Point in polygon

Algorithm 2 Point inside the polygon

$\hat{V} \Leftarrow$ Pick vertex
$p_0 \Leftarrow$ Calculate mid-point between picked vertex and it following vertex
$p_1 \Leftarrow p_0 + \delta y^+$
$point \Leftarrow (p_0, \hat{V}) \times (p_1, \hat{V})$
if ($point > 0$) **then**
  the point is outside the polygon, use δy^- to calculate p_2
  p_2 is inside the polygon
else if ($point < 0$) **then**
  p_1 is inside the polygon
else if ($point = 0$) **then**
  Pick next vertex (we are on a vertical line).
end if
