



**NTNU – Trondheim**  
Norwegian University of  
Science and Technology

# Lattice Boltzmann Simulations on a GPU

An optimization approach using C++ AMP

**Kristoffer Clausen Thyholdt**

Marine Technology (2 year)

Submission date: June 2012

Supervisor: Håvard Holm, IMT

Norwegian University of Science and Technology  
Department of Marine Technology



## **Problem Description**

The lattice Boltzmann Method is a computational fluid simulation method used to simulate fluid flow, well suited for parallel computing. With today's high performance graphic cards specialized in parallel computations, this creates opportunities to reach performance only possible with parallel supercomputers in the past.

This project will investigate the lattice Boltzmann method and different implementation techniques available, focusing on implementations able to achieve good performance by parallel execution.

A parallel lattice Boltzmann solver will be implemented using the appropriate techniques and the performance will be documented. The numerical validity of the code should be tested against known fluid flow solutions, and a visual representation of the fluid flow should be created.

The code is expected to be developed in C++ using a parallelization library called C++ AMP.

Assignment given: 7. February 2012

Supervisor: Håvard Holm



## Abstract

The lattice Boltzmann method has become a valuable tool in computational fluid dynamics, one of the reasons is due to the simplicity of its coding. In order to maximize the performance potential of today's computers, code has to be optimized for parallel execution.

In order to achieve parallel execution of the lattice Boltzmann method, the data dependency has to be solved. And to get good performance, the memory has to be organized for unit stride access. Here we investigate the most known algorithms for lattice Boltzmann, and implement a code which runs on a parallel graphics processor, using a library for parallelization called C++ AMP. Furthermore, we show how the code compares to known solutions of fluid flows to verify the numerical results. The optimized parallel code achieves a speed up of 650 times the un-optimized code, on a current generation high-end graphics card.



## Sammendrag

Lattice Boltzmann metoden har blitt en populær metode for Computational Fluid Dynamics (CFD). Mye grunnet den enkle måten å kode en løser på. For å kunne utnytte potensialet i dagens datamaskiner må koden optimaliseres for parallell utførelse.

For å kunne utføre lattice Boltzmann metoden parallelt på maskinen, må data avhengigheten løses. Og for å få god ytelse må minne organiseres for unit stride access. I denne oppgaven undersøkes de mest vanlige algoritmene for lattice Boltzmann, og en løser blir implementert for å kjøre på et parallelt grafikkort ved hjelp av et bibliotek ved navn C++ AMP. Videre testes koden opp mot kjente løsninger av væske strømminger. Den optimaliserte parallelle koden oppnår en hastighetsøkning på 650 ganger den uoptimaliserte koden, på et av dagens high-end grafikkort.





## Acknowledgements

This report is the result of work done at the Institute of Marine Technology at the Norwegian University of Science and Technology (NTNU) in Trondheim, Norway.

I would like to thank my supervisor Håvard Holm for introducing me to computational hydrodynamics and helping me to define an appropriate master project.

I would also like to thank Dr. Anne C. Elster for making the HPC laboratory available, so I was able to run my code on several computers, and I want to thank Øyvind Selmer and Anders Gustavsen who tested the code on their machines.

Last but not least, I want to thank Jens Erik Thyholdt and Lasse Roaas for having the patience to prof-read this report and correct my many spelling mistakes.

Kristoffer Thyholdt

Trondheim, Norway June 13, 2012



## Table of Contents

Problem Description.....	1
Abstract .....	3
Sammendrag .....	5
Acknowledgements.....	7
List of figures .....	11
List of tables .....	13
1. Introduction to parallel computing.....	15
1.1. Parallelization .....	16
2. C++ AMP .....	19
3. Achieving good performance from parallel programs.....	23
3.1. Parallelism.....	23
3.2. Data access efficiency.....	24
4. Computational Fluid Dynamics .....	27
4.1. Lattice Boltzmann Method .....	27
4.2. Theory.....	29
4.3. Boundary Conditions .....	33
4.4. The algorithm.....	34
4.5. Parameterization .....	35
4.5.1. Example .....	35
5. Implementation.....	37
5.1. Indexing .....	37
5.2. Data Layout.....	38
5.3. Algorithms.....	39
5.4. Arithmetic precision .....	44
6. Visualization .....	47

6.1. Vtk fileformat.....	49
7. Performance.....	51
8. Validating the code.....	53
8.1. Hagen-Poiseuille flow .....	53
8.2. Lid Driven Cavity Flow.....	55
8.3. Uniform flow around a smooth cylinder .....	57
8.4. Vortex Shedding Frequency.....	59
9. Conclusion .....	61
10. Bibliography.....	63
A. Appendix 1 – Example of vtk file .....	65
B. Appendix 2 – Hardware specifications.....	67
C. Appendix 3 – Selected Source Code.....	69

## List of figures

Figure 1: A representation of AoS and SoA memory layout for 3 elements with x, y and z stored for each element. Each cell represents one memory location.....	24
Figure 2: A discrete lattice grid where the sites have 9 lattice vectors .....	28
Figure 3: Two-dimensional D2Q9 model.....	30
Figure 4: The LBM collision and streaming step .....	32
Figure 5: LBM boundary conditions .....	33
Figure 6: Basic algorithm of the lattice Boltzmann method .....	34
Figure 7: Flow around a cylinder .....	35
Figure 8: Enumeration function to map a two-dimensional array in a one dimension.....	37
Figure 9: Three memory layouts for the lattice Boltzmann method .....	38
Figure 10: One iteration of the two-lattice and shift algorithm. ....	40
Figure 11: The swap algorithm has advanced to the blue node. First some of the distribution values are exchanged with those of the neighbors. Second, collision is executed for the node. Finally the algorithm proceeds to the next node. Notice how the lower distribution values are already exchanged. ....	41
Figure 12: A one dimensional view of the Eulerian and Lagrangian approach.....	42
Figure 13: Performance comparison of the implementations from (Mattila, et al., 2007) .....	43
Figure 14: Visualization of a velocity field around a cylinder .....	47
Figure 15: Streamlines created from the velocity field around a cylinder .....	48
Figure 16: Velocity vectors from the velocity field around a cylinder .....	48
Figure 17: Visual representation of Hagen-Poiseuille flow.....	53
Figure 18: Comparison of known and simulated velocity profile for the Poiseuille flow .....	54
Figure 19: A cavity with lengths L, where the lid moves with velocity $U_0$ .....	55
Figure 20: Comparison of u-velocity along vertical lines through the geometric center of the cavity .....	56
Figure 21: Comparison of v-velocity along horizontal lines through the geometric center of the cavity .....	56
Figure 22: Regimes of flow around a circular cylinder, taken from (Sumer & Fredsøe, 1997) .....	57
Figure 23: Visualization of stream around a cylinder for different Re, from LBM.....	58
Figure 24: Strouhal's number plotted for experimental and numerical results .....	59
Figure 25: Evolution of vortex shedding at Re 110, time is given in seconds.....	60



## List of tables

Table 1: Performance gains for optimizations .....	51
Table 2: Performance on different GPUs .....	52





## 1. Introduction to parallel computing

Since personal computers became common in the beginning of the 80's, the computational power has steadily increased. It was expected that software developed for the current generation of hardware, would run faster on the next generation, because of the ever-improving performance. This improvement has resulted in attainable and affordable computers with performance on the level of billions of floating point operations per second (gigaFLOPS).

The steady increase of software performance due to improved hardware diminished around 2005. The physical limitations were reached, e.g. the time used by electric signals to cross a chip, leading to a stop in increased clock speed. To further improve the performance, processors (CPU) were fitted with multiple cores able to do operations independently. Normal desktop computers became what were formerly known as parallel supercomputers.

However, having multiple cores available does not simply speed up applications. The software must be designed to take advantage of multiple cores, thus software developed to use only one core will not achieve the performance potential of today's processors.

Simultaneously with this change in improved CPU performance, a change occurred in graphic cards development. While the CPUs were fitted with two to eight cores, the graphical processing unit (GPU) was fitted with hundreds of cores. These cores differ a lot from the cores on a CPU. The GPU is developed to increase the performance of calculations related to graphics, like updating the color of a pixel on the screen.

For comparison a typical high-end CPU<sup>1</sup> has six cores and can execute 100 Gflops at double precision floating points, while a high-end GPU<sup>2</sup> have 448 cores and can execute 1 Tflop at single precision. The massive improvement in performance for the GPU is not only related to the number of cores, it is also related to the high memory bandwidth of the GPU. A typical CPU has a memory bandwidth of 20 gigabyte per second (GB/s) while a GPU have a bandwidth of 150 GB/s.

---

<sup>1</sup> Intel i7 Extreme

<sup>2</sup> Nvidia Quadro 6000

Even with these obvious advantages, a GPU is not superior to a CPU for all tasks. The GPU is developed to execute parallel calculations, where the same operation should be executed on a lot of data, known as SIMD (Same Instruction Multiple Data). For calculations of advanced data structures, multi-tasking and a lot of other areas, the CPU is still superior to the GPU. Another concern is the memory limitation for GPUs, the CPUs usually have 4-16 gigabyte of available memory, while GPUs typically have up to 2 gigabytes.

Accuracy can also be a problem on today's GPU's, as the majority only supports single precision, although it has become more common with double precision now that the GPU manufacturers has noticed the potential to use GPUs for high performance computing. However, executing operations on single precision floating numbers is usually faster than double precision computations, so if the accuracy is sufficient with single precision, this should be considered. (Itu, et al., 2011)

Fortunately a lot of the problems that involve big amounts of data are natural candidates for parallel processing, and well suited for the GPU. Scientific modeling and simulation, like computational fluid dynamics, use simple equations to model complicated problems with massive amounts of data. These problems can take hours or even days to finish computing. A significant speedup of the calculations means reduced runtime or higher accuracy since larger datasets can be processed.

### **1.1.Parallelization**

Creating parallel software can be a challenging task as most programming languages are designed for sequential operations. To ease the development several libraries and aids have been released the last couple of years. The following are some of the more popular libraries released:

- OpenMP – an open standard. Supports several programming languages and is platform independent. Is developed to only parallelize the CPU.
- OpenCL – an open standard for both CPU and GPU parallelization. Uses a programming language of its own, which resembles C. Support for a variety of graphic cards.
- DirectCompute – A Windows specific standard similar to OpenCL. Also uses its own language, resembling C, but there are significant differences both to C and OpenCL.

- CUDA – Nvidia’s own standard, used for parallelization of Nvidias graphic cards. Uses a programming language called CUDA C, which also resembles C. CUDA C however, is “higher level” than OpenCL and DirectCompute, meaning it is easier to use than the lower level languages of OpenCL and DirectCompute.

Each of these has restrictions and problems, OpenMP only support parallelization for CPUs, OpenCL and DirectCompute is quite complicated. CUDA only supports graphic cards from Nvidia. In addition, learning a new programming language is required.



## 2. C++ AMP

C++ AMP is developed as a library for C++ instead of using its own programming language to handle parallel operations. Because of this there will be few syntax changes, but some limitations are enforced to reflect the limitations in the hardware used for parallel computing. (Miller & Gregory, 2012)

The library is developed by Microsoft, and uses an open specification. They encourage developers for all platforms to implement the specification on their own systems. Currently there is very limited support for C++ AMP, as it is still under development and the only available version is in beta phase. Windows 7, as well as the coming Windows 8 is the only operating systems to officially support it. Visual Studio 11, which is currently only available in beta version, is the only tool to create software supporting C++ AMP. By means of hardware however, both of the biggest graphic card manufacturers, Nvidia and AMD, support C++ AMP.

When using C++ AMP to develop parallel software, the data-parallel hardware is referred to as an accelerator. This is usually the GPU, but the CPU can be used if the GPU is unsupported. The accelerator usually has separated memory from the CPU, so transfer of data between the two is time consuming. Therefore it is important to consider the cost of transferring data to the accelerator versus the time saved by doing the computations parallel on the accelerator.

C++AMP give an explicit control of data transfer between the CPU and the accelerator, and the calculations performed on each of them. The transfer can be done synchronous or asynchronous. The programming model makes it easy to choose between an ease-of-use approach and having full control to maximize the performance to the fullest with advanced fine tuning.

The following example is a function for adding two vectors using a sequential loop in C++.

```
void AddArrays(int n, int* pA, int* pB, int* pC)
{
    for (int i=0; i<n; i++)
    {
        pC[i] = pA[i]+pB[i];
    }
}
```

The equivalent code parallelized with C++ AMP would look like this.

```
#include <amp.h>
using namespace concurrency;

void AddArrays(int n, int* pA, int* pB, int* pC)
{
    array_view<int,1> a(n, pA);
    array_view<int,1> b(n, pB);
    array_view<int,1> c(n, pC);

    parallel_for_each(
        c.extent, [=](index<1> idx) restrict(amp)
        {
            c[idx] = a[idx] + b[idx];
        }
    );
}
```

The changes include:

- *#include amp.h* to use the library
- *using namespace concurrency*, this reduces the typing needed to call functions from the amp.h library.
- *array\_view* is used to define a container for the data to be copied between the CPU and the GPU.
- The for loop is changed with a *parallel\_for\_each* function, this function takes two arguments. The first is the extent, or the *compute domain* that defines the number of threads to create. The second is the *lambda expression* which defines the code to be run on the accelerator. The index statement defines the variable to hold the thread index (*idx*), this can be compared to the integer *i* in the sequential loop, holding the value of the current iteration. This is followed by the restrict keyword, defining the restrictions in the code that follows.

The restrictions are due to the kind of data types and operations the accelerator supports. Strings for example, are not supported by the accelerator, and in most cases with today's GPUs, the use of double precision floating points has to be restricted. The original mathematical logic remains unchanged.

C++ AMP supports two different methods to store and communicate with data on the accelerator. One is the *array\_view* used in the code example. The other is called just *array*. The difference between these is the handling of data transfer between the CPU and accelerator. The *array* is a container located on the accelerator and the data has to be copied between the accelerator and CPU manually, this gives more control over the data, and may improve performance. The *array\_view* is a wrapper, which automatically transfers the CPU container to the appropriate container on the accelerator. Transferring the data back to the CPU is done automatically when a reference to the CPU container is called. This is an easier alternative, but when dealing with functions to do operations on a container already stored on the accelerator this will perform unnecessary transfers between the two, thus lowering the performance.

Once the code has been compiled, it can run on all machines that support C++ AMP. There is no need to recompile the program for a different computer since the library handles all communication with the hardware. Bringing back the times where software already designed to perform a task, will be increasingly faster on the next generation of hardware.





### 3. Achieving good performance from parallel programs

Not all computing tasks will perform better in a parallel environment, to achieve good performance some prerequisites of the algorithm have to be met. There are also a variety of ways to make a working parallel algorithm run faster. Some of the requirements and optimizing strategies will be presented in this chapter.

#### 3.1.Parallelism

Scientific computing usually involves operations on large quantities of data. If this data can be manipulated independently, the operations can be performed in parallel execution. The data can be distributed evenly among the processors, and each processor or core can perform operations on an assigned part of the data. This is referred to as data parallelism (Rauber & Runger, 2010).

A lot of algorithms perform computations by iteratively traversing a large data set. This is often handled by a loop. In a sequential program this is executed in a sequential manner, where the  $i$ th iteration is executed when the  $(i-1)$ th iteration is completed. If there are no data dependencies, this loop could be performed in any arbitrary order and would be well suited for parallel computing. If however the loop needs to be executed in a specific order, the algorithm is better suited for sequential programming.

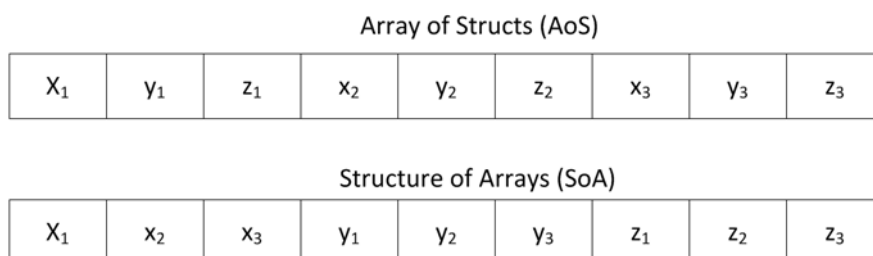
```
for(i=1;i<n;i++)          for(i=0;i<n;i++)
{                          {
    a[i] = b[i-1] + c[i];    a[i] = a[i+1] + b[i];
}                          }
```

Looking at the two code snippets above, the left snippet is not data dependent and can be run in any arbitrary order. The right snippet however, need to be run in a sequential manner, where the  $i$ th iteration is completed before the  $(i+1)$ th iteration, since the value  $a[i]$  is dependent on the value in  $a[i+1]$ . Data parallelism is crucial to get good performance from parallel computations.

### 3.2.Data access efficiency

Reading and writing to the memory is the bottle neck of today's computations, and data access is the most important factor to optimize parallel code (Hager & Wellein, 2011). To reduce the cost of retrieving data from the memory, caches have been introduced. A cache is a faster type of memory that is located closer to the processing unit. Because of the high cost of fast memory, the size of the cache is very small compared to the main memory. Modern GPUs and CPUs have multiple layers of caches, increasing in bandwidth and decreasing in size closer to the processing unit. When the processing unit needs data for an operation, it checks if the data is located in the cache first, if the data is not located in the cache, a cache miss occurs. When a cache miss occurs data is transferred from the main memory to the cache. Since the transfer is a costly operation, a sequence of data is transferred to the cache for each cache miss, instead of only the needed value. This sequence of data is called a cache line.

This knowledge is important to use when structuring the memory layout of the program. Every time a value that is not present in the cache is needed, a chunk of data is transferred from the main memory to the cache. If any of the data in the chunk is unused before it is overwritten, bandwidth was wasted. The number of cache misses should be minimized; to achieve this, unit stride access to the memory is preferred, meaning that the data is located in consecutive order. When the cache is retrieving the data, this is called spatial locality, meaning that elements within relatively close storage locations are going to be used for the coming operations. To best achieve unit stride access, a Structure of Arrays (SoA) should be used over the more convenient Array of Structs (AoS).



**Figure 1: A representation of AoS and SoA memory layout for 3 elements with x, y and z stored for each element. Each cell represents one memory location**

Another bottleneck to consider when using the GPU to perform calculations is the memory bus between the GPU and CPU. This bus is slower than the internal memory bus in the GPU, so data transfers between these two should be kept to a minimum. If the calculations can be done on the GPU without interaction from the CPU this would be optimal. However, the data computed on the GPU often need to be transferred to the CPU for some data operations, like writing results to the disk. The optimal strategy is to only transfer the data when it is needed.



## 4. Computational Fluid Dynamics

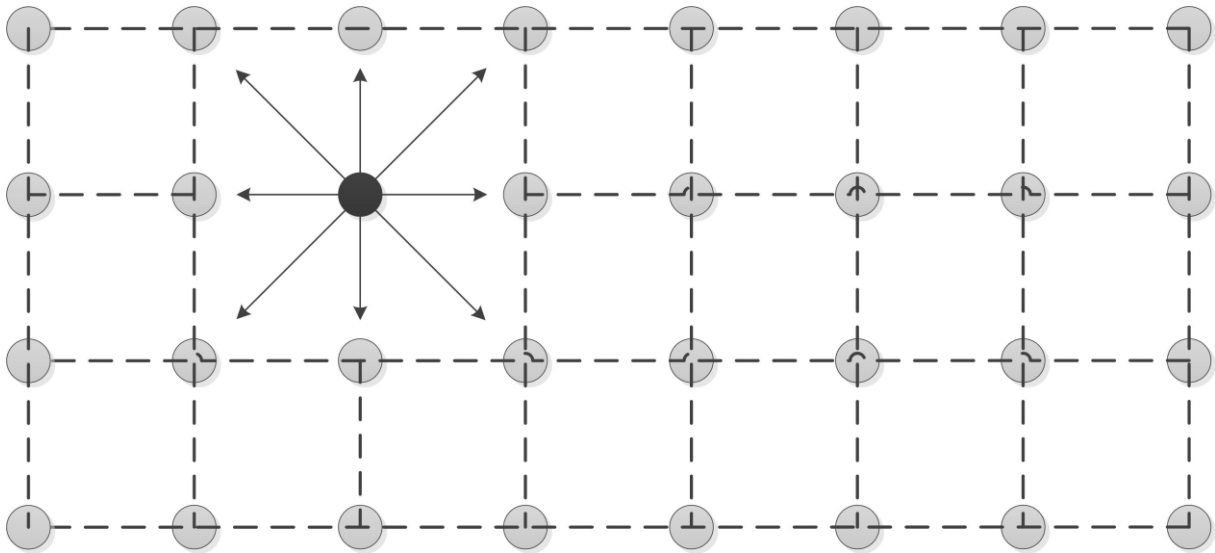
Computational Fluid Dynamics (CFD) is the use of computers to study fluid in motion. Due to the vast number of molecules in fluids, tracking the movement of each individual molecule at a microscopic level would be impractical. When looking at problems with size and time several orders of magnitude above the size of the molecules, it is possible to consider the fluid as a continuous medium, and ignore its discrete nature without significant errors (Papadopoulos, 2008). Using this assumption together with the laws of conservation of mass, momentum and energy, the non-linear partial differential equations known as the Navier-Stokes (NS) equations will model fluid motion accurately.

### 4.1. Lattice Boltzmann Method

In 1988, the Lattice Boltzmann method (LBM) was introduced by (McNamara & Zanetti, 1988). This model bridges the gap between microscopic and macroscopic approaches, by considering the behavior of a collection of molecules, instead of single molecules (Mohamad, 2011). LBM has a lot of advantages. It can easily be applied to complex domains, treat multi-phase and multi-component flow and it uses simple numerical codes, allowing efficient parallel implementation. On the downside however, more computer memory is needed as opposed to Navier Stokes methods. (Qian, et al., 1991)

The lattice Boltzmann method derived as a special case of lattice gas automata (LGA), the HPP model proposed by Hardy, Pomeau and de Pazzis as a new however flawed technique for numerical study of the Navier-Stokes equations. (Mohamad, 2011)

The basic principles behind LGA comprise of a discretization of time and space in steps to form a lattice and to discretize the fluid particles, positioned in certain points in space, called lattice sites or cells. The fluid particles are represented by a binary state, defining if a cell is occupied by a particle or not. The evolution of the simulation is done in discrete time steps where the cells are only allowed to travel along distinct lattice vectors, derived by a discretization of the velocity space. As shown in figure 2.



**Figure 2: A discrete lattice grid where the sites have 9 lattice vectors**

In lattice Boltzmann the binary state of the cells is replaced with real number particle distribution functions. So in contrast to the prior LGA method, not every single particle is simulated, rather a collection of particles, whose behavior is described by distribution functions.

The lattice Boltzmann methods can not only be seen as a successor of lattice gas automata. The method can be derived from the Boltzmann transport equation, an equation used in statistical mechanics that describes the evolution of the fluids microscopically based on the density and collision of particles (Körner, et al., 2005). From the Chapman-Enskog theory, one can also recover the Navier-Stokes equations from the lattice Boltzmann method.

## 4.2.Theory

From the classic Boltzmann equation used to describe the dynamics of particle position probability in phase space

$$\frac{\partial f(\mathbf{x}, \mathbf{u}, t)}{\partial t} + \mathbf{u} \cdot \nabla f(\mathbf{x}, \mathbf{u}, t) = Q(f, f) \quad (4.1)$$

where  $f(\mathbf{x}, \mathbf{u}, t)$  is the particle distribution function, which gives the number of particles per unit volume in phase space.  $\mathbf{u}$  is the particle velocity  $\mathbf{u} = \frac{\mathbf{p}}{m}$  with momentum  $\mathbf{p}$  and constant mass  $m$ .  $Q(f, f)$  express the collision operator, describing the interaction between colliding particles (He & Lou, 1997).

This equation is a complicated integro-differential equation because of the complicated collision operator. To efficiently avoid the complexity of this equation, an approximation introduced by Bhatnager, Gross and Krook is frequently used, often referred to as the BKG collision operator. This method expresses that the collisions tend to relax the particle distribution function to an equilibrium value. The approximation is expressed as a time-relaxation term (Bhatnager, et al., 1954):

$$\Omega(f) = -\frac{1}{\tau_c} [f(\mathbf{x}, \mathbf{u}, t) - f^{(eq)}(\mathbf{x}, \mathbf{u}, t)] \quad (4.2)$$

Where  $f^{(eq)}(\mathbf{x}, \mathbf{u}, t)$  denote the equilibrium distribution function.  $\tau_c$  is the relaxation time, the rate which the particle distribution function relaxes back towards equilibrium  $f^{(eq)}$  due to particle collisions. The relaxation time is often defined by the relaxation frequency where  $\omega = \frac{1}{\tau_c}$

In order to be computationally useful, the continuous Boltzmann equation has to be discretized. From the continuous Boltzmann equation with BKG approximation

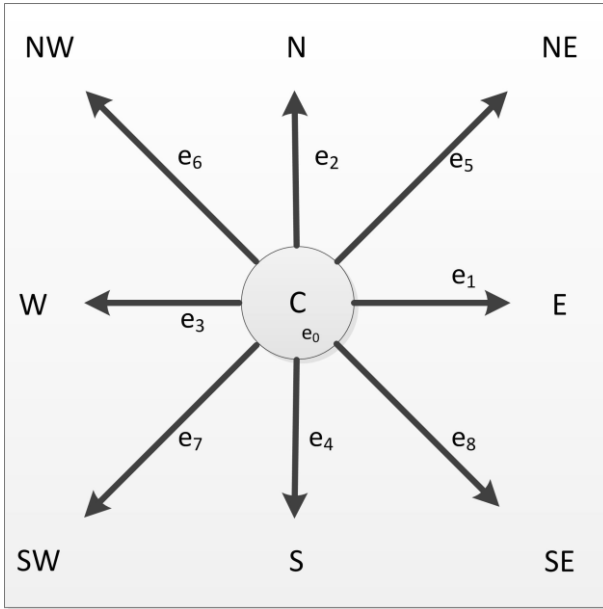
$$\frac{\partial f(\mathbf{x}, \mathbf{u}, t)}{\partial t} + \mathbf{u} \cdot \nabla f(\mathbf{x}, \mathbf{u}, t) = -\omega [f(\mathbf{x}, \mathbf{u}, t) - f^{(eq)}(\mathbf{x}, \mathbf{u}, t)] \quad (4.3)$$

the BKG lattice Boltzmann equation can be derived by a proper discretization of space, velocity space and time. (He & Lou, 1997)

The velocity space is discretized by introducing a finite set of velocity vectors  $\mathbf{e}_i$  for the continuous velocity space  $\mathbf{u}$ . The (velocity) discrete Boltzmann equation reads

$$\frac{\partial f_i(\mathbf{x}, t)}{\partial t} + \mathbf{e}_i \cdot \nabla f_i(\mathbf{x}, t) = -\omega \left[ f_i(\mathbf{x}, t) - f_i^{(eq)}(\mathbf{x}, t) \right] \quad (4.4)$$

A common classification used when discretizing the velocity into discrete velocity vectors, is the DdQq model (Pohl, et al., 2003), where d denotes the spatial dimension, and q the number of lattice velocities. For 2-dimensions, D2Q9 is a common model with 9 directions, shown in figure 3, where the directions are labeled according to compass notation with the corresponding velocity vectors  $\mathbf{e}_i$ . For 3-dimensions the D3Q19 model is commonly used.



**Figure 3: Two-dimensional D2Q9 model**

The equilibrium distribution function  $f_i^{(eq)}$ , represents the particle distribution at a given velocity and density inside a cell at equilibrium, defined by:

$$f_i^{(eq)} = t_p \rho (1 + F_i(\mathbf{u})), \quad F(\mathbf{u}) = \frac{\mathbf{u} \cdot \mathbf{e}_i}{c_s^2} + \frac{1}{2c_s^4} Q_{i\alpha\beta\gamma\delta\epsilon\zeta} \quad (4.5)$$

Where  $c_s$  is the sound speed of a fluid at equilibrium, defined as  $c_s = \frac{c}{\sqrt{3}}$ .  $c$  is the lattice speed defined by  $c = \frac{\Delta x}{\Delta t}$  where  $\Delta x$  is the spatial discretization length, and  $\Delta t$  is the time step (Körner, et al., 2005),  $c$  is usually normalized to 1.  $t_p$  is a weighting for each distribution function.



For the D2Q9 and D3Q19 model the equilibrium function is expressed as

$$f_i^{(eq)} = f_i^{(eq)}(\rho, \mathbf{u}) = t_p \rho \left( 1 + \frac{3}{c^2} \mathbf{e}_i \mathbf{u} + \frac{9}{2c^4} (\mathbf{e}_i \mathbf{u})^2 - \frac{3u^2}{2c^2} \right) \quad (4.6)$$

The time discretization of equation 4.4 is done by an implicit forward Euler, while the spatial discretization is done along the lattice velocities  $c_i$ . The special discretization of lattice Boltzmann causes the distribution to move exactly from one lattice point to the next in one time step.

$$\begin{aligned} \frac{f_i(x, t + \Delta t) - f_i(x, t)}{\Delta t} + \|\mathbf{e}_i\|_2 \cdot \frac{f_i(x + \Delta x_i, t + \Delta t) - f_i(x, t + \Delta t)}{\|\Delta x_i\|_2} \\ = -\omega [f_i - f_i^{(eq)}] \end{aligned} \quad (4.7)$$

With  $\Delta x_i = \mathbf{e}_i \Delta t$  and  $\Delta t = 1$  this results in the explicit lattice Boltzmann equation:

$$f_i(x + \mathbf{e}_i \Delta t, t + \Delta t) - f_i(x, t) = -\omega [f_i - f_i^{(eq)}] \quad (4.8)$$

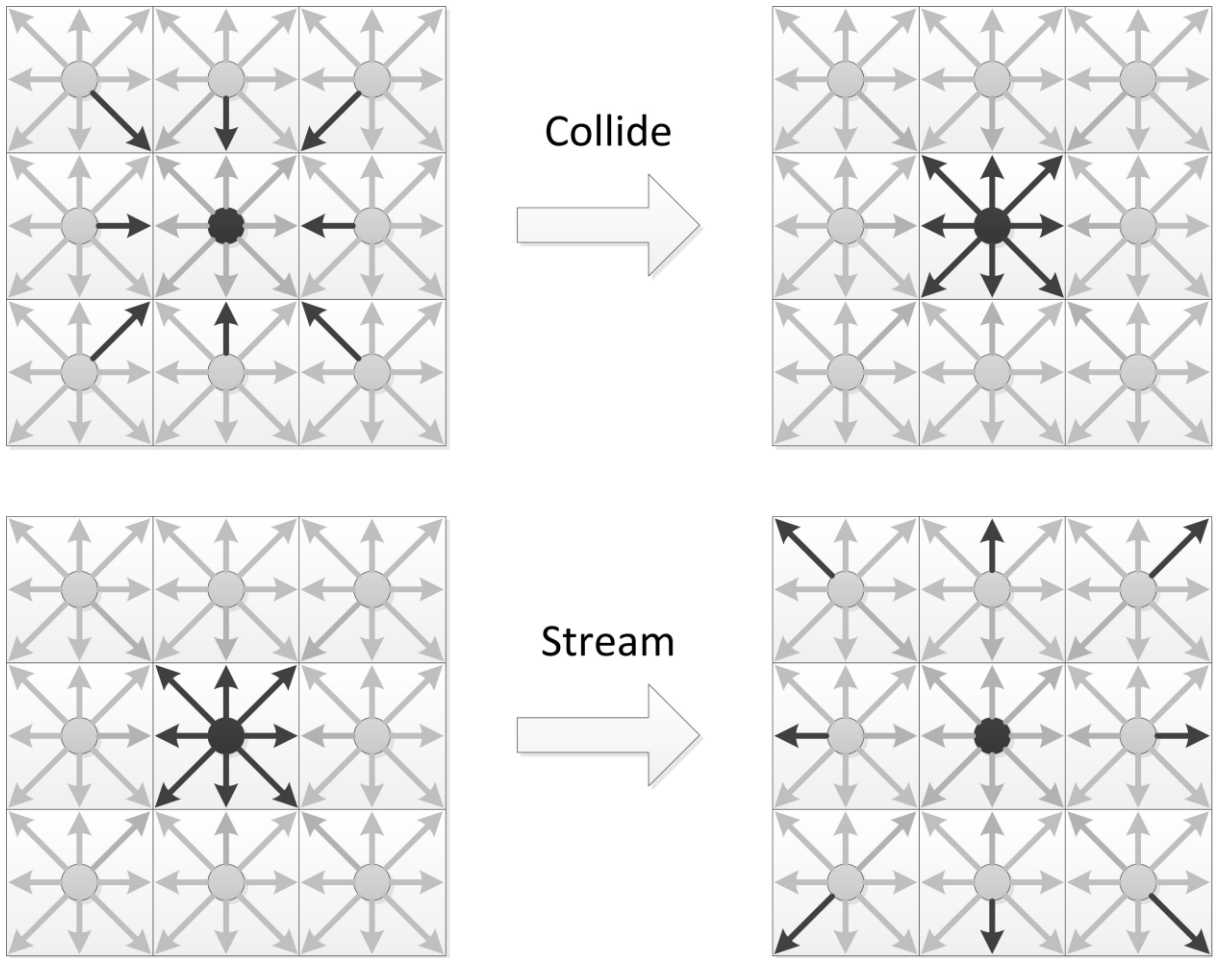
Solving this equation is usually done in two steps,

- The collide step

$$\tilde{f}_i(x, t + \Delta t) = f_i(x, t) - \omega [f_i - f_i^{(eq)}] \quad (4.9)$$

- The streaming step

$$f(x + \mathbf{e}_i \Delta t, t + \Delta t) = \tilde{f}_i(x, t + \Delta t) \quad (4.10)$$



**Figure 4: The LBM collision and streaming step**

In the collision step, fluid particle interactions are modeled and new distribution functions calculated according to the distribution functions of the last time step and the equilibrium distribution functions.

In the streaming step (also called propagation step), fluid particles are streamed from one cell to the neighboring cell.

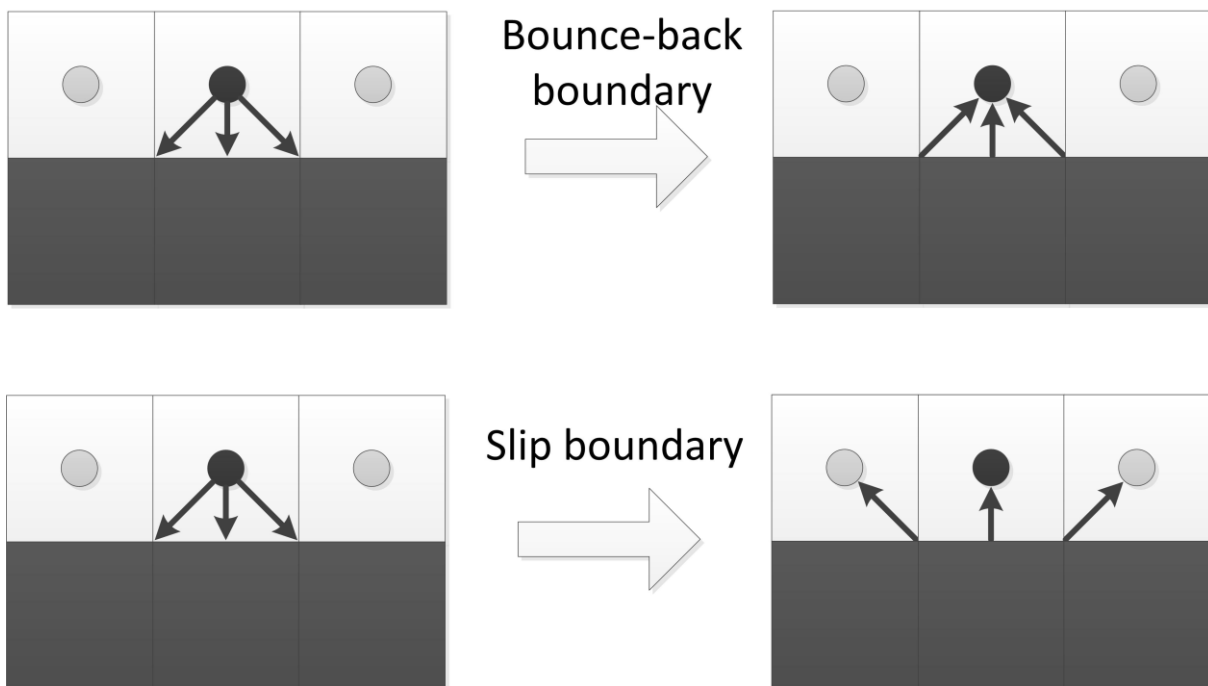
The macroscopic density  $\rho$  and momentum  $\rho u$  can be calculated by summation of the particle distribution functions of a cell.

$$\rho(\mathbf{x}, t) = \sum_i f_i(\mathbf{x}, t) \quad (4.11)$$

$$\mathbf{u}(\mathbf{x}, t)\rho(\mathbf{x}, t) = \sum_i \mathbf{e}_i f_i(\mathbf{x}, t) \quad (4.12)$$

### 4.3. Boundary Conditions

When simulating fluid within a boundary or fluid interacting with solid obstacles, a description of fluid behavior with the boundary is needed. The most common method to handle these interactions is the bounce back boundary condition (also called no-slip boundary condition). The method is aimed at simulation of rough surfaces, where the distribution functions are reflected back in the opposite direction, resulting in zero velocity of the fluid normal and tangential to the wall.

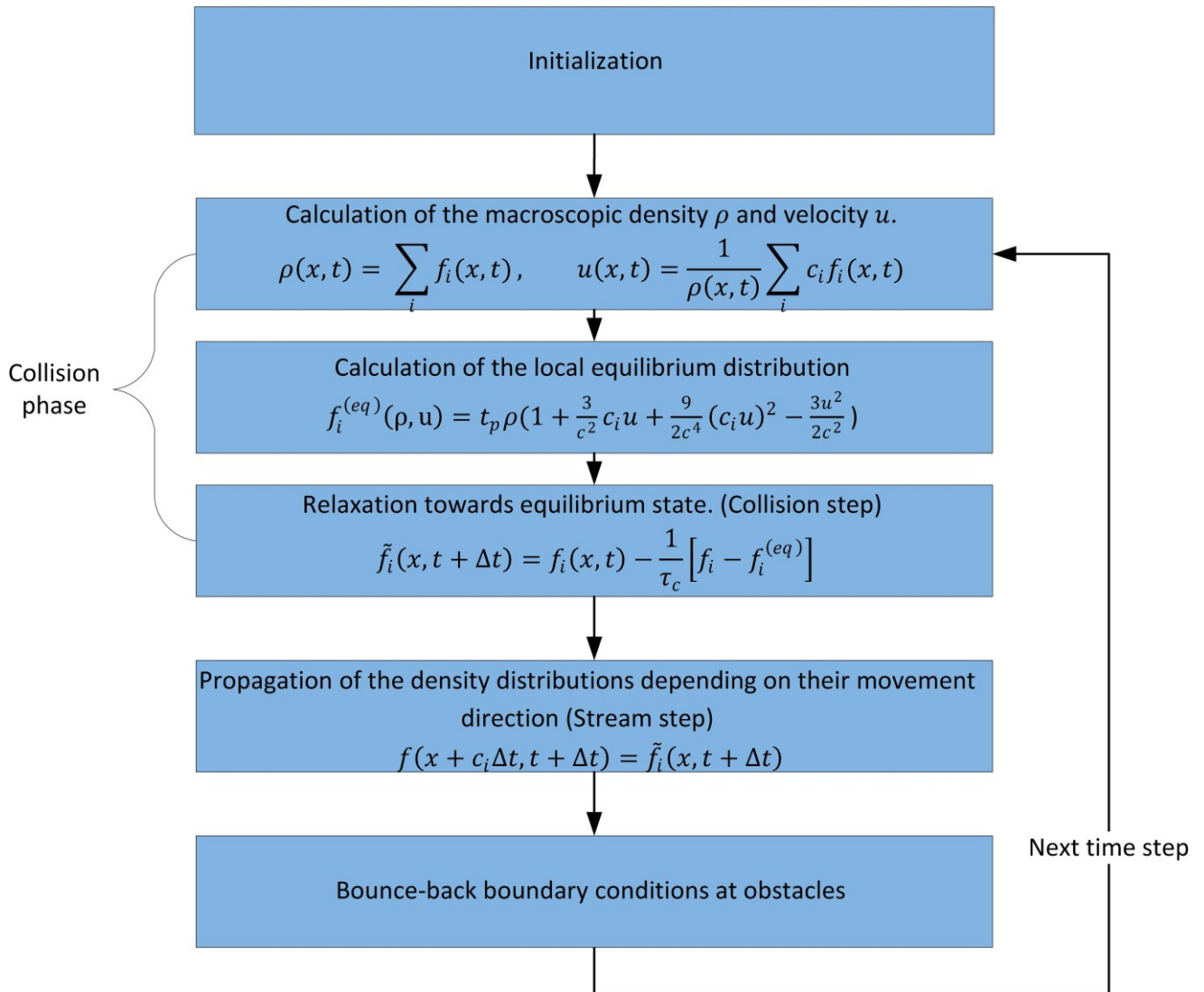


**Figure 5: LBM boundary conditions**

Another boundary condition is the slip boundary condition, which is used for walls without friction. With this boundary condition only the distribution function normal to the wall is reflected, resulting in zero velocity normal to the wall, while the tangential velocity remains unchanged. A usual application for the free slip boundary condition is at the symmetry axes in certain simulated domains, for example in the middle of a mirrored channel flow.

#### 4.4. The algorithm

A basic implementation of the discrete lattice Boltzmann algorithm is shown in figure 6. For each time step, a collision and streaming step is performed.



**Figure 6: Basic algorithm of the lattice Boltzmann method**

The order of the streaming and collision step can be switched, differentiating between LBM update orders as *stream-collide* or *collide-stream*.

## 4.5. Parameterization

A simulation is usually run to represent the physics of a real world problem. For the lattice Boltzmann method, base units are the lattice size and the time step. To convert between real world units and discrete units, one approach is to find the correct units from the dimensionless Reynolds number. (Wolf-Gladrow, 2005)

$$Re = \frac{u_m L}{\nu} \quad (4.13)$$

The viscosity in the discrete system is related to the relaxation time

$$\nu = \frac{(2\tau - 1)}{6} \quad (4.14)$$

Thus the relaxation time is restricted by  $\tau > 0.5$ . Another restriction is for the velocity to be below the lattice speed of sound  $c_s < \frac{1}{\sqrt{3}}$  due to the BKG approximation. However, it was found that the simulations became unstable already for values above 0.1, for low  $\tau$  values.

### 4.5.1. Example

The following example shows the procedure for how to model the flow around a cylinder with Reynolds number 200. Given the following scenario where diameter of cylinder  $D$ , viscosity  $\nu$ , and incoming velocity  $u_m$  is defined as follows.

$$D = 0.01 \text{ m}$$

$$u_m = 0.02 \frac{\text{m}}{\text{s}}$$

$$\nu = 10^{-6} \frac{\text{m}^2}{\text{s}}$$

$$Re = \frac{0.02 * 0.01}{10^{-6}} = 200$$

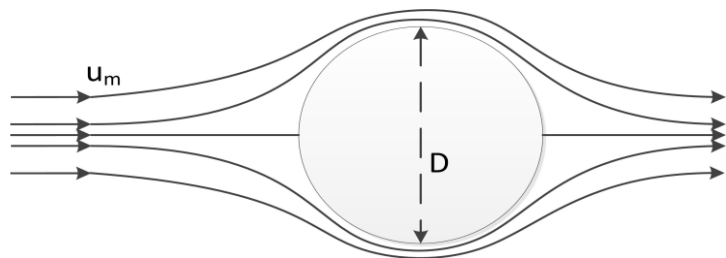


Figure 7: Flow around a cylinder

This is modeled inside a grid with domain size  $\Omega = 256^2$ , the cylinder will be modeled with a diameter  $D' = 32$  lattice points. The relaxation parameter is estimated, with the known restriction that it has to be above 0.5. Try using 0.6. With these parameters defined, the discrete viscosity, discrete lattice spacing and incoming velocity can be calculated.

$$v' = \frac{2 * 0.6 - 1}{6} = 0.0333333, \quad (4.14)$$

$$u' = \frac{Re * v'}{D'} = \frac{200 * 0.0333333}{32} = 0.20833, \quad (4.15)$$

$$\Delta x = \frac{0.01}{32 - 1} = 3.226 * 10^{-4} \quad (4.16)$$

The velocity is above 0.1, so the simulation might be unstable. To get stable values the relaxation time can be reduced or the resolution can be increased.

With the input parameters defined, the simulation can be run to simulate the behavior of a known system. For the result to be interpreted, the length of a discrete time step should be converted to seconds as well. This can be done from

$$\Delta t = \frac{\tau - 0.5}{3} * \frac{\Delta x^2}{v} \quad (4.17)$$

From equation 4.17 it is apparent that the time step length is dependent on the relaxation time. To do the simulation faster, a higher value for the relaxation time can be chosen, as long as the restriction for discrete velocity is satisfied.

The number of time steps needed to simulate a defined number of seconds would then be

$$\text{time steps} = \frac{\text{seconds}}{\Delta t} \quad (4.18)$$

## 5. Implementation

The implementation of the lattice Boltzmann method can be organized in two ways. Collide-stream, where the collision step precedes the streaming step, also referred to as the ‘push’ scheme. Or stream-collide, where the order is reversed, referred to as the ‘pull’ scheme.

The ‘pull’ scheme has been used throughout this thesis.

### 5.1. Indexing

Storing the data for a multi-dimensional lattice can be done in different ways. For a two dimensional lattice, the data can be stored in a two dimensional array, where the data for each node can be accessed using  $\text{node}[x][y]$ . However, since C++ AMP need to iterate over every single node, an easier approach is to store the data in a one dimensional array,  $\text{node}[\text{idx}]$ . Where each node has a unique positive integer value. Then the number of threads needed to run, is the same as the size of the array. In order to map two dimensional data in one dimension, an enumeration function is used. For a two-dimensional lattice, where  $d_x$  and  $d_y$  denote the number of lattice nodes in x and y direction, a specific node can be accessed by the enumeration function:

$$\text{idx} = y * d_x + x \quad (5.1)$$

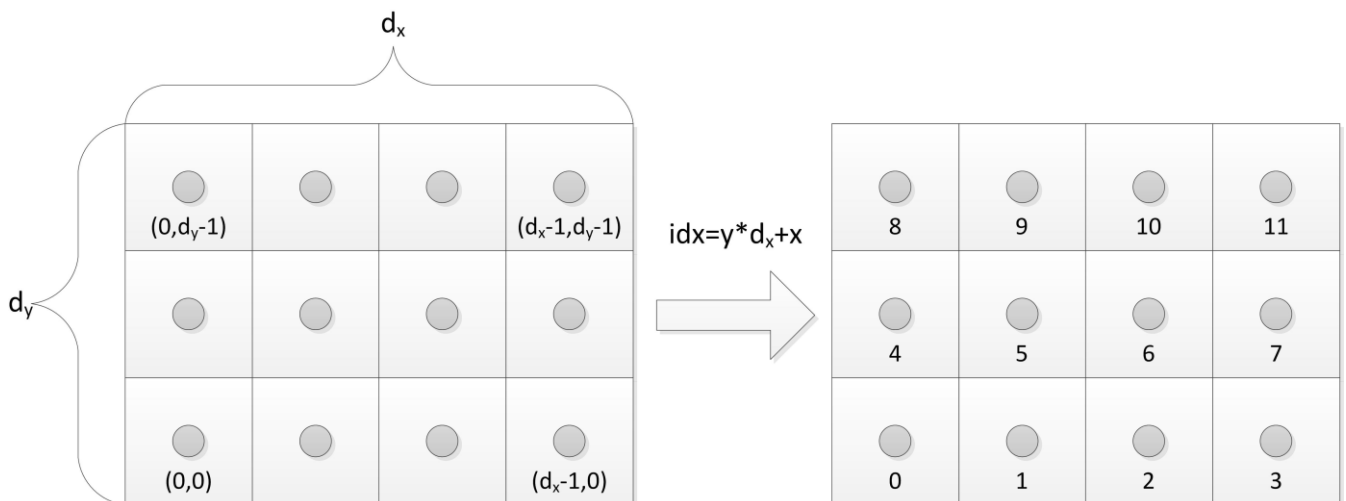
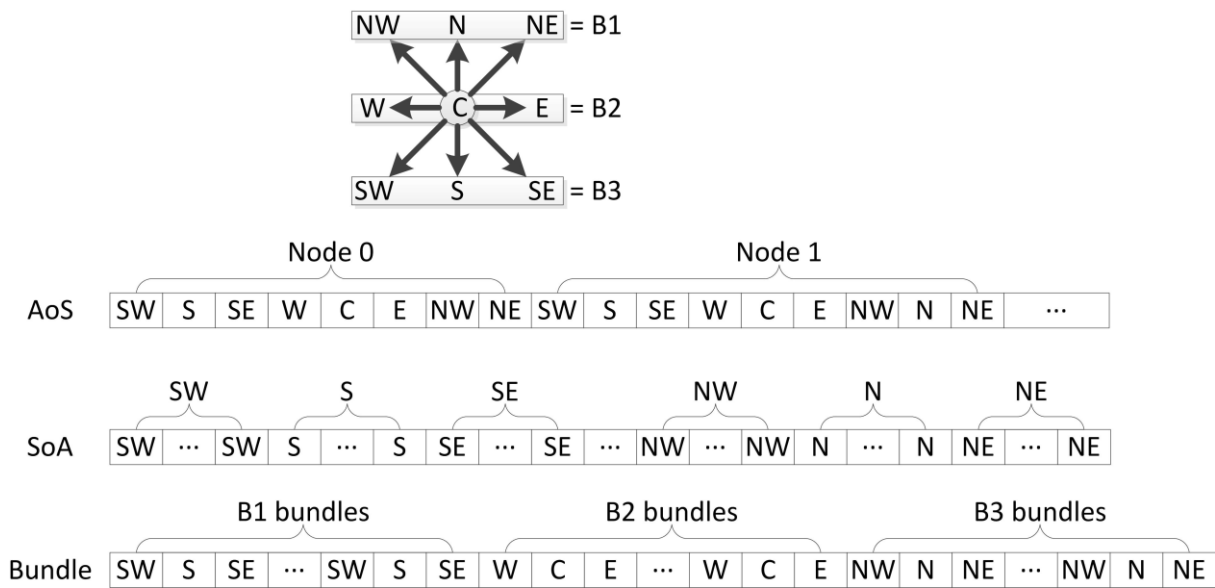


Figure 8: Enumeration function to map a two-dimensional array in a one dimension

## 5.2.Data Layout

Each node need to store a number of distribution values, depending on the discretization model chosen, for the D2Q9 model 9 values has to be stored for every lattice node. There are two natural candidates for the memory layout, Array of Structures and Structure of Arrays. For LBM the AoS layout would store all the distribution functions for a node in a consecutive order, while a SoA layout would store all distribution functions associated with a discrete lattice velocity for all lattice nodes in consecutive order.



**Figure 9: Three memory layouts for the lattice Boltzmann method**

The SoA layout has proven to be a lot more efficient for the lattice Boltzmann methods than AoS, especially for parallel implementations. This is related to the way memory is accessed through caches. Another alternative to a memory layout is the bundle layout proposed in (Mattila, et al., 2007). Here the distribution functions are grouped into small bundles stored consecutively for each node. Due to the order in which the lattice nodes are iterated, this method has been shown to be even faster than SoA implementations, due to less cache misses. (Mattila, et al., 2007)

C++ AMP has a limitation on how many writable arrays it can send to the accelerator. This is limited to 8 for DirectX 11.0 compatible cards, (increased to 64 for DirectX 11.1 compatible cards). Because of this limitation, a full SoA implementation was impossible for the D2Q9 implementation, since this would need 9 arrays.



### 5.3.Algorithms

A common issue with time evolution of dynamic variables is data dependence. For lattice Boltzmann the dynamic variables are the distribution functions. Looking at the two steps in the discrete lattice Boltzmann equation

$$\tilde{f}_i(x, t + \Delta t) = f_i(x, t) - \omega [f_i - f_i^{(eq)}] \quad (5.2)$$

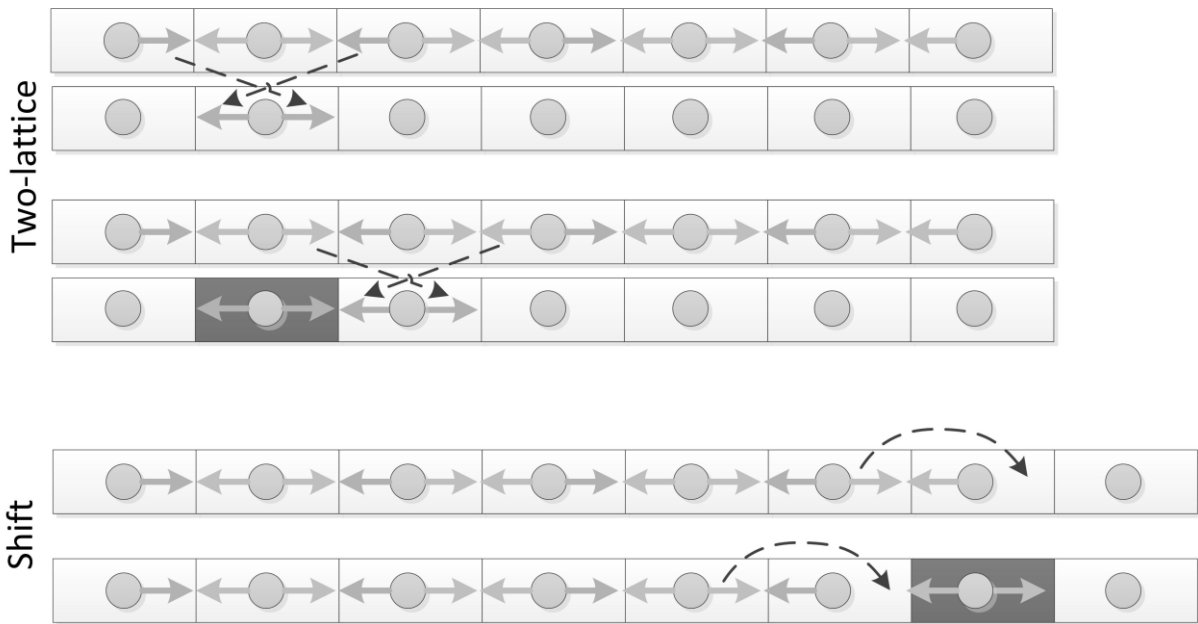
$$f(x + \mathbf{e}_i \Delta t, t + \Delta t) = \tilde{f}_i(x, t + \Delta t) \quad (5.3)$$

The streaming step gives rise to a coupling between the distribution values in neighboring nodes. To deal with this data dependence several algorithms has been proposed. There are at least five basic algorithms that deal with this: the two-lattice, two-step, Lagrangian, shift, and swap algorithms. The two-step and two-lattice algorithms are the two traditional implementation strategies (Mattila, et al., 2006).

In the two-lattice algorithm the data dependency is handled by storing the distribution values in two duplicated lattices. Thus two memory addresses are reserved for each distribution value. At even time steps the distribution values are read from one address and stored in the other, then reversing the procedure for odd time steps. The streaming and collision step can be fused together, so it is only necessary to iterate over the lattice once for each time step.

The two-step lattice takes another approach by iterating over the entire lattice in two steps. To begin with the streaming step is performed for the whole lattice. The values are propagated into neighboring nodes in a carefully selected order to ensure the values are updated from the appropriate values in the previous time step. When the streaming step has been executed for all lattice nodes, the collision step is executed for every lattice node.

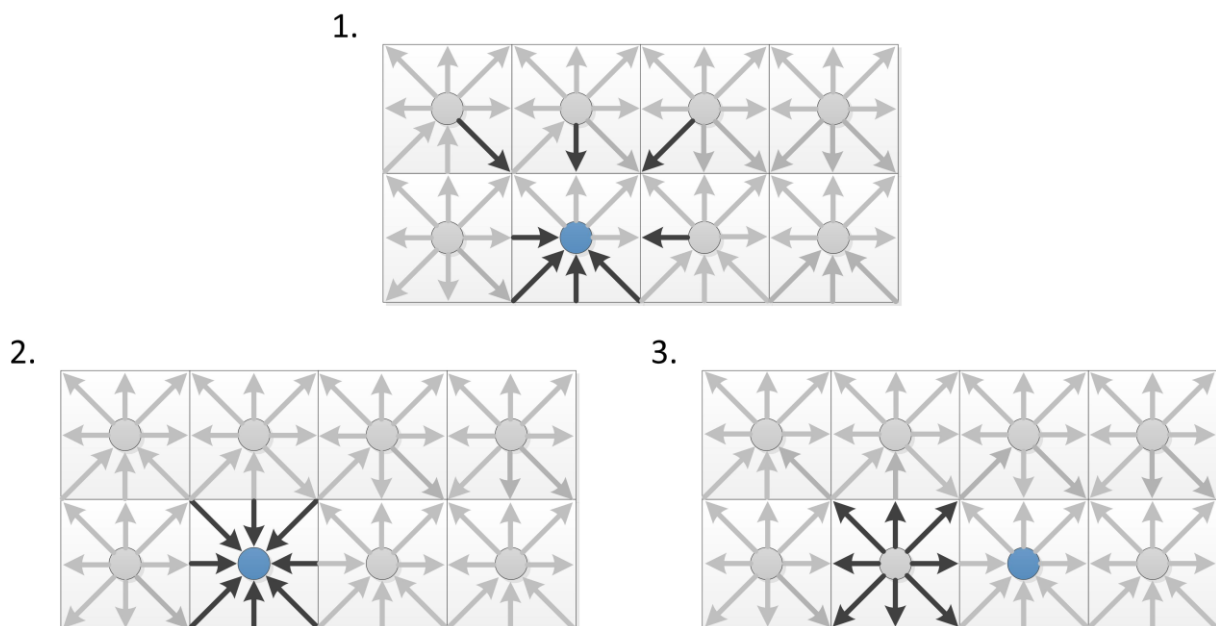
Both of these methods have their defects, the two-lattice algorithm suffer from excessive memory consumption, requiring nearly twice the amount of memory compared to the implementation of the two-step algorithm. The two-step algorithm needs to traverse the lattice twice for each time step, this creates overhead in the run-time of the algorithm. The choice between these two algorithms is a compromise of achieving high computational performance or reduced memory consumption.



**Figure 10: One iteration of the two-lattice and shift algorithm.**

In order to reduce the memory consumption of the two-lattice method, (Pohl, et al., 2003) introduced a compressed grid algorithm, later named the shift algorithm. Like in the two-lattice algorithm, the shift algorithm takes advantage of two memory addresses for each distribution value. But instead of storing the two sets of memory addresses separated, the sets intersect. The memory savings by using this technique is then proportional to the overlapping of the sets. The extended memory set is utilized with a 'buffer' zone, which is an additional allocation of memory for the shift operation. At even time steps the shift algorithm iterates over the lattice nodes, starting from the node with the highest enumeration number, and iterate in decreasing order. The values are stored from the right in the memory, starting in the buffer zone. When all lattice nodes are computed, the buffer zone will have moved to the left side of the array. For odd time steps, the iteration is reversed, starting at the lowest enumeration number, saving the values to the left, and iterating in increasing order. The shift algorithm has the advantage of using almost half the memory of the two-lattice algorithm. However, the algorithm has to be iterated using strictly enforced rules.

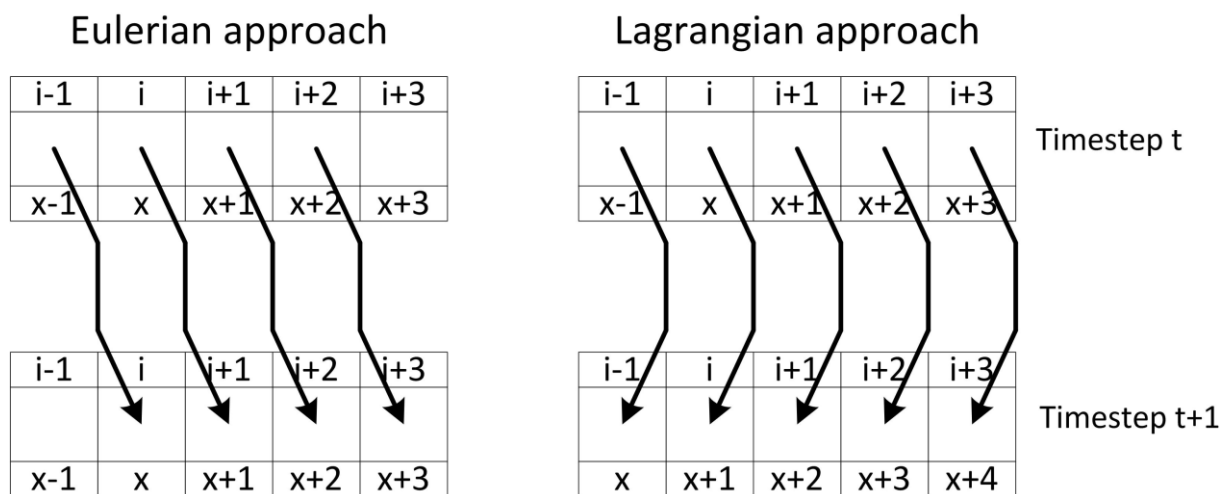
Another implementation of the lattice Boltzmann method is the swap algorithm (Mattila, et al., 2006). Instead of allocating additional memory, as the two-lattice, and in a lesser degree, the shift algorithm does. The swap algorithm takes advantage of a few temporal variables to fuse together the collision and streaming step from the two-step algorithm. As the iteration advances to a particular lattice node, the data dependence involving the node and its neighboring nodes are broken. This is done by exchanging some of the distribution values with the neighboring nodes. After swapping the distribution values, the collision procedure is performed at the node. It must be noted that the distribution values does not reside at the correct memory location after the swap, but at the opposite location. For the D2Q9 model, the north distribution value is resided at the south distribution value etc. This must be taken into account when executing the collision step and the boundary has to be treated in a separate way. This algorithm also needs to be iterated in a strict fashion.



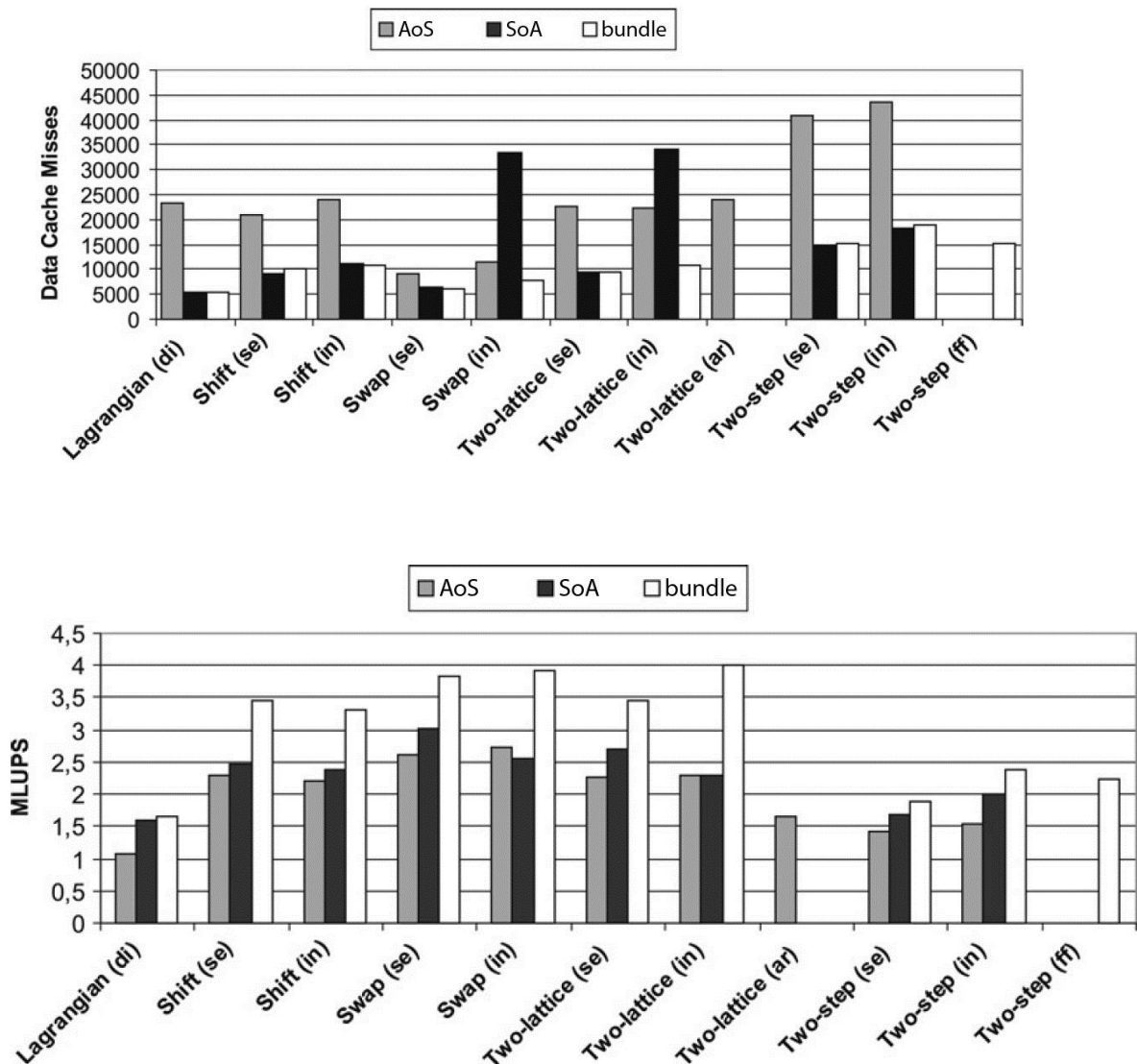
**Figure 11: The swap algorithm has advanced to the blue node. First some of the distribution values are exchanged with those of the neighbors. Second, collision is executed for the node. Finally the algorithm proceeds to the next node. Notice how the lower distribution values are already exchanged.**

All of these algorithms are based on an Eulerian approach, in the frame of reference where the lattice is at rest. In computational terms, the memory address is mapped to fixed physical space coordinates. So the values must migrate to its surrounding neighbors for each time step. Another strategy to implement the LBM is based on a Lagrangian approach, more specifically the Lagrangian algorithm as proposed by (Massaioli & Amati, 2002). With this

approach, each density distribution is represented in the frame of reference where the corresponding particles are at rest. For the memory address representing the density distribution, the values are stored in fixed locations, while the mapping to the physical space coordinates changes with time. So the streaming step is replaced with a coordinate mapping step, the collision is still completely local, but the particle distributions entering the collision process for each lattice node come from different places. Looking at figure 12, for the Lagrangian approach the data stay in the same locations, while the mapping of physical coordinates  $x$  to array index  $i$  is changed.



**Figure 12: A one dimensional view of the Eulerian and Lagrangian approach**



**Figure 13: Performance comparison of the implementations from (Mattila, et al., 2007)**

A comparison of the different algorithms and data layouts performed by (Mattila, et al., 2007) is shown in figure 13. These comparisons were performed for sequential code, and not for parallel implementations. So the results may deviate from the optimal solution for parallel programming. However, a good indication is few cache misses, resulting in better utilization of the memory. By looking at their results, the bundle data layout outperforms the other memory layouts for all algorithms.

When dealing with parallel computing, restrictions on the iteration sequence is hard to fulfill, and this would be better suited for sequential programming. Due to this the two-step, swap and shift algorithm are hard to implement. By looking at the results of the sequential

performance comparison, the two-lattice implementation clearly outperforms the Lagrangian algorithm. However, the Lagrangian algorithm was developed to perform better on a parallel computer using OpenMP, so it would be very interesting to see how this implementation would perform on a GPU. Due to its simplicity however, the two-lattice algorithm with the bundle memory layout was implemented for this thesis.

#### 5.4. Arithmetic precision

In order to run the lattice Boltzmann method on today's graphic cards, the use of single precision floating point arithmetic is necessary. Even when the graphic cards get full support for double precision it will be an advantage to use single precision due to less memory consumption. And the performance of double precision computations is often lower than single precision operations (Itu, et al., 2011). However, a straight forward implementation of lattice Boltzmann using single precision is vulnerable to round-off errors during arithmetic operations. The implementation is prone to round-off errors due to the discrete equilibrium function  $f_i^{(eq)}$ . A solution to reduce the round-off errors and achieve high accuracy using single-precision was proposed by (Skordos, 1993).

Looking at the collision step which reads

$$\tilde{f}_i = f_i - \omega [f_i - f_i^{(eq)}] \quad (5.4)$$

Where

$$f_i^{(eq)} = t_p \rho (1 + F_i(\mathbf{u})), \quad F(\mathbf{u}) = \frac{\mathbf{u} \cdot \mathbf{e}_i}{c_s^2} + \frac{1}{2c_s^4} Q_{i\alpha\beta u\alpha u\beta} \quad (5.5)$$

The terms inside the parenthesis of the equation above are of order  $O(1)$ ,  $O(Ma)$  and  $O(Ma^2)$ . Skordos found that the round-off errors occurred when  $\frac{u}{c}$  became small. With the Mach number getting smaller and smaller, the terms in  $f_i^{(eq)}$  gets more and more disparate in size and the round-off error increases. Single precision floating point numbers have the limitation of seven decimal digits. So it can store values like 1.234567 or  $1.234567 \cdot 10^{-14}$ . However, it is unable to store accurate number where the numbers disparate a lot, like 1,000001234567.

Skordos suggested a simple algebraic transformation to reduce the round off error, by subtracting a constant  $t_p \rho_0$  from each term of the equation. Where  $\rho_0$  is the reference or average density.

Using the  $F(u)$  term for the D2Q9 model and subtracting the constant  $t_p \rho_0$  gives:

$$g_i = f_i - t_p \rho_0, \quad g_i^{eq} := f_i^{eq} - t_p \rho_0 = t_p \left( \delta\rho + \rho \left[ \frac{\mathbf{e}_i u}{c^2} + \frac{(\mathbf{e}_i u)^2}{2c^4} - \frac{u^2}{2c^2} \right] \right) \quad (5.6)$$

The fluctuating part of the density  $\delta\rho = \rho - \rho_0$  is of order  $O(\text{Ma}^2)$ . Hence the terms inside the parenthesis is now of order  $O(\text{Ma})$  and  $O(\text{Ma}^2)$ . This is an improvement of the original equation. The density  $\rho$  and velocity  $u$  can be obtained from  $g_i$  as

$$\rho = \rho_0 + \delta\rho, \quad \delta\rho = \sum g_i, \quad u = \frac{\sum g_i \mathbf{e}_i}{\rho_0 + \delta\rho} \quad (5.7)$$

Due to the simple models tested in this thesis, implementing this was not necessary. The code in appendix 3 uses the standard single-relaxation-time BGK model.

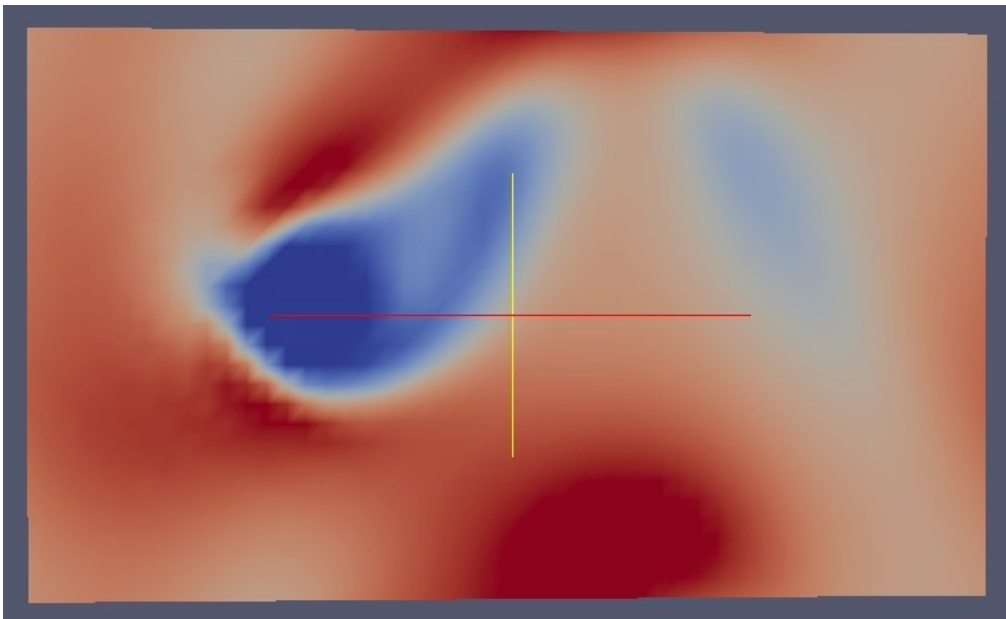




## 6. Visualization

In order to interpret the numerical results a visualization of the fluid flow is necessary. It is also useful to validate the code and to check how the implemented optimizations work compared to the non-optimized methods.

To visual the fluid a free data analysis and visualization program called ParaView was used. This application supports a simple file format that is easy to generate. It also offers a solution to create and play an animation of how the fluid flow changes over time.



**Figure 14: Visualization of a velocity field around a cylinder**

To visualize the flow it is beneficial to view streamlines and velocity vectors, these can easily be generated by ParaView from a velocity field.

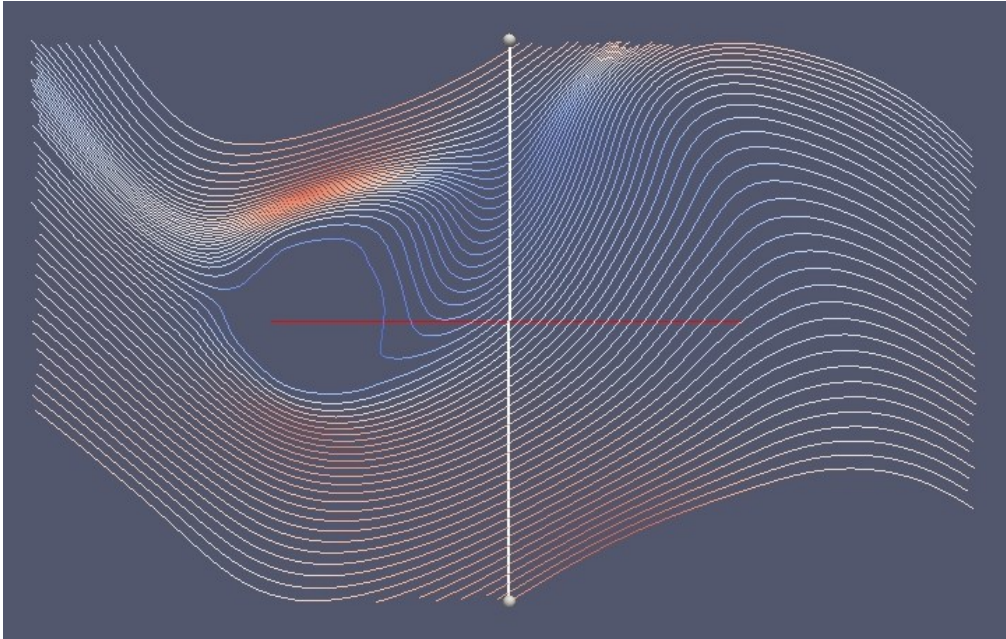


Figure 15: Streamlines created from the velocity field around a cylinder

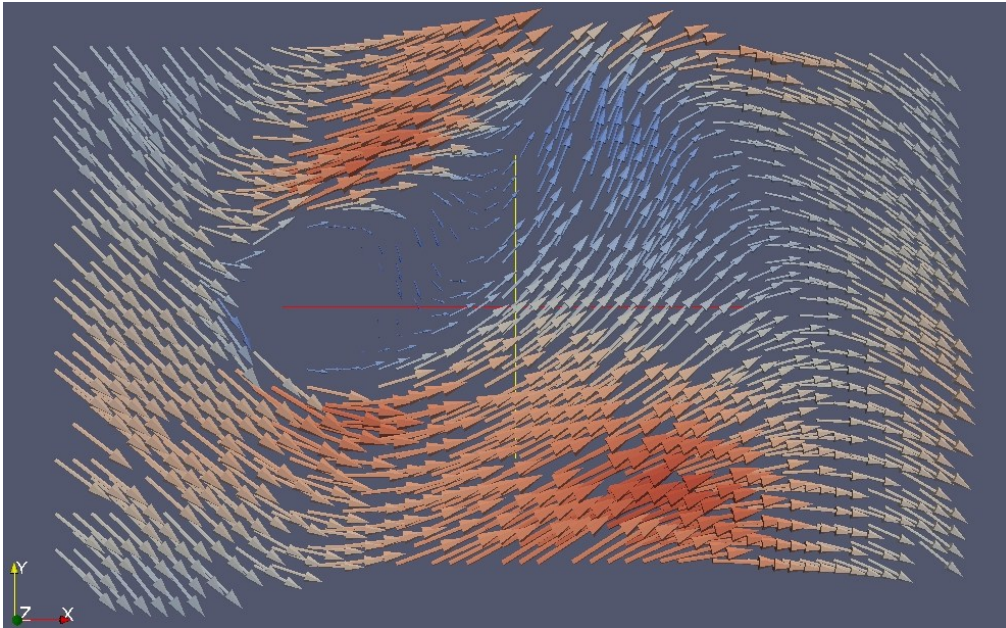


Figure 16: Velocity vectors from the velocity field around a cylinder

## 6.1.Vtk fileformat

The file format ParaView use for visualization is called vtk. This format is built up by 5 parts. (Kitware, u.d.)

- 1) The first part is file version and file identification. It contains a line: # vtk DataFile Version x.x. Where the latest version of the format is 3.0.
- 2) Second part of the file is the header. This contains text that describes the type of data found in the file.
- 3) Next is a description of the file format. This can be either ASCII or BINARY.
- 4) The fourth part describes the structure of the dataset. This start with a line containing the keyword DATASET, followed by a keyword explaining the type of dataset used for the grid.
- 5) The last part describes the dataset properties. Starting with the keyword POINT\_DATA, followed by the number of points. Next a keyword describing what kind of data, this can be scalar, vector, tensor, normal etc and the values for the data in these points.

To animate in ParaView a group of files with the same prefix and a number describing its place in the animation is used, like output01.vtk, output02.vtk, output03.vtk...

An example of a vtk file can be found in appendix 1.



## 7. Performance

The performance for lattice Boltzmann simulations is usually measured in million lattice node updates per second (MLUPS), which indicates how many lattice node collision and streaming steps are performed each second. This measure gives a good comparison between lattice Boltzmann implementations with the same dimension and lattice size.

$$MLUPS = \frac{n \cdot m}{t \cdot 10^6} \quad (7.1)$$

Where  $n$  is the number of lattice nodes,  $m$  is the number of iterations and  $t$  is the time in seconds spent on the streaming and collision steps.

The code developed for this thesis was based on the FORTRAN code `anb.f` from (Bernsdorf, 2008) ported to C++.

During the development of the code there were several major breakthroughs affecting the performance. The table below shows how the parallelization and optimizations affected the performance; all tests are run on a 512x512 lattice on a Nvidia Quadro 1000m graphic card.

**Table 1: Performance gains for optimizations**

Optimization	MLUPS	Speed UP
Unoptimized C++ code based on <code>anb.f</code>	2.78	
Parallelized using the C++ AMP library, with <code>array_view</code>	9.67	3.5x
Change from an <code>array_view</code> container to an <code>array</code> container to gain full control over when memory is transferred between CPU and GPU	32.39	3.3x
Change the memory layout from AoS to a bundle layout	65.66	2.0x
Use a two-lattice implementation instead of the algorithm from <code>anb.f</code>	134.08	2.0x

This shows a total increase of almost 50 times the starting performance. For comparison, the optimized sequential code was able to achieve 9.53 MLUPS.

Development of the code was done with the Nvidia Quadro 1000m graphics card. This is not a good graphics card to demonstrate the true power of GPU computing, as this is a fairly low-end card intended for laptops. To see the true potential, the application was tested on a variety of different computers. The tests were executed using the fully optimized code and single precision floating points. In the table below the results obtained from the tests are presented. A description of the hardware can be found in appendix 2. For comparison the results from a sequential CPU implementations is presented.

**Table 2: Performance on different GPUs**

<b>Grid size</b>	<b>CPU</b>	<b>Quadro 1000m</b>	<b>ATI Radeon™ HD 5770</b>	<b>ATI Radeon™ HD 6870</b>	<b>ATI Radeon™ HD 7970</b>
256x256	9.26	93.22	121.35	202.20	387.66
512x512	9.53	134.08	211.08	570.63	1437.80
1024x1024	10.35	157.83	256.68	870.73	1693.46
2048x2048	10.54	161.72	264.00	1003.35	1780.22
4096x4096	10.63	152.91	39.48	143.75	1831.25

These results show an incredible performance increase, the largest is achieved with the ATI Radeon 7970, which is in the latest generation of high-end cards from ATI. This card achieves a speed increase of over 170 times the speed of a sequential CPU implementation, and a speed increase of over 650 times the un-optimized code. This shows the importance of parallel computing and optimized memory access when dealing with large datasets. One can also see how a steady increase in performance is gained for new generations of hardware.

Notice the low MLUPS for some of the cards when the lattice gets big. This is assumed to be because of the limited memory on these cards. For the lattice with 4096x4096 nodes, the memory requirement is 1242 MB. The GPUs with lower memory than this has to transfer data between CPU and GPU for every time step, which affects the run time tremendously.

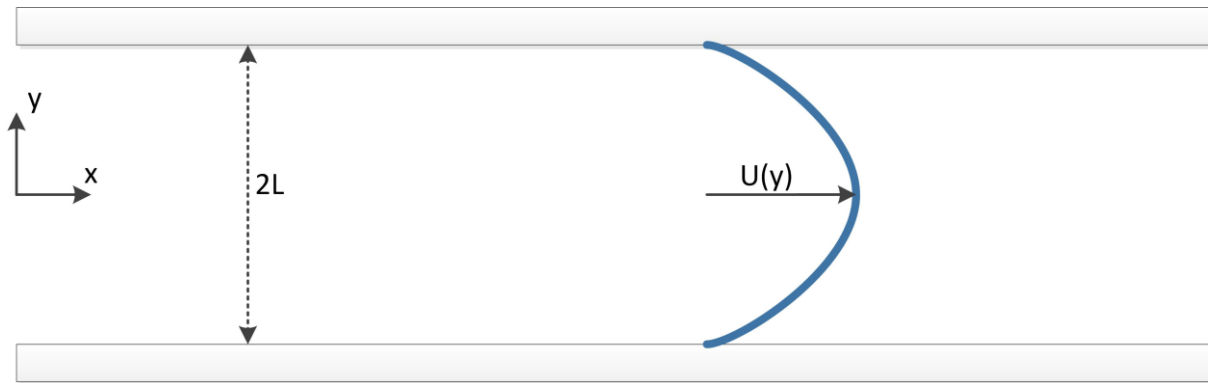
## 8. Validating the code

The numerical correctness of the lattice Boltzmann code was validated by comparing the results obtained with analytical, numerical or experimental methods.

Three studies were conducted 1. Poiseuille flow, 2. Lid driven cavity flow and 3. Uniform flow around a cylinder.

### 8.1.Hagen-Poiseuille flow

The Hagen-Poiseuille flow is a laminar flow through two parallel plates. The numerical simulation is driven by a fixed velocity profile at the inlet of the channel, with the bounce-back boundary condition applied at the walls.

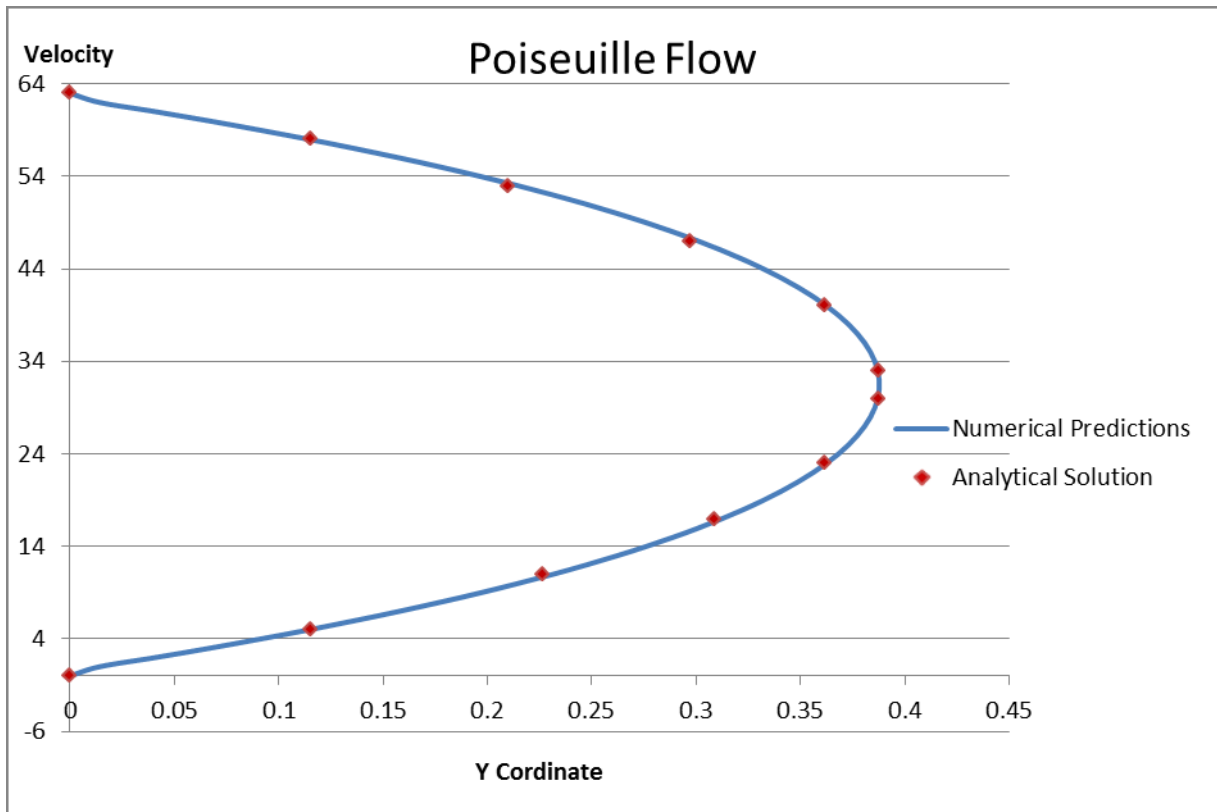


**Figure 17: Visual representation of Hagen-Poiseuille flow**

Since the analytical solution to the Poiseuille flow is available, it is an ideal case to test the validity of the code. The exact solution for the velocity, where the distance between the plates is 2L, is given by

$$U(y) = U_0 \left(1 - \frac{y^2}{L^2}\right) \quad (8.1)$$

Where  $U_0$  is the maximum velocity reached along the longitudinal axis of symmetry of the channel.



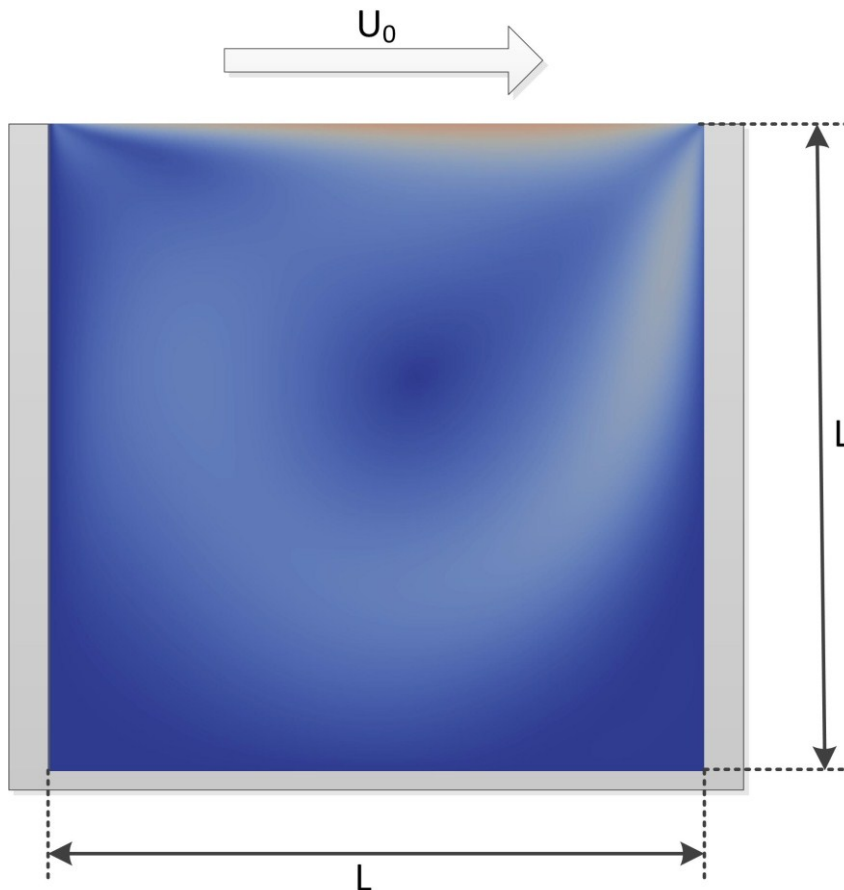
**Figure 18: Comparison of known and simulated velocity profile for the Poiseuille flow**

Looking at these results, the numerical solution obtained from the lattice Boltzmann implementation corresponds very well with the analytical solution.



## 8.2.Lid Driven Cavity Flow

The lid driven cavity flow consist of a two-dimensional viscous flow inside a cavity. The incompressible fluid is bounded by a square enclosure, and the top lid is moving with a constant velocity. The cavity flow is a standard benchmarking problem for CFD, the results here are compared with the numerical results from (Ghia, et al., 1982) found using Navier-Stokes equations.



**Figure 19: A cavity with lengths  $L$ , where the lid moves with velocity  $U_0$**

Present study has been carried out for a wide range of Reynolds numbers, for this thesis tests were conducted for  $Re = 100, 400$  and  $1000$ . The velocity components along the vertical and horizontal center lines for the tested  $Re$  are presented in figure 20 and 21.

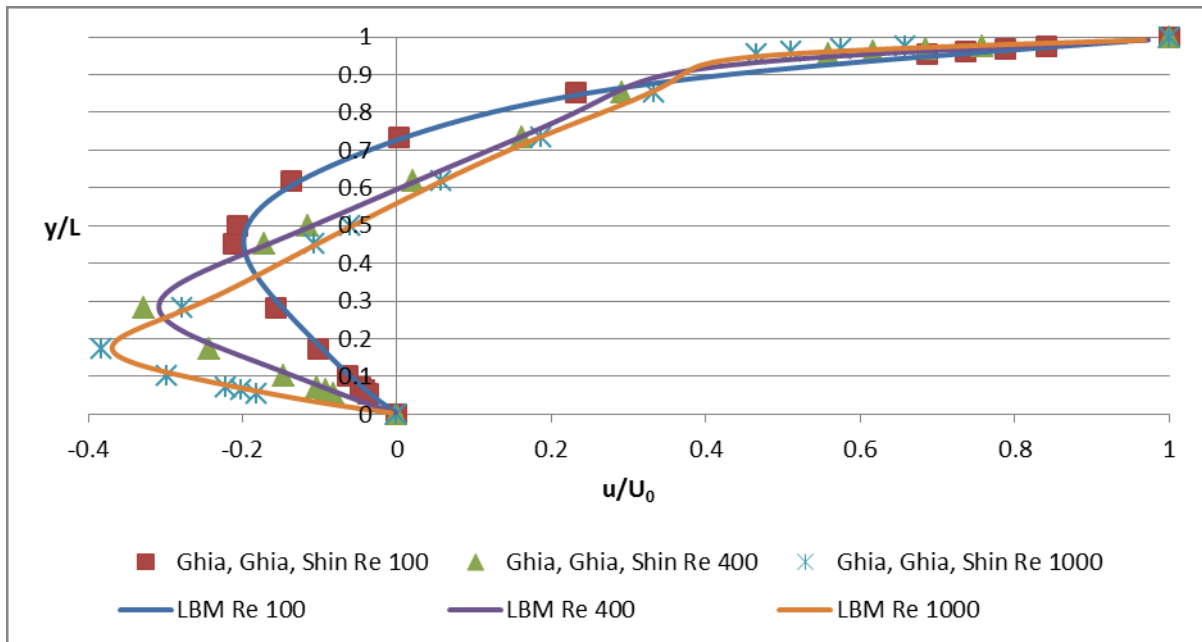


Figure 20: Comparison of u-velocity along vertical lines through the geometric center of the cavity

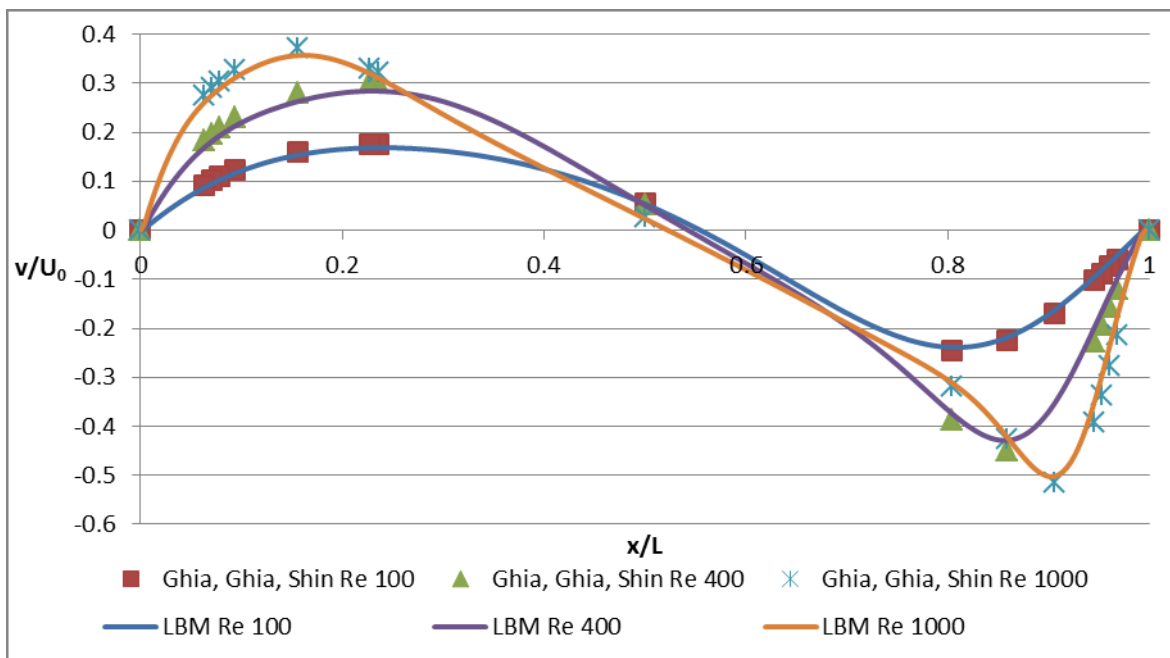


Figure 21: Comparison of v-velocity along horizontal lines through the geometric center of the cavity

The profiles obtained from lattice Boltzmann match really well with the results from (Ghia, et al., 1982).

### 8.3. Uniform flow around a smooth cylinder

For a uniform flow around a smooth cylinder, the fluid flow will change with increasing Reynolds number. The flow regimes experienced with different Re are summarized in the figure below.


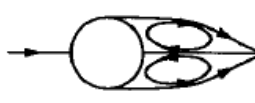



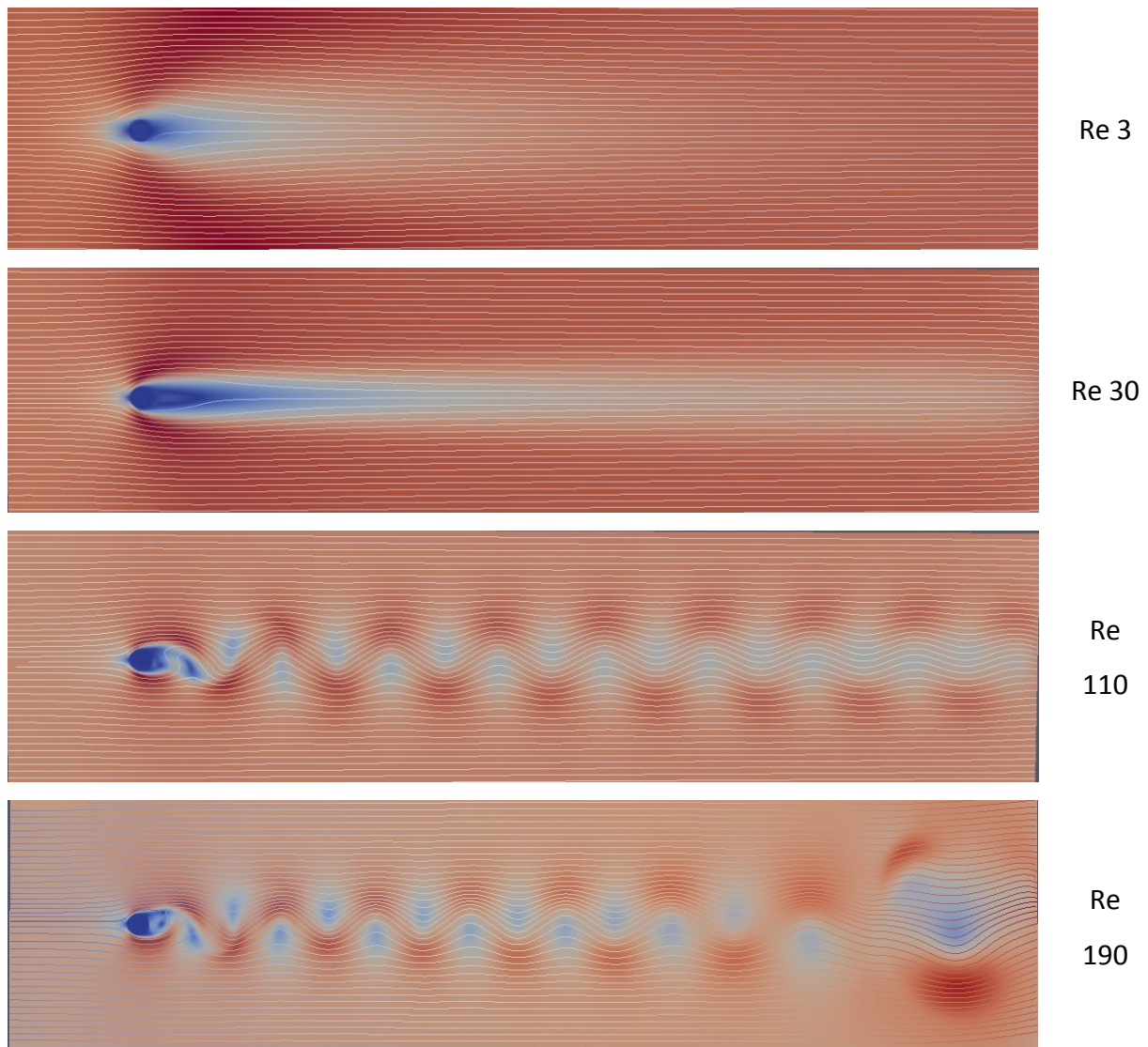
a) 	No separation. Creeping flow	Re < 5
b) 	A fixed pair of symmetric vortices	5 < Re < 40
c) 	Laminar vortex street	40 < Re < 200
d) 	Transition to turbulence in the wake	200 < Re < 300
e) 	Wake completely turbulent. A: Laminar boundary layer separation	300 < Re < 3 × 10 <sup>5</sup>  Subcritical

Figure 22: Regimes of flow around a circular cylinder, taken from (Sumer & Fredsøe, 1997)

For very low Re the stream will be laminar and follow the cylinder walls without separation. When increased above Re 5, separation will occur and two symmetric vortices will be formed behind the cylinder. The length of these vortices will increase up to Re ~45.

When increased above Re 45 the wake behind the cylinder will become unstable and the vortices will be shed alternately to either side at a certain frequency, a phenomenon referred to as vortex shedding. For Reynolds numbers 40 < Re < 200 the vortex street will be laminar behind the cylinder. When increased further 3 dimensional effects are affecting the flow, and the wake becomes turbulent. (Sumer & Fredsøe, 1997)

Tests were conducted for Re up to 190.



**Figure 23: Visualization of stream around a cylinder for different Re, from LBM**

The visualization of the stream from the lattice Boltzmann implementation corresponds well with the different flow regimes. For Re 3 the flow follows the cylinder walls without separation. For Re 30 two symmetrical vortices have been formed behind the cylinder and at Re 110 a laminar vortex street has been formed. The evolution of a flow for Re 110 can be seen in figure 24.

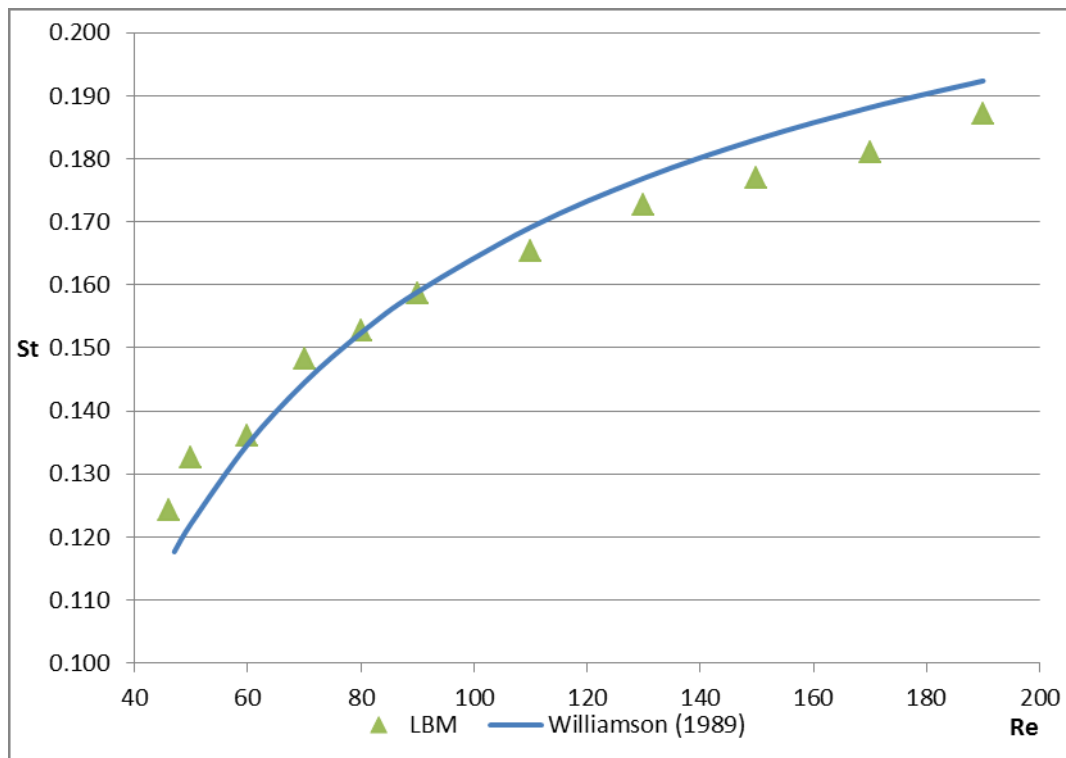
## 8.4. Vortex Shedding Frequency

Research on vortex shedding has been thoroughly studied, and a lot of experimental datasets are available. It's therefore natural to compare the numerical values with the available data. The frequency of vortex shedding is related to the Strouhal number with the following equation

$$St = \frac{fD}{V} \quad (8.2)$$

Where  $f$  is the vortex shedding frequency,  $D$  is the diameter of the cylinder and  $V$  is the velocity of the fluid flow.

The Strouhal number has been calculated for Reynolds number between 45 and 190 for the cylinder. The values have been compared with experimental data from (Williamson, 1989).



**Figure 24: Strouhal's number plotted for experimental and numerical results**

From the graph it is clear that the values correspond quite well for all Reynolds numbers.

Some deviation occurs however, it is assumed that this is related to inaccuracy when counting the vortices, inaccuracy for modeling a cylinder in a discrete grid, and the use of the bounce-back boundary condition.

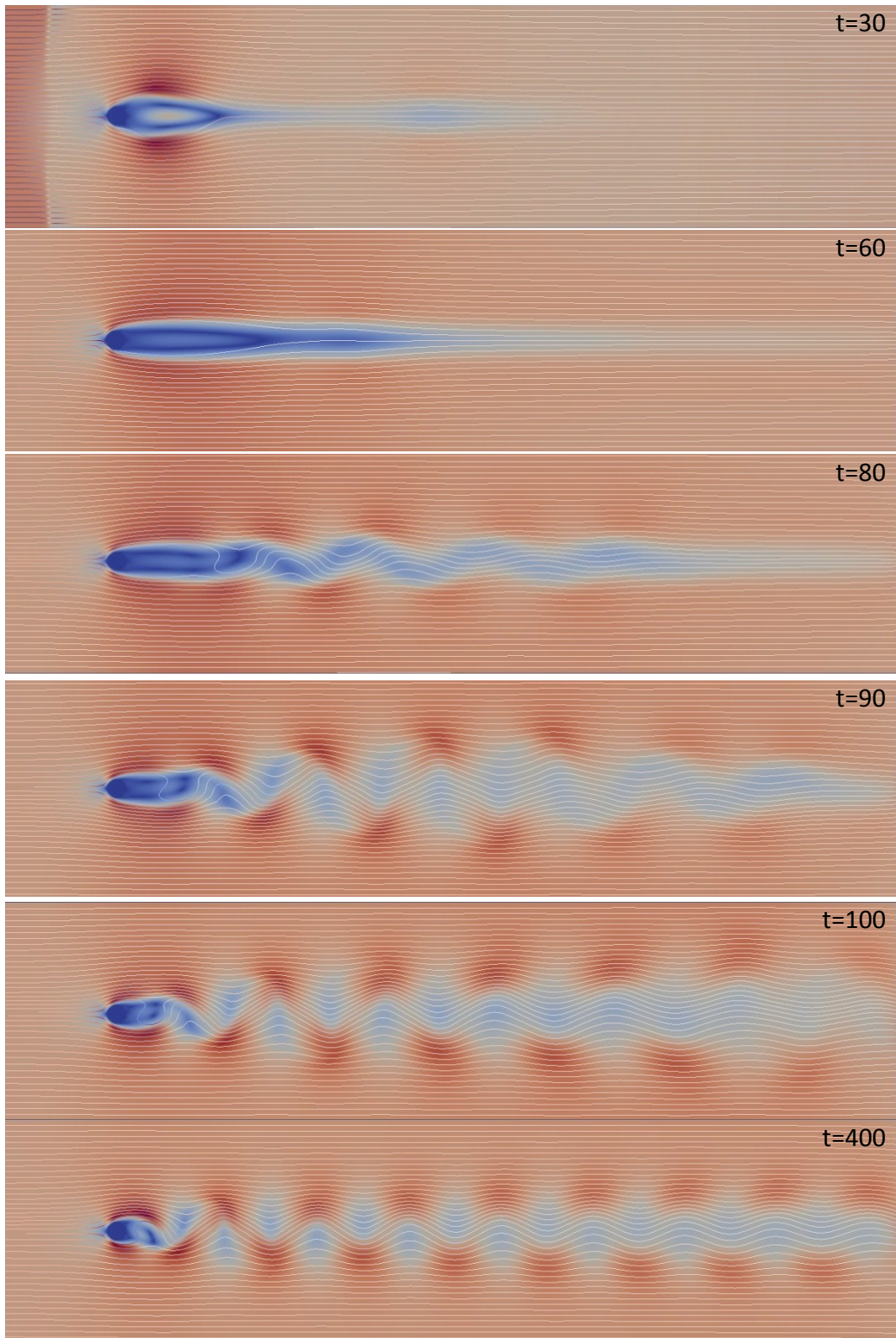


Figure 25: Evolution of vortex shedding at Re 110, time is given in seconds

## 9. Conclusion

In this thesis we investigated algorithms for implementation of the lattice Boltzmann method for parallel execution, and how memory and data dependency is an important factor for optimizing parallel algorithms. Compared to an un-optimized sequential CPU implementation, the parallel optimized version gained a speedup of 650 times, and over 190 times the speed up for an optimized sequential CPU implementation. This shows that parallel computing is highly valuable to be able to achieve the performance potential of today's computers.

Memory is the limiting factor for these implementations however, a grid with 4096 x 4096 lattice points need 2GB ram on the graphic card to operate at maximum speeds. And today's graphic cards usually do not support any more than 2GB.

The validation tests performed demonstrates that the results from the lattice Boltzmann method are good compared to the available data from experimental, analytical and other numerical simulations. However, some deviations occur when trying to model smooth cylinder shapes in the discrete grid.





## 10. Bibliography

Bernsdorf, J., 2008. *A lattice Boltzmann CFD teaching code*. [Online]

Available at: <http://bernsdorf.org/research/anb.f>

[Accessed 7 February 2012].

Bhatnager, P. L., Gross, E. P. & Krook, M., 1954. A Model for Collision Processes in Gases.. *Physical Reviews*, Volume 94, pp. 511-525.

Ghia, U., Ghia, N. & Shin, C. T., 1982. High-Re Solutions for Incompressible Flow Using the Navier-Stokes Equations and a Multigrid Method. *Journal of computational physics*, Volume 48, pp. 387-411.

Hager, G. & Wellein, G., 2011. *Introduction to High Performance Computing for Scientists and Engineers*. s.l.:CRC Press.

He, X. & Lou, L. S., 1997. Theory of the lattice Boltzmann method: From the Boltzmann equation to the lattice Boltzmann equation. *Physical Review*, Volume 56, pp. 6811-6817.

Itu, L. M., Suci, C., Moldoveanu, F. & Postelnicu, A., 2011. Comparison of single and double floating point precision performance for tesla architecture GPUs. 4(2).

Kitware, n.d. *VTK File Formats*. [Online]

Available at: <http://www.vtk.org/VTK/img/file-formats.pdf>

[Accessed February 2012].

Körner, C. et al., 2005. Parallel Lattice Boltzmann Methods for CFD Applications. In: *Numerical Solution of Partial Differential Equations on Parallel Computers*. Berlin: Springer, pp. 439-466.

Massaioli, F. & Amati, G., 2002. Achieving high performance in a LBM code using OpenMP.

Mattila, K., Hyväluoma, J., Timonen, J. & Rossi, T., 2007. Comparison of implementations of the lattice-Boltzmann method. 55(7).

Mattila, K. et al., 2006. An efficient swap algorithm for the lattice Boltzmann method. 176(3).

McNamara, G. R. & Zanetti, G., 1988. Use of the Boltzmann Equation to Simulate Lattice-Gas Automata. 61(20).

Miller, A. & Gregory, K., 2012. *C++ AMP Accelerated Massive Parallelism with Microsoft Visual C++*. s.l.:Microsoft Press.

Mohamad, A. A., 2011. *Lattice Boltzmann Method: Fundamentals and Engineering Applications with Computer Codes*. s.l.:Springer.

Papadopoulos, P., 2008. *ME185: Introduction to Continuum Mechanics*. [Online] Available at: <http://www.me.berkeley.edu/ME280B/notes.pdf>

Pohl, T. et al., 2003. Optimization and profiling of the cache performance of parallel lattice Boltzmann codes. *Parallel Processes*, Volume 13, pp. 549-560.

Qian, Y. H., D'Humières, D. & Lallemand, P., 1991. Lattice BGK Models for Navier-Stokes Equation. Volume 17.

Rauber, T. & Rüniger, G., 2010. *Parallel Programming - for Multicore and Cluster Systems*. s.l.:Springer.

Skordos, P. A., 1993. Initial and boundary conditions for the lattice Boltzmann method. *Physical Review*, 48(6), pp. 4823-4842.

Sumer, M. & Fredsøe, J., 1997. *Hydrodynamics around cylindrical structures*. s.l.:World Scientific Publishing Company.

Williamson, C. H., 1989. Oblique and parallel modes of vortex shedding in the wake of a circular cylinder at low Reynolds numbers. *J. Fluid Mech*, Volume 206, pp. 579-627.

Wolf-Gladrow, D., 2005. *Lattice-Gas Cellular Automata and Lattice Boltzmann Models - An Introduction*. s.l.:Alfred Wegener Institute for Polar and Marine Research.

## A. Appendix 1 – Example of vtk file

Following is an example of a vtk file for the velocity field of a grid with 50x30 points.

```
# vtk DataFile Version 2.0
Lattice_Boltzmann_fluid_flow
ASCII
DATASET RECTILINEAR_GRID
DIMENSIONS 50 30 1
X_COORDINATES 50 float
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
Y_COORDINATES 30 float
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
Z_COORDINATES 1 float
0
POINT_DATA 1500

VECTORS VecVelocity float

U1x U1y U1z
U2x U2y U2z
Uix Uiy Uiz
```

.....

Where  $U_{i_x}$  is the velocity in x direction,  $U_{i_y}$  is the velocity in y direction and  $U_{i_z}$  the velocity in z direction.



## B. Appendix 2 – Hardware specifications

	<b>Cores</b>	<b>Clock speed (MHz)</b>	<b>Compute power (GFLOPS)</b>	<b>Memory (GB)</b>	<b>Memory Bandwidth (GB/s)</b>
<b>Intel Core i7-2630QM</b>	4	2000	60*	12 (DDR3)	21.3
<b>Nvidia Quadro 1000m</b>	96	700	268	2 (DDR3)	28.8
<b>ATI Radeon™ HD 5770</b>	800	850	1360	1 (GDDR5)	76.8
<b>ATI Radeon™ HD 6870</b>	1120	900	2000	1 (GDDR5)	134.4
<b>ATI Radeon™ HD 7970</b>	2048	925	3790	3 (GDDR5)	264

\*No official number found, test performed by a third party.



## C. Appendix 3 – Selected Source Code

```
int twolattice(const struct st_parameters param, const array<struct grid_cell1,
1> &amp_node1, const array<struct grid_cell2, 1> &amp_node2, const
array<grid_cell3, 1> &amp_node3, array<struct grid_cell1, 1> &amp_tmp1,
array<struct grid_cell2, 1> &amp_tmp2, array<struct grid_cell3, 1> &amp_tmp3,
const array<int, 1> &amp_obst)
{
    parallel_for_each(
        // Define the compute domain, which is the set of threads that are
        // created.
        amp_tmp1.extent,
        // Define the code to run on each thread on the accelerator.
        [&amp_node1, &amp_node2, &amp_node3, &amp_tmp1, &amp_tmp2, &amp_tmp3,
        &amp_obst, param](index<1> idx) restrict(amp)
        {
            float c_sq = 1.0f/3.0f;           // Square speed of sound
            float w0 = 4.0f/9.0f;           // Weighting factors
            float w1 = 1.0f/9.0f;
            float w2 = 1.0f/36.0f;

            // Propagate
            int n = ( idx[0] < (param.ny*param.nx-param.nx) ) ?
            idx[0]+param.nx : (param.nx+idx[0] - param.nx*param.ny);
            int s = ( idx[0] > param.nx-1 ) ? idx[0]-param.nx :
            (param.nx*(param.ny-1) + idx[0]);
            int e_x = ( float((idx[0]+1) % param.nx) != 0.00f ) ? 1 : -
            param.nx+1;
            int w_x = ( float((idx[0]) % param.nx) != 0.00f ) ? -1 :
            param.nx-1;

            int e = idx[0]+e_x;
            int w = idx[0]+w_x;
            int nw = n+w_x;
            int ne = n+e_x;
            int sw = s+w_x;
            int se = s+e_x;

            float V0, V1, V2, V3, V4, V5, V6,V7,V8;
            V0 = amp_node2(idx).c;
            V1 = amp_node2(w).e;
            V2 = amp_node3(s).n;
            V3 = amp_node2(e).w;
            V4 = amp_node1(n).s;
            V5 = amp_node3(sw).ne;
            V6 = amp_node3(se).nw;
            V7 = amp_node1(ne).sw;
            V8 = amp_node1(nw).se;

            // Bounceback
            if (amp_obst[idx] == 1) {
                amp_tmp2[idx].c = V0;
                amp_tmp2[idx].e = V3;
                amp_tmp3[idx].n = V4;
                amp_tmp2[idx].w = V1;
                amp_tmp1[idx].s = V2;
                amp_tmp3[idx].ne = V7;
                amp_tmp3[idx].nw = V8;
                amp_tmp1[idx].sw = V5;
            }
        }
    }
}
```

```

        amp_tmp1[idx].se = V6;
    }

    // Collision
    if (amp_obst(idx) == 0) {
        float u_x,u_y;      // Avrage velocities in x and y
direction
        float u[9];        // Directional velocities
        float d_equ[9];    // Equalibrium densities
        float u_sq;        // Squared velocity
particular node
        float local_density; // Sum of densities in a

        // Calculate local density
        local_density = 0.0;
        local_density += V1;
        local_density += V2;
        local_density += V3;
        local_density += V4;
        local_density += V5;
        local_density += V6;
        local_density += V7;
        local_density += V8;
        local_density += V0;

        // Calculate x velocity component
        u_x = (V1+V5+V8-(V3+V6+V7))/local_density;
        // Calculate y velocity component
        u_y = (V2+V5+V6-(V4+V7+V8))/local_density;
        // Velocity squared
        u_sq = u_x*u_x +u_y*u_y;

        // Directional velocity components;
        u[1] = u_x;      // East
        u[2] = u_y;      // North
        u[3] = -u_x;     // West
        u[4] = -u_y;     // South
        u[5] = u_x + u_y; // North-East
        u[6] = -u_x + u_y; // North-West
        u[7] = -u_x - u_y; // South-West
        u[8] = u_x - u_y; // South-East

        // Equalibrium densities
        // Zero velocity density: weight w0
        d_equ[0] = w0 * local_density *(1.0f- u_sq/(2.0f*c_sq));
        // Axis speeds: weight w1
        d_equ[1] = w1 * local_density * (1.0f + u[1] / c_sq
            + (u[1] * u[1]) / (2.0f * c_sq * c_sq)
            - u_sq / (2.0f * c_sq));
        d_equ[2] = w1 * local_density * (1.0f + u[2] / c_sq
            + (u[2] * u[2]) / (2.0f * c_sq * c_sq)
            - u_sq / (2.0f * c_sq));
        d_equ[3] = w1 * local_density * (1.0f + u[3] / c_sq
            + (u[3] * u[3]) / (2.0f * c_sq * c_sq)
            - u_sq / (2.0f * c_sq));
        d_equ[4] = w1 * local_density * (1.0f + u[4] / c_sq
            + (u[4] * u[4]) / (2.0f * c_sq * c_sq)
            - u_sq / (2.0f * c_sq));
        // Diagonal speeds: weight w2
    }

```



```

d_equ[5] = w2 * local_density * (1.0f + u[5] / c_sq
    + (u[5] * u[5]) / (2.0f * c_sq * c_sq)
    - u_sq / (2.0f * c_sq));
d_equ[6] = w2 * local_density * (1.0f + u[6] / c_sq
    + (u[6] * u[6]) / (2.0f * c_sq * c_sq)
    - u_sq / (2.0f * c_sq));
d_equ[7] = w2 * local_density * (1.0f + u[7] / c_sq
    + (u[7] * u[7]) / (2.0f * c_sq * c_sq)
    - u_sq / (2.0f * c_sq));
d_equ[8] = w2 * local_density * (1.0f + u[8] / c_sq
    + (u[8] * u[8]) / (2.0f * c_sq * c_sq)
    - u_sq / (2.0f * c_sq));

amp_tmp2[idx].c = (V0 + param.omega * (d_equ[0] - V0));
amp_tmp2[idx].e = (V1 + param.omega * (d_equ[1] - V1));
amp_tmp3[idx].n = (V2 + param.omega * (d_equ[2] - V2));
amp_tmp2[idx].w = (V3 + param.omega * (d_equ[3] - V3));
amp_tmp1[idx].s = (V4 + param.omega * (d_equ[4] - V4));
amp_tmp3[idx].ne = (V5 + param.omega * (d_equ[5] - V5));
amp_tmp3[idx].nw = (V6 + param.omega * (d_equ[6] - V6));
amp_tmp1[idx].sw = (V7 + param.omega * (d_equ[7] - V7));
amp_tmp1[idx].se = (V8 + param.omega * (d_equ[8] - V8));
    }
}
);
    return 1;
}

```