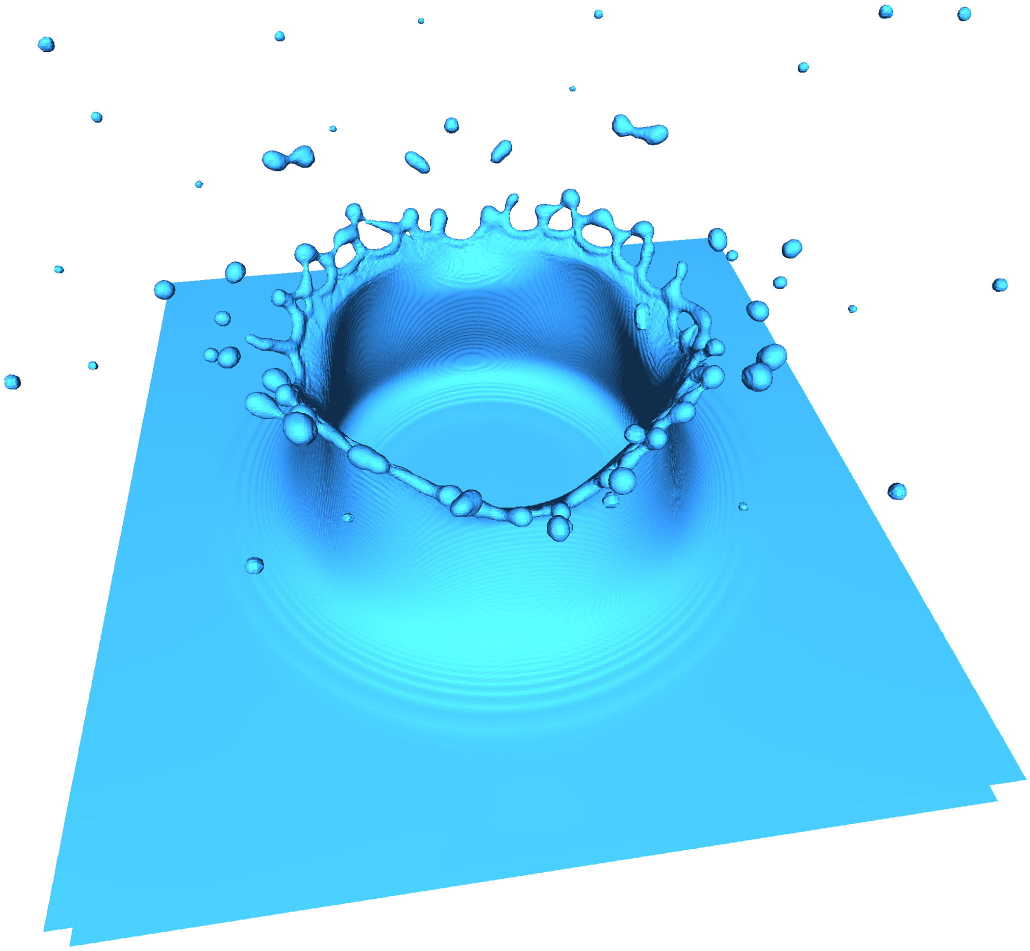
Masterarbeit
zur Erlangung des akademischen Grades
Master of Science (M. Sc.)

High Performance Free Surface LBM on GPUs
Hochleistungs-LBM mit freien Oberflächen auf GPUs



vorgelegt von

Moritz Lehmann
Geboren am: 16. April 1997
Matrikelnummer: 1384796

Abgabedatum: 09.12.2019

Erstprüfer: Prof. Dr. Stephan Gekle
Zweitprüfer: Prof. Dr. Arthur Peeters

Biofluid Simulation and Modeling, Theoretische Physik VI
Fakultät für Mathematik, Physik und Informatik
Universität Bayreuth

UNIVERSITÄT
BAYREUTH

Zusammenfassung

Die Lattice-Boltzmann-Methode (LBM) ist ein etabliertes Werkzeug zur Simulation von Flüssigkeiten, mit dem beliebige Geometrien und Simulationen über einen großen Bereich an Parametern hinweg möglich sind. Durch die Kombination von LBM mit dem Volume-of-Fluid (VoF) Modell können freie Oberflächen simuliert werden. Die algorithmische Struktur von LBM ermöglicht eine hardwarenahe Implementierung auf Grafikprozessoren (GPUs) unter Verwendung ihrer gesamten Leistungsfähigkeit. In dieser Arbeit wird LBM von Grund auf neu in OpenCL implementiert, einer Programmiersprache, die speziell für massiv parallele Hardware entwickelt wurde. Diese Implementierung heißt *FluidX3D*. Darin wird ein umfangreicher Katalog an GPU-spezifischen Optimierungen angewendet (Abschnitt 4), um maximale Effizienz zu erreichen (Abschnitt 5.6), was Simulationen von komplexen freie Oberflächenphänomenen wie Einschläge von Tropfen in nahezu Echzzeit ermöglicht, bei denen die unterschiedlichsten Effekte auftreten, darunter Kronen- und Jetbildung und die Plateau-Rayleigh Instabilität.

Beim Schreiben der Simulationssoftware werden verschiedene Varianten von LBM in Form von Geschwindigkeitsdiskretisierungen und Kollisionsoperatoren untersucht und charakterisiert (Abschnitte 3.2 und 3.3). Die gebräuchlichsten Erweiterungen von LBM werden implementiert, darunter verschiedene Randbedingungen (Abschnitt 3.4), Volumenkräfte (Abschnitt 3.5), die Berechnung von Kräften auf Wände (Abschnitt 3.6), ein Temperaturmodell zur Simulation von thermischer Konvektion (Abschnitt 3.7), die Immersed-Boundary Methode zur Simulation der Wechselwirkungen zwischen Flüssigkeit und Partikeln (Abschnitt 3.8), das Shan-Chen Modell zur Simulation der Koexistenz von Flüssigkeit und Dampf (Abschnitt 3.9) und schließlich der Schwerpunkt dieser Arbeit, das Volume-of-Fluid Modell zur Simulation freier Oberflächen mit einer scharfen Grenzschicht (Abschnitt 6). Die Hauptschwierigkeit von VoF ist neben der Herausforderung, es zusammen mit LBM mit massiver Parallelität auf der GPU zu betreiben, die Berechnung der Oberflächenkrümmung (Abschnitt 7), deren Kern ein Geometrieproblem ist, das Schnittvolumen von einer Ebene und einem Würfel als Teil der *piecewise linear interface construction* (PLIC), für das hier die gesamte analytische Lösung ausgearbeitet und präsentiert wird.

Die Basisfunktionalität von LBM wird anhand zweier analytisch lösbarer Problemstellungen validiert: der Poiseuille-Strömung in einem zylindrischen Kanal und der laminaren Strömung um eine Kugel herum (Abschnitte 9.1 und 9.2).
Zur Validierung des VoF-Modells werden zunächst die Massenerhaltung überprüft und die Genauigkeit verschienener Ansätze zur Krümmungsberechnung charakterisiert (Abschnitte 9.3 und 9.4). Anschließend wird das Modell quantitativ und qualitativ an einem System mit analytisch bekanntem Stabilitätsverhalten überprüft: der Plateau-Rayleigh Instabilität auf einem mit einer kleinen Störung versehenen Zylinder aus Flüssigkeit (Abschnitt 9.5).

Nach der Validierung der Implementierung werden folgende Systeme im Detail untersucht: Mit der LBM-Basisimplementierung wird simuliert, wie viel Kraft auf ein Mikroplastik-Partikel wirkt, das an der Wand eines rechteckigen Mikrokanals befestigt ist (Abschnitt 10.1) – ein Problem, für das es noch keine analytische Lösung gibt. Das VoF-Modell wird verwendet, um den schrägen Aufprall von einem Tropfen und die Formation der Krone beim Einschlag eines Tropfens zu reproduzieren, und die Simulationsergebnisse werden mit experimentellen Beobachtungen verglichen (Abschnitte 10.2 und 10.3).

Zuletzt werden noch einige qualitative Simulationen gezeigt, um zu verdeutlichen, wie vielseitig einsetzbar die hier entwickelte LBM Implementierung ist, und um einige faszinierende emergente Effekte der Hydrodynamik zu zeigen (Abschnitt 10.4).

Summary

The lattice Boltzmann method (LBM) is a well established tool for simulating fluids, with its ability to model arbitrary geometries and function across a wide range of simulation parameters. By combining the LBM with the Volume-of-Fluid (VoF) model, free surfaces can be simulated. The algorithmic structure of LBM allows for hardware-near implementation on graphics processing units (GPUs), using their full capabilities. In this work, LBM is implemented from the ground up in OpenCL, a programming language specifically designed for massively parallel hardware, and this implementation is called *FluidX3D*. A large catalog of GPU-specific optimizations is incorporated (section 4) in order to reach maximum efficiency (section 5.6), allowing for close to real-time simulations of complex free surface phenomena such as drop impacts with all the variety of emerging effects including crown- and jet-formation and the Plateau-Rayleigh instability.

In the process of writing the simulation software, various flavors of the LBM in the form of velocity sets and collision operators are investigated and characterized (sections 3.2 and 3.3). The most common extensions to the LBM are incorporated into the implementation, among them various boundary conditions (section 3.4) volume forces (section 3.5), evaluation of boundary forces (section 3.6), a temperature model for simulating thermal convection (section 3.7), the immersed-boundary method for simulating fluid-particle interaction (section 3.8), the Shan-Chen model for simulating the coexistence of liquid and vapor (section 3.9) and lastly the main focus of this work, the Volume-of-Fluid (VoF) model for simulating free surfaces with a sharp interface (section 6). The main difficulty of the latter, besides the challenge of running it alongside LBM with massive parallelism on the GPU, is surface curvature calculation (section 7), which has a geometry problem at its core, the plane-cube intersection as part of piecewise linear interface construction (PLIC), to which the complete analytic solution is elaborated and presented here.

The base functionality of LBM is thoroughly validated with two setups where the analytic solution is known, Poiseuille flow in a cylindrical channel and laminar flow around a sphere (sections 9.1 and 9.2).
For validating the VoF model, mass conservation is checked and the accuracy of different approaches for curvature calculation is characterized (sections 9.3 and 9.4). Then the model is verified qualitatively and quantitatively on a system with analytically known stability behavior: the Plateau-Rayleigh instability on an undulated cylinder of fluid (section 9.5).

After validation of the implementation, the following systems are studied in detail: With the base LBM implementation, a simulation is done to find the force acting on a microplastic particle attached to the wall of a rectangular microchannel (section 10.1) – a problem where no analytic solution is known yet. The VoF model is used to recreate oblique drop impact and crown splashing setups and the simulation results are compared to experimental observations (sections 10.2 and 10.3).

Lastly, some qualitative simulations are shown in order to demonstrate the vast diversity of use-cases of the here developed LBM implementation and in order to show a few of the fascinating emerging effects in hydrodynamics (section 10.4).

Contents

1 Introduction 8

2 List of physical Quantities and Nomenclature 9

3 The Lattice Boltzmann Method 10
- 3.1 LBM in a Nutshell . . . 10
- 3.2 Velocity Sets . . . 11
- 3.3 Collision Operators . . . 13
  - 3.3.1 Single Relaxation Time (SRT) . . . 13
  - 3.3.2 Two Relaxation Time (TRT) . . . 13
  - 3.3.3 Multi Relaxation Time (MRT) . . . 14
- 3.4 Boundaries in LBM . . . 19
  - 3.4.1 Equilibrium Boundaries . . . 19
  - 3.4.2 Non-moving Bounce-Back Boundaries . . . 19
  - 3.4.3 Moving Bounce-Back Boundaries . . . 19
- 3.5 Volume Force . . . 20
  - 3.5.1 Volume Force with SRT . . . 20
  - 3.5.2 Volume Force with TRT . . . 20
  - 3.5.3 Volume Force with MRT . . . 21
- 3.6 Forces on Boundaries . . . 22
- 3.7 Temperature for simulating thermal Convection . . . 23
- 3.8 Immersed-Boundary Method on the GPU . . . 24
  - 3.8.1 Velocity Interpolation . . . 24
  - 3.8.2 Force Spreading . . . 24
  - 3.8.3 GPU Implementation Notes . . . 25
- 3.9 Shan-Chen . . . 26
  - 3.9.1 Theory . . . 26
  - 3.9.2 GPU Implementation Notes . . . 26

4 LBM on the GPU 27
- 4.1 Challenges and Opportunities on parallel Hardware . . . 27
- 4.2 Glossary . . . 28
  - 4.2.1 GPU Kernel . . . 28
  - 4.2.2 OpenCL Memory Model . . . 28
  - 4.2.3 Memory Coalescence . . . 28
  - 4.2.4 Branching on GPUs . . . 29
  - 4.2.5 Race Conditions . . . 29
  - 4.2.6 Measuring LBM Performance – MLUPs/s . . . 29
- 4.3 Special LBM Optimizations on GPUs . . . 30
  - 4.3.1 Bottlenecks: Memory Bandwidth and Amount of Memory . . . 30
  - 4.3.2 Eliminating non-essential Memory Transfers . . . 30
  - 4.3.3 Avoidance of PCIe Data Transfer . . . 30
  - 4.3.4 Data Layout . . . 31
  - 4.3.5 Choice of Swap Algorithm . . . 31
  - 4.3.6 Why local/shared Memory is not useful in LBM . . . 32
  - 4.3.7 Direct/Indirect Memory Addressing . . . 33
  - 4.3.8 Micro-Optimization and Loop Unrolling . . . 33
  - 4.3.9 Arithmetic Optimization by exploiting numerical Loss of Significance . . . 34
  - 4.3.10 OpenCL Code Injection at Runtime . . . 35
  - 4.3.11 16-bit Floating-Point Storage for DDFs . . . 36

5 The *FluidX3D* Simulation Software **37**
5.1 Choice of OpenCL 37
5.2 List of GPU Kernels 37
5.2.1 Without Extensions 38
5.2.2 With Temperature 38
5.2.3 With Immersed-Boundary 38
5.2.4 With Shan-Chen 38
5.2.5 With free Surface 38
5.3 Multi-GPU Communication Requirements 38
5.4 Source File Dependencies 39
5.5 Example Simulation Setup: Crown Splashing from Section 10.3 41
5.6 Roofline Model and Implementation Efficiency 42

6 Volume-of-Fluid on the GPU **45**
6.1 Overview 45
6.2 Interface Advection 45
6.3 Flag Handling 46
6.3.1 Kernel 1: Modified stream_collide() 46
6.3.2 Kernel 2: surface_1() 46
6.3.3 Kernel 3: surface_2() 47
6.3.4 Kernel 4: surface_3() 47

7 Curvature Calculation for modeling Surface Tension **48**
7.1 Analytic Curvature of a Paraboloid 48
7.2 Approximation over Volume beneath Paraboloid 49
7.2.1 Anisotropic Approximation over Volume beneath Paraboloid (approximation, very fast) . 49
7.2.2 Isotropic Approximation over Volume beneath Paraboloid (Failure) 50
7.3 Curvature Calculation via Paraboloid Fit 51
7.3.1 Calculating the Interface Normal Vector from a 3^3 Neighborhood 51
7.3.2 Curvature from Least-Squares Paraboloid Fit 51
7.3.3 Obtaining neighboring Interface Points: PLIC Point Neighborhood 52
7.3.4 Obtaining neighboring Interface Points: Marching-Cubes (Failure) 53
7.4 Piecewise Linear Interface Construction (PLIC) 54
7.4.1 Plane-Cube Intersection 54
7.4.2 Plane-Sphere Intersection 57

8 Parametrization Procedure **58**

9 Error Validation **59**
9.1 Poiseuille Flow in 2D and 3D 59
9.1.1 Parametrization 59
9.1.2 Analytic Solution 59
9.1.3 Error Definition and Convergence Criteria 59
9.1.4 Simulations 60
9.2 Forces on Boundaries via Stokes Drag 63
9.2.1 Analytic Solution 63
9.2.2 Strategy 63
9.2.3 Error Definition and Convergence Criteria 64
9.2.4 Simulation Parameters 64
9.2.5 Results – $R = const$, L is varied 65
9.2.6 Results – $L/R = const$, R is varied 65
9.2.7 Results – Velocity Field Errors 66
9.3 VoF Mass Conservation Test 68
9.4 Curvature Calculation Error 69
9.5 Plateau-Rayleigh Instability of a perturbed Cylinder of Fluid 71

10 Simulations and Results **76**
10.1 Force on a Particle attached to the Wall of a rectangular Microchannel . . . 76
10.1.1 Experimental Setup . . . 76
10.1.2 Poiseuille Flow in a rectangular Channel . . . 77
10.1.3 Estimation of the expected Force on the Particle . . . 77
10.1.4 Strategy and Simulation Setup . . . 78
10.1.5 Results . . . 79
10.2 Oblique Drop Impact . . . 84
10.2.1 Setup . . . 84
10.2.2 Difficulties arising from the Setup . . . 85
10.2.3 Simulation Results . . . 85
10.3 Crown Formation by Drop Impact on a shallow Pool . . . 86
10.3.1 Setup . . . 86
10.3.2 Simulation Results . . . 87
10.4 Simulations to demonstrate the Diversity of LBM Use-Cases . . . 88

11 Outlook: Microplastic Transport Mechanisms **93**

12 Conclusions **94**

13 References **95**

14 Acknowledgements **100**

15 Appendix: PLIC Inversion with Mathematica **101**

16 Eidesstattliche Erklärung **102**

1 Introduction

Fluid dynamics still is one of the millennium problems with the Navier-Stokes equations remaining unsolved to this day for the vast majority of cases. It covers all length scales, from astrophysics, the chaotic swirls and bands in the atmosphere of Jupiter, across turbulence in the movement of water in rivers or natural gas in pipelines, down to the microscopic scale where bacteria have developed asymmetric movement strategies to circumvent the time-reversibility of flow in the Stokes-limit. Analytic solutions only exist for simplified cases of the equations and then only with very few boundary geometries. Once free surfaces – i.e. the dynamic interface between a liquid and a gas – are added, for example when a raindrop impacts a puddle, theory reaches to its end. We are surrounded by these fascinating and seemingly simple free surface phenomena every day, yet theoretical understanding still has a long way to go.

Making theoretical predictions about such complex systems however is not entirely out of reach due to the since the 1960s exponentially growing capabilities of in silico computation [1]. Simulating a fluid is a parallelizable problem, meaning the work can be split up to many independent processors, each calculating only a small part of the whole. With the rise of graphics processing units (GPUs) in recent years [2, 3] – mainly driven by the computer game industry – single silicon chips with parallel compute power in the order of 10 $TFLOPs/s$ (10 trillion floating-point operations per second) have become affordable to the general public. Especially the memory amount and memory bandwidth available to GPUs has made a great leap with some models now offering 48 GB of memory and a bandwidth beyond 1 TB/s. In terms of both compute power and memory bandwidth, CPUs are lagging behind by about an order of magnitude.

While other fluid solvers, such as finite elements, already are parallelizable on the CPU, the lattice Boltzmann method (LBM), which is used in this work, fits especially well to run on GPUs – with some microarchitectures like Nvidia Volta even allowing for almost perfect efficiency without losses. This is in large contrast compared to traditional CPU-implementations of the LBM, most of which run at single-digit efficiency due to losses from parallelization and communication between CPUs, additionally to memory access being an order of magnitude slower on CPUs.
In this work, LBM is written from the ground up in OpenCL, the industry-standard programming language for parallel hardware, including GPUs, but also multi-core CPUs and field programmable gate arrays (FPGAs). This way, it is possible to measure performance impact of every part of LBM individually and figure out which optimization strategies to apply for which part of the code.

Not only are the many varieties of the basic LBM implementation in the form of velocity sets and collision operators compared, but LBM is also combined with a variety of extensions such as temperature, the Shan-Chen method, the immersed-boundary method and lastly the main focus of this work, the Volume-of-Fluid (VoF) model for simulating free surfaces with a sharp interface. VoF is especially difficult to integrate in a completely parallelized manner. With the implementation issues put aside, the big challenge of VoF comes down to surface curvature calculation on a Cartesian lattice, which in its core has an until now unsolved geometry problem, the plane-cube intersection, to which the full analytic solution is elaborated here.

2 List of physical Quantities and Nomenclature

quantity	SI-units	defining equation(s)	description	chapter
$\vec{x}$	m	$\vec{x} = (x, y, z)^T$	3D position in Cartesian coordinates	3
t	s	–	time	3
Δx	m	$\Delta x := 1$	lattice constant (in lattice units)	3
Δt	s	$\Delta t := 1$	simulation time step (in lattice units)	3
c	$\frac{m}{s}$	$c := \frac{1}{\sqrt{3}} \frac{\Delta x}{\Delta t}$	lattice speed of sound (in lattice units)	3
ρ	$\frac{kg}{m^3}$	$\rho = \sum_i f_i$	mass density	3
p	$\frac{kg}{m\,s^2}$	$p = c^2 \rho$	pressure	3
$\vec{u}$	$\frac{m}{s}$	$\vec{u} = \sum_i \vec{c}_i f_i$	velocity	3
f_i	$\frac{kg}{m^3}$	(1)	density distribution functions (DDFs)	3
f_i^{eq}	$\frac{kg}{m^3}$	(3)	equilibrium DDFs	3
i	1	$0 \leq i < q$, $\vec{e}_i = -\vec{e}_{\bar{i}}$	LBM streaming direction index	3
q	1	$q \in \{7, 9, 13, 15, 19, 27\}$	number of LBM streaming directions	3.2
$\vec{c}_i$	$\frac{m}{s}$	(11)	streaming velocities	3.2
$\vec{e}_i$	m	$\vec{e}_i = \vec{c}_i \, \Delta t = -\vec{e}_{\bar{i}}$	streaming directions	3.2
w_i	1	(10), $\sum_i w_i = 1$	velocity set weights	3.2
τ	s	$\tau = \frac{\nu}{c^2} + \frac{\Delta t}{2}$	LBM relaxation time	3.3
ν	$\frac{m^2}{s}$	$\nu = \frac{\mu}{\rho}$	kinematic shear viscosity	3.3
μ	$\frac{kg}{m\,s}$	$\mu = \rho \, \nu$	dynamic viscosity	3.3
Ω_i	$\frac{kg}{m^3}$	(4)	LBM collision operator	3.3
Λ	1	(16)	TRT 'magic parameter'	3.3.2
M	$q \times q$	(29)-(34)	LBM MRT transformation matrix	3.3.3
S	$q \times q$	$S := diag(\frac{\Delta t}{\tau_i})$	LBM MRT relaxation matrix	3.3.3
$\vec{f}$	$\frac{kg}{m^2\,s^2}$	$\vec{f} = \frac{\vec{F}}{V}$	force per volume	3.5
$\vec{F}$	$\frac{kg\,m}{s^2}$	$\vec{F} = m \, \vec{a} = \vec{f} \, V$	force	3.5
F_i	$\frac{kg}{m^3\,s}$	(40)	LBM forcing terms	3.5
$\vec{p}$	$\frac{kg\,m}{s}$	$\vec{p} = V \, \rho \, \vec{u}$	momentum	3.6
V	m^3	$V = L_x \, L_y \, L_z$	simulation box volume	3.6
$(L_x, L_y, L_z)^T$	$(m, m, m)^T$	$L_x \, L_y \, L_z = V$	simulation box dimensions	3.6
T	K	$T = \sum_i g_i > 0$	absolute temperature	3.7
g_i	K	(55)	temperature DDFs	3.7
g_i^{eq}	K	(56)	equilibrium temperature DDFs	3.7
α	$\frac{m^2}{s}$	(59)	thermal diffusion coefficient	3.7
β	$\frac{1}{K}$	(60)	thermal expansion coefficient	3.7
g	$\frac{m}{s^2}$	$g := 9.81 \frac{m}{s^2}$	gravitational acceleration	3.7
n	1	$n = x + (y + z \, L_y) \, L_x$	linear index for 3D position in space	4.3.4
φ	1	$0 \leq \varphi \leq 1$	VoF fill level	6.1
m	$\frac{kg}{m^3}$	$m = \rho \, \varphi$	VoF fluid mass	6.1
σ	$\frac{kg}{s^2}$	(95)	surface tension coefficient	7
κ	$\kappa = \frac{1}{R}$	(98)	mean curvature	7
$\vec{n}$	m	(120)	surface normal vector	7.3.1
L	m	$L = 2\,R$	some spatial distance	8
Re	1	$Re = \frac{u\,L}{\nu}$	Reynolds number	8
Ma	1	$Ma = \frac{u}{c}$	Mach number	8
We	1	$We = \frac{\rho\,u^2\,L}{\sigma}$	Weber number	8
Fr	1	$Fr = \frac{u}{\sqrt{g\,L}}$	Froude number	8
Ca	1	$Ca = \frac{\rho\,\nu\,u}{\sigma}$	Capillary number	8
R	m	$R = \frac{L}{2}$	channel or sphere radius	9.1.2
D	m	$D = 2\,R$	channel or sphere diameter	9.1.2
r	m	$r = \sqrt{x^2 + y^2 + z^2}$	radial position	9.1.2
Q	$\frac{m^3}{s}$	(179), (192)	volume flow rate	9.1.2
E	1	L_1/L_2 norm	error	9.1.3

3 The Lattice Boltzmann Method

The lattice Boltzmann method (LBM) is gaining increasing popularity among competing simulation techniques for computational fluid dynamics (CFD) due to it being well-fitted to run on massively parallel computer hardware. It provides great flexibility in terms of extensions, for example for simulating temperature or free surfaces – the latter will be the very core of this thesis. Still understanding the basics of LBM and its different flavors is very important, since in the simulation software *FluidX3D* the majority of them is implemented in a modular approach. Here in this chapter, LBM is introduced and its different varieties are presented with all information necessary for implementation. Proper validation of *FluidX3D* is done in chapter 9. The mathematical connection between LBM, the Boltzmann equation and the Navier-Stokes equations is given by the Chapman-Enskog analysis, which is presented in great detail in [4, p.105-119] and [5] and thus won't be elaborated further here.

3.1 LBM in a Nutshell

The LBM discretizes space into a Cartesian lattice and for each lattice point introduces so called density distribution functions (DDFs), also called fluid populations, denoted with the symbol f_i. These are realized as floating-point numbers in memory and can be imagined as a bunch of fluid molecules. The index i denotes the direction to a neighboring point on the lattice to which the f_i stream. When at a lattice point all of the f_i have arrived from neighboring lattice points, they are collided and redistributed, ready to stream out again to the next neighbors. The DDFs, which formally have the unit of mass density, are essential for the inner workings of LBM and from them directly the desired macroscopic quantities such as density and velocity (eq. (2)) or later the force density (eq. (52)) are calculated.

At a single lattice point at the position $\vec{x}$ and time t, LBM first streams in the DDFs from neighboring lattice points at positions $(\vec{x}-\vec{e}_i)$ (eq. (1)), whereby $\vec{e}_i = \vec{c}_i\,\Delta t$ are the streaming directions (eq. (11)). The capital superscript A denotes that f_i^{A} are stored in memory at the pointer position A. The streamed-in DDFs f_i^{temp} are temporarily stored in registers (details see section 4.2.2) and from them the local density ρ and velocity $\vec{u}$ are calculated (eq. (2)), which themselves are the input parameters for the so-called equilibrium DDFs f_i^{eq} in eq. (3). w_i denote the lattice weights (eq. (10)) and $c := \frac{1}{\sqrt{3}}\frac{\Delta x}{\Delta t}$ is the lattice speed of sound. Finally, on f_i^{temp} and f_i^{eq} the collision operator Ω_i (details see section 3.3) is applied and the result is written back into memory, although to a different memory address denoted as f_i^{B} (eq. (4)). This second memory location B is required in order to eliminate any data dependencies between neighboring lattice points when all of them are processed in parallel with random order of execution. These equations are the LBM:

1. Streaming

$$f_i^{\mathrm{temp}}(\vec{x},t) = f_i^{\mathrm{A}}(\vec{x}-\vec{e}_i,\, t) \tag{1}$$

2. Collision

$$\rho(\vec{x},t) = \sum_i f_i^{\mathrm{temp}}(\vec{x},t) \qquad \vec{u}(\vec{x},t) = \frac{1}{\rho}\sum_i \vec{c}_i\, f_i^{\mathrm{temp}}(\vec{x},t) \tag{2}$$

$$f_i^{\mathrm{eq}}(\rho(\vec{x},t),\, \vec{u}(\vec{x},t)) = w_i\,\rho\cdot\left(\frac{\vec{u}\circ\vec{c}_i}{c^2} + \frac{(\vec{u}\circ\vec{c}_i)^2}{2\,c^4} + 1 - \frac{\vec{u}\circ\vec{u}}{2\,c^2}\right) \tag{3}$$

$$f_i^{\mathrm{B}}(\vec{x},\, t+\Delta t) = f_i^{\mathrm{temp}}(\vec{x},t) - \Omega_i \tag{4}$$

After the LBM step is completed for all lattice points, the memory pointers A and B are swapped and the next LBM step is executed. The here presented variant of the LBM algorithm is called one-step-pull and is the fastest implementation on GPUs. Details and alternatives are discussed in section 4.3.5.

Once at simulation startup, $f_i^{\mathrm{A}}(\vec{x}, t=0)$ need to be initialized. This is done by setting

$$f_i^{\mathrm{A}}(\vec{x}, t=0) := f_i^{\mathrm{eq}}(\rho(\vec{x}, t=0),\, \vec{u}(\vec{x}, t=0)) \tag{5}$$

whereby $\rho(\vec{x}, t=0)$ and $\vec{u}(\vec{x}, t=0)$ need to be defined to fit the simulated setup. Note that the density ρ is connected to the pressure p via

$$p = c^2\rho \tag{6}$$

3.2 Velocity Sets

LBM allows for various levels of velocity discretization, the so-called velocity sets. These are denoted as $DdQq$, where d is the number of spatial dimensions (either 2 or 3) and q is the number of streaming directions determining the range of the streaming direction index i:

$$0 \leq i < q \tag{7}$$

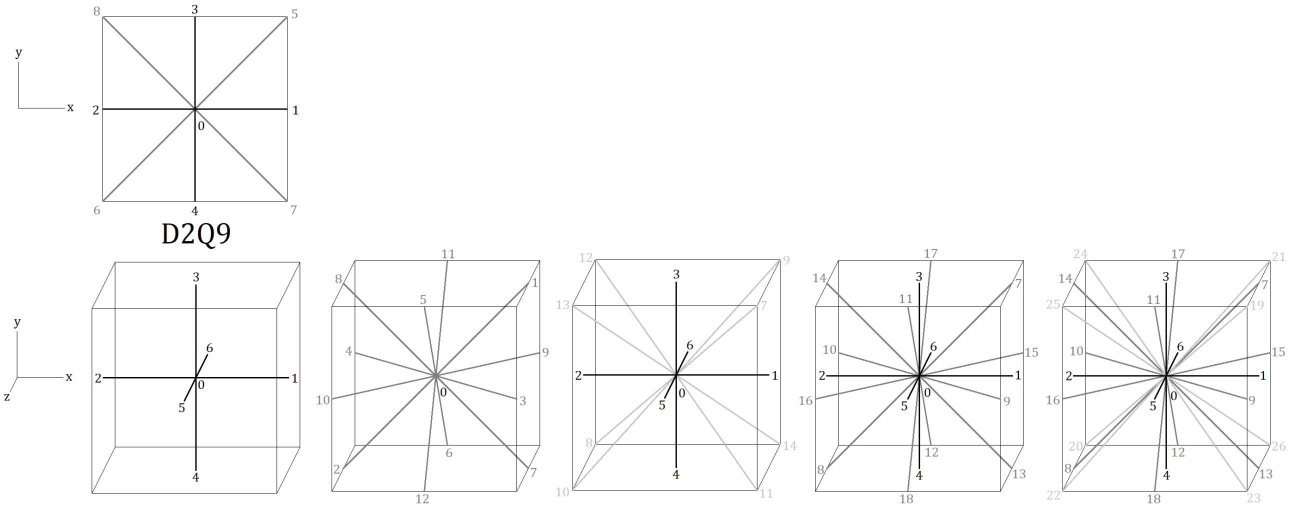
Each velocity set defines $\vec{c}_i = \vec{e}_i \, \Delta t$, w_i, M and S differently. The matrices M and S are introduced later in section 3.3.3. The streaming directions $\vec{e}_i$ for the different velocity sets are illustrated by figure 1 and defined in equation (11). Here they are defined such that

$$\vec{e}_i = -\vec{e}_{i+1} =: -\vec{e}_{\bar{i}} \tag{8}$$

for all odd i, but this definition is not used everywhere in literature. The lattice weights w_i are defined in equation (10) [4, p.88], whereby they are normalized:

$$1 = \sum_i w_i \tag{9}$$

Velocity sets with larger q are more accurate, but require proportionally more memory as well and take proportionally longer to compute. D2Q9 is the standard for 2D simulations and D3Q19 is the standard for 3D simulations. D3Q7 and D3Q13 are only rarely used due to their poor accuracy and stability, D3Q15 is reasonably stable, but does not have the best isotropic properties and D3Q27 is only used in edge cases where isotropy and energy are of great concern.



D2Q9

D3Q7 D3Q13 D3Q15 D3Q19 D3Q27

Figure 1: The velocity sets implemented in *FluidX3D*.

$$w_i = \begin{cases} \left\{\frac{4}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}\right\} & \text{for D2Q9} \\ \left\{\frac{1}{4}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}\right\} & \text{for D3Q7} \\ \left\{\frac{1}{2}, \frac{1}{24}, \frac{1}{24}, \frac{1}{24}, \frac{1}{24}, \frac{1}{24}, \frac{1}{24}, \frac{1}{24}, \frac{1}{24}, \frac{1}{24}, \frac{1}{24}, \frac{1}{24}, \frac{1}{24}\right\} & \text{for D3Q13} \\ \left\{\frac{2}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{72}, \frac{1}{72}, \frac{1}{72}, \frac{1}{72}, \frac{1}{72}, \frac{1}{72}, \frac{1}{72}, \frac{1}{72}\right\} & \text{for D3Q15} \\ \left\{\frac{1}{3}, \frac{1}{18}, \frac{1}{18}, \frac{1}{18}, \frac{1}{18}, \frac{1}{18}, \frac{1}{18}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}\right\} & \text{for D3Q19} \\ \left\{\frac{8}{27}, \frac{2}{27}, \frac{2}{27}, \frac{2}{27}, \frac{2}{27}, \frac{2}{27}, \frac{2}{27}, \frac{1}{54}, \frac{1}{54}, \frac{1}{54}, \frac{1}{54}, \frac{1}{54}, \frac{1}{54}, \frac{1}{54}, \frac{1}{54}, \frac{1}{54}, \frac{1}{54}, \frac{1}{54}, \frac{1}{54}, \right. & \\ \qquad \left. \frac{1}{216}, \frac{1}{216}, \frac{1}{216}, \frac{1}{216}, \frac{1}{216}, \frac{1}{216}, \frac{1}{216}, \frac{1}{216}\right\} & \text{for D3Q27} \end{cases} \tag{10}$$

$$\vec{c}_i = \frac{\vec{e}_i}{\Delta t} = \frac{\Delta x}{\Delta t} \cdot \left\{ \begin{array}{ll}
\left\{\begin{matrix} 0 & 1 & -1 & 0 & 0 & 1 & -1 & 1 & -1 \\ 0 & 0 & 0 & 1 & -1 & 1 & -1 & -1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{matrix}\right\} & \text{for D2Q9} \\[2ex]
\left\{\begin{matrix} 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{matrix}\right\} & \text{for D3Q7} \\[2ex]
\left\{\begin{matrix} 0 & 1 & -1 & 1 & -1 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & -1 & 1 & -1 & 1 \end{matrix}\right\} & \text{for D3Q13} \\[2ex]
\left\{\begin{matrix} 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & -1 & 1 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 & 1 & -1 & -1 & 1 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & -1 & 1 & 1 & -1 & 1 & -1 \end{matrix}\right\} & \text{for D3Q15} \\[2ex]
\left\{\begin{matrix} 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & -1 & 1 & -1 & 1 \end{matrix}\right\} & \text{for D3Q19} \\[2ex]
\left\{\begin{matrix} 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & -1 & 1 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & -1 & 1 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & -1 & 1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & 1 & -1 \end{matrix}\right\} & \text{for D3Q27}
\end{array} \right. \tag{11}$$

3.3 Collision Operators

The collision operators work with a relaxation time τ or several relaxation times τ_i. τ is determined by the kinematic shear viscosity $\nu > 0$ of the simulated fluid:

$$\tau = \frac{\nu}{c^2} + \frac{\Delta t}{2} > \frac{\Delta t}{2} \tag{12}$$

In the following subsections, the collision operators SRT, TRT and MRT are presented in detail.

3.3.1 Single Relaxation Time (SRT)

The SRT (also called BGK) collision operator

$$\Omega_i = \frac{\Delta t}{\tau} \left(f_i^{\text{temp}}(\vec{x}, t) - f_i^{\text{eq}}(\vec{x}, t) \right) \tag{13}$$

is the simplest collision operator with just a single relaxation time dictated by ν via equation (12).

3.3.2 Two Relaxation Time (TRT)

The TRT relaxation operator [4, p.425-427] uses two relaxation times τ^+ and τ^-, which are related to each other by the 'magic parameter' Λ:

$$\tau^+ := \tau \tag{14}$$

$$\tau^- := \left(\frac{\Lambda}{\frac{\tau}{\Delta t} - \frac{1}{2}} + \frac{1}{2} \right) \Delta t \tag{15}$$

$$\Lambda := \begin{cases} \frac{1}{4} & \text{best stability} \\ \frac{3}{16} & \text{exact bounce-back boundary locations for Poiseuille flow} \\ \frac{1}{12} & \text{best advection (removes third-order error)} \\ \frac{1}{6} & \text{best diffusion (removes fourth-order error)} \end{cases} \tag{16}$$

With these two relaxation times, the even- and odd-order moments

$$f_i^+(\vec{x}, t) := \frac{1}{2} \left(f_i^{\text{temp}}(\vec{x}, t) + f_{\bar{i}}^{\text{temp}}(\vec{x}, t) \right) \tag{17}$$

$$f_i^-(\vec{x}, t) := \frac{1}{2} \left(f_i^{\text{temp}}(\vec{x}, t) - f_{\bar{i}}^{\text{temp}}(\vec{x}, t) \right) \tag{18}$$

$$f_i^{\text{eq}+}(\vec{x}, t) := \frac{1}{2} \left(f_i^{\text{eq}}(\vec{x}, t) + f_{\bar{i}}^{\text{eq}}(\vec{x}, t) \right) \tag{19}$$

$$f_i^{\text{eq}-}(\vec{x}, t) := \frac{1}{2} \left(f_i^{\text{eq}}(\vec{x}, t) - f_{\bar{i}}^{\text{eq}}(\vec{x}, t) \right) \tag{20}$$

are relaxed individually:

$$\Omega_i = \frac{\Delta t}{\tau^+} \left(f_i^+(\vec{x}, t) - f_i^{\text{eq}+}(\vec{x}, t) \right) + \frac{\Delta t}{\tau^-} \left(f_i^-(\vec{x}, t) - f_i^{\text{eq}-}(\vec{x}, t) \right) \tag{21}$$

Here $\bar{i}$ denotes the streaming direction opposite to i, which in this work is $\vec{e}_i = -\vec{e}_{i+1} =: -\vec{e}_{\bar{i}}$ for all odd i, but this definition is not always used in literature. TRT comes at no significant additional computational cost compared to SRT, but is generally more accurate and more stable and thus preferred. By default, $\Lambda = \frac{3}{16}$ is chosen in the implementation.

3.3.3 Multi Relaxation Time (MRT)

The MRT operator

$$\Omega_i = \left(M^{-1} S\, M \left(f^{\text{temp}}(\vec{x},t) - f^{\text{eq}}(\vec{x},t)\right)\right)_i \tag{22}$$

at first glance looks simple on paper, but is the most tedious to implement. In equation (22), M is a $q \times q$ transformation matrix from population space into moment space and $S = \operatorname{diag}(\frac{\Delta t}{\tau_i})$ is a $q \times q$ diagonal matrix containing all relaxation times with which the moments are relaxed individually. It is claimed that MRT is most accurate, but the relaxation times τ_i have to be specifically tuned for every type of simulation and are sort of a black box, especially for the numerous non-physical moments. The moments

$$m := M\, f = \begin{cases} (\rho, e, \epsilon, j_x, q_x, j_y, q_y, p_{xx}, p_{xy})^T & \text{for D2Q9} \\ (\rho, j_x, j_y, j_z, e, 3\,p_{xx}, p_{ww}, p_{xy}, p_{yz}, p_{zx}, m_x, m_y, m_z)^T & \text{for D3Q13} \\ (\rho, e, \epsilon, j_x, q_x, j_y, q_y, j_z, q_z, p_{xx}, p_{ww}, p_{xy}, p_{yz}, p_{zx}, m_{xyz})^T & \text{for D3Q15} \\ (\rho, e, \epsilon, j_x, q_x, j_y, q_y, j_z, q_z, p_{xx}, \pi_{xx}, p_{ww}, \pi_{ww}, p_{xy}, p_{yz}, p_{xz}, m_x, m_y, m_z)^T & \text{for D3Q19} \end{cases} \tag{23}$$

correspond to quantities such as density ρ, energy e, energy squared ϵ, momentum $j_{x/y/z}$, heat flux $q_{x/y/z}$, momentum flux $p_{xx/ww/xy/xz/yz/zx}$ and a bunch of other non-physical higher-order polynomials such as $m_{x/y/z}$ or $\pi_{xx/ww}$. The meanings of the moments for D3Q7 and D3Q27 are a bit unclear in literature since these velocity sets are only rarely used, leaving the MRT definitions for these two incomplete.

Each type of moment is relaxed individually by a relaxation rate

$$\omega_i := \frac{\Delta t}{\tau_i} \tag{24}$$

which

- for conserved quantities (density ρ and momentum $j_{x/y/z}$) is 0,
- for the momentum flux $p_{xx/ww/xy/yz/zx}$ is given by $\omega := \frac{\Delta t}{\tau}$ (τ is determined by the kinematic shear viscosity),
- for the energy e is dictated by the kinematic bulk viscosity

  $$\nu_{\text{B}} = \frac{2}{3}\,\nu \tag{25}$$

  which again is defined by the kinematic shear viscosity ν, leading to

  $$\omega_e := \frac{\Delta t}{\frac{2}{3}\left(\tau - \frac{\Delta t}{2}\right) + \frac{\Delta t}{2}} \tag{26}$$

- for non-physical moments such as $m_{x/y/z}$ and $\pi_{xx/ww}$ usually is set to 1 (instant relaxation) and
- for all other physical moments such as ϵ and q is a tuning parameter.

All together then these are the relaxation matrix definitions:

$$S = \begin{cases} \operatorname{diag}(0, \omega_e, \omega_\epsilon, 0, \omega_q, 0, \omega_q, \omega, \omega) & \text{for D2Q9} \\ \operatorname{diag}(0, 0, 0, 0, \omega_e, \omega, \omega, \omega, \omega, \omega, \omega_m, \omega_m, \omega_m) & \text{for D3Q13} \\ \operatorname{diag}(0, \omega_e, \omega_\epsilon, 0, \omega_q, 0, \omega_q, 0, \omega_q, \omega, \omega, \omega, \omega, \omega, \omega_m) & \text{for D3Q15} \\ \operatorname{diag}(0, \omega_e, \omega_\epsilon, 0, \omega_q, 0, \omega_q, 0, \omega_q, \omega, \omega_\pi, \omega, \omega_\pi, \omega, \omega, \omega, \omega_m, \omega_m, \omega_m) & \text{for D3Q19} \end{cases} \tag{27}$$

Not counting S since it is diagonal, in the MRT operator two matrix multiplications are required, which, although they may entirely be done in registers, are potentially computationally costly if they shift the overall arithmetic intensity too far up (see section 5.6). The matrices M and S have to be hard-coded for all velocity sets and can be found in [4, p.420][6] (D2Q9), [7] (D3Q7), [8, p.30] (D3Q13), [4, p.669-672] (D3Q15 and D3Q19) and [9] (D3Q27). Close attention has to be paid on the order in which the indexing i of the streaming directions is done. If the order is different, in M the order of columns must be changed accordingly. The matrices compatible with the here used $\vec{c}_i$ are given by equations (29) to (34).

In order to avoid the second matrix multiplication, since both M and S do not change over time, at runtime right before simulation start $Q := M^{-1}S\,M$ is calculated and injected into the OpenCL C code (see section 4.3.10). This way, the GPU only has to calculate

$$\Omega_i = \left(Q\left(f^{\text{temp}}(\vec{x},t) - f^{\text{eq}}(\vec{x},t)\right)\right)_i \tag{28}$$

in every simulation step, cutting the required floating-point operations almost in half. The reduction of floating-point operations is even larger if a volume force (see section 3.5) is needed, here avoiding two of three matrix multiplications. For the single remaining matrix multiplication, loop unrolling (see section 4.3.8) is essential. With these optimizations applied, LBM overall is still memory-bound, meaning that the MRT operator is just as fast as SRT and TRT (illustrated in figure 8).

$$M_{\mathrm{D2Q9}} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ -4 & -1 & -1 & -1 & -1 & 2 & 2 & 2 & 2 \\ 4 & -2 & -2 & -2 & -2 & 1 & 1 & 1 & 1 \\ 0 & 1 & -1 & 0 & 0 & 1 & -1 & 1 & -1 \\ 0 & -2 & 2 & 0 & 0 & 1 & -1 & 1 & -1 \\ 0 & 0 & 0 & 1 & -1 & 1 & -1 & -1 & 1 \\ 0 & 0 & 0 & -2 & 2 & 1 & -1 & -1 & 1 \\ 0 & 1 & 1 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 \end{pmatrix} \tag{29}$$

$$M_{\mathrm{D3Q7}} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 \\ 6 & -1 & -1 & -1 & -1 & -1 & -1 \\ 0 & 2 & 2 & -1 & -1 & -1 & -1 \\ 0 & 0 & 0 & 1 & 1 & -1 & -1 \end{pmatrix} \tag{30}$$

$$M_{\mathrm{D3Q13}} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & -1 & 1 & -1 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & -1 & 1 & -1 & 1 \\ -12 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & -2 & -2 & 1 & 1 & 1 & 1 & -2 & -2 \\ 0 & 1 & 1 & -1 & -1 & 0 & 0 & 1 & 1 & -1 & -1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & -1 & -1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 \\ 0 & 1 & -1 & -1 & 1 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & -1 & 1 & 1 & -1 \end{pmatrix} \tag{31}$$

$$M_{\mathrm{D3Q15}} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ -2 & -1 & -1 & -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 16 & -4 & -4 & -4 & -4 & -4 & -4 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & -1 & 1 \\ 0 & -4 & 4 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & -1 & 1 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 & 1 & -1 & -1 & 1 & 1 & -1 \\ 0 & 0 & 0 & -4 & 4 & 0 & 0 & 1 & -1 & 1 & -1 & -1 & 1 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & -1 & 1 & 1 & -1 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & -4 & 4 & 1 & -1 & -1 & 1 & 1 & -1 & 1 & -1 \\ 0 & 2 & 2 & -1 & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & -1 & 1 & -1 & 1 \end{pmatrix} \tag{32}$$

$$M_{\mathrm{D3Q19}} = \begin{pmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
-30 & -11 & -11 & -11 & -11 & -11 & -11 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 \\
12 & -4 & -4 & -4 & -4 & -4 & -4 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 \\
0 & -4 & 4 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 \\
0 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & 1 & -1 \\
0 & 0 & 0 & -4 & 4 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & 1 & -1 \\
0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & -1 & 1 & -1 & 1 \\
0 & 0 & 0 & 0 & 0 & -4 & 4 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & -1 & 1 & -1 & 1 \\
0 & 2 & 2 & -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 & -2 & -2 & 1 & 1 & 1 & 1 & -2 & -2 \\
0 & -4 & -4 & 2 & 2 & 2 & 2 & 1 & 1 & 1 & 1 & -2 & -2 & 1 & 1 & 1 & 1 & -2 & -2 \\
0 & 0 & 0 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 0 & 0 & 1 & 1 & -1 & -1 & 0 & 0 \\
0 & 0 & 0 & -2 & -2 & 2 & 2 & 1 & 1 & -1 & -1 & 0 & 0 & 1 & 1 & -1 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & -1 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & 1 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & -1 & 1 & 1 & -1
\end{pmatrix} \tag{33}$$

$$M_{\mathrm{D3Q27}} = \begin{pmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & -1 & 1 \\
0 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & -1 & 1 & 1 & -1 \\
0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & -1 & 1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & 1 & -1 \\
-2 & -1 & -1 & -1 & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 2 & 2 & -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 & -2 & -2 & 1 & 1 & 1 & 1 & -2 & -2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 0 & 0 & 1 & 1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & -1 & -1 & 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\
0 & -4 & 4 & 0 & 0 & 0 & 0 & -1 & 1 & -1 & 1 & 0 & 0 & -1 & 1 & -1 & 1 & 0 & 0 & 2 & -2 & 2 & -2 & 2 & -2 & -2 & 2 \\
0 & 0 & 0 & -4 & 4 & 0 & 0 & -1 & 1 & 0 & 0 & -1 & 1 & 1 & -1 & 0 & 0 & -1 & 1 & 2 & -2 & 2 & -2 & -2 & 2 & 2 & -2 \\
0 & 0 & 0 & 0 & 0 & -4 & 4 & 0 & 0 & -1 & 1 & -1 & 1 & 0 & 0 & 1 & -1 & 1 & -1 & 2 & -2 & -2 & 2 & 2 & -2 & 2 & -2 \\
0 & 4 & -4 & 0 & 0 & 0 & 0 & -2 & 2 & -2 & 2 & 0 & 0 & -2 & 2 & -2 & 2 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & -1 & 1 \\
0 & 0 & 0 & 4 & -4 & 0 & 0 & -2 & 2 & 0 & 0 & -2 & 2 & 2 & -2 & 0 & 0 & -2 & 2 & 1 & -1 & 1 & -1 & -1 & 1 & 1 & -1 \\
0 & 0 & 0 & 0 & 0 & 4 & -4 & 0 & 0 & -2 & 2 & -2 & 2 & 0 & 0 & 2 & -2 & 2 & -2 & 1 & -1 & -1 & 1 & 1 & -1 & 1 & -1 \\
4 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
-8 & 4 & 4 & 4 & 4 & 4 & 4 & -2 & -2 & -2 & -2 & -2 & -2 & -2 & -2 & -2 & -2 & -2 & -2 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & -4 & -4 & 2 & 2 & 2 & 2 & 1 & 1 & 1 & 1 & -2 & -2 & 1 & 1 & 1 & 1 & -2 & -2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -2 & -2 & 2 & 2 & 1 & 1 & -1 & -1 & 0 & 0 & 1 & 1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -2 & -2 & 0 & 0 & 0 & 0 & 2 & 2 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -2 & -2 & 0 & 0 & 0 & 0 & 2 & 2 & 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -2 & -2 & 0 & 0 & 0 & 0 & 2 & 2 & 0 & 0 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & -1 & 1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & -1 & 1 & -1 & 1
\end{pmatrix} \tag{34}$$

3.4 Boundaries in LBM

In order to distinguish regular lattice points from boundary lattice points, a flags field is allocated in memory. The smallest data type available in OpenCL is `uchar`, an 8-bit unsigned integer. Each of these bits can be used separately to distinguish lattice points of different types. Non-moving bounce-back boundaries occupy only 1 bit of the available eight.

3.4.1 Equilibrium Boundaries

Equilibrium boundaries [10] fix the flagged lattice points to their initial density and velocity by always calculating the density distribution functions as the equilibrium DDFs for flagged lattice points.

$$f_i^{\mathrm{B}}(\vec{x}, t+\Delta t) = \begin{cases} f_i^{\mathrm{temp}}(\vec{x}, t) - \Omega_i & \text{if } \vec{x} \text{ is a regular lattice point} \\ f_i^{\mathrm{eq}}(\rho_0, \vec{u}_0) & \text{if } \vec{x} \text{ is a } \textit{equilibrium boundary} \text{ lattice point} \end{cases} \tag{35}$$

Any incoming DDFs are dismissed, making this method non-reflecting. This method qualitatively works, but its accuracy – tested with Poiseuille flow – leaves much to be desired.

3.4.2 Non-moving Bounce-Back Boundaries

Non-moving no-slip bounce-back boundary conditions in LBM are rather straight forward and are integrated right into the streaming step by checking the flag of the neighbor in the respective streaming direction. The flag lattice can be arbitrary, providing an almost infinite variety of simulatable geometries. This is the formulation for non-moving BB-boundaries with the pull scheme:

$$f_i^{\mathrm{temp}}(\vec{x},\, t) := \begin{cases} f_i^{\mathrm{A}}(\vec{x} - \vec{e}_i,\, t) & \text{if } (\vec{x} - \vec{e}_i) \text{ is a regular lattice point} \\ f_{\bar{i}}^{\mathrm{A}}(\vec{x},\, t) & \text{if } (\vec{x} - \vec{e}_i) \text{ is a } \textit{wall} \text{ lattice point} \end{cases} \tag{36}$$

3.4.3 Moving Bounce-Back Boundaries

Specifying a non-zero wall velocity seemingly does not add a lot of difficulty, extending eq. (36) by an additional term containing the boundary velocity [4, p.180].

$$f_i^{\mathrm{temp}}(\vec{x},\, t) := \begin{cases} f_i^{\mathrm{A}}(\vec{x} - \vec{e}_i,\, t) & \text{if } (\vec{x} - \vec{e}_i) \text{ is a regular lattice point} \\ f_{\bar{i}}^{\mathrm{A}}(\vec{x},\, t) - \frac{2\, w_{\bar{i}}\, \rho_{\mathrm{wall}}}{c^2}\, (\vec{c}_{\bar{i}} \circ \vec{u}_{\mathrm{wall}}(\vec{x} - \vec{e}_i,\, t_0)) & \text{if } (\vec{x} - \vec{e}_i) \text{ is a } \textit{wall} \text{ lattice point} \end{cases} \tag{37}$$

In eq. (37), $\vec{u}_{\mathrm{wall}}$ denotes the specified wall velocity, $\rho_{\mathrm{wall}} := 1$ is the average density throughout the simulation box and $c := \frac{1}{\sqrt{3}} \frac{\Delta x}{\Delta t}$ is the lattice speed of sound, so the equation can be simplified to

$$f_i^{\mathrm{temp}}(\vec{x},\, t) := \begin{cases} f_i^{\mathrm{A}}(\vec{x} - \vec{e}_i,\, t) & \text{if } (\vec{x} - \vec{e}_i) \text{ is a regular lattice point} \\ f_{\bar{i}}^{\mathrm{A}}(\vec{x},\, t) - 6\, w_{\bar{i}}\, (\vec{c}_{\bar{i}} \circ \vec{u}_{\mathrm{wall}}(\vec{x} - \vec{e}_i,\, t_0)) & \text{if } (\vec{x} - \vec{e}_i) \text{ is a } \textit{wall} \text{ lattice point} \end{cases} \tag{38}$$

which means that the neighbor wall velocity $\vec{u}_{\mathrm{wall}}(\vec{x}_1 - \vec{e}_i,\, t_0)$ must be read from memory for all regular lattice points adjacent to a wall with non-zero velocity. In order to speed up the algorithm, all lattice points adjacent to a wall with non-zero velocity are specially flagged during initialization so that for lattice points adjacent to only zero velocity walls no extra memory transfers are required.

Even though the velocity $\vec{u}_{\mathrm{wall}}(\vec{x}_1 - \vec{e}_i,\, t_0)$ is read from *wall* neighbors while $\vec{u}(\vec{x}_2, t)$ may be written to memory for another lattice point in parallel, a race condition (see section 4.2.5) is not possible because $\vec{u}$ is never overwritten in memory for *wall* lattice points.

3.5 Volume Force

The volume force – or rather force per volume, here denoted as $\vec{f}$ – is not only useful for creating a flow, it also is the foundation for numerous LBM extensions such as temperature, the immersed-boundary method or the Shan-Chen method. The standard for volume forces in LBM is the *Guo forcing* scheme [11]. It integrates a force per volume $\vec{f}$ right into the respective collision operator. The force needs to be accounted for twice: Firstly, as an additional term to equation (2)

$$\rho(\vec{x},t)=\sum_i f_i^{\text{temp}}(\vec{x},t) \qquad \vec{u}(\vec{x},t)=\frac{1}{\rho}\sum_i \vec{c}_i\, f_i^{\text{temp}}(\vec{x},t)+\frac{\Delta t}{2\,\rho}\,\vec{f} \tag{39}$$

and secondly as an additional term to the density distribution functions directly, the original equation for which [4, p.233] is quite unreadable due to inflated Einstein notation. In a more readable vector form for efficient implementation it states

$$F_i=\frac{w_i}{c^4}\left(\left(\vec{c}_i\circ\vec{f}\right)\left(\vec{c}_i\circ\vec{u}+c^2\right)-c^2\left(\vec{u}\circ\vec{f}\right)\right) \tag{40}$$

The forcing terms F_i then need to be relaxed according to the respective collision operator as shown in the following subsections.

3.5.1 Volume Force with SRT

In the SRT operator, the equations (4) and (13) combined

$$f_i^{\text{B}}(\vec{x},\,t+\Delta t)=f_i^{\text{temp}}(\vec{x},t)-\frac{\Delta t}{\tau}\left(f_i^{\text{temp}}(\vec{x},t)-f_i^{\text{eq}}(\vec{x},t)\right) \tag{41}$$

are extended [4, p.233] to

$$f_i^{\text{B}}(\vec{x},\,t+\Delta t)=f_i^{\text{temp}}(\vec{x},t)-\frac{\Delta t}{\tau}\left(f_i^{\text{temp}}(\vec{x},t)-f_i^{\text{eq}}(\vec{x},t)\right)+\left(1-\frac{\Delta t}{2\,\tau}\right)\Delta t\,F_i \tag{42}$$

3.5.2 Volume Force with TRT

In the TRT operator, the equations (4) and (21) combined

$$f_i^{\text{B}}(\vec{x},\,t+\Delta t)=f_i^{\text{temp}}(\vec{x},t)-\frac{\Delta t}{\tau^+}\left(f_i^{+}(\vec{x},t)-f_i^{\text{eq}+}(\vec{x},t)\right)-\frac{\Delta t}{\tau^-}\left(f_i^{-}(\vec{x},t)-f_i^{\text{eq}-}(\vec{x},t)\right) \tag{43}$$

are extended [12] to

$$\begin{aligned} f_i^{\text{B}}(\vec{x},\,t+\Delta t)=f_i^{\text{temp}}(\vec{x},t)-\frac{\Delta t}{\tau^+}\left(f_i^{+}(\vec{x},t)-f_i^{\text{eq}+}(\vec{x},t)\right)-\frac{\Delta t}{\tau^-}\left(f_i^{-}(\vec{x},t)-f_i^{\text{eq}-}(\vec{x},t)\right)+\\ +\left(1-\frac{\Delta t}{2\,\tau^+}\right)\Delta t\,F_i^+ +\left(1-\frac{\Delta t}{2\,\tau^-}\right)\Delta t\,F_i^- \end{aligned} \tag{44}$$

with

$$F_i^+ := \frac{1}{2}\left(F_i+F_{\bar{i}}\right) \tag{45}$$

$$F_i^- := \frac{1}{2}\left(F_i-F_{\bar{i}}\right) \tag{46}$$

3.5.3 Volume Force with MRT

In the MRT operator, equations (4) and (22) combined

$$f_i^{\mathrm{B}}(\vec{x},\,t+\Delta t) = f_i^{\mathrm{temp}}(\vec{x},t) - \left(M^{-1} S\, M\left(f^{\mathrm{temp}}(\vec{x},t) - f^{\mathrm{eq}}(\vec{x},t)\right)\right)_i \tag{47}$$

are extended [4, p.424] to

$$\begin{aligned}
f_i^{\mathrm{B}}(\vec{x},\,t+\Delta t) &= f_i^{\mathrm{temp}}(\vec{x},t) - \left(M^{-1} S\, M\left(f^{\mathrm{temp}}(\vec{x},t) - f^{\mathrm{eq}}(\vec{x},t)\right) + M^{-1}\left(\frac{1}{2}\,S - 1\right) M\,\Delta t\,F\right)_i = \\
&= f_i^{\mathrm{temp}}(\vec{x},t) - \left(M^{-1} S\, M\left(f^{\mathrm{temp}}(\vec{x},t) - f^{\mathrm{eq}}(\vec{x},t)\right) + \frac{1}{2}\,M^{-1} S\, M\,\Delta t\,F - M^{-1} M\,\Delta t\,F\right)_i = \\
&= f_i^{\mathrm{temp}}(\vec{x},t) - \left(Q\left(f^{\mathrm{temp}}(\vec{x},t) - f^{\mathrm{eq}}(\vec{x},t)\right) + \frac{\Delta t}{2}\,Q\,F - \Delta t\,F\right)_i = \\
&= f_i^{\mathrm{temp}}(\vec{x},t) - \left(Q\left(f^{\mathrm{temp}}(\vec{x},t) - f^{\mathrm{eq}}(\vec{x},t) + \frac{\Delta t}{2}\,F\right) - \Delta t\,F\right)_i
\end{aligned} \tag{48}$$

and rearranged with $Q := M^{-1} S\, M$ pre-calculated such that there is only a single matrix multiplication remaining.

3.6 Forces on Boundaries

Boundary forces in LBM are calculated with the *momentum exchange algorithm* [4, p.215-217][13, 14], taking advantage of the LBM distribution functions directly. The sum of the DDFs f_i^{temp} weighted by their accompanying streaming velocities $\vec{c}_i$ is equal to the total momentum density at a lattice point $\vec{x}$ at time t.

$$\frac{\vec{p}(\vec{x},t)}{V} = \rho(\vec{x},t)\,\vec{u}(\vec{x},t) = \sum_i \vec{c}_i\, f_i^{\text{temp}}(\vec{x},t) \tag{49}$$

Here $V = \Delta x^3$ is the volume of a single lattice point. In general, momentum $\vec{p}$ is the integral of force $\vec{F}$ over time, but during a LBM time step Δt this force is constant due to the discretization of time, meaning the integral is simplified to a multiplication with Δt. The force per volume $\vec{f}$ then is equal to the momentum density divided by the time step Δt:

$$\vec{f} = \frac{\vec{p}(\vec{x},t)}{V\,\Delta t} = \frac{1}{\Delta t}\sum_i \vec{c}_i\, f_i^{\text{temp}}(\vec{x},t) \tag{50}$$

On bounce-back boundaries, the distribution functions are reflected, meaning that a *wall* lattice point receives twice the opposite force density of any incoming DDF. Neighboring *wall* lattice points contribute with the equilibrium DDFs at density $\rho = 1$ and velocity $\vec{u} = 0$, which is equal to the lattice weights w_i:

$$f_i^{\text{eq}}(\rho = 1, \vec{u} = 0) = w_i \tag{51}$$

The force per volume for a single *wall* lattice point $\vec{f}_{\text{wall}}$ then becomes

$$\vec{f}_{\text{wall}} = -\frac{2}{\Delta t}\sum_i \begin{cases} \vec{c}_i\, w_i & \text{if } (\vec{x} - \vec{e}_i) \text{ is a \textit{wall} lattice point} \\ \vec{c}_i\, f_i^{\text{temp}}(\vec{x},t) & \text{otherwise} \end{cases} \tag{52}$$

Finally, the total force $\vec{F}(t)$ on a boundary, being composed of many *wall* lattice points, is equal to their force densities integrated over the total volume. Since the volume $V = \Delta x^3$ is the same for all lattice points, this integral becomes a summation

$$\vec{F}(t) = V \sum_{\vec{x}_{\text{ROI}}} \vec{f}_{\text{wall}}(\vec{x},t) \tag{53}$$

whereby the sum runs only over a local region of interest $\vec{x}_{\text{ROI}}$, for example a sphere composed of many *wall* lattice points. In the implementation, eq. (52) is calculated in parallel on the GPU and eq. (53) is calculated on the CPU.

3.7 Temperature for simulating thermal Convection

The lattice Boltzmann method itself contains no definition of temperature, i.e. assuming the simulation domain to be isothermal. In order to include temperature, a separate D3Q7 sublattice is introduced in addition to the regular DDM lattice [15]. This sublattice performs temperature advection and diffusion based on the thermal conductivity and thermal expansion coefficient. The coupling to LBM is done via an additional local volume forces induced by local thermal expansion and gravity.

1. Streaming

$$g_i^{\text{temp}}(\vec{x},t) = g_i^{\text{A}}(\vec{x}-\vec{e}_i,\, t) \tag{54}$$

2. Collision

$$T(\vec{x},t) = \sum_i g_i^{\text{temp}}(\vec{x},t) \tag{55}$$

$$g_i^{\text{eq}}(T(\vec{x},t),\, \vec{u}(\vec{x},t)) = w_i\, T \cdot \left(1 - \frac{\vec{u}\circ\vec{u}}{c^2}\right) \tag{56}$$

$$g_i^{\text{B}}(\vec{x},\, t+\Delta t) = g_i^{\text{temp}}(\vec{x},t) - \Omega_i \tag{57}$$

   with the SRT collision operator

$$\Omega_i = \frac{\Delta t}{\tau^{\text{T}}}\left(g_i^{\text{temp}}(\vec{x},t) - g_i^{\text{eq}}(\vec{x},t)\right) \tag{58}$$

   whereby the single relaxation time

$$\tau^{\text{T}} = 2\,\frac{\Delta t^2}{\Delta x^2}\,\alpha + \frac{\Delta t}{2} \tag{59}$$

   is set by the thermal diffusion coefficient α. TRT and MRT [7] variants are also possible analogous to the DDFs in section 3.3.

The temperature is coupled back into LBM in the form of an additional force per volume (buoyancy)

$$\vec{f}_{\text{temperature}} = \rho\,\vec{g}\,\beta \cdot (T - T_0) \tag{60}$$

with $\vec{g}$ denoting the gravitational acceleration vector, β denoting the thermal expansion coefficient of the simulated fluid and $T_0 := 1$ being the average temperature. The volume force is applied as described in section 3.5.

Examples for simulations with the temperature extension are shown by figures 49 and 50.

3.8 Immersed-Boundary Method on the GPU

The immersed-boundary method (IBM) [16] is an extension to LBM in order to have particles (or vertices of a tesselated surface) with non-discrete positions interacting with the LBM fluid which is only defined at discrete positions on a lattice. This interaction is two-way: The flow advects the particles, but the particles themselves locally apply force to the fluid (for example buoyancy of the particles or bending forces in a membrane), thereby changing the local fluid velocity. These two interactions are called *velocity interpolation* and *force spreading*.

3.8.1 Velocity Interpolation

A particle at a non-discrete position $\vec{x}_p = (x_p,\, y_p,\, z_p)^T$ in 3D is always surrounded by a cube of eight LBM lattice points $i, j, k \in \{0, 1\}$, at which the fluid velocity $\vec{u}(\vec{x}_{ijk})$ is known. To determine the velocity at the particle position, here trilinear interpolation[1] [17] is used. First, from the particle position $\vec{x}_p$ the eight surrounding lattice points $\vec{x}_{ijk}$ are determined:

$$\vec{x}_a = (x_a,\, y_a,\, z_a) := \left(x_p - \frac{1}{2} + \frac{3}{2}\,L_x,\; y_p - \frac{1}{2} + \frac{3}{2}\,L_y,\; z_p - \frac{1}{2} + \frac{3}{2}\,L_z\right)^T \tag{61}$$

$$\vec{x}_b = (x_b,\, y_b,\, z_b) := ((\texttt{int})\,x_a,\, (\texttt{int})\,y_a,\, (\texttt{int})\,z_a)^T \tag{62}$$

$$\vec{x}_{ijk} := ((x_b + i)\,\%\,L_x,\; (y_b + j)\,\%\,L_y,\; (z_b + k)\,\%\,L_z)^T \tag{63}$$

Here $(\texttt{int})\,x$ denotes the integer casting operation, which rounds down a floating point number x, and $\%$ denotes the modulo operator. The offsets are required to avoid negative input to the casting operator and to align the particle coordinate system, in which the origin is defined as the center of the LBM lattice, with the LBM coordinate system. The modulo operator is required to assure seamless periodic boundary conditions. Next, the interpolation factors are calculated:

$$\vec{x}_1 = (x_1,\, y_1,\, z_1) := \vec{x}_a - \vec{x}_b \tag{64}$$

$$\vec{x}_0 = (x_0,\, y_0,\, z_0) := (1,\, 1,\, 1)^T - \vec{x}_1 \tag{65}$$

Finally, the interpolated velocity at the particle position $\vec{u}_p$ is given by the following expression:

$$\begin{aligned}\vec{u}_p = {} & x_0\, y_0\, z_0\, \vec{u}(\vec{x}_{000}) + \\ & + x_0\, y_0\, z_1\, \vec{u}(\vec{x}_{001}) + \\ & + x_0\, y_1\, z_0\, \vec{u}(\vec{x}_{010}) + \\ & + x_0\, y_1\, z_1\, \vec{u}(\vec{x}_{011}) + \\ & + x_1\, y_0\, z_0\, \vec{u}(\vec{x}_{100}) + \\ & + x_1\, y_0\, z_1\, \vec{u}(\vec{x}_{101}) + \\ & + x_1\, y_1\, z_0\, \vec{u}(\vec{x}_{110}) + \\ & + x_1\, y_1\, z_1\, \vec{u}(\vec{x}_{111})\end{aligned} \tag{66}$$

With this velocity, the particle is advected one time step Δt.

3.8.2 Force Spreading

Force spreading is sort of the inverse process to trilinear interpolation. For the advected particle position, again first the eight surrounding lattice points $\vec{x}_{ijk}$ are determined using equations (61) to (64). Next, the interpolation factor for each of the eight corners

$$d_{ijk} := (1 - |x_1 - i|)\,(1 - |y_1 - j|)\,(1 - |z_1 - k|) \tag{67}$$

is calculated and finally the additional force $\vec{F}_p$ which the particle exhibits on the fluid is spread across the eight corners:

$$\vec{F}_{\text{add}}(\vec{x}_{ijk}) := d_{ijk}\,\vec{F}_p \tag{68}$$

This additional volume force $\vec{F}_p(\vec{x}_{ijk}) = \sum \vec{F}_{\text{add}}(\vec{x}_{ijk})$ is incorporated into the LBM via the Guo forcing scheme as described in section 3.5.

---

[1] Other interpolation stencils are also possible, but more computationally costly. The simplest form of interpolation, nearest neighbor interpolation, is fine for graphical purposes, but insufficient for IBM.

3.8.3 GPU Implementation Notes

In order to preserve the performance advantage of LBM on the GPU, transferring the velocity field to the CPU, then calculating IBM on the CPU side and transferring back the forces must be avoided (see section 4.3.3). Instead, IBM is fully implemented on the GPU, eliminating any PCIe data transfer. Without interactions between IBM particles (like when they are vertices of a membrane), the entire IBM extension, including velocity interpolation and force spreading, can be implemented as a single kernel (see section 4.2.1) parallelized over the number of particles rather than the number of LBM lattice points. While the velocity interpolation part uses the LBM velocity field *read-only*, the force spreading part is a lot more problematic. For multiple nearby particles, a force (eq. (68)) is added to the same memory address from multiple GPU threads in parallel in random order, leading to a race condition (see section 4.2.5).
The solution to this problem is the `atomic_add()` function, an addition to the standard GPU instruction set available since 2007, so nowadays supported by every GPU. `atomic_add()` reads the value currently in memory, adds another value and writes the result back into memory, all while blocking access to that memory address for other threads during the operation. In OpenCL however `atomic_add()` is only supported for 32-bit (and on some GPUs 64-bit) integer data types and not floating-point. There is however a clever workaround using the integer `atomic_cmpxchg()` function combined with the C99 `union` [18]:

```c
void __attribute__((always_inline)) atomic_add_f(volatile global float* addr, const float val) { // not deterministic because the order of addition can vary: (a+b)+c is rounded differently than a+(b+c)
  union { // https://streamhpc.com/blog/2016-02-09/atomic-operations-for-floats-in-opencl-improved/
    uint  u32;
    float f32;
  } next, expected, current;
  current.f32 = *addr;
  do {
    next.f32 = (expected.f32=current.f32)+val; // ...*val for atomic_mul_f()
    current.u32 = atomic_cmpxchg((volatile global uint*)addr, expected.u32, next.u32);
  } while(current.u32!=expected.u32);
}
```

Listing 1: Atomic addition for floating-point numbers in OpenCL.

3.9 Shan-Chen

3.9.1 Theory

The Shan-Chen method [4, p.367-386][19] is an extension to LBM to simulate multi-phase flows, for example liquid water and water vapor. The idea of Shan-Chen is to artificially sustain density gradients: At a liquid-vapor interface, there is a large density gradient, which in LBM via eq. (6) is the same as a pressure gradient. Pressure gradients in fluids are – except for when a volume force is acting – inherently unstable, and this is where Shan-Chen introduces a local countering force based on a pseudo-potential

$$\psi(\vec{x}) := (1 - e^{-\frac{\rho(\vec{x})}{\rho_0}})\, \rho_0 \tag{69}$$

which depends on the local density. The average density $\rho_0 = 1$ is a constant. With this pseudo-potential evaluated at the local lattice point $\vec{x}$ and its neighbors $(\vec{x}+\vec{e}_i)$, an additional local force (per volume) is defined:

$$\begin{aligned}\vec{f}_{\mathrm{SC}}(\vec{x}) :=& -G\,\psi(\vec{x})\sum_i \vec{e}_i\, w_i\, \psi(\vec{x}+\vec{e}_i) = & (70)\\ =& -G\,(1 - e^{-\frac{\rho(\vec{x})}{\rho_0}})\,\rho_0 \sum_i \vec{e}_i\, w_i\, (1 - e^{-\frac{\rho(\vec{x}+\vec{e}_i)}{\rho_0}})\,\rho_0 = \\ =& -G\,(1 - e^{-\frac{\rho(\vec{x})}{\rho_0}})\,\rho_0^2 \left(\sum_i \vec{e}_i\, w_i - \sum_i \vec{e}_i\, w_i\, e^{-\frac{\rho(\vec{x}+\vec{e}_i)}{\rho_0}}\right) = \\ =& -G\,(1 - e^{-\frac{\rho(\vec{x})}{\rho_0}})\,\rho_0^2 \left(0 - \sum_i \vec{e}_i\, w_i\, e^{-\frac{\rho(\vec{x}+\vec{e}_i)}{\rho_0}}\right) = \\ =& \; G\,(1 - e^{-\frac{\rho(\vec{x})}{\rho_0}})\,\rho_0^2 \sum_i \vec{e}_i\, w_i\, e^{-\frac{\rho(\vec{x}+\vec{e}_i)}{\rho_0}} & (71)\end{aligned}$$

Here $G < -4$ is a parameter defining the interface properties. For $G \geq -4$ there is no stable phase separation and for $G < -7$ the simulation usually will become unstable. The density ratio is approximately $\rho_{fluid}/\rho_{gas} \approx 70$ – although it can be much higher for more sophisticated pseudo-potential models [20] – and the surface tension coefficient is in the order of $\sigma \approx 0.1$, depending on G. Boundary conditions are realized by setting the density of *wall* lattice points either to ρ_{fluid} (wetting boundaries) or to ρ_{gas} (non-wetting boundaries) or somewhere in between. The force $\vec{f}_{\mathrm{SC}}$ is added to the other forces at play and ingested into LBM via the Gou forcing scheme as described in section 3.5.

In Shan-Chen, flux through the interface layer is possible, meaning that droplets can evaporate and condense elsewhere. The mass is conserved analytically apart from floating-point errors. However the interface layer is distributed across several lattice points, meaning it is rather diffuse. This makes drop impact simulations with Shan-Chen impractical, because the lattice dimensions would have to be enormous in order to resolve small droplets.

3.9.2 GPU Implementation Notes

While the implementation of the Shan-Chen method is rather straight-forward once volume forces with the Guo scheme have been implemented, on the GPU special attention is required for the temporal sequence of execution without data dependencies between neighboring lattice points. In order to make Shan-Chen thread-safe, it is essential to partly split the main `stream_collide()` kernel. The density ρ needs to be calculated first for every lattice point before `stream_collide()` is executed and in `stream_collide()` the density must not be written into memory to avoid race conditions.

4 LBM on the GPU

4.1 Challenges and Opportunities on parallel Hardware

LBM is a massively parallel algorithm and it needs massively parallel hardware to run efficiently. Traditional LBM implementations run on multiple CPUs in parallel and every CPU works on a local domain of the simulation box. To achieve at least reasonable speeds, the parallelization is not only done across multiple CPU cores on one CPU, but across multiple CPUs on different compute nodes with communication in between the nodes. However hardware efficiency on CPU implementations is quite low[2], even when the code is vectorized. Losses are mainly due to the required communication between CPU nodes, but also come from the cache hierarchy of the CPU microarchitecture itself.
LBM is the perfect fit for the GPU, where each GPU core calculates a single LBM lattice point (illustrated in figure 2). By making the right choices of swap algorithm and data structures, up to 99.7 %[3] of the theoretically available performance are leveraged by *FluidX3D*. On top of that, both the floating-point performance and memory bandwidth typically are an order of magnitude higher for GPUs[4], resulting in a speedup of two magnitudes compared to a multi-core CPU or three magnitudes compared to a single CPU core. This means that a GPU performs the same LBM simulation at approximately 2 %[5] the electricity cost compared to using a CPU cluster. This substantial advantage, both in speed and cost, drives the motivation to get through the difficulty of developing the GPU code.

GPUs offer only very limited freedom in being programmed. The OpenCL language is built upon the essentials of C99, with a few specific additions for the memory model. Complicated calculations have to be broken down into simple syntax. Restrictions [27] of the language include:

- only one-dimensional arrays with constant length are supported
- no dynamic allocation
- no classes or objects
- no recursion
- no pointers to functions

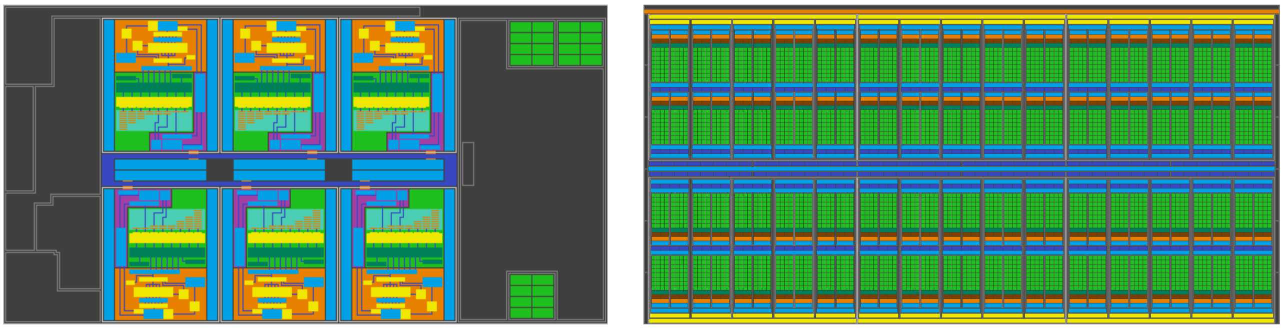


Figure 2: The microarchitectures of an Intel Coffee Lake CPU (i7-8700K, left) and an Nvidia Pascal GPU (GP102, right) illustrated as block diagrams [28–30]. While the CPU only has a few, but very sophisticated cores connected by a ring bus, the GPU has thousands of cores (green) operating in groups of 32 (so-called warps) and sharing cache (light blue). While each CPU core can execute instructions separately, on the GPU all cores within a warp have to execute the same instructions (single instruction multiple data (SIMD)).

---

[2] OpenLB for example reaches 142479 $MLUPs/s$ on 2732 12-core Intel Xeon E5-2690 v3 CPUs with a combined memory bandwidth of 185.78 TB/s [21–23], which is equivalent to 13.1 % efficiency.

[3] Measured on a Nvidia Tesla V100 GPU, see figure 9. The Tesla V100 offers 900 GB/s of memory bandwidth and achieves 5247 $MLUPs/s$ with D3Q19.

[4] As of November 2019, the fastest CPU is the Intel Xeon Platinum 9282 (56 cores, 9.3 $TFLOPs/s$, 282 GB/s) [24] while the fastest GPU is the Nvidia Tesla V100S (5120 cores, 16.4 $TFLOPs/s$, 1134 GB/s) [25]. Other GPUs with very fast memory are the AMD Radeon Instinct MI50/MI60 and Radeon VII (1024 GB/s) [26].

[5] A single Intel Xeon E5-2690 v3 CPU has a thermal design power (TDP) of 135 W [23] and a single Nvidia Tesla V100 has a TDP of 250 W [25].

4.2 Glossary

4.2.1 GPU Kernel

A kernel in OpenCL is the entry point for the GPU. Kernels are massively parallel in nature, being equivalent to the inner part of a `for` loop in C++. Example: In C++ what looks like

```
void example(float* data, const int N) {
  for(int n=0; n<N; n++) {
    data[n] += 1.0f;
  }
}
```

in OpenCL C looks like this:

```
kernel void example(global float* data) {
  const int n = get_global_id(0);
  data[n] += 1.0f;
}
```

The upper bound N is handed to the kernel call on the CPU side. The data elements automatically get distributed to the thousands of cores on the GPU. With this in mind, it also becomes obvious that if the GPU has 5120 cores and $N = 128$, GPU utilization will be very poor. Synchronization within a kernel is only possible for threads within a thread block and not globally across all N threads[6]. If global synchronization is required, the kernel must be split into two kernels which are called one after the other. For performance reasons, kernel splitting should be avoided whenever possible.

4.2.2 OpenCL Memory Model

OpenCL defines five distinct memory types [32]. These directly relate to different components of the available hardware [33]. The table below gives an overview:

memory type	description	host permissions	device permissions
host	main system memory on the CPU side very slow CPU ↔ GPU transfer over PCIe bus generally no direct access on the GPU	dynamic allocation read/write access	no allocation no access
global	dedicated video memory of the GPU slow (400-800 clock cycles latency [34–36]) accessible for all GPU threads	dynamic allocation read/write access	no allocation read/write access
constant	part of the dedicated video memory of the GPU cached for faster access than global memory[7] accessible for all GPU threads	dynamic allocation read/write access	static allocation read-only access
local/shared	part of the L1 cache of the GPU [33] fast access (4-8 clock cycles latency [35, 36]) only accessible for threads within a thread block	dynamic allocation no access	static allocation read/write access
private	registers of the GPU very fast access (0 clock cycle latency [35]) only accessible for a single thread	no allocation no access	static allocation read/write access

4.2.3 Memory Coalescence

When on the GPU consecutive threads access data at consecutive global memory addresses within a 128-byte aligned segment, the single accesses are coalesced into one, significantly reducing overall latency [35, 36]. This hardware optimization has to be considered for the data layout when implementing an algorithm on the GPU, because only coalesced access will leverage the full memory bandwidth capabilities. If the requirements for memory coalescence are not met, the transfer is called misaligned, and bandwidth is substantially reduced.

---

[6] Although not commonly used, there is a workaround for global synchronization using integer atomics [31].

[7] Read-only global memory access can be cached just like constant memory for multiple accesses within a kernel by using the C99 `restrict` type qualifier [37, p.141][38].

4.2.4 Branching on GPUs

Although `if-else` branching is supported on GPUs, it comes at severe performance impact [35, 36]. The reason is that threads always run in groups of 32 (warps) and within such a warp branching is not allowed, meaning that if one of the 32 threads has to execute a different branch than the others, all threads have to execute both branches and discard the results from the wrong branch afterwards. In past GPU microarchitectures such as Nvidia Kepler, the negative impact of branching is much larger because they have a larger warp size.

4.2.5 Race Conditions

A race condition occurs whenever two threads running in parallel write two different values to the same memory address. In this case, the value last written to memory determines the output, but it can be either of the two randomly. A race condition makes the program output non-deterministic and has to be avoided at all cost. For CPU code this is only rarely an issue, but in GPU code it can easily be overlooked. Example in LBM: Two neighboring lattice points assign flags to their neighbors; the common neighbors may receive two different values. Solution: both lattice points need to assign the same flag to their neighbors (or at least the same flag bit); then the order of assignment does not matter. A simple check for any race conditions is to run a non-stationary simulation with highly chaotic output twice and compare the results. Mismatch means there is a race condition present somewhere.

4.2.6 Measuring LBM Performance – MLUPs/s

The unit for determining LBM performance is called mega lattice updates per second ($MLUPs/s$) and refers to how many LBM lattice points are processed in one second. This unit is independent of the simulation box size. The theoretically achievable performance limit $[p_\text{LBM}] = MLUPs/s$ is calculated based on the (combined) memory bandwidth of the CPU(s)/GPU(s) and the velocity set DdQq

$$p_\text{LBM} = 10^{-6} \cdot \frac{b_\text{HW}}{9\,q\,\frac{bytes}{\text{lattice point}}} \tag{72}$$

whereby b_HW denotes the memory bandwidth of the device. The 9 comes from 4 bytes per DDF times two (load and store) plus 1 byte per flag (load only). Figure 3 shows the LBM performance of *FluidX3D* compared to the theoretical maximum for a variety of different hardware.

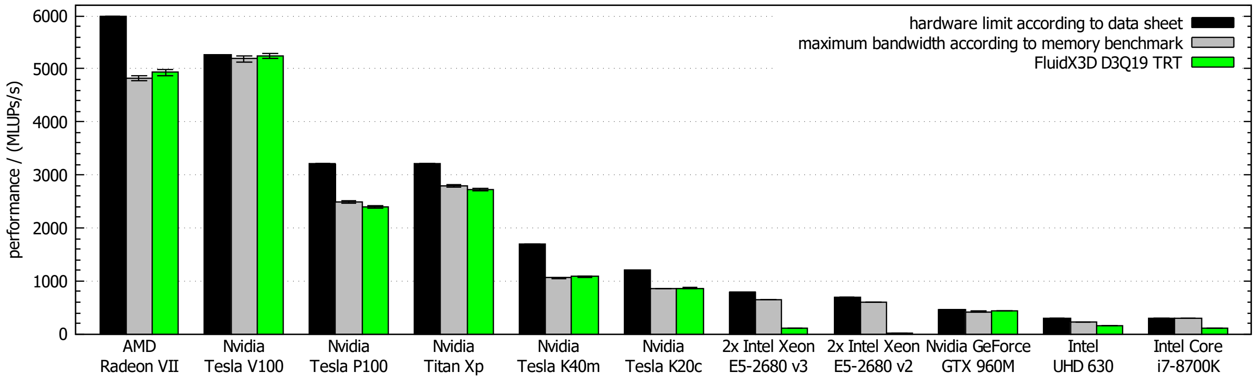


Figure 3: Performance of *FluidX3D* on various GPUs and CPUs. Some hardware cannot hold up to the values claimed in the data sheet and the actual performance measured in a benchmark program is lower. For more details see figure 5.

4.3 Special LBM Optimizations on GPUs

GPU programming is an art in itself and in many ways very different and much more complicated than regular CPU programming. While for the CPU for example vectorization is a common optimization technique, on the GPU literally everything is vectorized. On the other hand, modern CPUs manage the caching hierarchy automatically, while the GPU in this regard is much less advanced, making the explicit cache management a responsibility of the programmer. Based on the nature of GPU microarchitectures, there is an entire suite of optimizations, each with very specific criteria and use-cases. In this chapter, the catalog of optimization is applied specifically to LBM.

4.3.1 Bottlenecks: Memory Bandwidth and Amount of Memory

As will become evident in section 5.6, LBM without extensions is bottlenecked by memory access only. With this in mind, the main focus is to eliminate any non-essential memory transfers and to speed up the essential memory transfers by making use of special data access patterns.

The size of the simulation box and thus the volumetric resolution is limited by the amount video memory available on the GPU. As of today, the record goes to the Nvidia Quadro RTX 8000 with 48 *GB* of memory [39]. This hardware limit can only be pushed so much further by for example extending the code to run on multiple GPUs, which for the *FluidX3D* software was done by Fabian Häusl very successfully [40]. However memory requirements scale cubically with the simulation box dimensions. The only way to go further is to use an adaptive lattice, an optimization that is planned to be implemented in the future.

4.3.2 Eliminating non-essential Memory Transfers

LBM solely operates on the DDFs. Although the density and velocity locally have to be calculated in every step for every lattice point (eq. 2) it is not mandatory to write them to memory in every simulation time step. In fact, the memory transfers are orders of magnitude more costly than calculating density and velocity in registers in the first place.

Another non-essential data transfer is the flags, more specifically using the 32-bit `uint` data type when really only a few bits are in use. The smallest data type available in OpenCL is the 8-bit integer `uchar`, which is just enough to store the flags even for all of the extensions combined.

4.3.3 Avoidance of PCIe Data Transfer

Data transfer between CPU and GPU happens over the PCIe bus (illustrated in figure 4), the bandwidth of which is limited substantially (unidirectional 8 GB/s for PCIe 3.0 x16 and 16 GB/s for PCIe 4.0 x16 plus latency; twice of that for bidirectional transfers). CPU ↔ GPU transfer are required for initialization and for storing simulation results on the hard drive. During the simulation however, regular PCIe data transfer must be avoided at all cost. Especially for some extensions to LBM like the immersed-boundary method, transferring the velocity field to the CPU in every simulation step in order to perform the particle integration on the CPU side would totally ruin performance. Instead it is much faster to do all calculations on the GPU, including the IBM particle integration, even if it might not be very efficient itself. By not using any PCIe data transfer, there is a lot of performance to spare.

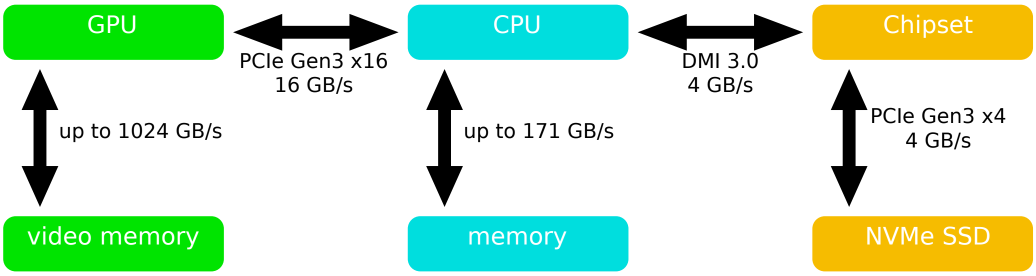


Figure 4: Memory transfer bandwidths of a modern PC illustrated. The best performance for LBM is achieved if only the connection between the GPU and video memory is used during simulation.

4.3.4 Data Layout

There are two main types of data layout: array of structures (AoS) and structure of arrays (SoA) [41–43]. For LBM density distribution functions $f_i(n)$[8] with a D3Q7 velocity set in a $3x1x1$ sized lattice, AoS means that the order in memory is

$f_0(0)$	$f_1(0)$	$f_2(0)$	$f_3(0)$	$f_4(0)$	$f_5(0)$	$f_6(0)$	$f_0(1)$	$f_1(1)$	$f_2(1)$	$f_3(1)$	$f_4(1)$	$f_5(1)$	$f_6(1)$	...

and for SoA the order in memory is

$f_0(0)$	$f_0(1)$	$f_0(2)$	$f_1(0)$	$f_1(1)$	$f_1(2)$	$f_2(0)$	$f_2(1)$	$f_2(2)$	$f_3(0)$	$f_3(1)$	$f_3(2)$	$f_4(0)$	$f_4(1)$	...

Each thread accesses all streaming directions i, but only for a single position n. Thus in the AoS layout, consecutive threads access consecutive *blocks* of memory, so there is no coalescence. In the SoA layout on the other hand, consecutive threads do access consecutive memory addresses for each i individually, meeting the requirements for memory coalescence. In practice, SoA makes LBM on the GPU more than 3 times faster compared to when AoS is used.

4.3.5 Choice of Swap Algorithm

The major choice for the LBM implementation is the swap algorithm, swap standing for the streaming step in which neighboring density distribution functions are swapped. While the collision operator only works on the DDFs locally, the streaming step reads from / writes to neighboring lattice points. This is a challenge to handle on massively parallel hardware, where all lattice points are calculated at roughly the same time, with the exact temporal order being random. The simple solution to resolve any data dependencies is to use two data fields f^{A} and f^{B} in global memory, in even time steps only read from f^{A} and write to f^{B} and in odd time steps the other way around by swapping pointers in the kernel arguments after each time step. Such a solution, in which the order of execution of work-items does not matter, is called *thread-safe*.

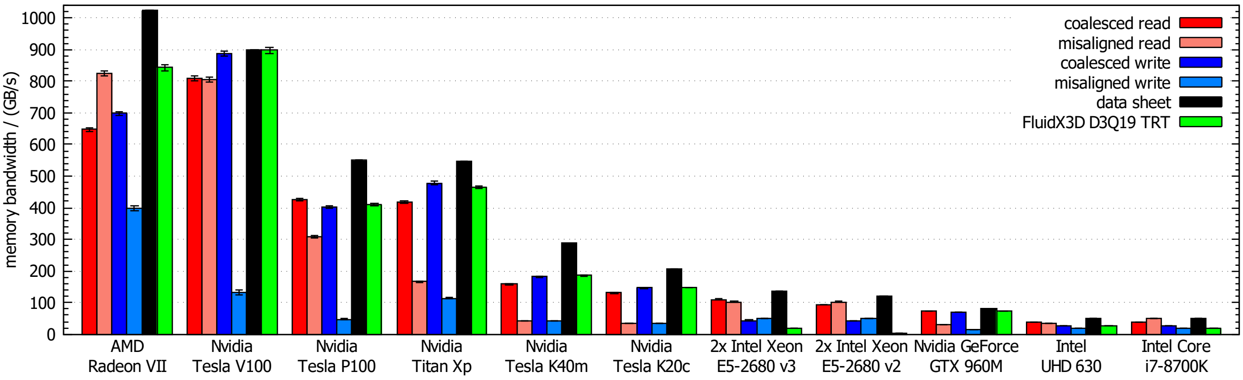


Figure 5: Read and write performance of a variety of GPUs and CPUs measured using a self-written OpenCL benchmark tool. For each of the measurements (coalesced/misaligned read/write) a small kernel is designed and the time of execution is measured. In order to get the values closest to the data sheet, the fastest execution time of several hundred kernel executions is chosen as the result. From the knowledge of how much data transfer of which type is contained in the kernels, the respective memory bandwidths are calculated. Besides the data sheet values (black), the memory bandwidth in use by *FluidX3D* (with the one-step-pull swap algorithm) is also included in the plot (green). The takeaway is that the performance penalty for misaligned writes is much larger than for misaligned reads on most hardware.

The naive implementation of LBM would be to perform streaming and collision separately (so-called two-step swap), reading and writing every DDF twice per simulation step. Since memory bandwidth is the bottleneck of LBM, memory access has to me minimized. By combining streaming and collision into a single kernel, memory access is cut in half (one-step swap). The intermediate DDFs are kept locally in registers for the collision.
Then there is the question of what to do first, streaming or collision. One-step can be both stream-collide (one-step-pull) or collide-stream (one-step-push); both implementations achieve exactly the same with exactly

---

[8] $n = x + (y + z\,L_y)\,L_x$ is the linear index for the 3D position $\vec{x} = (x, y, z)^T$, x, y, z being integers in dimensionless lattice units and L_x, L_y, L_z being the lattice dimensions. The subscript i denotes the streaming direction index.

the same floating-point errors. Still the pull variant is better than the push on GPUs and there is a reason for that.

While access to the DDFs belonging to the local lattice point is coalesced with the SoA data layout, access to the neighbor DDFs, which is inevitable in the streaming step, is at least partly misaligned [42, 44–46]. Now it is important to observe that the performance penalty for misaligned global memory access is different for reads and writes (see figure 5). On most GPUs the performance penalty for misaligned writes is much higher than for misaligned reads, which is due to the use of the L2 cache in reading operations [44]. The one-step algorithm with its pull and push variants provides a free choice where to have the misaligned access – in reads (pull) or writes (push). This is why one-step-pull is the fastest swap algorithm and therefore the choice in *FluidX3D*.

There are also swap algorithms which are more symmetric in their use of misaligned reads/writes, namely esoteric twist [47], AA-pattern [48] and more modern variants of the latter [49], all of which have in common that they do not need a second copy of the DDFs in memory, cutting memory requirements almost in half but also adding a lot more complexity to the implementation. In terms of performance, these are expected to be somewhere in between one-step-push and one-step-pull.
A very detailed description and analysis of all different swap algorithms is given in [50], so here the choices are only briefly presented in table 1. There are also two more swap algorithms specifically designed for CPUs, the so-called swap [51, 52] and compressed-grid [53], both of which are not thread-safe, meaning that they cannot be used on GPUs at all; thus they are not included in table 1.

algorithm	memory requirements	memory transfers	coalesced reads	partly misaligned reads	coalesced writes	partly misaligned writes
two-step-pull	double DDFs	$4\,q$	$q+1$	$q-1$	$2\,q$	0
two-step-push	double DDFs	$4\,q$	$2\,q$	0	$q+1$	$q-1$
one-step-pull	double DDFs	$2\,q$	1	$q-1$	q	0
one-step-push	double DDFs	$2\,q$	q	0	1	$q-1$
esoteric twist	single DDFs	$2\,q$	$(q+1)/2$	$(q-1)/2$	$(q+1)/2$	$(q-1)/2$
AA-pattern	single DDFs	$2\,q$	$(q+1)/2$	$(q-1)/2$	$(q+1)/2$	$(q-1)/2$

Table 1: Average amount of memory access types per LBM time step for various swap algorithms.

4.3.6 Why local/shared Memory is not useful in LBM

Access to local memory is very fast and possible for all GPU threads within a thread block, enabling local communication between threads. It is useful to avoid

A multiple accesses to the same memory address in global memory (example: cache tiling for multiplication of large matrices [54]; here approximately a 10 times speedup can be achieved) or to avoid

B misaligned global memory access and instead do a coalesced memory transfer from global into local memory and then do the misaligned access in local memory (such that memory bandwidth to global memory is enhanced, but speedup is much less than in case A).

For the DDFs of LBM however none of these use-cases apply. In every time step, each DDF is loaded from and written to memory exactly once, so no multiple accesses occur (point A does not apply). Due to the linear space index, turning misaligned into coalesced global access (point B) in the streaming step would only work for DDFs shifted along the x-direction, and then only if the box size in x-direction is equal to the thread block size, otherwise at the edge of the thread block communication between thread blocks is necessary, requiring a second kernel to be executed every time step [46], which is undesirable and does cost more performance than is gained with the potentially faster coalesced transfer.

There is one place where local memory could be useful in LBM: the flags. In every LBM step, for every lattice point, the entire DdQq neighborhood of flags has to be read from global memory in order to decide for boundaries in the streaming step. Parts of the neighborhoods of adjacent lattice points are the same, meaning that the same memory address in global memory is accessed multiple times from adjacent threads (case A). With linear space indexing $n = x + (y + z\,L_y)\,L_x$, threads are adjacent on the x-axis, meaning that each thread only needs to load a $1 \cdot 3 \cdot 3$ slice perpendicular to the x-axis of its flag neighborhood from global into local memory and can access neighboring slices within local memory. Special care needs to be taken at the lower and upper ends of the thread block: At the lower/upper end, neighbors to the lower/upper side are outside of the thread block and need to be loaded additionally into local memory by the two threads located at the lower/upper end of the thread block. With bit masking for the neighbor index calculation, this is realized within a single branching. Special consideration also needs to be taken to not have any threads returning *before* the entire flag loading process is completed (for example, if threads whose lattice point is *wall* return right away, parts of the local memory buffer will remain in an uninitialized state, creating a race condition). Table 2 below shows the theoretically achievable speedup for every lattice set in the two edge cases $THREAD\_BLOCK\_SIZE = 2$ and $THREAD\_BLOCK\_SIZE = \infty$. Real-world however does not hold up well. Flags are in the `uchar` format with 1 byte each; DDFs are in the `float` format with 4 byte each.

velocity set	saved transfer overall in bytes ($TBS = 2$)	saved transfer overall in bytes ($TBS = \infty$)	real-world performance gain/loss
D2Q9	$3/81 = 3.7\%$	$6/81 = 7.4\%$	0.0%
D3Q7	$1/63 = 1.6\%$	$2/63 = 3.2\%$	-1.3%
D3Q13	$0/117 = 0.0\%$	$4/117 = 3.4\%$	0.0%
D3Q15	$1/135 = 0.7\%$	$6/135 = 4.4\%$	-0.8%
D3Q19	$5/171 = 2.9\%$	$10/171 = 5.8\%$	0.1%
D3Q27	$9/243 = 3.7\%$	$18/243 = 7.4\%$	0.0%

Table 2: Theoretical and measured overall speedups for the different velocity sets when the shared memory optimization for the flags is applied.

Although this optimization looks promising on paper, it does not make any significant difference, sometimes even worsening performance. All velocity sets except D2Q9 and D3Q27, due to their neighbor stamp geometry, cannot even use the full potential of local memory. Moreover, the flags – if the data type `uchar` is used – only represent a minority of the total data transfer per simulation step, and using the local memory optimization in this case also entails additional branching and required synchronization of the thread block. On top, implementation difficulty is substantial. In summary, local memory is not useful for the base implementation of LBM.

4.3.7 Direct/Indirect Memory Addressing

Direct memory addressing refers to when from the thread ID through some integer arithmetic all memory locations, be it of the local lattice point or its neighbors, are calculated directly without making use of a remote lookup table in global memory. This is beneficial if the computational complexity for index calculation is low and the algorithm is bottlenecked by memory bandwidth. *FluidX3D* utilizes direct memory addressing.
There is also indirect memory addressing where an additional lookup table in global memory is used. This is beneficial when there is no clear algebraic pathway from the thread ID to the memory address, which in LBM is beneficial for simulations where a significant volume fraction of the simulation box taken up by *wall* lattice points in order to not have to store them explicitly.

4.3.8 Micro-Optimization and Loop Unrolling

Although the main focus is not on arithmetic optimization, some LBM extensions combined can shift the arithmetic intensity over into the compute limit if no arithmetic optimization is present at all. Micro-optimization refers to the technique of replacing arithmetic operations with different ones that occupy less clock cycles while doing the same calculations. A few examples:

- Divisions are more computationally costly than multiplications.
- Redundant parts of equations should be pre-calculated only once.

- GPUs support the reciprocal square root operation in hardware, making it just as fast as a multiplication.
- Trigonometric functions occupy hundreds of cycles and need to be avoided whenever possible, for example by applying trigonometric identities.
- Branching can be extremely costly on GPUs and can often be avoided by bit masking.

The general rule is: never trust the compiler to do any optimization for you.

Loop unrolling is another powerful micro-optimization tool. It makes the compiler unroll loops with fixed length, in the process pre-calculating all integer indices and removing floating-point additions with zero, which takes a lot of load from the GPU later on. In cases where in the loop there is a lot of floating-point and indexing arithmetic (for example the matrix multiplication[9] in the MRT collision operator), performance is significantly increased. However there are also some instances where loop unrolling will reduce performance by over-elongating the assembly to the point where it does not anymore fit into instruction cache entirely. This is the case when for example in the loop there is a lot of branching.

4.3.9 Arithmetic Optimization by exploiting numerical Loss of Significance

Mathematically, sums are commutative, but when working with finite precision floating-point formats, it is an interesting observation that sums here are not commutative and adding numbers in a different order will yield a slightly different result with a different error. Moreover, for floating-point formats half of all possible values are within the interval $[-1,\ 1]$, meaning that numerical loss of significance[10] for small numbers is less. This can be used as an optimization technique – not with the purpose to make an algorithm faster, but rather to make it more accurate at the exact same cycle count – whereby the goal is for large summations to keep intermediate results as small as possible. In the case of LBM, there are two prominent locations in the algorithm to apply this optimization. The first is at the density summation (details further below)

$$\rho = \sum_i f_i \tag{73}$$

and the second is where the velocity is calculated as the sum of fluid populations weighted with the streaming directions.

$$\vec{u} = \frac{1}{\rho} \sum_i \vec{c_i} f_i \tag{74}$$

When inserting $\vec{c_i}$ explicitly in eq. (74), we get for the D3Q19 velocity set

$$\begin{aligned}
u_x &= \frac{1}{\rho}\left(f_1+f_7+f_9+f_{13}+f_{15}-(f_2+f_8+f_{10}+f_{14}+f_{16})\right)\\
u_y &= \frac{1}{\rho}\left(f_3+f_7+f_{11}+f_{14}+f_{17}-(f_4+f_8+f_{12}+f_{13}+f_{18})\right)\\
u_z &= \frac{1}{\rho}\left(f_5+f_9+f_{11}+f_{16}+f_{18}-(f_6+f_{10}+f_{12}+f_{15}+f_{17})\right)
\end{aligned}$$

but if the order of summation instead is changed to alternating + and − like this

$$\begin{aligned}
u_x &= \frac{1}{\rho}\left(f_1-f_2+f_7-f_8+f_9-f_{10}+f_{13}-f_{14}+f_{15}-f_{16}\right)\\
u_y &= \frac{1}{\rho}\left(f_3-f_4+f_7-f_8+f_{11}-f_{12}+f_{14}-f_{13}+f_{17}-f_{18}\right)\\
u_z &= \frac{1}{\rho}\left(f_5-f_6+f_9-f_{10}+f_{11}-f_{12}+f_{16}-f_{15}+f_{18}-f_{17}\right)
\end{aligned}$$

---

[9] In the case of unrolled matrix multiplications, the PTX assembly [38] mostly contains the fused-multiply-add (fma) instruction, which is the only instruction that can leverage the full floating-point capabilities of any CPU/GPU as it executes a multiplication and an addition within a single clock cycle. Cache tiling is not an option for the MRT matrix multiplication since it is entirely calculated in registers already.

[10] When subtracting two very similar numbers, the difference might be only the last few digits, canceling out all other digits. Example 1: $0.12345678 - 0.12345600 = 0.00000078 = 7.80000000\text{E-}7$; even though both initial numbers have 8 significant digits, the result only has 2. Example 2: $1.00000000 + 1.23456789\text{E-}7 = 1.00000012$.

the entire simulation will be measurably more accurate.

There is another optimization regarding numerical loss of significance especially for LBM, which significantly improves accuracy. The trick is to not deal with the density distribution functions f_i directly, but with them shifted down by the lattice weights, namely $f_i^{\text{shifted}} = f_i - w_i$. The reason for this is that generally speaking, the density ρ is always not far off from 1 and velocities $\vec{u}$ are small meaning that the difference of the equilibrium density distribution functions and the equilibrium density distribution functions for $\rho = 1$ and $\vec{u} = 0$ is also small.

$$f_i^{\text{eq-shifted}}(\rho, \vec{u}) := f_i^{\text{eq}}(\rho, \vec{u}) - f_i^{\text{eq}}(\rho = 1, \vec{u} = 0) = f_i^{\text{eq}}(\rho, \vec{u}) - w_i \tag{75}$$

Now right within eq. (75) there is a subtraction of two similarly sized numbers, resulting in loss of significance within the equation. But now $f_i^{\text{eq-shifted}}$ and therefore also f_i^{shifted} after the collision operator are very small numbers close to zero, meaning that loss of significance is reduced everywhere else in the code significantly. The only places where a change due to the shifting has to be introduced are the density summation, where the sum of all lattice weights $\sum_i w_i$ has to be added again, and the calculation of forces on boundaries in eq. 52. The lattice weights w_i are normalized, so that conveniently $\sum_i w_i = 1$, resulting in

$$\rho = 1 + \sum_i f_i^{\text{shifted}} \tag{76}$$

and this is where again the summation order comes in. When adding the +1 before the summation, very small numbers $f_i^{\text{shifted}} \ll 1$ are added to a 1, resulting in large loss of significance. But when instead the +1 is added after summation, loss of significance will be mitigated significantly:

$$\rho = \left(\sum_i f_i^{\text{shifted}} \right) + 1 \tag{77}$$

In the velocity summation, all of the lattice weight shifts cancel out, making this optimization easy to implement.

4.3.10 OpenCL Code Injection at Runtime

The OpenCL C code which runs on the GPU is not compiled along with the C++ code, but later at runtime. Besides the great advantage of largely better compatibility with CPUs/GPUs/FPGAs from many vendors, this means that after C++ compilation, the OpenCL C code can still be modified at runtime right before it gets compiled for the GPU. The OpenCL C compiler has a preprocessor just like C99, so `#define` can be used inside the OpenCL C code. The last two aspects combined mean that constants, such as for example the relaxation time τ, can be embedded into the OpenCL C code at runtime, meaning these constants will later reside in the instruction cache and do not have to be loaded from global/constant memory.

In order to embed the `string` of OpenCL C code into the executable at C++ compilation instead of reading it in from a separate source file at runtime, the common way was to embed it as a `string` literal with quotation marks `"code"` for every single line. This is not only bothersome, but also prevents text highlighting in the editor. The next better way would be to use a *raw string literal* with the syntax `R"(code)"`, which at least works over multiple lines, but still prevents text highlighting. There is however a smarter way, the so-called *macro stringification*. The C++ macro[11]

```
#define R(...)  string(#__VA_ARGS__" ")
```

combined with a bit of subsequent character replacement[12] enables text highlighting while the syntax for the code stays very similar to the familiar raw string literal with `R(code)`.

---

[11] The `...` means that `R(separated, code)` can have multiple ”arguments” separated with `,`. Without it, if the OpenCL C code contains a `,` character, the method would not work. `__VA_ARGS__` denotes all ”arguments” separated by an arbitrary number of `,` characters, and `#` converts all of the ”arguments” including the `,` separators into a string literal. Finally, `" "` and `string()` allow for concatenating multiple `R(code)+R(code)` with the `+` operator.

[12] All spaces need to be replaced with the newline character `\n`, except after preprocessor commands.

4.3.11 16-bit Floating-Point Storage for DDFs

The major part of memory bandwidth (and thus execution time) is taken up by reading/writing the DDFs from/to video memory. With the assumption that the majority of the error of LBM originates not from floating-point arithmetic itself but rather from the second-order collision operator and velocity discretization, it seems prudent to reduce floating-point accuracy where it matters most. The trick is to convert the FP32 DDFs to FP16 just before a memory store is happening and write them as FP16 values with reduced precision into memory, then later load them as FP16 from memory and right afterwards convert them back into FP32 and do all of the arithmetic unchangedly in FP32. It is very convenient that modern GPUs can do the FP32↔FP16 conversion in hardware within a single clock cycle. Accuracy can be improved by designing a custom FP16 floating-point format with reduced exponent range but therefore improved mantissa accuracy. Some people are still not confident with using FP32 in the first place instead of FP64, so this will be a future topic to explore in more detail. Until then, all of the simulations for this thesis are performed with regular FP32.

5 The *FluidX3D* Simulation Software

In order to gain fundamental understanding about every single line of code, its functions, impact on performance and dependencies, not an existing simulation package is extended, but the LBM code is written completely from the ground up in the GPU programming language OpenCL. With the main motivation of achieving maximum efficiency, LBM is implemented in a modular approach where combinations of velocity sets and collision operators can easily be selected and extensions can be enabled when needed by uncommenting a single line of code. While the entirety of the GPU code is quite extensive, after splicing by the C++ compiler the active[13] part of the OpenCL C code is still comprehensible. So far, the implemented extensions include various boundary conditions, volume forces, boundary forces, a temperature model, the immersed-boundary method, the Shan-Chen model and the Volume-of-Fluid model. Although existing software such as ESPResSo [55] or OpenLB [56–58] is rich in features, it is not designed primarily with efficiency in mind. This is not an issue for running a single simulation for a few weeks for one particular use-case, but will quickly become a problem once a large number of large-scale simulations is required for scientific case-studies. The speedup of going from a CPU to a GPU implementation is substantial, somewhere between two and three magnitudes. Additionally, the VoF model, in which progress has been made only very recently, is not available in these existing simulation packages, so it would have to be implemented anyways and the implementation is much easier in a familiar code environment.

5.1 Choice of OpenCL

There are two choices for programming GPUs: CUDA and OpenCL [37, 59]. CUDA is proprietary to Nvidia hardware whereas OpenCL is the open-source industry standard. There are claims that for Nvidia GPUs CUDA would have a performance advantage [60], but there very likely not the full set of optimizations possible in OpenCL was applied. When proper optimization is done, there is no performance advantage left. The compilers for CUDA and OpenCL produce very different assembly, but overall there is no significant performance advantage of one over the other [61]. The large advantage of OpenCL is its widespread adoption across hardware. Not only GPUs from Nvidia, AMD and Intel can execute OpenCL code, but also CPUs from AMD and Intel with full autovectorization support [62, 63] as well as FPGAs from various vendors. Hence, for *FluidX3D* the choice is OpenCL.

5.2 List of GPU Kernels

Each kernel reads certain quantities from memory, performs some arithmetic and writes other quantities back into memory. All instances of a kernel run in parallel on every lattice point with the exact order of execution being random. To avoid race conditions, it is important to distinguish if a quantity is read from / written to only the local lattice point for which the kernel is executed or also its neighbors. There are no problems if a kernel reads a quantity x only locally and then writes x back locally. Race conditions occur – if not explicitly prevented – if a kernel

A reads a quantity x from its neighbors and at the same time writes x locally,

B reads a quantity x locally and at the same time writes x to its neighbors,

C writes a quantity x to its neighbors or

D reads a quantity x from its neighbors and at the same time also writes x to its neighbors.

For the DDFs f_i and also for g_i with the pull-variant swap, point A is prevented by having two copies of these quantities in memory. Case B would occur in the push-variant swap and would be prevented the same way. Cases C and D are especially difficult to circumvent and happen in the `integrate_ibm()` and `surface_1()`/`surface_2()` kernels in the IBM and free surface extensions respectively. The solutions here are for `integrate_ibm()` to use atomic addition and for `surface_1()`/`surface_2()` to only change independent bits of the flags, in essence treat every bit of the flags as a separate quantity.
The tables below provide an overview of the kernel arguments. Notes:

* only read from *wall* neighbors if moving boundaries are enabled

** only written if specifically enabled

*** only read but not written for temperature equilibrium boundaries

---

[13] toggled with C++ preprocessor directives

5.2.1 Without Extensions

kernel	only reads locally	reads locally and from neighbors	only writes locally	writes locally and to neighbors
`initialize()`	ρ, $\vec{u}$	flags	f_i^{A}, flags, $\vec{u}$	
`stream_collide()`		f_i^{A}, flags, $\vec{u}$*	f_i^{B}, ρ**, $\vec{u}$**	

5.2.2 With Temperature

kernel	only reads locally	reads locally and from neighbors	only writes locally	writes locally and to neighbors
`initialize()`	ρ, $\vec{u}$, T	flags	f_i^{A}, flags, $\vec{u}$, g_i^{A}	
`stream_collide()`	T***	f_i^{A}, flags, $\vec{u}$*, g_i^{A}	f_i^{B}, ρ**, $\vec{u}$**, g_i^{B}, T***	

5.2.3 With Immersed-Boundary

kernel	only reads locally	reads locally and from neighbors	only writes locally	writes locally and to neighbors
`initialize()`	ρ, $\vec{u}$	flags	f_i^{A}, flags, $\vec{u}$, $\vec{f}$	
`stream_collide()`	$\vec{f}$	f_i^{A}, flags, $\vec{u}$*	f_i^{B}, ρ**, $\vec{u}$**, $\vec{f}$	
`integrate_ibm()`	$\vec{x}_p$	$\vec{u}$		$\vec{f}$

5.2.4 With Shan-Chen

kernel	only reads locally	reads locally and from neighbors	only writes locally	writes locally and to neighbors
`initialize()`	ρ, $\vec{u}$	flags	f_i^{A}, flags, $\vec{u}$	
`stream_collide()`		f_i^{A}, flags, ρ, $\vec{u}$*	f_i^{B}, $\vec{u}$**	
`calculate_rho_u()`	f_i^{B} flags		ρ, $\vec{u}$	

5.2.5 With free Surface

kernel	only reads locally	reads locally and from neighbors	only writes locally	writes locally and to neighbors
`initialize()`	ρ, $\vec{u}$, φ	flags	f_i^{A}, flags, $\vec{u}$, φ, m, m_{ex}	
`stream_collide()`	m	f_i^{A}, flags, $\vec{u}$*, φ, m_{ex}	f_i^{B}, flags, ρ, $\vec{u}$, m	
`surface_1()`		flags		flags
`surface_2()`		ρ, $\vec{u}$, flags	f_i^{B}	flags
`surface_3()`	ρ, m	flags	$\vec{u}$, flags, φ, m, m_{ex}	

5.3 Multi-GPU Communication Requirements

For the multi-GPU implementation of *FluidX3D*, which has been done by Fabian Häusl [40], it is important to know which of the quantities have to be exchanged at the simulation domain boundaries at which point in time. The criteria for a required exchange of a quantity x between two consecutive kernels X and Y are:

A Y reads x from its neighbors and X previously modified x.

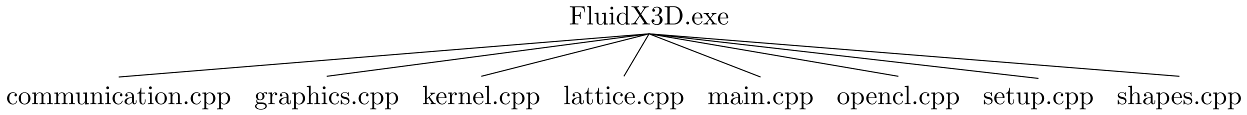
B Y reads x and X previously modified x in its neighbors.

These criteria also hold true if the kernels X and Y are the same kernel.

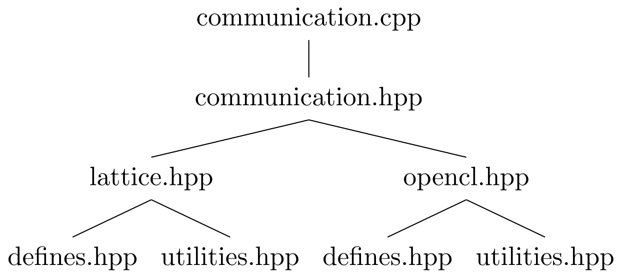
5.4 Source File Dependencies

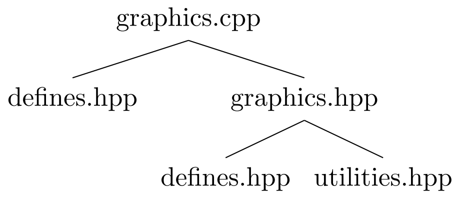
The source code of *FluidX3D* consists of 8 source files and 10 header files comprising approximately 11000 lines of code, thereof about 3000 lines in OpenCL C:

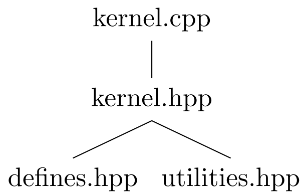
communication.cpp	constains MPI communication stuff for using multiple GPUs
graphics.cpp	contains a graphics library (only works on Windows, ignore for Linux development)
kernel.cpp	contains most of the OpenCL GPU code, including the LBM algorithm
lattice.cpp	implements functions of the Lattice class which is defined in lattice.hpp
main.cpp	contains the entry point (main_physics())
opencl.cpp	contains code for setting up OpenCL and compiling the OpenCL code
setup.cpp	contains the simulation setup (boundary geometry, initial conditions)
shapes.cpp	contains methods for initializing boundary geometries, for example pipes or spheres
communication.hpp	header for communication.cpp
defines.hpp	contains all settings, for example D3Q19 or on which device *FluidX3D* should run
graphics.hpp	header for graphics.cpp (only works on Windows, ignore for Linux development)
kernel.hpp	header for kernel.cpp, contains the stringification macro and other string preparation
lattice.hpp	contains the Lattice class definition
main.hpp	contains an Info struct for console prints without affecting the LBM core
opencl.hpp	includes the OpenCL C++ bindings and defines some OpenCL auxiliary functions
setup.hpp	header for setup.cpp to reduce confusion in setup.cpp
shapes.hpp	header for shapes.cpp, is included in setup.cpp
utilities.hpp	contains math, string and other functions for making programming easier

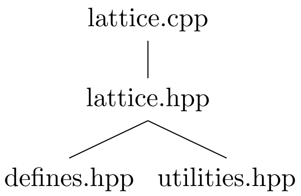


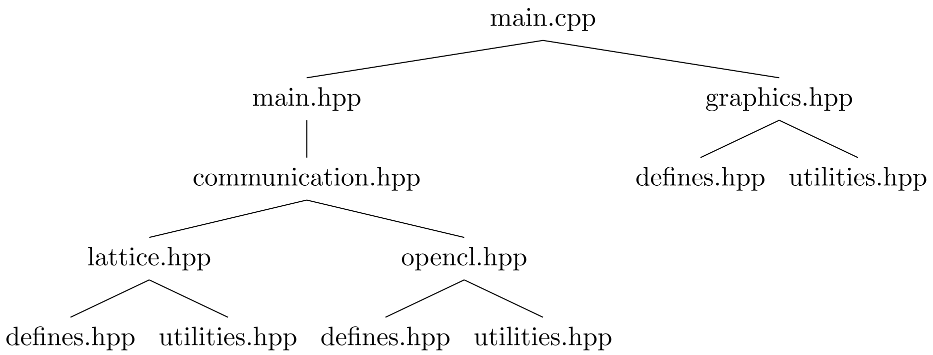
The individual source files include various header files and the header files also include other header files:

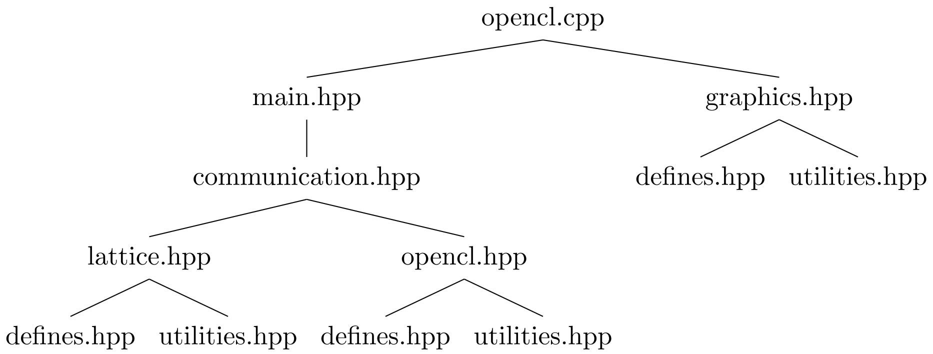


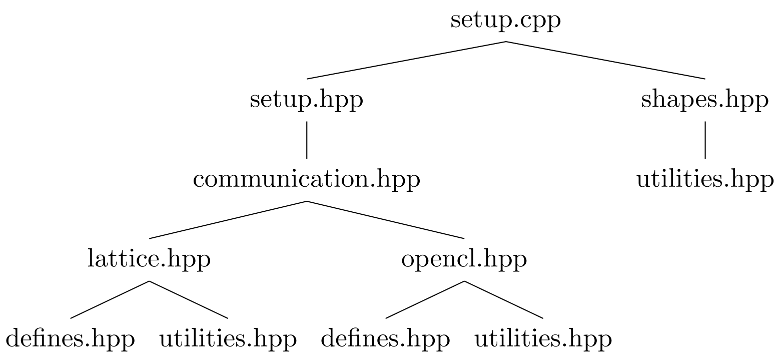


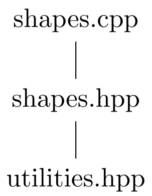

kernel.cpp
kernel.hpp
defines.hpp
utilities.hpp


lattice.cpp
lattice.hpp
defines.hpp
utilities.hpp


main.cpp
main.hpp
graphics.hpp
communication.hpp
defines.hpp
utilities.hpp
lattice.hpp
opencl.hpp
defines.hpp
utilities.hpp
defines.hpp
utilities.hpp


opencl.cpp
main.hpp
graphics.hpp
communication.hpp
defines.hpp
utilities.hpp
lattice.hpp
opencl.hpp
defines.hpp
utilities.hpp
defines.hpp
utilities.hpp


setup.cpp
setup.hpp
shapes.hpp
communication.hpp
utilities.hpp
lattice.hpp
opencl.hpp
defines.hpp
utilities.hpp
defines.hpp
utilities.hpp


shapes.cpp
shapes.hpp
utilities.hpp

5.5 Example Simulation Setup: Crown Splashing from Section 10.3

Listing 2 shows the setup for the simulation in section 10.3 as an example of how a setup is built in the code in `setup.cpp`. At first, the experimental parameters in SI-units are listed. After defining a length scale, a velocity and a density in simulation units, the three independent base units m, kg, and s are calculated by **`units.set_m_kg_s(...)`**. Then the rest of the parameters is converted from SI-units into simulation units and the simulation is initialized by calling `initialize(...)`, which allocates the memory and compiles the OpenCL C code. Then the simulation geometry is built. The function `sphere_plic(...)` returns the fill level φ of all lattice points inside a sphere or on the sphere surface, allowing for generating a sphere of *fluid* LBM points and setting the fill levels for the *interface* points such that the surface is smooth at initialization. With the first call of `run(0)`, the initialization state of the simulation is automatically transferred from CPU to GPU memory, but no LBM time step is executed yet. At initialization, the interface layer above the liquid pool is also created, among others. Finally, with `run(int n)` the simulation is executed for predefined time intervals and with the function call `write_image()` a picture of the current simulation state is written to the hard drive.

```
void main_setup() { // setup from "Crown behavior and bubble entrainment during a drop impact on a liquid film"

  // parameters from paper in SI-units
  const float si_rho = 1177.9f; // fluid density [kg/m^3]
  //const float si_mu = 8.36E-4f; // fluid dynamic viscosity [Pa*s], Pa=kg/(m*s^2)
  //const float si_nu = units.si_nu_mu(si_mu, si_rho); // fluid kinematic shear viscosity [m^2/s]
  const float si_sigma = 66.1E-3f; // fluid surface tension [N/m], N=kg*m/s^2
  const float si_g = 9.81f; // gravity acceleration [m/s^2]
  const float alpha = 0.0f; // impact angle in degrees, 0 = vertical

  const float si_D = 3.1E-3f; // drop diameter [m] (3.3-3.7)
  const float si_Lx = 30.0E-3f; // simulation box width [m]
  const float si_Lz = 25.0E-3f; // simulation box height [m]
  const float si_H = 2.0E-3f; // liquid pool height [m]

  // define velocity either directly or via We or Fr or free fall drop height
  //const float si_u = 1.86f; // impact velocity [m/s] (2.1-3.8)
  const float si_We = 2010.0f; // Weber number We = rho*u^2*L/sigma
  const float si_u = units.si_u_We(si_We, si_D, si_sigma, si_rho); // define impact velocity by Weber number
  //const float si_Fr = 20.0f; // Froude number Fr = u/sqrt(g*L)
  //const float si_u = units.si_u_Fr(si_Fr, si_D, si_g); // define impact velocity by Froude number
  //const float si_h = 0.79f; // drop height
  //const float si_u = units.si_u_h(si_h, si_g); // define impact velocity by free fall drop height

  //print_info("Re = "+to_string_old(units.si_Re(si_D, si_u, si_nu)));
  const float si_Re = 1168.0f;
  const float si_nu = units.si_nu_Re(si_Re, si_D, si_u);

  // determine a length, a velocity and the mean density in simulation units
  const float Lx = 400.0f; // simulation box width
  const float u = 0.15f; // impact velocity
  const float rho = 1.0f; // density
  units.set_m_kg_s(Lx, u, rho, si_Lx, si_u, si_rho); // calculate 3 independent conversion factors (m, kg, s)

  // calculate values for remaining parameters in simulation units
  const float nu = units.nu(si_nu); // kinematic shear viscosity in simulation units
  const float sigma = units.sigma(si_sigma); // surface tension coefficient in simulation units
  const float f = units.f(si_rho, si_g); // force per volume
  const float Lz = units.x(si_Lz); // simulation box height
  const float H = units.x(si_H); // liquid pool height
  const float R = 0.5f*units.x(si_D); // drop radius

  // ############################################ define simulation box size, viscosity and volume force ##############################################
  initialize(to_uint(Lx), to_uint(Lx), to_uint(Lz), nu, 0.0f, 0.0f, -f, sigma, 0.05f); // largest box size on Titan Xp with FP32: 384^2, FP16: 464^3
  // ##################################################################################################################################################
  for(uint n=0, x=0, y=0, z=0, sx=lattice.total_x(), sy=lattice.total_y(), sz=lattice.total_z(), s=sx*sy*sz; n<s&&mpi_is_master; n++, x=n%(s/sz)%sx, y=
        ↪ n%(s/sz)/sx, z=n/sx/sy) {
    // ############################################################### define geometry ################################################################
    lattice.rho[n] = rho; // set density everywhere
    float b = sphere_plic(x, y, z, 0.5f*sx, 0.5f*sy-R*tanf(alpha*pi/180.0f), H+R+2, R-0.5f);
    if(b!=-1.0f) {
      lattice.u[  s+n] =  sinf(alpha*pi/180.0f)*u; // initial x-velocity of drop
      lattice.u[2*s+n] = -cosf(alpha*pi/180.0f)*u; // initial z-velocity of drop
      if(b==1.0f) {
        lattice.flags[n] = TYPE_F;
        lattice.phi[n] = 1.0f;
      } else {
        lattice.flags[n] = TYPE_I;
        lattice.phi[n] = b;
      }
    }
    if(z==0) lattice.flags[n] = TYPE_W; // bottom of simulation box is wall
    else if(z<H) lattice.flags[n] = TYPE_F; // define liquid pool
  } // ################################################################################################################################################
  run(0);

  float times[6] = {0.0f, 0.3f, 1.0f, 3.0f, 7.5f, 10.0f}; // define timestamps for pictures
  uint t_last = 0;
  float si_t = 0.0f;
  for(uint n=1; n<6; n++) {
    const uint dt = units.t(0.001f*(times[n]-0.13f))-t_last; // convert time intervals from SI-units to simulation units
    t_last += dt;
    run(dt); // run LBM for dt time steps
    write_image(); // write a picture of the current simulation status to the hard drive
  }
} /**/
```

Listing 2: The setup for the simulation in section 10.3 as an example of how simulation setups are built with C++ code in `setup.cpp`.

5.6 Roofline Model and Implementation Efficiency

The roofline model [41, 64] is a way to find out how efficiently an algorithm runs on hardware. The hardware dictates two limits: memory bandwidth $[b_\text{HW}] = GB/s$ (how fast data can be written to / loaded from memory) and compute performance $[p_\text{HW}] = FLOPs/s$ (how many floating-point operations per second the processor can handle). Then there is the so-called arithmetic intensity $[a_\text{HW}] = FLOPs/B$ which is defined as the quotient of compute performance and memory bandwidth. The hardware limits can be either found in the data sheet or calculated as follows:

$$b_\text{HW} = \text{number of memory channels} \cdot \text{memory frequency} \cdot \text{bus width in Byte} \tag{78}$$

$$p_\text{HW} = \text{number of cores} \cdot \text{frequency} \cdot \text{instructions per cycle} \tag{79}$$

$$a_\text{HW} = \frac{p_\text{HW}}{b_\text{HW}} \tag{80}$$

Which of the two limits – bandwidth or compute – applies to an algorithm is decided by the arithmetic intensity $[a_\text{A}] = FLOPs/B$ of the algorithm, namely how many floating point operations are performed for every Byte loaded from / written to memory.

$$a_\text{A} \begin{cases} < a_\text{HW} & \text{means that memory bandwith is the bottleneck} \\ > a_\text{HW} & \text{means that compute performance is the bottleneck} \end{cases} \tag{81}$$

High arithmetic intensity means that the algorithm does a lot of floating-point arithmetic, but little to no memory transfers; an example is prime number generation. Low arithmetic intensity on the other hand means that there are lots of memory transfers, but little floating-point arithmetic; here the prominent example is LBM. For LBM, the algorithm properties are calculated as

$$p_\text{A} = \frac{\text{lattice point updates}}{s} \cdot \frac{FLOPs}{\text{lattice point update}} \tag{82}$$

$$a_\text{A} = \frac{FLOPs\,/\,\text{lattice point update}}{\text{memory transfers}\,/\,\text{lattice point update}} \tag{83}$$

but this can also be generalized to other algorithms. For the LBM implementation, the number of floating-point operations per lattice point (figure 6) and the number of bytes transferred from and to memory per lattice point are counted by a self-written C++ program, which automatically sifts through the PTX assembly code generated for Nvidia GPUs using regular expressions.

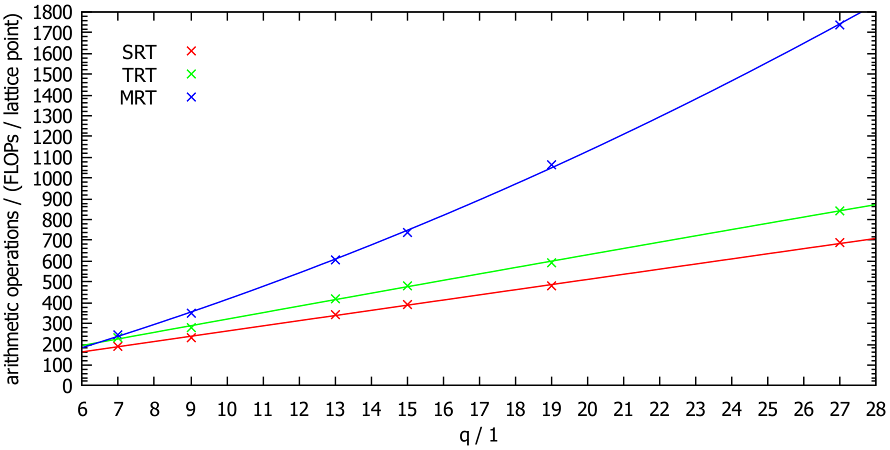


Figure 6: The number of arithmetic operations per lattice point of the LBM implementation *FluidX3D* plotted for all velocity sets and collision operators. For SRT and TRT, the number of arithmetic operations is linear in q while for MRT it is quadratic in q due to the single matrix multiplication involved.

Technically, floating-point operations are defined only as arithmetic operations performed on floating-point numbers, but here all arithmetic operations, including floating-point, integer and bit operations, are counted as *FLOPs*. On GPUs, integer and bit operations occupy the same execution units as floating-point operations, so

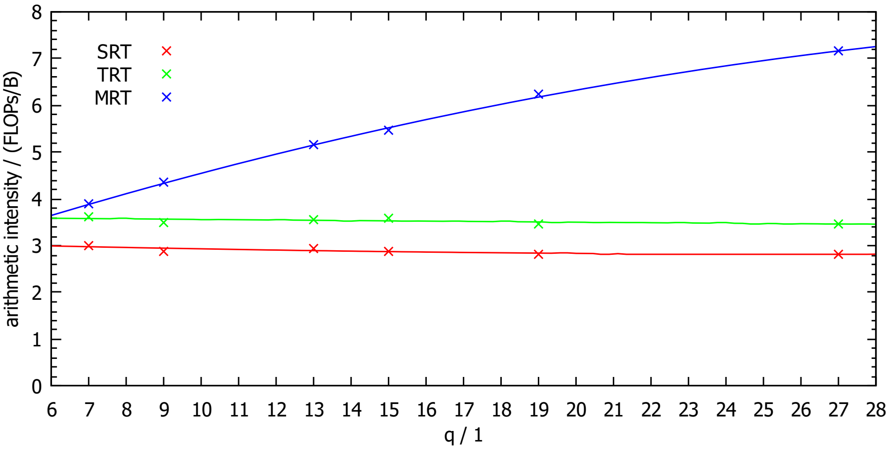


Figure 7: Arithmetic intensity a_A of the LBM algorithm plotted for all velocity sets and collision operators. Here the amount of micro-optimization put into the code becomes evident. For every DDF, there are only $9\,a_A$ floating-point operations in the code for calculating equations (36) and (2) to (4).

summarizing all of these operations as one type gives a more complete picture. For the performance measurements, the box sizes 8192^2 for 2D and 256^3 for 3D are chosen. By combining the measurements for *FLOPs* and memory transfers, the arithmetic intensity of the implementation is determined (figure 7).

With the algorithm properties defined, finally the roofline plot (figure 8) is done. The proximity of the algorithm data points to the hardware limit 'roofline' shows the implementation efficiency for the specified hardware, in the case of the Nvidia Titan Xp this is 84.1 %. The Titan Xp clocks down its memory when under heavy load in order to avoid overheating of the memory modules, so here the data sheet value is not really true. On other hardware such as the Nvidia Tesla V100, an efficiency of 99.7 % is achieved (figure 9).

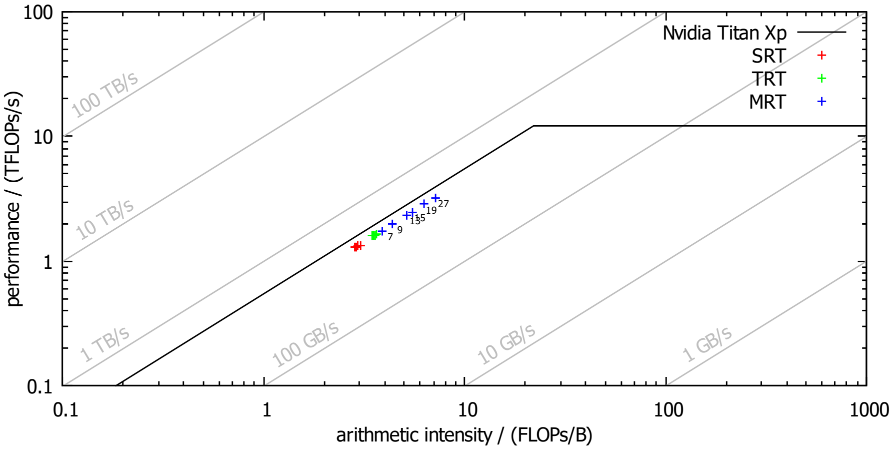


Figure 8: Roofline plot for the Nvidia Titan Xp with the LBM performance plotted for all velocity sets and collision operators. The part left from the kink in the black line is in the memory bandwidth limit while the part right from the kink is in the floating-point limit. LBM performance does not scale with the floating-point performance (y-axis), but with the memory bandwidth illustrated here as gray diagonals. The performance of all collision operators is almost identical. For SRT and TRT, the arithmetic intensity for all velocity sets is nearly the same, resulting in all of the points being located at roughly the same spots. For MRT, arithmetic intensity increases with q, shifting the points up and to the right along the constant-memory-bandwidth-diagonal for larger values of q. As long as the LBM data points are left from the kink in the black line, performance remains unchanged. However this kink is not in unreachable distance when LBM extensions are in use; typically it is located somewhere around 20 *FLOPs/B* depending on the device.

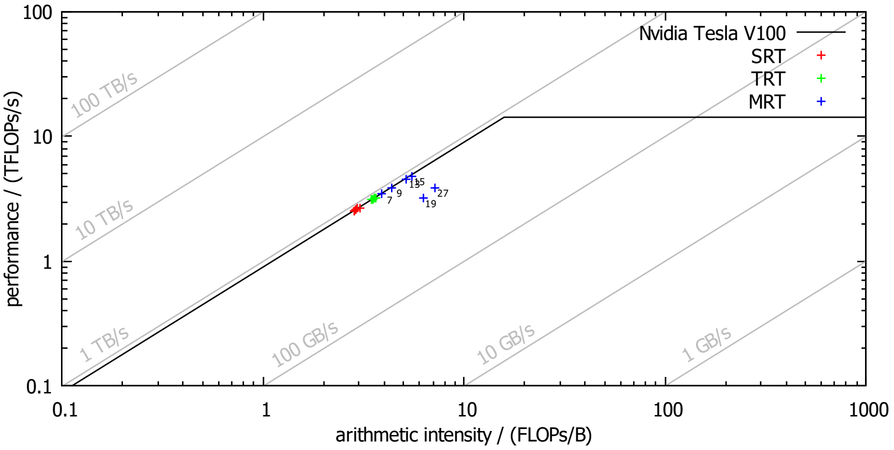


Figure 9: Roofline plot for the Nvidia Tesla V100 with the LBM performance plotted for all velocity sets and collision operators. Efficiency is excellent at 99.7 %, except for D3Q19 and D3Q27 with the MRT operator. Somehow the compiler here does not calculate the matrix multiplication in registers but stores the matrix Q in constant memory as becomes evident in the generated PTX assembly, despite it being explicitly defined to be in private memory in the code. Register file size per streaming multiprocessor for the Pascal and Volta microarchitectures is the same [65, 66], so the matrix Q not fitting in register space is not the explanation.

6 Volume-of-Fluid on the GPU

6.1 Overview

Volume-of-Fluid (VoF) is a model to simulate a sharp, freely moving interface between a fluid and gas phase in a Cartesian lattice [67–70]. The interface is ensured to be exactly one lattice point thick at any time (illustrated in figure 10). As an indicator for each lattice point type, the fill level φ is introduced, whereby for *fluid* lattice points $\varphi = 1$, for *interface* $1 > \varphi > 0$ and for *gas* $\varphi = 0$:

$$\varphi(\vec{x},t) := \frac{m(\vec{x},t)}{\rho(\vec{x},t)} \begin{cases} = 1 & \text{if } \vec{x} \text{ is } \textit{fluid} \\ \in]0,\,1[& \text{if } \vec{x} \text{ is } \textit{interface} \\ = 0 & \text{if } \vec{x} \text{ is } \textit{gas} \end{cases} \tag{84}$$

Here ρ is the density provided by LBM and m is the also newly introduced fluid mass. m is a conserved quantity and cannot be gained or lost, only moved within the simulation box. Although storing φ in memory for each lattice point would be sufficient, due to parallelization on the GPU also m needs to be stored in memory and moreover m_ex, the later introduced excess mass. m and φ are initialized either by initial fill levels or, if they are not explicitly defined, by the flags:

$$m(\vec{x},\, t=0) = \rho(\vec{x},\, t=0) \cdot \varphi(\vec{x},\, t=0) = \begin{cases} 1 & \text{if } \vec{x} \text{ is } \textit{fluid} \\ \frac{1}{2} & \text{if } \vec{x} \text{ is } \textit{interface} \\ 0 & \text{if } \vec{x} \text{ is } \textit{gas} \end{cases} \tag{85}$$

Additionally to the fill level φ, flag bits are introduced in order to perform type conversions between *fluid*↔*interface*↔*gas* and to be able to check the state of lattice points by loading only the flags (1 byte) instead of φ (4 bytes) from memory.

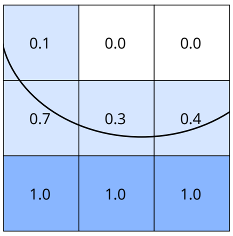


Figure 10: The idea of the Volume-of-Fluid model illustrated in 2D: A sharp interface (black curved line) divides the *gas* phase (white cells) from the *fluid* phase (dark blue cells). All cells through which the interface extends are called *interphase* cells (light blue). Every lattice cell gets a fill level $\varphi \in [0,\,1]$ assigned, which is $\varphi = 0$ for *gas*, $\varphi = 1$ for *fluid* and $\varphi \in]0,\,1[$ for *interphase* – based on where exactly the sharp interface cuts through.

6.2 Interface Advection

The interface advection is integrated into the `stream_collide()` kernel and calculated via the mass flux between neighboring lattice points.

$$m(\vec{x},\, t{+}\Delta t) = m(\vec{x},\, t) + \sum_i \begin{cases} f_{\bar{i}}^\text{temp}(\vec{x},\, t) - f_i^\text{A}(\vec{x},\, t) & \text{if } \vec{x} \text{ is } \textit{fluid} \\ \left(f_{\bar{i}}^\text{temp}(\vec{x},\, t) - f_i^\text{A}(\vec{x},\, t)\right) \cdot \begin{cases} 1 & \text{if } (\vec{x}+\vec{e}_i) \text{ is } \textit{fluid} \\ \frac{\varphi(\vec{x}+\vec{e}_i,\, t)+\varphi(\vec{x},\, t)}{2} & \text{if } (\vec{x}+\vec{e}_i) \text{ is } \textit{interface} \\ 0 & \text{if } (\vec{x}+\vec{e}_i) \text{ is } \textit{gas} \end{cases} & \text{if } \vec{x} \text{ is } \textit{interface} \end{cases} \tag{86}$$

For *interface* lattice points, DDFs $f_{\bar{i}}^\text{temp}(\vec{x},\, t)$ which have been streamed in from a *gas* lattice point at $(\vec{x}+\vec{e}_i)$ are not valid and have to be reconstructed by equilibrium DDFs $f_i^\text{eq}(\rho_\text{gas},\, \vec{u})$ which depend on the local fluid velocity $\vec{u}(\vec{x},\, t)$ and gas density $\rho_\text{gas}(\vec{x},\, t)$:

$$f_{\bar{i}}^\text{temp}(\vec{x},\, t) := f_{\bar{i}}^\text{eq}(\rho_\text{gas},\, \vec{u}) + f_i^\text{eq}(\rho_\text{gas},\, \vec{u}) - f_i^\text{A}(\vec{x},\, t) \qquad \text{if } (\vec{x}+\vec{e}_i) \text{ is } \textit{gas} \tag{87}$$

Others [68, 69] postulate to also reconstruct DDFs which have been streamed in from other interface neighbors (and therefore are well defined values) which form an angle with the local normal vector of less than 90° using equation (87). However this produces strong anisotropic artifacts – also observed by [67] – and moreover breaks mass conservation.
ρ_{gas} denotes the *gas* density, which is

$$\rho_{\text{gas}}(\vec{x},\, t) = \frac{1}{c^2}\, p_{\text{gas}}(\vec{x},\, t) = \frac{1}{c^2}\, p_0 - \frac{1}{c^2}\, \Delta p(\vec{x},\, t) = 1 - 6\,\sigma\,\kappa(\vec{x},\, t) \tag{88}$$

whereby $c := \frac{1}{\sqrt{3}}$ is the lattice speed of sound, $p_0 := \frac{1}{3}\rho_0 := \frac{1}{3}$ is the ambient pressure and $\Delta p = 2\,\sigma\,\kappa$ is the Young-Laplace pressure containing the surface tension constant σ and local mean curvature $\kappa = \frac{1}{R}$, the latter of which is studied in great detail in chapter 7.

6.3 Flag Handling

All possible flags for every lattice point are contained within a single `uchar`, the smallest OpenCL data type holding 8 bits. VoF requires 5 of these, although in theory 4 would also be sufficient. These are:

{*fluid*, *interface*, *gas*, *interface* → *fluid*, *interface* → *gas*}

On the GPU all lattice points are calculated at roughly the same time and the exact order of execution of neighboring lattice points may vary randomly, so any data dependencies between neighbors lead to difficulties. Unfortunately, keeping the interface sharp at all time requires lots of data dependencies between neighbors. If these are not handled correctly, race conditions will occur, making the simulation non-deterministic, meaning that any two identical simulations will lead to different results. GPU parallelization makes deterministic and error-free flag handling much more difficult than in non- or semi-parallel code for CPUs.
For correct flag handling on the GPU, in each time step four global synchronization points are mandatory, in other words there need to be three more kernel in addition to modifications in `stream_collide()`. Each synchronization point will decrease performance. Previous GPU implementations did require a total of seven kernels [71].

6.3.1 Kernel 1: Modified stream_collide()

Based on the interface advection, at the end of the `stream_collide()` kernel, exclusively for *interface* points, two interface change flags may be set, either *interface*→*fluid* or *interface*→*gas*:

$$m(\vec{x},\, t + \Delta t) \begin{cases} > (1+\epsilon)\, \rho(\vec{x},\, t+\Delta t) & \implies \text{set } \textit{interface}{\to}\textit{fluid} \text{ flag} \\ < (0-\epsilon)\, \rho(\vec{x},\, t+\Delta t) & \implies \text{set } \textit{interface}{\to}\textit{gas} \text{ flag} \end{cases} \tag{89}$$

Here $\epsilon \approx 0.01 \ll 1$ is a small number with the purpose to avoid flickering back and forth between flags, i.e. to stabilize the interface. Directly changing the state from *interface* to *fluid* or *gas* is not possible, because then afterwards a *fluid* point could be located right next to a *gas* point. This is circumvented by only flagging which changes should be made and resolving conflicts later, so now only a cell flagged with *interface* and *interface*→*fluid* can be located next to a cell flagged with *interface*→*gas*. This conflict has to be resolved and is the reason why three more kernel are mandatory.

6.3.2 Kernel 2: surface_1()

The `surface_1()` kernel only works on points with the *interface*→*fluid* flag set. For these points, it checks the flags of all streaming neighbors. Neighbors with the flags *interface*→*gas* AND NOT *gas* represent a conflict, so for these neighbors the *interface*→*gas* flag is cleared.
Moreover, neighbors with the *gas* flag set need to be turned into *interface*. However, due to otherwise occurring race conditions the conversion cannot happen within this kernel and is instead flagged to be carried out later by enabling the *interface*→*gas* flag additionally to the already set *gas* flag, a combination otherwise never used. This combination marks the lattice point to be converted to *interface* later on.

6.3.3 Kernel 3: surface_2()

The `surface_2()` kernel at first only works on lattice points flagged with *interface→gas* AND *gas*, the former *gas* lattice points which need to be turned into *interface*. Their previously undefined DDFs f_i need to be initialized by calculating the equilibrium DDFs

$$f_i := f_i^{\text{eq}}(\rho_{\text{avg}}, \vec{u}_{\text{avg}}) \tag{90}$$

with ρ_{avg} and $\vec{u}_{\text{avg}}$ being the average density and velocity of all *fluid* OR *interface* neighbors[14] (which have these properties well defined). Because for the averaging also *interface* neighbors are considered, within the `surface_2()` kernel the change from *interface→gas* AND *gas* to *interface* still is not allowed because it would lead to a race condition. The lattice point instead remains with the flags *interface→gas* AND *gas* set and its conversion is done later.

The second type of lattice point which `surface_2()` checks for is points ONLY flagged with *interface→gas*. If any neighbor of such a point is flagged with either *fluid* or *interface→fluid*, both *fluid* and *interface→fluid* flags of that neighbor are cleared and the *interface* flag is set, turning these neighbors into *interface* points.

6.3.4 Kernel 4: surface_3()

The first task of the `surface_3()` kernel is to finally change lattice points with *interface→gas* AND *gas* flags to *interface* by clearing the *interface→gas* AND *gas* flags and setting the *interface* flag. The flag is then immediately written to memory.

The remaining tasks of `surface_3()` are to perform the remaining indicated flag changes and to ensure strict mass conservation, working with *interface→fluid*, *interface→gas*, *fluid* and *interface* lattice points.

First, for points flagged with *interface→fluid* both the *interface→fluid* and *interface* flags are cleared and the *fluid* flag is set, turning them into *fluid* points. The excess mass is calculated as

$$m_{\text{ex}} := m - \rho \tag{91}$$

and then the mass is constrained to equal density $m := \rho$ and the fill level is set to one $\varphi := 1$.

For points flagged with *interface→gas* both the *interface→gas* and *interface* flags are cleared and the *gas* flag is set, turning them into *gas* points. The excess mass is set to the remaining mass

$$m_{\text{ex}} := m \tag{92}$$

and then the mass, velocity and fill level are zeroed: $m := 0$, $\vec{u} := 0$, $\varphi = 0$.

For points flagged with *interface*, no flag change is necessary, but still any eventual excess mass

$$m_{\text{ex}} := \begin{cases} m - \rho - \epsilon & \text{if } m > \rho + \epsilon \\ m + \epsilon & \text{if } m < -\epsilon \\ 0 & \text{otherwise} \end{cases} \tag{93}$$

is calculated, the mass is clamped to its approved range $m \in [-\epsilon,\, \rho+\epsilon]$ and the fill level is calculated from mass and density:

$$\varphi := \begin{cases} 1 & \text{for } \textit{fluid} \\ min(max(0,\, \frac{m}{\rho}),\, 1) & \text{for } \textit{interface} \\ 0 & \text{for } \textit{gas} \end{cases} \tag{94}$$

Finally, the excess mass m_{ex} needs to be distributed to all *fluid* or *interface* neighbors or neighbors which, in parallel, within the same step have not yet been turned into *fluid* or *interface*, each flagged by a unique flag or flag combination. The latter part is very tricky, but with these checks

$$(\text{flags}\,\&(I|F|I{\to}F)||(\text{flags}\,\&I{\to}G\&\&\,\text{flags}\,\&G))\,\&\&\,!(\text{flags}\,\&I{\to}G\,\&\&\,!(\text{flags}\,\&G))$$

with F denoting *fluid*, I denoting *interface* and G denoting *gas*, no race conditions occur, keeping the entire algorithm deterministic. The number of such neighbors is counted and the excess mass on the local lattice point is divided by this number.

In the very beginning of the following `stream_collide()` kernel, this reduced excess mass from all neighbors is added to the mass of all *fluid* or *interface* points locally, spreading it through the lattice until it reaches an *interface* point, where it finally gets absorbed. This way, the total mass in the simulation box is conserved analytically, verified to no changes but floating-point errors after several million time steps.

---

[14] For simplicity, all *fluid* and *interface* neighbors are weighted equally in the average calculation. Weighting by φ would also be possible, but since the velocity gradient across three lattice points usually is not that large anyways, the additional complexity and memory transfers are not justified.

7 Curvature Calculation for modeling Surface Tension

The key difficulty of modeling a free surface on a discretized lattice is to obtain the surface curvature, which is a necessary ingredient for calculating the surface tension via the Young-Laplace pressure

$$\Delta p = 2\,\sigma\,\kappa \tag{95}$$

with $\kappa = \frac{1}{R}$ denoting the local mean curvature and σ denoting the surface tension parameter of the simulated fluid. The equation is easy, but figuring out κ from the discretized interface geometry is not. Specifically, discretized interface here means that only a local 3^3 neighborhood of fill levels $\varphi \in [0,1]$ is known in addition to the point in the center of this neighborhood being an *interface* lattice point. So a single floating-point number needs to be extracted from the information contained in 27 floating-point numbers.

$$\varphi_0,\,...,\,\varphi_{26} \;\;\rightarrow\;\; \kappa \tag{96}$$

There many completely different approaches come to mind and since it is not yet clear which of them is suited best, all of them will be examined in the following sections in great detail. The different algorithms are later in section 9.4 compared quantitatively.

7.1 Analytic Curvature of a Paraboloid

A paraboloid curve is described by

$$z = f(x,y) = A\,x^2 + B\,y^2 + C\,x\,y + H\,x + I\,y + J \tag{97}$$

where A, B, C, H, I and J are fitting parameters. For such a 2D surface in 3D space in the form $(x,\,y,\,z = f(x,y))$ (so-called Monge patch), the mean curvature [72–76] is

$$\kappa := \frac{f_{xx}\left(f_y^2+1\right) + f_{yy}\left(f_x^2+1\right) - 2\,f_{xy}\,f_x\,f_y}{2\left(\sqrt{f_x^2+f_y^2+1}\right)^3} \tag{98}$$

The partial derivatives of eq. (97) evaluated at the point $(x=0,\,y=0)$ are

$$f_{xx}\,\big|_{x=y=0} = 2\,A \tag{99}$$

$$f_{yy}\,\big|_{x=y=0} = 2\,B \tag{100}$$

$$f_{xy}\,\big|_{x=y=0} = C \tag{101}$$

$$f_x\,\big|_{x=y=0} = \,2\,A\,x + C\,y + H\,\big|_{x=y=0} \,=\, H \tag{102}$$

$$f_y\,\big|_{x=y=0} = \,2\,B\,y + C\,x + \,I\,\big|_{x=y=0} \,=\, I \tag{103}$$

so that the mean curvature for the paraboloid at the origin is

$$\kappa := \frac{A\,(I^2+1) + B\,(H^2+1) - C\,H\,I}{\left(\sqrt{H^2+I^2+1}\right)^3} \tag{104}$$

which means that [67] are using a wrong equation and [77] does not use the common definition of the mean curvature.
There are now two strategies for finding the required fitting parameters: The first one is by calculating the fluid volume beneath the paraboloid in the 3^3 neighborhood and the second one is to apply a least-squares fit on a neighborhood of points on the interface.

7.2 Approximation over Volume beneath Paraboloid

7.2.1 Anisotropic Approximation over Volume beneath Paraboloid (approximation, very fast)

The easiest approach that may come to mind is to consider a 3^3 neighborhood of the lattice point in question and see this neighborhood as a 3^3 cubic box. When the center lattice point inside this box is an interface point and the surface within this box is assumed to be of parabolic shape (eq. (97)), this paraboloid curve is uniquely defined by the fluid volume in the 3^3 box beneath it, which is just the sum of all fill levels φ_i.

$$V_{\text{cube}} = \sum_{i=0}^{26} \varphi_i \tag{105}$$

When volume of the 3^3 cube is truncated by the paraboloid in the vertical direction, the intersection volume as a function of the parameters which define the paraboloid is given by the following volume integral:

$$\begin{aligned}
V_{\text{cube}} &= \int_{-\frac{3}{2}}^{z(x,y)} \int_{-\frac{3}{2}}^{\frac{3}{2}} \int_{-\frac{3}{2}}^{\frac{3}{2}} dx\,dy\,dz = \\
&= \int_{-\frac{3}{2}}^{\frac{3}{2}} \int_{-\frac{3}{2}}^{\frac{3}{2}} \left(Ax^2 + By^2 + Cxy + Hx + Iy + J + \frac{3}{2} \right) dx\,dy = \\
&= \int_{-\frac{3}{2}}^{\frac{3}{2}} \left[Ax^2 y + By^3 + Cxy^2 + Hxy + \frac{1}{2} I y^2 + (J + \frac{3}{2}) y \right]_{-\frac{3}{2}}^{\frac{3}{2}} dx = \\
&= \int_{-\frac{3}{2}}^{\frac{3}{2}} \left(3Ax^2 + \frac{9}{4} B + 3Hx + 3(J + \frac{3}{2}) \right) dx = \\
&= \left[Ax^3 + \frac{9}{4} Bx + \frac{3}{2} Hx^2 + 3(J + \frac{3}{2}) x \right]_{-\frac{3}{2}}^{\frac{3}{2}} = \\
&= \frac{27}{4}(A + B) + 9\,J + \frac{27}{2}
\end{aligned} \tag{106}$$

Interestingly, the parameters C, H and I cancel out and are thus without loss of generality set to zero, substantially simplifying equation (104):

$$\kappa := \frac{A\,(I^2+1) + B\,(H^2+1) - C\,H\,I}{\left(\sqrt{H^2+I^2+1}\right)^3} \overset{\text{C,H,I=0}}{=} A + B \tag{107}$$

Finally, when all above equations are combined, a direct and simple expression for the curvature is obtained:

$$\kappa = \frac{4}{27} \left(\sum_{i=0}^{26} \varphi_i - 9\,J - \frac{27}{2} \right) \tag{108}$$

However this expression is not isotropic, meaning that errors are made if the surface normal is not parallel to either one of the three coordinate axis directions, which is almost always the case. To mostly cancel out this error, the neighborhood of the velocity set $D3Q19$ works best by accounting for the four missing unit cubes in the bottom corners of the 3^3 neighborhood. J is the vertical offset of the paraboloid, which ideally would be the output of piecewise linear interface construction (PLIC), where it is calculated by the intersection of a plane and a cube, but the algorithm turned out to provide best results for the simple (anisotropic) linear approximation

$$J = \varphi_0 - \frac{1}{2} \tag{109}$$

resulting in the following algorithm:

$$\kappa = \frac{4}{27} \left(\sum_{i=0}^{18} \varphi_i - 9\,\varphi_0 - 5 \right) \tag{110}$$

The great advantage of this algorithm besides its speed and simplicity is that it does handle surface wetting extremely well by assigning a fill level $\varphi_{wall} \in [0, 1]$ for *wall* lattice points, very similar to how wetting is handled in Shan-Chen (section 3.9.1). $\varphi_{wall} = 0$ corresponds to non-wetting surfaces and $\varphi_{wall} = 1$ to wetting surfaces.

7.2.2 Isotropic Approximation over Volume beneath Paraboloid (Failure)

The next idea is to try to make the algorithm from chapter 7.2.1 isotropic by considering not a cubic 3^3 neighborhood, but a spherical neighborhood with radius $R = \frac{3}{2}$ instead. Therefore, the cubic cells get different weights assigned, corresponding to their overlap volume with the spherical neighborhood. The cube-sphere intersection problem is very difficult and therefore it is not solved analytically to obtain the three different weights for lattice points in the corners, on the edges or on the sides of the 3^3 neighborhood. Instead, the coefficients are calculated using Monte-Carlo sampling of 10^{11} random points, yielding approximately 7 digits of accuracy:

$$w_{\text{center}} = 1 \tag{111}$$

$$w_{\text{side}} \approx 0.9428903 \tag{112}$$

$$w_{\text{edge}} \approx 0.5087824 \tag{113}$$

$$w_{\text{corner}} \approx 0.1717880 \tag{114}$$

The parabolic equation is the same as in chapter 7.2.1, eq. (97), but the volume summation gets additional weights

$$V_{\text{sphere}} = \sum_{i=0}^{26} w_i\, \varphi_i \tag{115}$$

and the volume integral is now limited by a sphere on the lower side and by the paraboloid on the upper side.

$$x = r\cos(\theta) \qquad y = r\sin(\theta) \qquad z = z \tag{116}$$

$$\begin{aligned}
V_{\text{sphere}} &= \int_{-\sqrt{\frac{9}{4}-x^2-y^2}}^{f(x,y)} \int_0^{2\pi} \int_0^{\frac{3}{2}} r\,dr\,d\theta\,dz = \\
&= \int_0^{2\pi} \int_0^{\frac{3}{2}} \left(Ax^2 + By^2 + Cxy + Hx + Iy + J + \sqrt{\frac{9}{4} - x^2 - y^2} \right) r\,dr\,d\theta = \\
&= \int_0^{2\pi} \int_0^{\frac{3}{2}} \left(\left(A\cos^2(\theta) + B\sin^2(\theta) + C\cos(\theta)\sin(\theta)\right) r^2 + \left(H\cos(\theta) + I\sin(\theta)\right) r + J + \sqrt{\frac{9}{4} - r^2} \right) r\,dr\,d\theta = \\
&= \int_0^{\frac{3}{2}} \left(\pi r^3 (A+B) + 2\pi r \left(J + \sqrt{\frac{9}{4} - r^2} \right) \right) dr = \\
&= \left[\frac{\pi}{4} r^4 (A+B) + \pi r^2 J - \frac{2\pi}{3} \left(\frac{9}{4} - r^2 \right)^{\frac{3}{2}} \right]_0^{\frac{3}{2}} = \\
&= \frac{81\pi}{64}(A+B) + \frac{9\pi}{4} J + \frac{9\pi}{4} = \\
&= \frac{9\pi}{4} \left(\frac{9}{16}(A+B) + J + 1 \right)
\end{aligned} \tag{117}$$

Again, the parameters C, H and I cancel out and are set to zero. This leads again to eq. (107), yielding a direct expression for the curvature:

$$\kappa = \frac{16}{9} \left(\frac{4}{9\pi} \sum_{i=0}^{26} w_i\, \varphi_i - J - 1 \right) \tag{118}$$

Note that above equation makes an error for large curvatures, as the volume on the edge is limited to the bottom by a sphere and to the top by a cylinder limited to the height of the paraboloid and not to the sphere. J again is the vertical offset of the paraboloid and is defined by the fill level of the center cell. The exact solution to get the offset parameter from the fill level would again be PLIC, so $J = d_0(V_0 = \varphi_0)$. However it turned out to have the least visual artifacts, when PLIC is done with a spherical lattice cell with unit volume instead of a cubic one. The analytic solution for this is given in section 7.4.2. Overall the visual artifacts from this approach are more pronounced than in the algorithm from section 7.2.1, so this algorithm is considered a failure.

7.3 Curvature Calculation via Paraboloid Fit

The most common algorithm in literature [67, 70] is the curvature calculation via least-squares paraboloid fit from a neighborhood of points located on the interface. It starts by assuming the local interface to be a paraboloid, the specifics of which will be given in the following subsections.
Finding an appropriate set of neighboring points on the interface is especially challenging. There are two ways of obtaining the relative locations of neighboring interface points: PLIC and marching-cubes. Either one of these will be discussed in detail in the following sections.

7.3.1 Calculating the Interface Normal Vector from a 3^3 Neighborhood

Calculating the normal vector on an interface lattice point in a 3^3 neighborhood in which all fill levels φ_i are known basically works by applying the gradient to the fill levels. This is called the center of mass (CM) method:

$$\vec{n}_{\text{CM}} := -\frac{\sum_{i=1}^{26} \vec{c}_i \, \varphi_i}{|\sum_{i=1}^{26} \vec{c}_i \, \varphi_i|} \tag{119}$$

Another more accurate approach is the Parker-Youngs (PY) approximation [70, 78] which assigns different weights to the gradient components:

$$\vec{n}_{\text{PY}} := -\frac{\sum_{i=1}^{26} w_i \, \vec{c}_i \, \varphi_i}{|\sum_{i=1}^{26} w_i \, \vec{c}_i \, \varphi_i|} \tag{120}$$

with

$$w_i := \begin{cases} 4 & \text{for } |\vec{c}_i| = 1 \qquad \text{The order of weights 4,2,1} \\ 2 & \text{for } |\vec{c}_i| = \sqrt{2} \qquad \text{has been corrected;} \\ 1 & \text{for } |\vec{c}_i| = \sqrt{3} \qquad \text{order is wrong in [70].} \end{cases} \tag{121}$$

According to [70], the average error for CM is approximately 4° while for PY it is approximately 1°. For the surface curvature algorithms below, the more accurate and equally fast PY method is used.

7.3.2 Curvature from Least-Squares Paraboloid Fit

The least-squares method [79] is a procedure of fitting an analytic curve – here a 2D surface on 3D space – to a set of discretized points located somewhat nearby the analytic curve. The general idea is to define the total error as a general expression of all fitting parameters and the entire set of discretized points and then find its global minimum by zeroing its gradient.
The analytic curve first needs to be written in a dot product form

$$z(x, y) = \vec{x} \circ \vec{Q} \tag{122}$$

with $\vec{x}$ being defined as the vector of parameters that define the curve and $\vec{Q}_i$ being a vector containing expressions only dependent on a discretized set of points (x_i, y_i) located close the curve. In this notation, the error E between the z-positions of the analytic curve $\vec{x} \circ \vec{Q}$ and a set of z-positions of at least N neighboring interface points z_i is defined by summing up the squared differences

$$E(\vec{x}) = \sum_{i=0}^{N} (\vec{x} \circ \vec{Q}_i - z_i)^2 \tag{123}$$

whereby N denotes the dimensionality which is equal to the number of desired fitting parameters. The gradient of the error E is calculated and set to zero, where the error must have a global minimum:

$$\nabla E(\vec{x}) = 2 \sum_{i=0}^{N} (\vec{x} \circ \vec{Q}_i - z_i) \, \vec{Q}_i = 0 \tag{124}$$

With some algebra, this equation is then transformed into a linear equation

$$\left(\sum_{i=0}^{N}\vec{Q}_i\vec{Q}_i^{\,T}\right)\vec{x}=\sum_{i=0}^{N}z_i\,\vec{Q}_i \tag{125}$$

$$\mathbf{M}:=\sum_{i=0}^{N}\vec{Q}_i\vec{Q}_i^{\,T}\qquad\vec{b}:=\sum_{i=0}^{N}z_i\,\vec{Q}_i \tag{126}$$

$$\mathbf{M}\,\vec{x}=\vec{b} \tag{127}$$

which is solved by LU-decomposition and provides the desired solution $\vec{x}$ that uniquely defines the curve.
Note that the matrix $\mathbf{M}$ is always symmetrical, meaning that only the upper half and diagonal have to be calculated explicitly and the lower half is copied over. This reduces computational cost significantly due to every matrix element being a sum over an expression depending on all fitted points. In case there are less than N data points available, the regular fitting will not work. Instead, then the amount of fitting parameters is decreased to match the number of available data points by reducing dimensionality in the LU-decomposition. The ignored fitting parameters will remain zero.
Finally, from the solution vector $\vec{x}$ the constants defining the fitted curve are extracted and the curvature is calculated from them using equation (104).

7.3.3 Obtaining neighboring Interface Points: PLIC Point Neighborhood

Piecewise linear interface construction (PLIC) works on a 3^3 neighborhood of an interface lattice point (illustrated in 2D in figure 11). Within this neighborhood, only other interface points are considered. The difficult part now is to accurately obtain the heights z_i of at least five neighboring points located on the true interface. First, the normal vector $\vec{n}$ of the center interface point is calculated via the Parker-Youngs approximation (eq. (120). The first vector of the new coordinate system $\vec{b}_z$ is defined as this normal vector. Then, the cross product with an arbitrary vector such as

$$\vec{r_n}:=(0.56270900,\;0.32704452,\;0.75921047)^T \tag{128}$$

which is always non-colinear with $\vec{b}_z$ just by random chance is calculated to provide second and third orthonormal vectors

$$\vec{b}_z:=\vec{n} \tag{129}$$

$$\vec{b}_y:=\frac{\vec{b}_z\times\vec{r}_n}{|\vec{b}_z\times\vec{r}_n|} \tag{130}$$

$$\vec{b}_x:=\vec{b}_y\times\vec{b}_z \tag{131}$$

forming the new coordinate system in which the z-axis is colinear with the surface normal and the center interface point is in the origin. Now the relative positions (equal to the D3Q27 streaming directions) of all neighboring interface lattice points are gathered and transformed into the rotated coordinate system. During this step, the approximate interface position of neighboring interface points (streaming directions) is also corrected to the much more accurate interface position via the PLIC plane-cube intersection solution (section 7.4.1):

$$\vec{p_i}=(x_i,\,y_i,\,z_i)^T:=\left(\vec{e_i}\circ\vec{b}_x,\;\vec{e_i}\circ\vec{b}_y,\;\vec{e_i}\circ\vec{b}_z+d_0(\varphi_i,\,\vec{n})-d_0(\varphi_0,\,\vec{n})\right)^T \tag{132}$$

Here i is only the subset of $\{0,..,26\}$ for which $0<\varphi_i<1$ is true (interface points). $\vec{e_i}$ denotes the D3Q27 streaming directions and $d_0(V_0,\,\vec{n})$ denotes the PLIC function (equation (158)). Note that $d_0(\varphi_0,\,\vec{n})$ only needs to be calculated once while $d_0(\varphi_i,\,\vec{n})$ has to be calculated for each neighboring interface point and that the normal vectors of neighboring interface lattice points are approximated to be equal to the normal vector of the center lattice point. In theory, going with the separately calculated neighbor normal vectors – which would require either an additional data buffer for normal vectors in global memory or alternatively a 5^3 neighborhood which would break the multi-GPU capabilities of the code – should be more accurate, but in practice it makes no noticeable difference; surprisingly with the exact normal vectors a spherical drop in zero gravity will even start to wobble on its own. The set of points $\vec{p_i}$ is then inserted into the fitting procedure.

The fitted paraboloid here lacks a vertical offset parameter as that is handled already by the center point being defined as the origin, reducing computational cost to a LU-decomposition of dimensionality $N = 5$. The paraboloid has the form

$$z(x, y) = Ax^2 + By^2 + Cxy + Hx + Iy =: \vec{x} \circ \vec{Q} \tag{133}$$

with

$$\vec{x} := (A,\, B,\, C,\, H,\, I)^T \tag{134}$$

$$\vec{Q}_i := (x_i^2,\, y_i^2,\, x_i\, y_i,\, x_i,\, y_i)^T \tag{135}$$

The solution vector $\vec{x}$ and thus the fitting parameters are calculated following the procedure in section 7.3.2. Finally, the constants A, B, C, H and I are inserted into the analytic equation for the curvature (104), completing the algorithm.

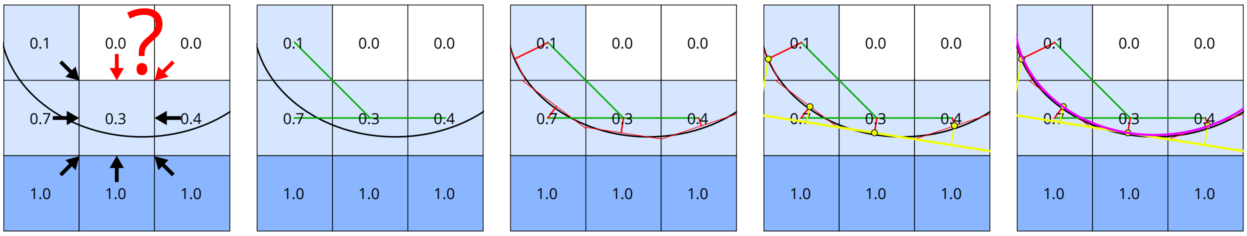


Figure 11: The curvature calculation procedure with PLIC illustrated in 2D. From left to right: 1) The DDFs to be streamed in from *gas* neighbors to the *interface* point in the center are undefined and need to be reconstructed with equation (87), for which the local gas density and thus the local curvature is required (eq. (88)). For obtaining the local curvature, the steps are to 2) identify all *interface* neighbors (eq. (84)), 3) correct the relative interface neighbor positions with the PLIC offset (section 7.4.1), 4) rotate/translate these now PLIC-corrected points into a coordinate system (eq. (132)) with the PLIC-corrected center point being the origin and the z-axis being colinear with the local surface normal (section 7.3.1) and finally 5) perform a paraboloid fit with these points (section 7.3.2).

7.3.4 Obtaining neighboring Interface Points: Marching-Cubes (Failure)

This is an alternative option to identify points on the interface in a universe where PLIC has not been solved yet. Local points on the interface here are calculated based on how the famous marching-cubes algorithm [80] calculates them: by linear interpolation. For two neighboring lattice points $\vec{x}_1$ and $\vec{x}_2$ in 3D space with fill levels $\varphi_1 \neq \varphi_2$, where one of them is smaller than $\frac{1}{2}$ and the other one is larger than $\frac{1}{2}$ (the interface is exactly at $\varphi = \frac{1}{2}$), by linear interpolation the point located exactly on the interface in between is calculated:

$$\vec{x} = \vec{x}_1 + \frac{\frac{1}{2} - \varphi_1}{\varphi_2 - \varphi_1} (\vec{x}_2 - \vec{x}_1) \tag{136}$$

In a 3^3 neighborhood, there are 54 straight, 72 edge diagonal and 32 space diagonal connection candidates. Best results however turned out when space diagonal connections were excluded as the distance between these points is already $\sqrt{3}$, so interpolation turns out poor.

The remaining coordinate system transformation and fitting procedure here are identical to chapter 7.3.3, except with an additional vertical offset parameter J in the fitting, meaning that now the dimensionality of the fitting matrix is increased to $N = 6$.

$$z(x, y) = Ax^2 + By^2 + Cxy + Hx + Iy + J =: \vec{x} \circ \vec{Q} \tag{137}$$

$$\vec{x} := (A,\, B,\, C,\, H,\, I,\, J)^T \tag{138}$$

$$\vec{Q}_i := (x_i^2,\, y_i^2,\, x_i\, y_i,\, x_i,\, y_i,\, 1)^T \tag{139}$$

The remaining part follows the procedure of chapter 7.3.2.

The results from this algorithm are quite bad. It is both way slower than the fitting with PLIC due to heavy use of branching and much less accurate; artifacts such as flat sides on a spherical droplet in zero gravity are clearly visible. Thus this approach is considered failure. Although the marching cubes algorithm works perfectly fine for visualization of the interface, the interpolation is not sufficiently accurate for curvature calculation.

7.4 Piecewise Linear Interface Construction (PLIC)

7.4.1 Plane-Cube Intersection

PLIC stands for piecewise linear interface construction – first occurring in literature for 2D in 1982 [81] and for 3D in 1984 [82] – and is the problem of calculating the offset alongside the given normal vector of a plane intersecting a unit cube for a given truncated intersection volume. There are five possible intersection cases, of which the numbers (1), (2) and (5) have been already solved in the original 1984 work by Youngs [82], but the cubic polynomial cases (3) and (4) – resigned as impossible to algebraically invert [83] – in the majority of literature are approximated by Newton-Raphson. This and other comparatively slow iterative approximations [84, 85] would severely bottleneck LBM on the GPU. Here the complete analytic solution with its full derivation is elaborated.

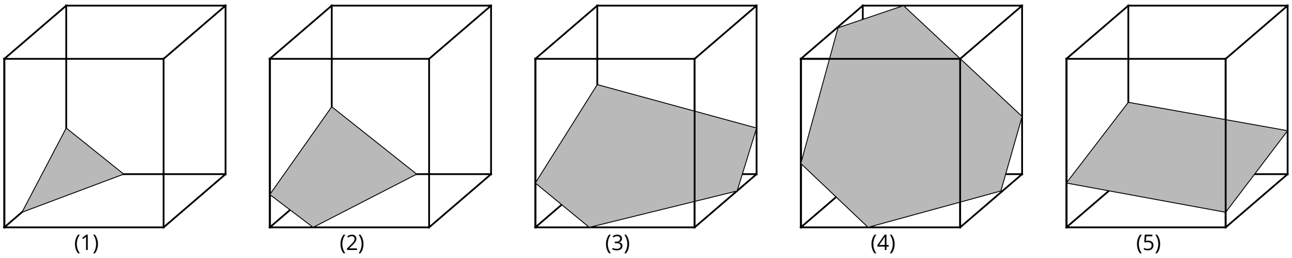


Figure 12: All possible intersection cases of a plane and a unit cube. The truncated volume of cases (1) to (4) is a tetrahedral pyramid with zero (1), one (2), two (3) or all three (4) corners extending outside of the unit cube being cut-off tetrahedral pyramids themselves.

Inputs to the PLIC algorithm are the volume of intersection V_0 and the (normalized) normal vector of the plane $\vec{n} = (n_x, n_y, n_z)^T$. The desired output is the plane offset from the origin alongside the normal vector d_0.

$$V_0,\ (n_x, n_y, n_z)^T \ \rightarrow\ d_0 \tag{140}$$

In order to derive the complete analytic solution, first the inverse of the problem is formulated in equations and then, to be able to validate the later presented solution, approximated iteratively via nested intervals. So at first, the intersection volume is calculated from the plane offset and normal vector as inputs. To reduce the amount of possible cases and to avoid having to consider all possible intersections of the plane and cube edges, the normal vector is component-wise mirrored into positive space and its components are sorted ascending for their magnitude such that $0 < n_1 \leq n_2 \leq n_3 \leq 1$. To later avoid diverging fractions, all components must be non-zero, which is enforced numerically by limiting the smaller two to a minimum value of 10^{-5}. For n_3 no check is required since $\vec{n}$ is normalized.

$$n_1 := \max(\min(|n_x|,\ |n_y|,\ |n_z|),\ 10^{-5}) \tag{141}$$

$$n_3 := \max(|n_x|,\ |n_y|,\ |n_z|) \tag{142}$$

$$n_2 := \max(|n_x| + |n_y| + |n_z| - n_1 - n_3,\ 10^{-5}) \tag{143}$$

Furthermore, since the function $V_0(d_0)$ is symmetric and increasing monotonically, the volume is limited to the lower half $V \in [0, \frac{1}{2}]$ and the upper half is reconstructed from symmetry.

$$V := \frac{1}{2} - \left| V_0 - \frac{1}{2} \right| \tag{144}$$

With this, the nested intervals can begin. Search is limited to the lower half $d_0 \in [-\frac{n_1+n_2+n_3}{2}, 0]$ and the number of iterations is fixed to 24 which is sufficient for convergence just below 32-bit floating point accuracy. At each iteration, only the intersection volume at midpoint is evaluated, using the information that the function rises monotonically. For volume evaluation, the coordinate origin is first shifted from (0, 0, 0) to $(\frac{1}{2}, \frac{1}{2}, \frac{1}{2})$:

$$d = d_0 + \frac{n_1 + n_2 + n_3}{2} \tag{145}$$

Next, the intersection points of the plane with the coordinate axes are determined:

$$s_1 := \frac{d}{n_1} \quad \geq \quad s_2 := \frac{d}{n_2} \quad \geq \quad s_3 := \frac{d}{n_3} \tag{146}$$

Now comes the actual volume calculation. The approach is to calculate the volume of the tetrahedral pyramid formed by the plane and the coordinate system axes and, if necessary, subtract the volumes of zero, one, two or all three corners that extend beyond 1. For the case of two corners, an additional check is required which also mutually excludes the last possible case of the bottom four corners of the cube being beneath the plane.

$$V = \begin{cases} \frac{1}{6}\, s_1\, s_2\, s_3 & \text{if } s_1 \leq 1 \\ \frac{1}{6}\, s_2\, s_3 \left(s_1 - (s_1 - 1) \left(1 - \frac{1}{s_1}\right)^2 \right) & \text{if } s_1 > 1 \text{ and } s_2 \leq 1 \\ \frac{1}{6}\, s_3 \left(s_1\, s_2 - (s_1 - 1)\, s_2 \left(1 - \frac{1}{s_1}\right)^2 - (s_2 - 1)\, s_1 \left(1 - \frac{1}{s_2}\right)^2 \right) & \text{if } s_2 > 1 \text{ and } s_3 \leq 1 \text{ and } s_1\,(s_2 - 1) \leq s_2 \\ \frac{1}{6} \left(s_1\, s_2\, s_3 - (s_1 - 1)\, s_2\, s_3 \left(1 - \frac{1}{s_1}\right)^2 - (s_2 - 1)\, s_1\, s_3 \left(1 - \frac{1}{s_2}\right)^2 - (s_3 - 1)\, s_1\, s_2 \left(1 - \frac{1}{s_3}\right)^2 \right) & \text{if } s_3 > 1 \\ \frac{1}{2}\, s_3 \left(2 - \frac{1}{s_1} - \frac{1}{s_2}\right) & \text{otherwise} \end{cases} \tag{147}$$

Finally, after the nested intervals the symmetry condition is applied to cover the case $V_0 > \frac{1}{2}$ which flips the sign of d. Here ? : denotes the ternary operator. This completes the (slow, but at least working) nested intervals algorithm.

$$d_0 := V_0 \leq \frac{1}{2}\ ?\ d : -d \tag{148}$$

To speed up the computation, the inversion of equation (147) is done. Therefore, s_1, s_2 and s_3 are first substituted and the expression is simplified, yielding

$$V = \frac{1}{6\, n_1\, n_2\, n_3} \cdot \begin{cases} d^3 & (1) \text{ if } \quad d \leq n_1 \\ (d^3 - (d - n_1)^3) & (2) \text{ if } n_1 < d \leq n_2 \\ (d^3 - (d - n_1)^3 - (d - n_2)^3) & (3) \text{ if } n_2 < d \leq \min(n_1 + n_2,\, n_3) \\ (d^3 - (d - n_1)^3 - (d - n_2)^3 - (d - n_3)^3) & (4) \text{ if } n_3 < d \\ 6\, n_1\, n_2\, (d - \frac{1}{2}\,(n_1 + n_2)) & (5) \text{ if } \min(n_1 + n_2, n_3) < d \leq n_3 \end{cases} \tag{149}$$

which is already quite a bit more friendly. The conditions for the five cases are mutually exclusive. This equation is now inverted for each case individually. Cases (1), (2) and (5) are easy, but cases (3) and (4) are non-trivial third order polynomials with three complex solutions each. Luckily, the general structure of their solutions is identical (equation (151)). However, a complex solution such as outputted by Mathematica (section 15) is nonsense here since the expected result is a real number – a problem known as the *casus irreducibilis* – and OpenCL cannot deal with complex numbers natively. But with the following equation, the third complex solutions from Mathematica of cases (3) and (4) respectively are converted into real expressions – the trigonometric solution:

$$\mathrm{f}\,(x, y, a, b, c) := c - 2\, \frac{a + b\, \sqrt[3]{x^2 + y^2}}{\sqrt[6]{x^2 + y^2}}\, \sin\left(\frac{\pi}{6} - \frac{1}{3}\, \mathrm{atan2}(y, x)\right) = \tag{150}$$

$$= c - a\, \frac{(1 - i\sqrt{3})}{\sqrt[3]{x + i\, y}} - b\,(1 + i\sqrt{3})\, \sqrt[3]{x + i\, y} \tag{151}$$

For better readability, a few expressions are pre-defined. Hereby the normalization condition $n_1^2 + n_2^2 + n_3^2 = 1$ is applied. Taking the absolute value under the square root in equations (153), (156) and (157) is not necessary in a mathematical sense, but in the actual code, floating-point exception handling is turned off for performance reasons and the resulting `NaN` of a square root of a negative number would not be captured in the case condition, leading to a false result.

$$x_3 := 81\, n_1\, n_2\, (n_1 + n_2 - 2\, V\, n_3) > 0 \tag{152}$$

$$y_3 := \sqrt{|23328\, (n_1\, n_2)^3 - x_3^2|} \geq 0 \tag{153}$$

$$t_4 := 9\, (n_1 + n_2 + n_3)^2 - 18 \tag{154}$$

$$x_4 := 324\, n_1\, n_2\, n_3\, (1 - 2\, V) \geq 0 \tag{155}$$

$$y_4 := \sqrt{|4\, t_4^3 - x_4^2|} \geq 0 \tag{156}$$

Finally then, the complete analytic solution to the 3D PLIC problem is given by

$$d = \begin{cases} \sqrt[3]{6\,V\,n_1\,n_2\,n_3} & (1)\text{ if } \qquad d \leq n_1 \\ \frac{n_1}{2} + \frac{1}{\sqrt{12}}\sqrt{|24\,V\,n_2\,n_3 - n_1^2|} & (2)\text{ if } n_1 < d \leq n_2 \\ \mathrm{f}\,(x_3, y_3, \sqrt[3]{54}\,n_1\,n_2, \frac{1}{\sqrt[3]{432}}, n_1 + n_2) & (3)\text{ if } n_2 < d \leq \min(n_1 + n_2,\, n_3) \\ \mathrm{f}\,(x_4, y_4, \frac{1}{\sqrt[3]{864}}\,t_4, \frac{1}{\sqrt[3]{3456}}, \frac{n_1+n_2+n_3}{2}) & (4)\text{ if } n_3 < d \\ V\,n_3 + \frac{n_1+n_2}{2} & (5)\text{ if } \min(n_1 + n_2, n_3) < d \leq n_3 \end{cases} \tag{157}$$

$$d_0 := \left(V_0 \leq \frac{1}{2}\ ?\ 1 : -1\right)\left(d - \frac{n_1 + n_2 + n_3}{2}\right) \tag{158}$$

in conjunction with equations (141) to (144), (150) and (152) to (156).

Now it looks a bit strange that the conditions for the five different cases are determined by the result itself. This just means that each case has to be evaluated successively and for the resulting value d the respective condition has to be tested. If the condition is true, calculation is stopped and d is returned. If the condition is false, the next case has to be evaluated and so on, until the last case is reached, which is always true. The order in which the cases are computed and checked does not matter and can be optimized to calculate the most difficult and infrequent cases last, when the probability is high that one of the easier and more frequent cases has already been chosen. The condition for the last case does not have to be checked since it is mutually excluded by the conditions of the four previous cases. The order (5)→(2)→(1)→(3)→(4) proved to be fastest on GPUs. For even more speedup, all redundant mathematical operations are reduced to a minimum by pre-calculating them to variables (micro-optimization). If $x_4 > 0$ is artificially ensured, then instead of $\mathrm{atan2}(y, x)$ the faster $\mathrm{atan}(y/x)$ can be called, giving the entire simulation a 15% speedup. In case branching would be undesirable, bit masking is also an option, but branching turned out to be faster even on GPUs. Performance-wise, the analytic solution leads to a 4 times speedup of the VoF-LBM simulation compared to when nested intervals are used. Listing 3 shows the fully optimized OpenCL C implementation of the analytic 3D PLIC solution.

```c
float __attribute__((always_inline)) offset_cube(const float V, const float n1, const float n2, const float n3) {
  const float n1pn2=n1+n2, n1xn2=n1*n2, n3xV=n3*V, minn1pn2n3=fmin(n1pn2, n3);
  const float d5 = n3xV+0.5f*n1pn2;
  if(d5>minn1pn2n3&&d5<=n3) return d5; // case (5)
  const float d2 = 0.5f*n1+0.28867513f*sqrt(fdim(24.0f*n2*n3xV, sq(n1)));
  if(d2>n1&&d2<=n2) return d2; // case (2)
  const float d1 = cbrt(6.0f*n1xn2*n3xV);
  if(d1<=n1) return d1; // case (1)
  const float x3 = 81.0f*n1xn2*(n1pn2-2.0f*n3xV); // x3>0
  const float y3 = sqrt(fdim(23328.0f*cb(n1xn2), sq(x3))); // y3>=0
  const float u3 = cbrt(sq(x3)+sq(y3));
  const float d3 = n1pn2-(7.5595264f*n1xn2+0.26456684f*u3)*rsqrt(u3)*sin(0.5235988f-0.33333334f*atan(y3/x3)); // x3>0
  if(d3>n2&&d3<=minn1pn2n3) return d3; // case (3)
  const float t4 = 9.0f*sq(n1pn2+n3)-18.0f;
  const float x4 = fmax(n1xn2*n3*(324.0f-648.0f*V), 1.1754944E-38f); // avoid edge case V==0.5 to make x4>0
  const float y4 = sqrt(fdim(4.0f*cb(t4), sq(x4))); // y4>=0
  const float u4 = cbrt(sq(x4)+sq(y4));
  const float d4 = 0.5f*(n1pn2+n3)-(0.20998684f*t4+0.13228342f*u4)*rsqrt(u4)*sin(0.5235988f-0.33333334f*atan(y4/x4)); // x4>0
  /*if(d4>n3)*/ return d4; // case (4)
}
float __attribute__((always_inline)) plic_cube(const float V0, const float3 n) { // volume V0 in [0,1], normal vector n --> plane offset d
  const float ax=fabs(n.x), ay=fabs(n.y), az=fabs(n.z), V=0.5f-fabs(V0-0.5f); // eliminate symmetry cases
  const float n1 = fmax(fmin(fmin(ax, ay), az), 1E-5f);
  const float n3 = fmax(fmax(ax, ay), az);
  const float n2 = fmax(ax-n1+ay+az-n3, 1E-5f);
  const float d = offset_cube(V, n1, n2, n3); // calculate PLIC with reduced symmetry
  return copysign(d-0.5f*(n1+n2+n3), V0-0.5f); // apply symmetry for V0>1/2
}
```

Listing 3: Fully optimized OpenCL C implementation of the analytic 3D PLIC solution.

7.4.2 Plane-Sphere Intersection

PLIC can also be done for spherical cells with unit volume.

$$1 = V = \frac{4}{3}\pi r^3 \tag{159}$$

$$r = \sqrt[3]{\frac{3}{4\pi}} \tag{160}$$

The offset along the plane normal vector is the desired result while the volume of intersection is given. Since we have a sphere, the normal vector direction of the plane does not matter at all.

$$V_0 \;\rightarrow\; d_0 \tag{161}$$

Calculating the inverse PLIC (getting the volume V_0 from a given offset $d_0 \in [-r, r]$) is straight-forward and covered by the 'spherical cap' equation:

$$V_0 = \frac{\pi}{3}\left(r + d_0\right)^2 \left(2r - d_0\right) \in [0, 1] \tag{162}$$

Calculating the inverse function of the above equation again is quite difficult due to it being a third order polynomial with three complex solutions, but by leveraging the same trick as in the plane-cube intersection case (equations (150) and (151)), this real expression is obtained:

$$d_0 = \sqrt[3]{\frac{6}{\pi}} \sin\left(\frac{\pi}{6} - \frac{1}{3}\,\text{atan2}\left(2\sqrt{V_0 - V_0^2},\; 2\,V_0 - 1\right)\right) \tag{163}$$

Approximating a cubic LBM node as a unit sphere and solving PLIC for this sphere however proved to be not sufficient, resulting in non-round droplets with artifacts being clearly visible. Nevertheless, the plane-sphere intersection PLIC algorithm might be useful for some completely different applications.

8 Parametrization Procedure

Floating-point arithmetic is more accurate for numbers close to 1. Half of all floating-point numbers available are located in the interval $[-1, 1]$. This is the reason to not perform LBM with the raw numbers in SI-units, but instead with so-called *simulation units*. In between a conversion step is required.

The numbers in *simulation units* are unit-less. This is possible because units are artificial and nature does not know about them. Their only purpose is to compare quantities for similarity. For example, fluids at different time- and length-scales behave identical as long as the Reynolds number Re stays the same [86]. There is an entire list of such unit-less numbers (at least 69) which stay invariant under unit transformation, but only a few of them are relevant here:

$$Re = \frac{u\,L}{\nu} \qquad \text{Reynolds number} \tag{164}$$

$$Ma = \frac{u}{c} \qquad \text{Mach number} \tag{165}$$

$$We = \frac{\rho\,u^2 L}{\sigma} \qquad \text{Weber number} \tag{166}$$

$$Fr = \frac{u}{\sqrt{g\,L}} \qquad \text{Froude number} \tag{167}$$

$$Ca = \frac{\rho\,\nu\,u}{\sigma} \qquad \text{Capillary number} \tag{168}$$

Here ρ is the average density, u is the typical velocity, L is the typical length scale, ν is the kinematic shear viscosity, c is the lattice speed of sound, σ is the surface tension coefficient and g is the gravitational acceleration.

For the unit conversion in LBM [87] only the three SI-base-units $[m]$, $[kg]$ and $[s]$ are required. They are in the following treated just like variables and the brackets are used to more easily distinguish them from real physical quantities. The SI-units of all here relevant quantities are composed of only these three base-units, meaning that once they have been determined, any quantity can be converted from SI-units to simulation units by dividing by its SI-units. The conversion factors $[m]$, $[kg]$ and $[s]$ are determined as the ratio of three independent quantities in SI-units and in simulation units. The simplest way is to use a length scale L, a density ρ and a velocity u:

$$[m] := \frac{L^{\text{SI}}}{L^{\text{sim}}} \tag{169}$$

$$[kg] := \frac{\rho^{\text{SI}}}{\rho^{\text{sim}}}\,[m^3] \tag{170}$$

$$[s] := \frac{u^{\text{sim}}}{u^{\text{SI}}}\,[m] \tag{171}$$

The reason to use L, ρ and u is that these three parameters do not have a lot of freedom in simulation units. $\rho^{\text{sim}} = 1$ is strictly required for many LBM extensions, $L^{\text{sim}} < 10^5$ is limited by the memory available and $3\cdot 10^{-4} < u^{\text{sim}} < \frac{1}{\sqrt{3}}$ is limited by the LBM algorithm itself and by floating-point accuracy (see figure 16).

With the three base-units determined, any physical quantity can easily converted back and forth, for example the kinematic shear viscosity ν:

$$\nu^{\text{SI}} = \nu^{\text{sim}} \left[\frac{m^2}{s}\right] \tag{172}$$

A complete list of all relevant physical quantities is given in chapter 2. Caution needs to be taken to assure that the quantities in simulation units are all within their stable ranges, especially ν^{sim} should not be too small. If ν^{sim} gets too small, then the velocity u^{sim} needs to be adjusted accordingly.
The volume force in simulation units also must not be too small ($|\vec{f}| \geq 10^{-5}$), otherwise it will cancel out in equations (39) and (40) due to numeric loss of significance.

In *FluidX3D*, the entire unit conversion process is encapsulated into a unit conversion class, in which the SI-units of all relevant physical quantities are embedded for both conversion directions. This significantly reduces effort for implementing simulation setups.

9 Error Validation

9.1 Poiseuille Flow in 2D and 3D

For stationary laminar flow through a (cylindrical) channel in 2D and 3D (Poisseuille flow) with no-slip boundary conditions (fluid velocity at the boundaries is equal to the boundary velocity, here zero), the analytic solution to the Stokes equations is known. This makes Poiseuille flow the ideal system for testing numerical simulations.

9.1.1 Parametrization

Defining a Poiseuille flow in a cylinder requires three independent quantities from this list: Reynolds number Re, Mach number Ma, channel radius R, maximum flow velocity $u_{\max}$ at the channel center ($r = 0$), kinematic shear viscosity ν or LBM relaxation time τ. The remaining quantities are calculated with these two expressions:

$$u_{\max} = \frac{Ma}{\sqrt{3}} \tag{173}$$

$$\nu = \frac{2\,R\,u_{\max}}{Re} = \frac{\tau - \frac{1}{2}}{3} \tag{174}$$

9.1.2 Analytic Solution

The velocity profile $u(r)$ [88, p.182-183] only differs by a factor of 2 for 2D and 3D cases:

$$u(r) = \frac{f}{4\,\rho\,\nu}\,(R^2 - r^2) \cdot \begin{cases} 2 & \text{for 2D} \\ 1 & \text{for 3D} \end{cases} \tag{175}$$

Here r denotes the radial distance from the channel center at y_0 or (y_0, z_0) respectively and R denotes the channel radius. L_x, L_y and L_z denote the simulation box dimensions and $\rho = 1$ is the average fluid density.

$$r = \begin{cases} y - y_0 & \text{for 2D} \\ \sqrt{(y-y_0)^2 + (z-z_0)^2} & \text{for 3D} \end{cases} \tag{176}$$

$$L_x = 1 \qquad L_y (= L_z) := 2\,(R+1) \tag{177}$$

The force per volume $f := |\vec{f}|$ is calculated by setting $r = 0$ and solving equation (175) for $\vec{f}$:

$$f = \frac{2\,\rho\,\nu\,u_{\max}}{R^2} \cdot \begin{cases} 1 & \text{for 2D} \\ 2 & \text{for 3D} \end{cases} \tag{178}$$

Also quite often the flow rate Q through the channel is given or desired:

$$Q = \begin{cases} \int_{-R}^{R} \quad u(r)\,dr = \dots = \frac{2}{3}\,\frac{f R^3}{\rho\,\nu} = \frac{4}{3}\,R \;\; u_{\max} & \text{for 2D} \\ \int_{0}^{R} 2\,\pi\,r\,u(r)\,dr = \dots = \frac{\pi}{8}\,\frac{f R^4}{\rho\,\nu} = \frac{\pi}{2}\,R^2\,u_{\max} & \text{for 3D} \end{cases} \tag{179}$$

Note that in the 2D case the flow rate Q_{2D} is the area flow rate or volume flow rate per unit length with units $[Q_{\text{2D}}] = \frac{m^2}{s}$ while in the 3D case Q_{3D} is the volume flow rate with units $[Q_{\text{3D}}] = \frac{m^3}{s}$.

9.1.3 Error Definition and Convergence Criteria

The total error E of the simulated velocity profile $u_{\text{sim}}(r)$ is calculated as the L_2 norm [4, p.138]:

$$E(u) := \sqrt{\frac{\sum_{r=0}^{R} |u_{\text{sim}}(r) - u_{\text{theo}}(r)|^2}{\sum_{r=0}^{R} |u_{\text{theo}}(r)|^2}} \tag{180}$$

The sum runs across one slice of the channel and the velocities are the absolute velocities $u := |\vec{u}|$. The error will only decrease during simulation, so here it is sufficient to run the simulation until the error reaches its minimum. The error is calculated every $N = 1000$ simulation time steps.

9.1.4 Simulations

Simulations of the Poiseuille flow are done in both 2D and 3D and are used as a tool to compare the different velocity sets and collision operators. A parabolic flow profile is expected and this is exactly what the simulations show (figures 13 to 15). The flow is created by a volume force (section 3.5).

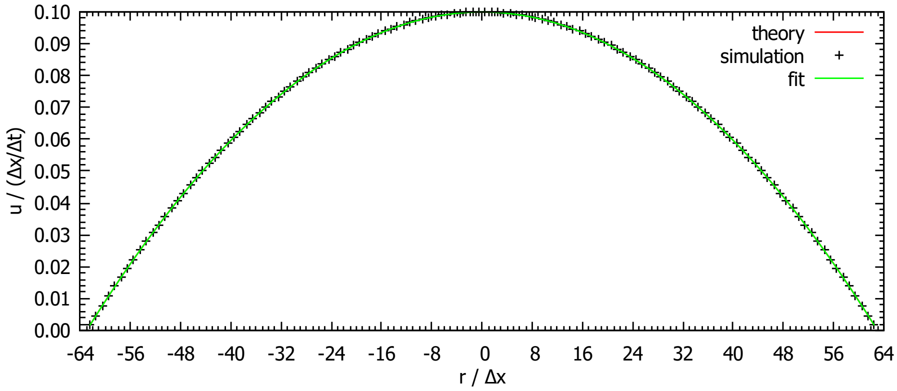


Figure 13: Example for the velocity profile in a 2D channel (D2Q9 TRT) at $R = 63$, $u_{\max} = 0.1$, $\tau = 1$. The red line for the theoretical velocity profile is completely covered by the green line fitted onto the simulated velocity. The error for this particular simulation is $E = 0.027\,\%$.

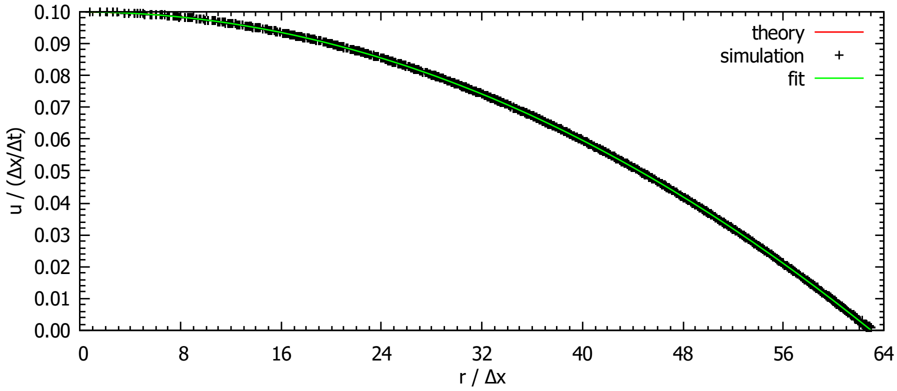


Figure 14: Example for the velocity profile in a cylindrical 3D channel (D3Q19 TRT) at $R = 63$, $u_{\max} = 0.1$, $\tau = 1$. The red line for the theoretical velocity profile is completely covered by the green line fitted onto the simulated velocity. The error for this particular simulation is $E = 0.164\,\%$.

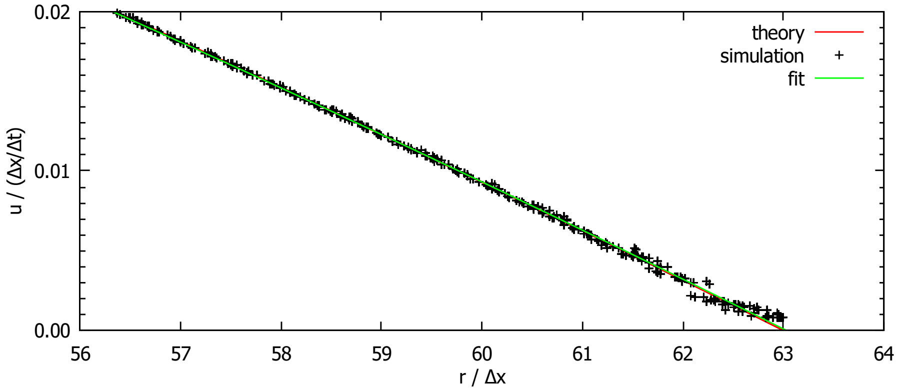


Figure 15: Figure 14 zoomed in at the outer edge of the channel. When closely observing the simulated velocities close here, one can notice the data points there becoming a bit fuzzy. This is due to the staircase-effect at the voxelated cylinder wall and examined further in figure 18.

In figures 16 and 17 the errors for different velocity sets and collision operators are compared. There is a quite large range of velocities $0.0003 \leq u \leq 0.5$ where the error is very low. It also becomes evident that D3Q7 and D3Q13 are insufficient for useful simulations and D3Q19 is even better than D3Q27 for the most part. The collision operators for the 3D velocity sets differ only insignificantly in error while for D2Q9 the MRT operator is much better than TRT and SRT.

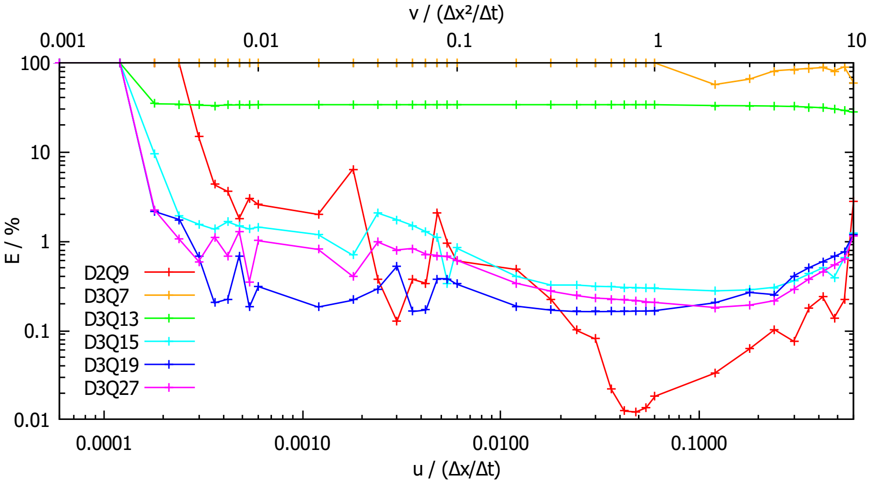


Figure 16: The velocity sets with the TRT collision operator at $R = 63$ compared for varying center velocity $u_{\max}$ and varying kinematic shear viscosity ν, while $Re = 7.56$ and therefore the simulated physics are kept constant. For velocities outside of the range $0.0003 \leq u_{\max} \leq 0.5$, the error drastically increases. D3Q19 holds up better than D3Q27 and D2Q9 has a narrow range in velocity where the error is especially small. D3Q7 and D3Q13 can't handle the Poiseuille flow at all.

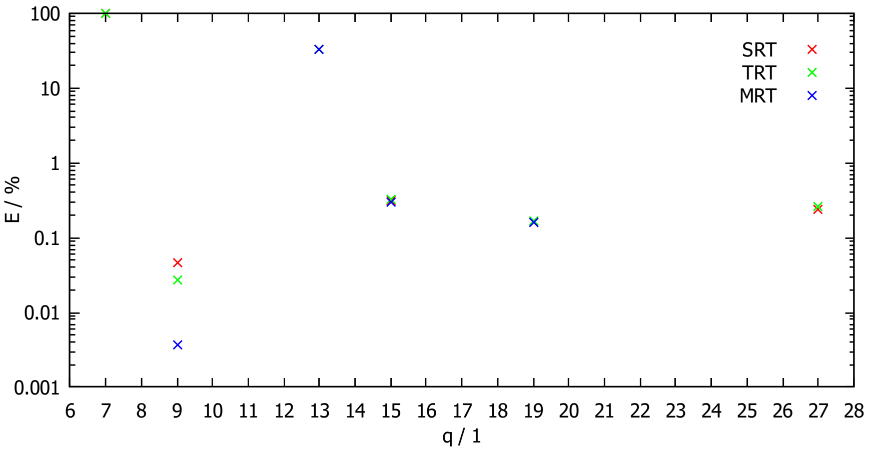


Figure 17: The errors for the different velocity sets q and collision operators SRT, TRT and MRT compared for $R = 63$, $u_{\max} = 0.1$, $\tau = 1$. For D3Q7 and D3Q27 the MRT operator is missing due to an incomplete definition of the relaxation matrix S. D3Q7 and D3Q13 are insufficient for useful LBM simulations. The collision operators only significantly differ in error for D2Q9. Quite surprisingly, D3Q19 has a slightly lower error than D3Q27 for all collision operators.

Lastly, artifacts embedded in the error definition (eq. (180)) are quantified (figure 18). For small R, the error drastically increases. There are three reasons for this:

1) errors due to spatial disctretization,
2) floating-point errors are larger for small velocities and
3) the staircase-effect at the voxelated curved cylinder surface.

For 3D, all three effects are at play: The number of lattice points in the bulk of the cylinder scale with R^2 while the number of lattice points at the circumference only scales with R, and all points contribute in equal amounts to the total error. For 2D simulations, there is no staircase-effect and also no difference in scaling for the numbers of points, leaving only point 1) as an error source, which is visible in the smaller slope if the curve for D2Q9.

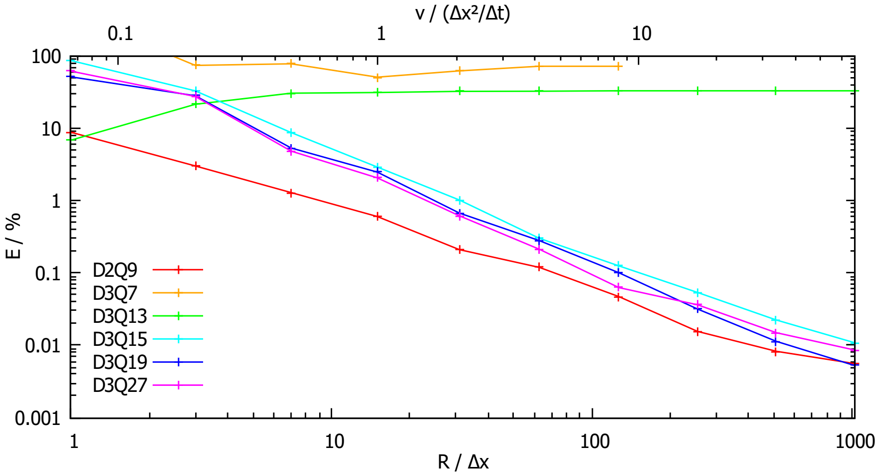


Figure 18: The staircase-effect is an artifact in the error of the 3D simulations. Here the the TRT collision operator is used. The channel radius R and the kinematic shear viscosity ν are varied while the center velocity $u_{\max} = 0.25$ and the Reynolds number $Re = 7.56$ and therefore the simulated physics are kept constant.

9.2 Forces on Boundaries via Stokes Drag

The validation of forces on boundaries (section 3.6) is done with a problem to which the analytic solution is known – the Stokes drag force on a sphere in laminar flow. Figure 19 illustrates what is being simulated in this section.

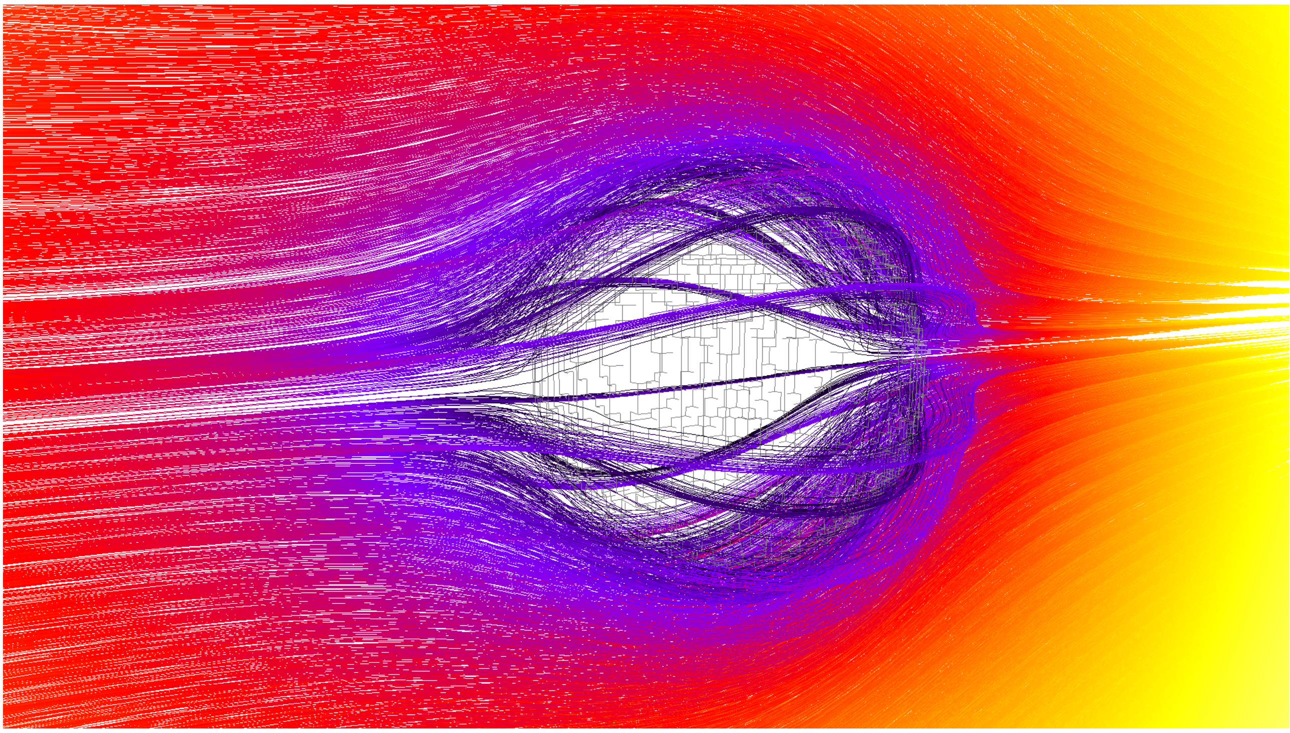


Figure 19: Laminar flow past a sphere of radius $R = 16$ in a cubic simulation box of side length $L = 384$ visualized with streamlines. The flow direction is left-to-right, although due to the symmetry at $Re = 0.01$ one could not tell from the image. The coloring indicates velocity. Visualization is done with the real-time OpenCL graphics engine of *FluidX3D*.

9.2.1 Analytic Solution

For a sphere of radius R resting at the coordinate origin, when it is in laminar flow with density ρ_0 and velocity $\vec{u}_0$ at infinite distance from the origin, the analytic density and velocity fields [88, p.230-235][89][90, p.168-171][91, p.36-38] are known as the following:

$$\vec{x} := (x,\, y,\, z)^T \qquad\qquad r := \sqrt{x^2+y^2+z^2} \tag{181}$$

$$\rho(\vec{x}) = \rho_0 - \frac{9\,\rho_0\,\nu\,R\,(\vec{u}_0 \circ \vec{x})}{2\,r^3} \tag{182}$$

$$\vec{u}(\vec{x}) = \vec{u}_0 - \frac{3}{4}\left(\left(\frac{R}{r} + \frac{R^3}{3\,r^3}\right)\vec{u}_0 + \left(\frac{R}{r^3} - \frac{R^3}{r^5}\right)(\vec{u}_0 \circ \vec{x})\,\vec{x}\right) \tag{183}$$

The analytic solution for the drag force $\vec{F}$ on the sphere is given by

$$\vec{F} = 6\,\pi\,\rho\,\nu\,R\,\vec{u}_0 \tag{184}$$

9.2.2 Strategy

The goal of this test is to validate the accuracy of boundary forces in LBM (section 3.6), meaning that the error must explicitly be the error of the force on the sphere and not the error of the velocity field. This in turn means that a volume force (section 3.5) cannot be used to drive the flow, because the sphere is the only object in the simulation box that slows down the flow. When a volume-force-driven flow becomes stationary, any boundary geometry in the simulation box will always absorb exactly the amount of force that was put into the fluid via

volume force, making the error always (almost) zero. Of course, one could then measure the stationary velocity profile and calculate the error from velocity, but that would only validate the bounce-back boundaries and not fulfill the purpose of validating the boundary forces.
Instead, the flow needs to be driven by moving bounce-back boundaries (section 3.4.3) at the outer edge of the simulation box, which in turn have been validated with Poiseuille flow. Furthermore, it is not sufficient to set the velocity at these boundaries to the velocity $\vec{u}_0$ infinitely far away from the sphere, because the simulation box can never be infinitely large – the error would then converge to about 10 % (force would be too large) even for the largest possible simulation box that fits into memory. Even in an infinitely large simulation box, the streamlines are never perpendicular to $\vec{u}_0$; they always curve around the sphere, although curvature gets less further out. It would not make sense to enforce straight streamlines at finite distance to the sphere, thereby artificially constricting the flow. To avoid this, the velocity at the boundaries is set to the analytic solution $\vec{u}(\vec{x})$ from equation (183), allowing streamlines to curve out and back into the simulation box at the boundaries.

9.2.3 Error Definition and Convergence Criteria

The relative error E of a measured property x_sim of the simulation to the theoretically expected property x_theo here is calculated as the L_1 norm:

$$E(x) := \frac{|x_\text{sim} - x_\text{theo}|}{x_\text{theo}} \tag{185}$$

Since it is not clear that the error will only decrease during simulation, it is not sufficient to run the simulation until the error reaches a local minimum. Instead, the following definition is used for determining error convergence: Both the absolute slope and the absolute curvature

$$\left|\frac{\partial E}{\partial t}\right| < \epsilon\, E \qquad\qquad \left|\frac{\partial^2 E}{\partial t^2}\right| < \epsilon\, E \tag{186}$$

of the error E must be smaller than $\epsilon\, E$ with $\epsilon \ll 1$ being a small number. The derivatives are calculated as the first- and second-order backward difference of the last three error values, which are computed every N simulation time steps.

9.2.4 Simulation Parameters

All simulations are performed with D3Q19 TRT at Reynolds number $Re = 0.01$. This leaves the freedom of choosing either the kinematic shear viscosity ν or velocity $u_0 := |\vec{u}_0|$. Simulations have shown (figure 20) that the error E does not depend on ν as long as $0 < \nu \approx 10^0$ and $0.0003 \leq |\vec{u}_0| \leq 0.5$ remain in reasonable ranges (figure 16). For section 9.2.5, $\nu = 1$ (except for $R = 32$ where $\nu = 10$) and for 9.2.6, $\nu = 10$ is used in order to not have too small velocity. The error is calculated as $E(|\vec{F}|)$ every $N = 100$ LBM time steps. For achieving sufficient convergence, $\epsilon = 10^{-4}$ is chosen. The sphere radius is denoted as R and the simulation box is cubic with side length L. The flow is created along the x-axis with positive velocity.

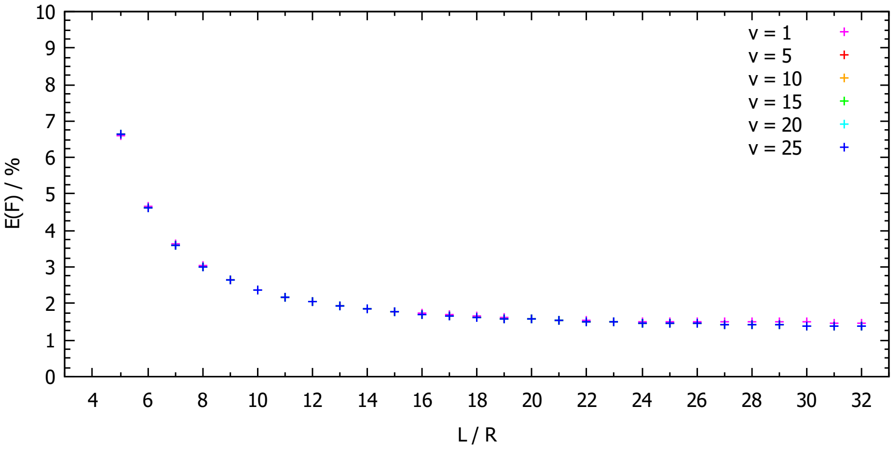


Figure 20: Simulations for $R = 8$ and $Re = 0.01$ for various $\nu = \{1, 5, 10, 15, 20, 25\}$ with u_0 adjusted accordingly. The observation is that as long as no physical property is changed ($Re = const$), the error does not significantly change as well.

9.2.5 Results – $R = const$, L is varied

Figure 21 shows the error behavior when the sphere radius is kept constant and the simulation box size is varied. As expected, for a larger simulation box, the error goes down to a plateau induced by the staircase effect of sphere voxelation.

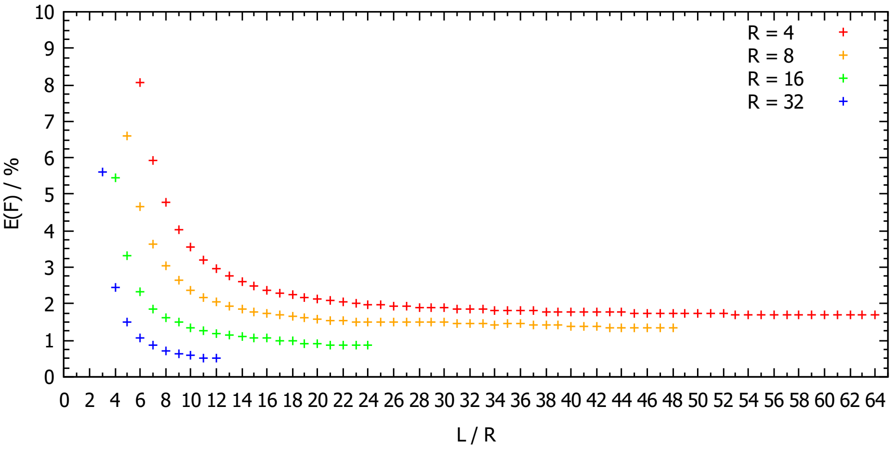


Figure 21: For a fixed sphere radius $R = \{4, 8, 16, 32\}$ the box size is varied. As expected, for a larger box size L the error decreases down to a plateau. Moreover, the error is smaller for larger spheres, indicating the plateau being caused by the staircase-effect of the sphere in limited voxel resolution.

9.2.6 Results – $L/R = const$, R is varied

Figure 22 shows various simulations for exactly the same physics ($L/R = const$) with varying voxel resolution of the sphere. As expected, the error overall decreases with increasing R (voxel resolution), with the addition that the error is especially small when R is a multiple of 12 – for which there surely is some geometrical explanation.

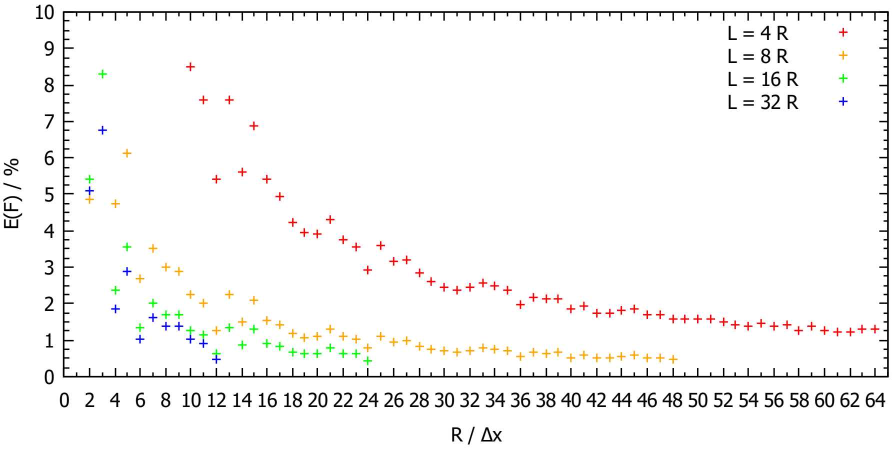


Figure 22: Here the ratio of the simulation box size and the sphere radius L/R is fixed and R is varied, meaning that along a data line the simulated physics are identical. Confirming the indication of figure 21 that the plateau of the error is caused by the staircase-effect; here the observation is that for identical physics the error decreases when the sphere is better resolved (R is larger). The data points are not monotonic however and certain integer radii (multiple of 12) will make the error especially small. Also in agreement to figure 21, larger L/R results in a smaller error.

9.2.7 Results – Velocity Field Errors

Figure 23 shows the velocity field around the sphere as well as its error for various sphere radii. The simulated velocity fields are indistinguishable from the theoretical solution with the bare eye; only separate plots of the error show that the error in velocity is largest in close vicinity to the sphere surface.

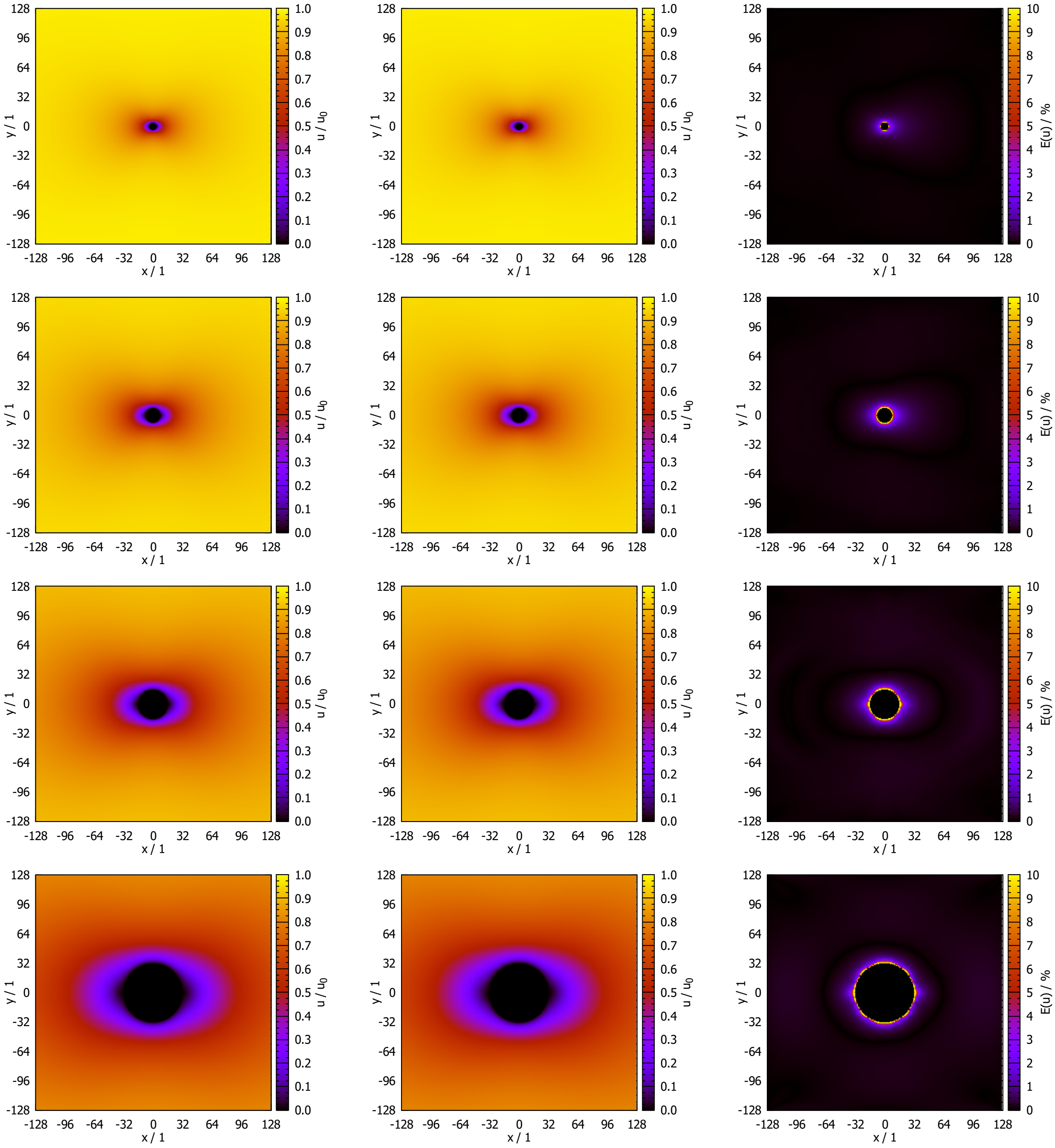


Figure 23: A slice through the velocity field of laminar flow past a sphere of radius $R = \{4,\ 8,\ 16,\ 32\}$ (top to bottom) at $Re = 0.01$. The flow direction is from left to right. The columns are the normalized theoretical velocity magnitude $|\vec{u}_{\text{theo}}|/u_0$ (left), the normalized simulated velocity magnitude $|\vec{u}_{\text{sim}}|/u_0$ (middle) and the error of the velocity magnitude $E(|\vec{u}|)$ (right). For the first three rows, $\nu = 1$ and for the last row $\nu = 10$ in order to avoid the floating-point errors of too small fluid velocity. The error is largest in close vicinity to the sphere surface, where the velocity magnitude is smallest, and its distribution is not exactly symmetrical due to $Re > 0$. Next to the voxelated sphere surface, single voxels with unusually high error are observed due to the staircase-effect.

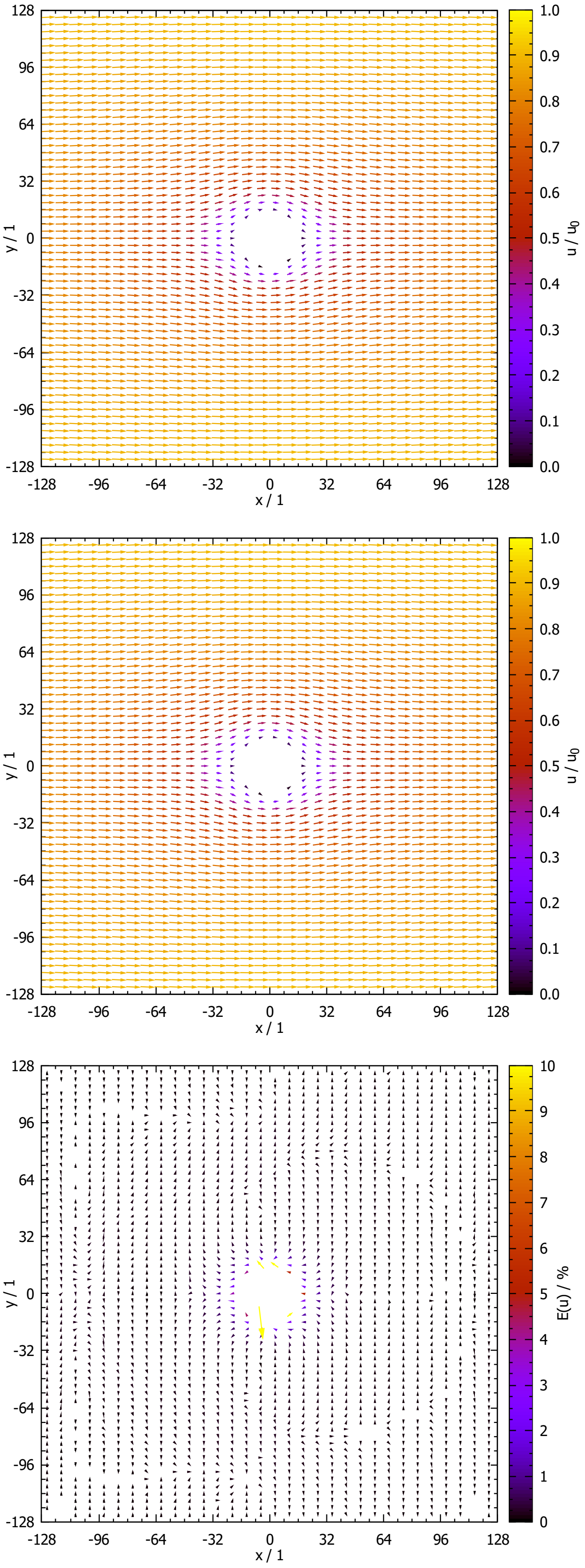


Figure 24: A slice through the velocity field of laminar flow past a sphere of radius $R = 16$ at $Re = 0.01$. The flow direction is left to right, indicated with arrows. The theoretical velocity field is plotted on the top and the simulated velocity field is plotted in the middle. On the bottom, the relative difference of the theoretical and simulated velocity fields is plotted as a vector field. Errors are largest in close vicinity of the sphere surface.

9.3 VoF Mass Conservation Test

An easy test to check the validity of VoF is fluid mass conservation. Fluid in an enclosed simulation box without inflow or outflow should neither drain into Nirwana nor become more. A necessary requirement for this check is to have a simulation where the fluid and the interface layer are always moving, which is met by the periodic faucet setup (figure 25). Here the periodic boundaries are used in z-direction and gravity points downward, making the fluid never rest. The simulation box dimensions are (96, 192, 128), the kinematic shear viscosity in simulation units is $\nu = 0.02$, the volume force in simulation units is $f = 0.001$ and the Reynolds number is approximately $Re \approx 500$ in order to have some turbulent flow.

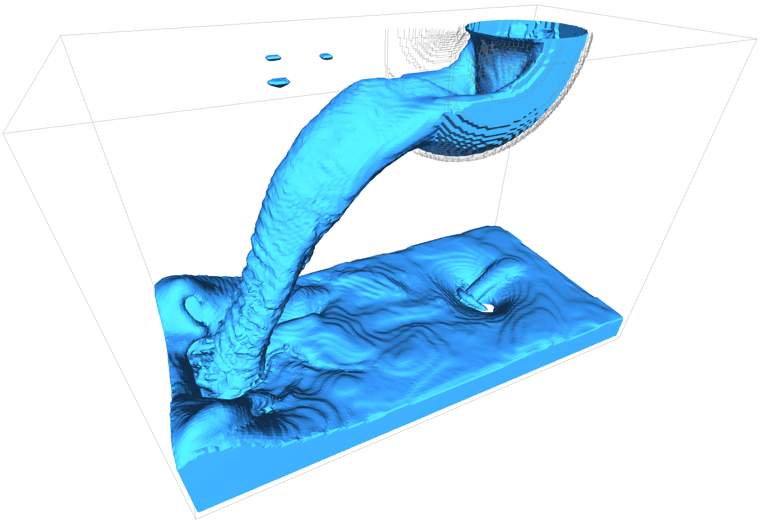


Figure 25: An especially challenging setup for testing mass conservation, specifically designed to make the algorithm fail if there is any flaw. Turbulent free-surface flow is sustained by a vertical volume force and vertical periodic boundaries. The fluid circles through the faucet approximately every 1000 time steps.

The fluid mass fraction M is calculated as the total mass divided by the simulation box volume

$$M := \frac{1}{L_x\, L_y\, L_z} \sum_{\{\mathit{fluid},\ \mathit{interface}\}} \varphi\,\rho \quad \in [0,\, 1] \tag{187}$$

and is plotted in figure 26 for 14 million LBM time steps. Without surface tension ($\sigma = 0$), the fluid mass fraction during this time stays within 2 % with random fluctuations and no clear upwards or downwards trend. Small random fluctuations are expected due to floating-point errors. However with surface tension included (algorithm from section 7.3.3, $\sigma = 0.01$), mass is not conserved. The fluid drains into the aether to the point where the periodic flow is no longer sustained, and plateaus after there is no more surface movement.

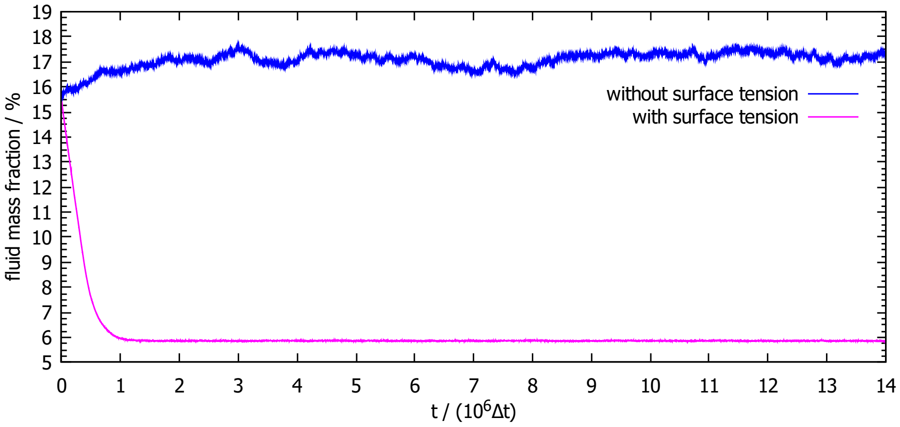


Figure 26: The fluid mass fraction of the simulation illustrated in figure 25 measured over 14 million LBM time steps. Each simulation took about three hours of compute time on the Titan Xp.

9.4 Curvature Calculation Error

The accuracy of the four different approaches for curvature calculation in chapter 7 is evaluated by comparing the calculated curvature at the surface of a sphere of fluid with the analytic mean curvature

$$\kappa_{\text{theo}} = \frac{1}{R} \tag{188}$$

whereby R is the sphere radius. The four algorithms are given numbers:

0 7.2.1

1 7.2.2

2 7.3.3

3 7.3.4

The initialization of the fill levels φ and flags is done by inverse PLIC to ensure a smooth surface even at simulation startup. Each of the algorithms then is given time to relax the sphere with simulation parameters $\tau = 1$ and $\sigma = 0.001$. The error of the curvature is defined as the L_1[15] error norm across all interface points

$$E(\kappa) := \frac{\sum |\kappa_{\text{sim}} - \kappa_{\text{theo}}|}{\sum \kappa_{\text{theo}}} \tag{189}$$

and error convergence is defined as described in 9.2.3 with $\epsilon = 10^{-5}$ with the error being calculated every $N = 100$ time steps. If the error does not converge within 100000 time steps, the error is defined as the average of the last 100 error values. The errors are plotted in figure 27.

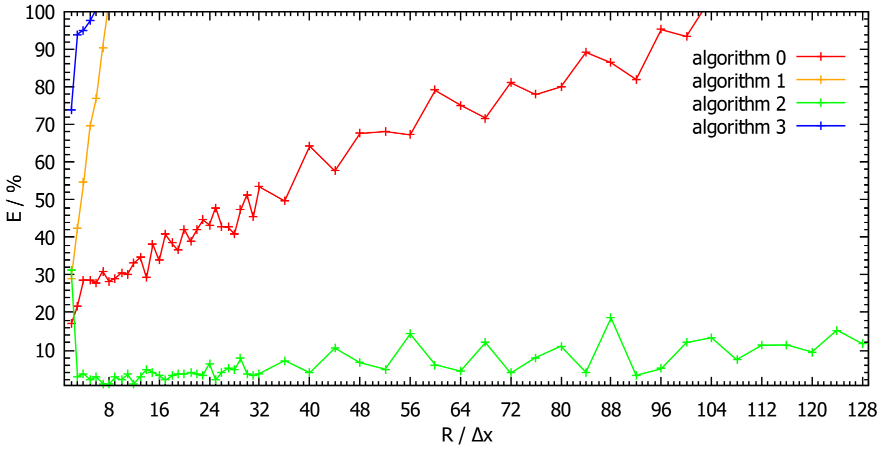


Figure 27: The L_1 error of the four curvature calculation algorithms 0 (7.2.1), 1 (7.2.2), 2 (7.3.3) and 3 (7.3.4) for different sphere radii R. The errors of algorithms 1 and 3 are totally off the chart, algorithm 0 at least gets the order of magnitude right and algorithm 2 is the way to go with an error below 5 % for $R \leq 32$.

It becomes obvious that 7.3.3 is the algorithm of choice. For larger sphere radii, fluctuations in the error increase, but surface tension effects are also much less pronounced for large radii, so this has less of an impact on simulations.

In figure 29 an example of the spatial error distribution on the sphere surface for $R = 16$ (illustrated in figure 28) is shown for algorithms 0 and 2. The overall error of algorithm 0 is an order of magnitude larger than the error of algorithm 2. Moreover, algorithm 0 even after simulation convergence has a few single lattice points with extremely large error. In comparison, for algorithm 2 the error is distributed very homogeneously after simulation convergence.

---

[15] In the L_2 error norm, the denominator would remain constant for increasing R: $\sum |\kappa_{\text{theo}}|^2 \approx 4\,\pi\,R^2\,|\kappa_{\text{theo}}|^2 = 4\,\pi\,R^2\,\frac{1}{R^2} = 4\,\pi$. This means that as more terms are summed up in the numerator for larger R (there are more surface points for larger R), the L_2 error would increase just because of the here ill-fated L_2 error definition.

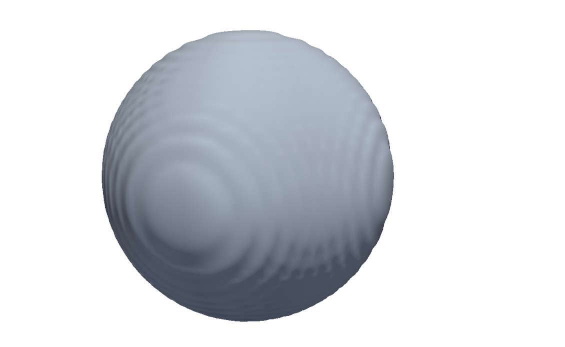


Figure 28: The initialization state sphere with $R = 16$ is visualized with marching-cubes in ParaView to illustrate that the initialization of a smooth surface works.

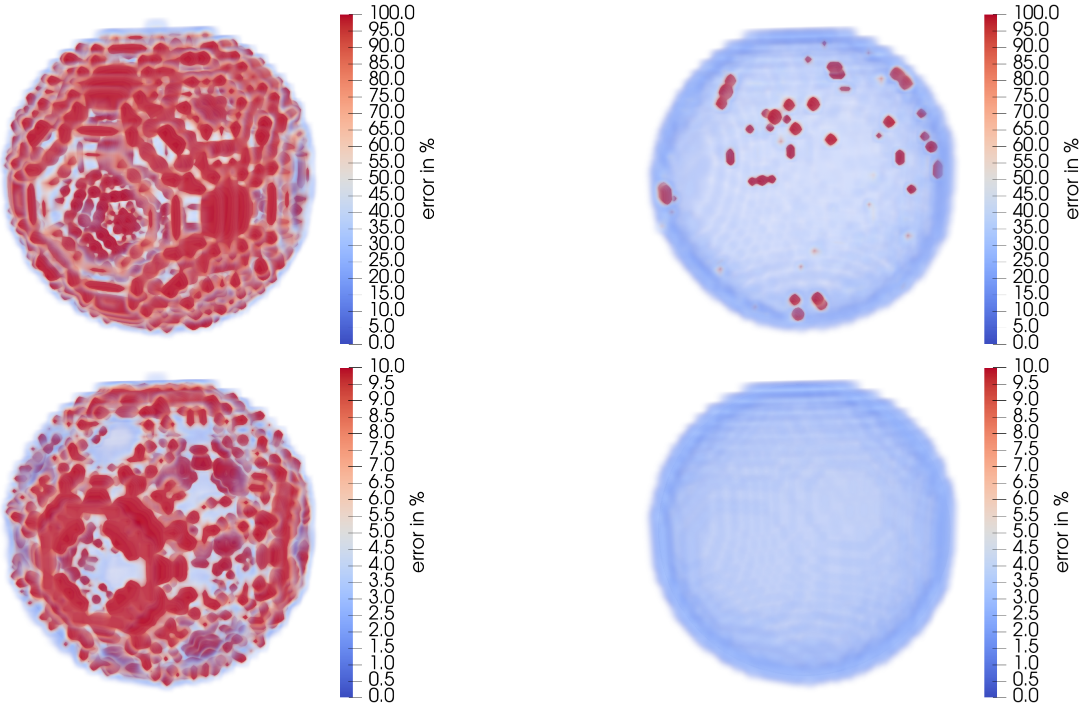


Figure 29: Example for the error distribution on the sphere surface for $R = 16$. The top row shows algorithm 0 (7.2.1) and the bottom row shows algorithm 2 (7.3.3). On the left column, the errors are plotted for the initial state of the sphere as depicted in figure 28 and on the right column the errors are shown after the simulation has converged (these errors are plotted in figure 27). Notice that the error scale bar for algorithm 0 is ten times higher than for algorithm 2.

9.5 Plateau-Rayleigh Instability of a perturbed Cylinder of Fluid

When a thin stream of water flows out of the tap, it gets increasingly thinner and at some point breaks up into individual droplets. This phenomenon is called the Plateau-Rayleigh instability and is a result of surface tension. This effect also happens on a perturbed cylinder of fluid without gravity, which is examined here in order to validate both the VoF implementation and the curvature calculation algorithm from section 7.3.3. The simulations are done with D3Q19 TRT, density $\rho = 1$, initial velocity $\vec{u} = 0$ everywhere, a relaxation time of $\tau = 1$ and surface tension of $\sigma = 0.1$.
The theory predicts that $\lambda_{\text{max}} \approx 9\,R$ is the perturbation wavelength of maximum growth rate and that perturbations with $\lambda < 2\,\pi\,R$ are unstable and decay [92]. The simulations (figures 30 to 32), which take only a few seconds of computation time each, show very good agreement with the theoretical predictions.

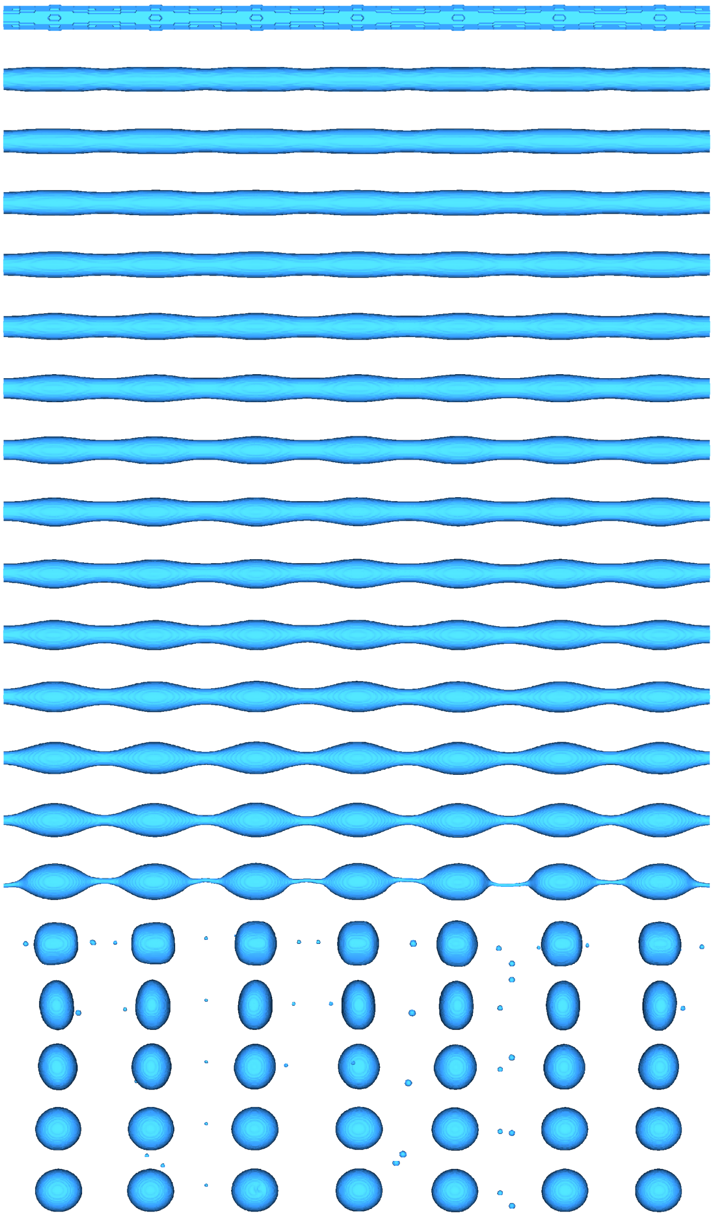


Figure 30: A periodic cylinder of fluid (length $L = 512$, radius $R = \frac{512}{7 \cdot 9}$) is initially perturbed with a sine wave (amplitude $A = 0.05\,R$, wavelength $\lambda = 9\,R$). The interface lattice points are initialized with a fill level of $\varphi = \frac{1}{2}$, making the first frame at $t = 0$ (top image) look jagged. Shortly after initialization, this jaggedness completely disappears and the perturbation grows until the cylinder separates into seven droplets. The images (top to bottom) are each separated by 100 LBM time steps. Visualization is done with the marching-cubes algorithm [80], implemented in the real-time OpenCL graphics engine of *FluidX3D*.

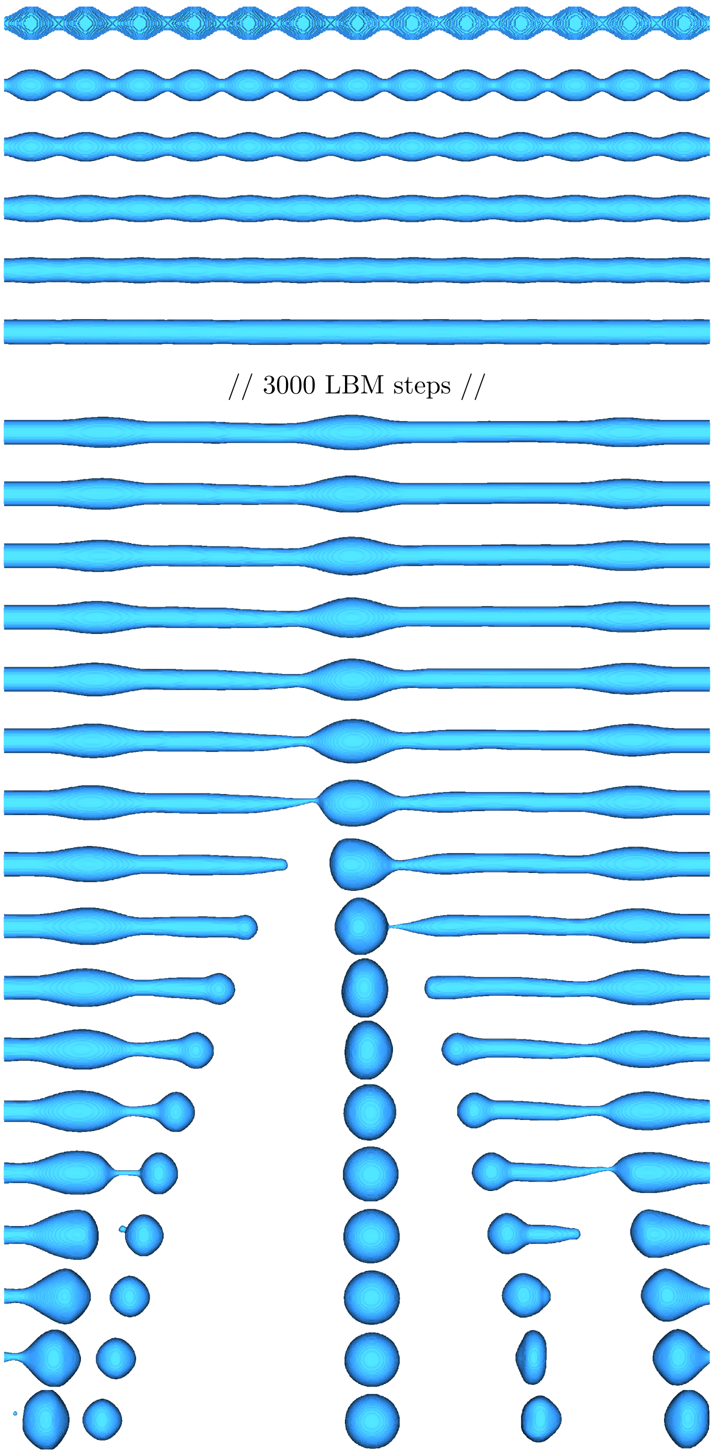


Figure 31: When the initial perturbation wavelength is $\lambda < 2\,\pi\,R$, the perturbations are unstable and expected to decay. This is exactly what is observed here. A periodic cylinder of fluid (length $L = 512$, radius $R = \frac{512}{13\cdot 5}$) is initially perturbed with a sine wave (amplitude $A = 0.5\,R$, wavelength $\lambda = 5\,R$). Again, the interface lattice points are initialized with a fill level of $\varphi = \frac{1}{2}$, making the first frame at $t = 0$ (top image) look jagged. As expected, the initially quite large perturbation quickly decays and the fluid takes on the shape of a cylinder. After running the simulation for longer, the cylinder at some point irregularly separates into droplets, but with a larger wavelength than the initial perturbation. The images (top to bottom) are each separated by 100 LBM time steps.

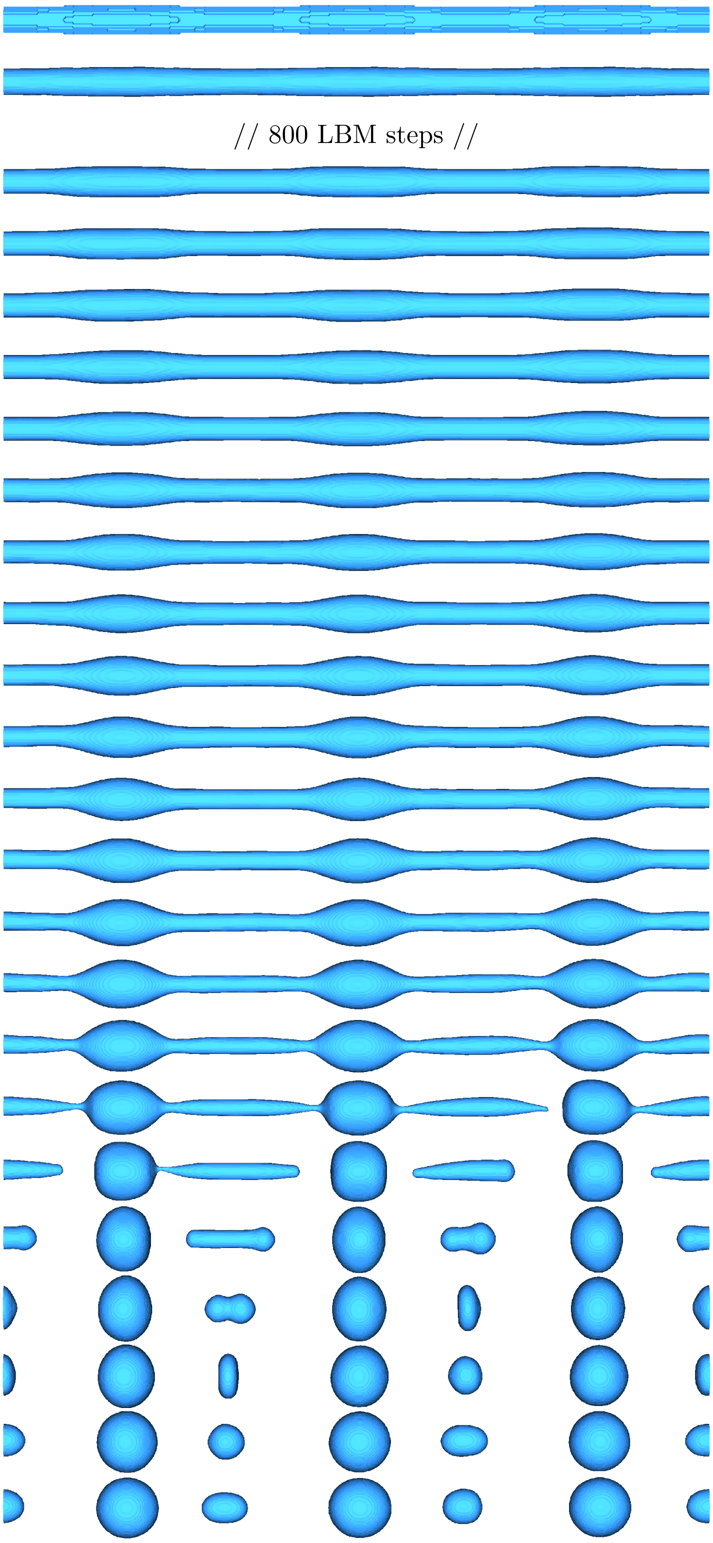


Figure 32: Here a periodic cylinder of fluid (length $L = 512$, radius $R = \frac{512}{3 \cdot 18}$) is initially perturbed with a sine wave of long wavelength (amplitude $A = 0.05\,R$, wavelength $\lambda = 18\,R$). The first frame at $t = 0$ (top image) again looks jagged due to fill level initialization of $\varphi = \frac{1}{2}$ of interface lattice points. With the wavelength being approximately twice the wavelength of maximum growth rate, the drops of the initial perturbation separate and in between smaller drops are created. The images (top to bottom) are each separated by 100 LBM time steps.

Now the growth rate is examined quantitatively. Therefore, the cylinder (or later sphere) radius at the maximum of the initial undulation is measured as the average of the diameters in x- and z-directions. The cylinder of fluid is aligned along the y-axis. This measurement is done after every single LBM time step for 3000 time steps for $\lambda \in \{1, 2, 3, ..., 25\}$. The initial radius of the cylinder is $R = 8$ and the initial perturbation amplitude is $A = 0.1\,R$, the relaxation time is $\tau = 1$ and the surface tension coefficient is $\sigma = 0.1$. The simulation results are plotted in figure 33.

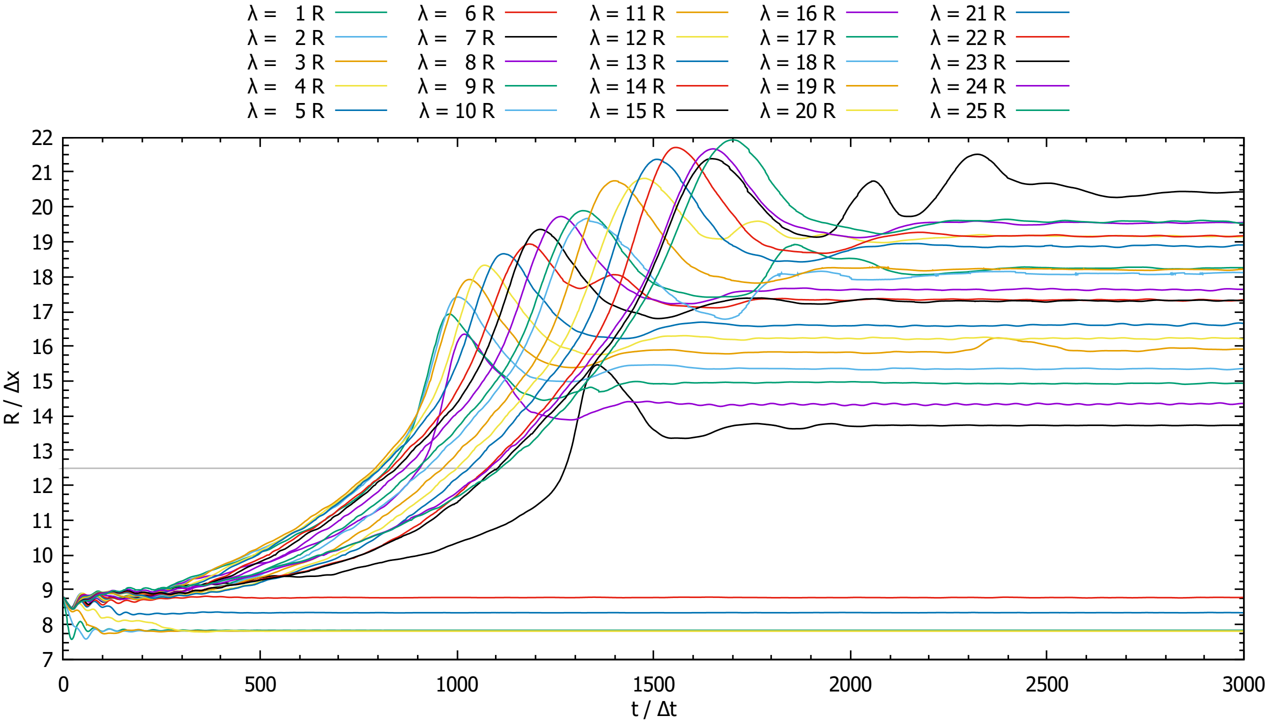


Figure 33: The center radius R of the fluid cylinder (and later sphere) measured over time. For different wavelengths λ of the initial undulation, R will either decay from its initial perturbated state at $R(t = 0\,s) = 8.8\,m$ down to $R = 8.0\,m$ or increase exponentially until the cylinder separates into individual droplets. The separation radius is approximately $R_{\text{sep}} = 12.5\,m$ and indicated by the horizontal gray line. After separation, the slope of the curve increases even more until the curve reaches a maximum, after which the drop bounces a bit and finally relaxes to a constant radius. For some curves, for example $\lambda = 11\,R$, there is a ripple later in the curve caused by satellite droplets fusing with the main drop after separation.

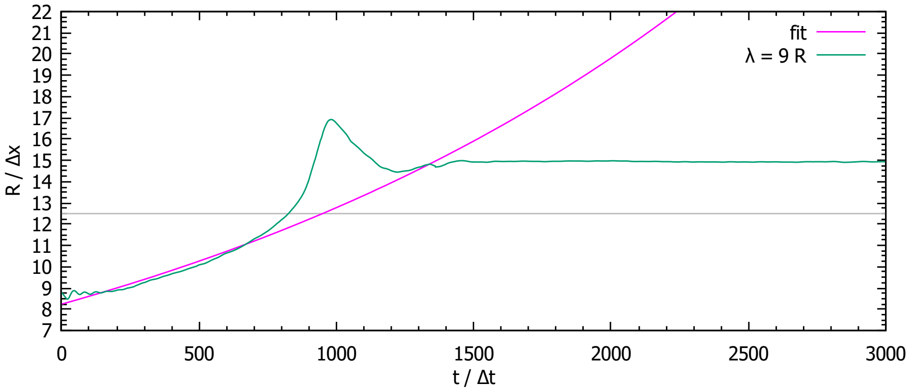


Figure 34: An example of the fitting for $\lambda = 9\,R$ with equation (190).

For obtaining the growth rate, an exponential fit of the form

$$R(t) = R_0\, e^{k\,t} \tag{190}$$

is done on each of the curves, but only in the range before the separation of the cylinder where $R < 12.5\,m$, as indicated by the horizontal gray line in the plots. An example of one of the fits is shown in figure 34. The last step is to plot the growth rates k for all of the λ values (figure 35). This shows that an initial undulation with $\lambda < 2\,\pi$ is indeed unstable. The maximum growth rate deviates a bit from the theoretical $\lambda_{\text{max}}^{\text{theo}} \approx 9\,R$ at $\lambda_{\text{max}}^{\text{sim}} = 11\,R$.

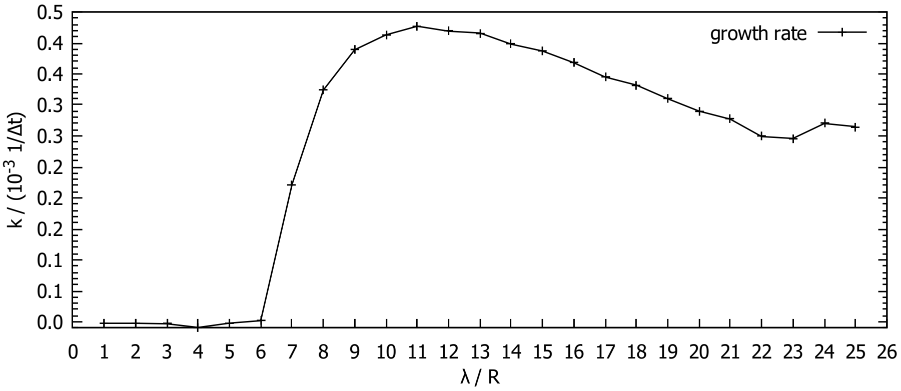


Figure 35: The growth rate k from equation (190) plotted for different initial undulation wavelengths λ.

10 Simulations and Results

10.1 Force on a Particle attached to the Wall of a rectangular Microchannel

For both laminar flow in a rectangular channel and laminar flow around a sphere there are solutions to the Stokes equations, but not for the combination of both, i.e. a sphere attached to the wall of a rectangular channel. This is the experiment carried out by Wolfgang Gross and Simon Wieland in the labs of the Experimentalphysik I here at the university. They want to attach microplastic particles to the membranes of living mouse macrophage cells in a rectangular microchannel and by creating a flow through the channel make the particles pull on the membranes. Therefore they need to know the pulling force of the particles, which is the result of the simulations here.

10.1.1 Experimental Setup

The experimental setup consists of a spherical microplastic particle with radius $R^{\text{SI}} \in \{[0.5,\ 1.5],\ 5.0\}\,\mu m$ fixed to the center of the bottom wall of a rectangular microchannel with the dimensions $(19.0,\ 1.0,\ 0.1)^T mm$. The flow direction is along the x-axis and the flow rate is $Q^{\text{SI}} \in [0.1,\ 50.0]\,\frac{\mu L}{s}$. For the fluid, the density and viscosity of water [93] at $T^{\text{SI}} = 37°C$ are assumed. In SI-units and without prefixes the given parameters are:

- sphere radius $R^{\text{SI}} \in \{[0.5,\ 1.5],\ 5.0\} \cdot 10^{-6}\,m$
- channel dimensions (flow is in x-direction) $L_x^{\text{SI}} = 19.0 \cdot 10^{-3}\,m$, $L_y^{\text{SI}} = 1.0 \cdot 10^{-3}\,m$, $L_z^{\text{SI}} = 0.1 \cdot 10^{-3}\,m$
- volume flow rate $Q^{\text{SI}} \in [0.1,\ 50.0] \cdot 10^{-9}\,\frac{m^3}{s}$
- fluid density $\rho^{\text{SI}} = 993.36\,\frac{kg}{m^3}$
- fluid dynamic viscosity $\mu^{\text{SI}} = 0.6922 \cdot 10^{-3}\,\frac{kg}{m\,s}$
- fluid kinematic shear viscosity $\nu^{\text{SI}} = \frac{\mu^{\text{SI}}}{\rho^{\text{SI}}} = 6.968 \cdot 10^{-7}\,\frac{m^2}{s}$

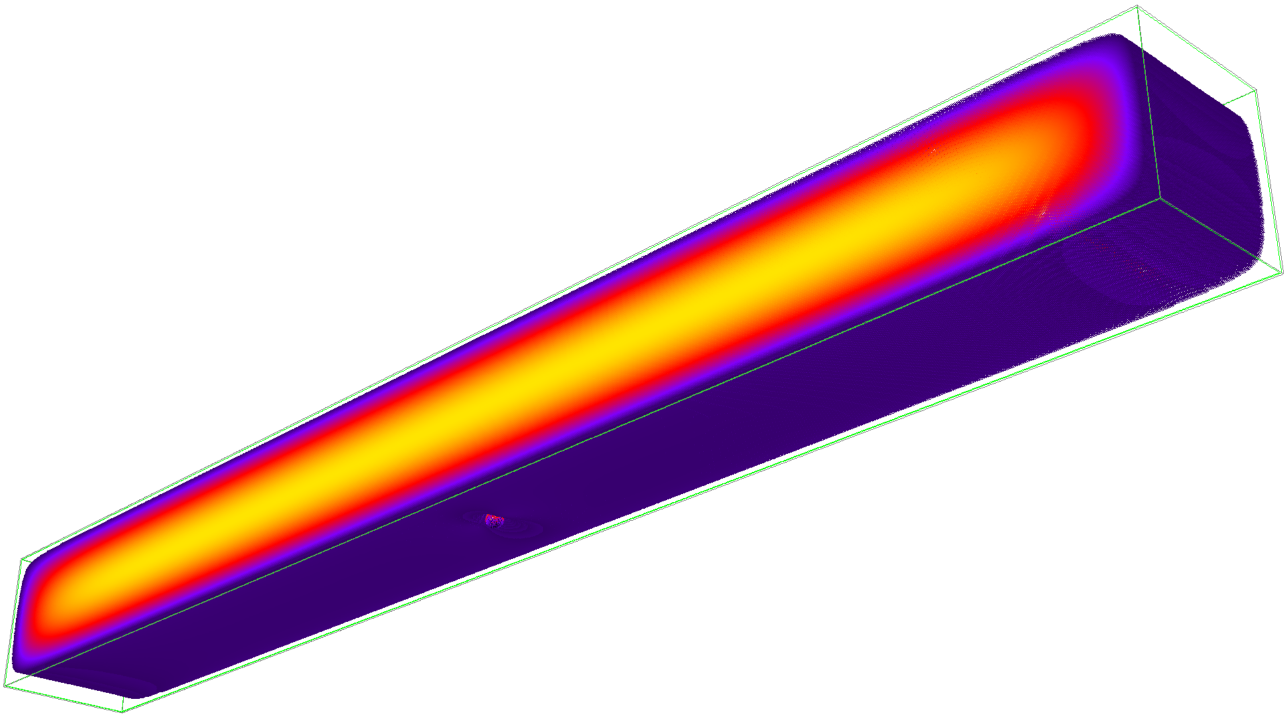


Figure 36: A $0.1\,mm$ short section of the channel illustrated for the largest particle at $R^{\text{SI}} = 5\,\mu m$ with flow rate $Q^{\text{SI}} = 50\,\frac{\mu L}{s}$. The colors of the fluid represent the velocity magnitude and the particle is colored by local force magnitude. The particle is barely visible with its radius of $R^{\text{sim}} = 8$ in simulation units compared to the simulation box dimensions $(160,\ 1600,\ 160)^T$. Only a small section of the channel fits into memory. The flow direction is left-to-right. Visualization is done with the real-time OpenCL graphics engine of *FluidX3D*.

10.1.2 Poiseuille Flow in a rectangular Channel

The analytic solution for laminar flow in a rectangular channel [94, 95] is known as

$$u_x(y,z)=\frac{4\,L_z^2\,f_x}{\pi^3\rho\,\nu}\lim_{N\to\infty}\sum_{n=1,3,5,\dots}^{2\,N-1}\frac{1}{n^3}\left(1-\frac{\cosh\left(\frac{n\,\pi\,y}{L_z}\right)}{\cosh\left(\frac{n\,\pi\,L_y}{2\,L_z}\right)}\right)\sin\left(\frac{n\,\pi\,\left(z+\frac{L_z}{2}\right)}{L_z}\right) \tag{191}$$

whereby $-\frac{L_y}{2}\leq y\leq\frac{L_y}{2}$ and $-\frac{L_z}{2}\leq z\leq\frac{L_z}{2}$ are the coordinates in the channel cross section and f_x is the force per volume in x-direction (equal to the pressure gradient), which is defined by the volume flow rate Q_x [94] in x-direction:

$$f_x=\frac{12\,\rho\,\nu\,Q_x}{L_y\,L_z^3}\left(1-\lim_{N\to\infty}\sum_{n=1,3,5,\dots}^{2\,N-1}\frac{192\,L_z}{\pi^5\,n^5\,L_y}\tanh\left(\frac{n\,\pi\,L_y}{2\,L_z}\right)\right)^{-1} \tag{192}$$

The infinite sum obviously cannot be computed, so instead for eq. (191), $N=23$ is set, which is limited not by compute time but by the $\cosh(x)$ function blowing up for too large x, resulting in floating-point overflow even with FP64 double precision. For eq. (192), N is neither limited by compute time (f_x needs to be calculated only once per simulation) nor by floating-point overflow and $N=256$ should be sufficient. The sums are calculated in reverse (large n first) in order to mitigate numerical loss of significance (see section 4.3.9).

10.1.3 Estimation of the expected Force on the Particle

In order to at least figure out the order of magnitude of the expected drag force on the sphere, the force is estimated with the Stokes drag equation (184) with the velocity at the sphere center. In order to approximate this velocity, the channel is assumed to have infinite width ($L_z\ll L_y$), meaning the Poiseuille flow is approximated to be only 2D (equations (179) and (175)). The force per volume f for a given (area) flow rate Q_{2D} is

$$f=\frac{3\,\rho\,\nu\,Q_{\text{2D}}}{2\,\left(\frac{L_z}{2}\right)^3}=\frac{12\,\rho\,\nu\,Q_{\text{2D}}}{L_z^3} \tag{193}$$

Note that $[Q_{\text{2D}}]=\frac{m^2}{s}$ is the volume flow rate per unit length:

$$Q_{\text{2D}}=\frac{Q}{L_y} \tag{194}$$

The velocity evaluated one sphere radius away from the bottom wall at $z=0$ then is

$$u\,\Big|_{r=-\frac{L_z}{2}+R}=\frac{f}{2\,\rho\,\nu}\left(\left(\frac{L_z}{2}\right)^2-\left(-\frac{L_z}{2}+R\right)^2\right)=6\,Q\,\frac{R\,(L_z-R)}{L_y\,L_z^3} \tag{195}$$

which finally is inserted into eq. (184):

$$F\approx 36\,\pi\,\rho\,\nu\,Q\,\frac{R^2\,(L_z-R)}{L_y\,L_z^3} \tag{196}$$

For $Q^{\text{SI}}=50\,\frac{\mu L}{s}$ and $R^{\text{SI}}=1.5\,\mu m$ the force will be approximately $F^{\text{SI}}\approx 0.8675\,nN$. Since in the experiment the channel is not periodic in the direction of the y-axis, the flow is slower close to the channel walls perpendicular to the y-axis, meaning that the true flow velocity in the middle of the channel is larger than the estimation; numerical evaluation of the velocity from eq. (191) results in a slightly larger force of $F^{\text{SI}}\approx 0.9257\,nN$. It is also unclear how accurate the assumption of eq. (184) is for a sphere with flow happening only on one side.

10.1.4 Strategy and Simulation Setup

In order to resolve the particle with radius $R = 0.5\,\mu m$ as a single voxel and still simulate the entire rectangular channel, just over $1\,TB$ of video memory would be required, meaning that simulating the entire channel (illustrated in figure 36) is not an option. Instead, only the neighborhood of the particle is simulated and at the edge of the simulation domain the velocity is set via (artificially introduced) moving bounce-back-boundaries. The simulation box dimensions are defined as

$$L_x = L_y = k \cdot R \tag{197}$$

$$L_z = \left(\frac{k}{2} + 1\right) \cdot R \tag{198}$$

with $k \gg 1$ being a number that is chosen as large as possible and a non-moving boundary at $z = 0$, which means that the simulation box boundaries **do not** necessarily coincide with the channel boundaries. If k is large enough, the simulation box size would ultimately be confined to the channel dimensions in simulation units (not the case here). The particle is placed at the position

$$\vec{x}_0 = \left(\frac{L_x}{2},\, \frac{L_y}{2},\, R+1\right)^T \tag{199}$$

The velocity at the simulation box boundaries other than at $z = 0$ for the same reason as discussed in section 9.2.2 cannot be set to the velocity $\vec{u}_x(y,z)$ of the rectangular-channel Poiseuille flow (eq. (191)), which would artificially constrict the flow and thereby result in a too large force on the sphere (A). Periodic boundaries in x- and y-direction would have the same effect as setting the velocity to $\vec{u}_x(y,z)$.

However here the boundary velocity also cannot be set to the analytic velocity for laminar flow around a sphere (eq. (183)) with the z-dependent Poiseuille flow velocity $\vec{u}_x(y=0,z)$ as input, which would mean that the flow would not be constricted even infinitely far away from the sphere, resulting in a force too small (B). In the experimental setup ultimately the walls of the rectangular channel enforce straight streamlines and these walls are at finite distance away from the sphere.

In order to decrease the possible corridor of force between (A) and (B), the boundaries should be as far away as possible, but the particle also has to be resolved as good as possible, so $R^{\text{sim}} = 16$ for all simulations is a good compromise. The true force will be somewhere in between, meaning both above variants are used to confine the possible corridor and an interpolation of the velocities of both variants will give the best results within this range (C). The interpolation is done for every R individually and the interpolation factors are determined by the volume fraction of the simulated volume and the total volume of the rectangular channel. The interpolation factors for the velocity boundaries at $R^{\text{sim}} = 16$ are

$$w(u_{(\text{A})}) = \{3.80,\ 7.60,\ 11.39,\ 37.98\}\,\%$$

$$w(u_{(\text{B})}) = \{96.20,\ 92.40,\ 88.61,\ 62.02\}\,\%$$

As a fourth boundary condition, (C) is simulated with the addition that the surface on which the particle rests is not flat (D). For the surface roughness it is estimated that the surface features have the length scale of $0.5\,\mu m$ as illustrated in figure 37. In the simulation, the roughness is created by using simplex noise [96], which is a more computational efficient variant of Perlin noise [97].

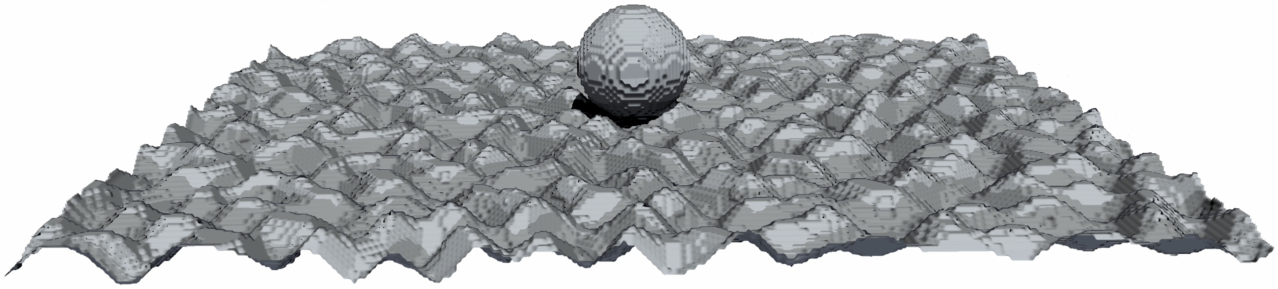


Figure 37: The particle with $R^{\text{SI}} = 1.5\,\mu m$ ($R^{\text{sim}} = 16$) sitting on a rough surface (D) illustrated with ParaView.

Now the plan is to run four simulation rows with

$$R^{\mathrm{SI}} \in \{0.5,\, 1.0,\, 1.5,\, 5.0\}\, \mu m$$

each for a volume flow rate of

$$Q^{\mathrm{SI}} \in \{0.1,\, 1.0,\, 2.0,\, 3.0,\, 4.0,\, 5.0,\, 7.0,\, 10.0,\, 15.0,\, 20.0,\, ...,\, 50.0\}\, \frac{\mu L}{s}$$

for all different boundary definitions ((A), (B), (C), (D)), so a total of 256 simulations. During each simulation row, the velocity u^{sim} in simulation units remains constant while the kinematic shear viscosity ν^{sim} in simulation units is varied. As known from section 9.2.4, u^{sim} should not be too small and ν^{sim} should not vary too much. By setting $\nu^{\mathrm{sim}} = 1$ for the midway flow rate $Q^{\mathrm{SI}} = 25\, \frac{\mu L}{s}$, the corresponding velocity

$$u^{\mathrm{sim}} = \frac{\nu^{\mathrm{sim}}\, R^{\mathrm{SI}}}{\nu^{\mathrm{SI}}\, R^{\mathrm{sim}}} \cdot u_x^{\mathrm{SI}}(y^{\mathrm{SI}} = 0, z^{\mathrm{SI}} = 0,\, L_y^{\mathrm{SI}}, L_z^{\mathrm{SI}},\, Q^{\mathrm{SI}}) \tag{200}$$

is determined, which corresponds to the velocity in simulation units in the channel center for all simulations in one row. These velocities for the different R^{SI} are numerically evaluated to be

$$u^{\mathrm{sim}}\Big|_{y=z=0} = \{0.017949,\, 0.035897,\, 0.053846,\, 0.179486\}$$

Then, the fluid velocity in simulation units that hits the center of the particle is numerically evaluated to be

$$u^{\mathrm{sim}}\Big|_{y=0,\, z=-\frac{L_z^{\mathrm{sim}}}{2}+R^{\mathrm{sim}}} = \{0.000356,\, 0.001419,\, 0.003182,\, 0.034098\} > 0.0003$$

for all Q^{SI}. When during a simulation row Q is varied, the kinematic shear viscosity in simulation units varies between

$$\nu^{\mathrm{sim}} \in [0.5,\, 250.0] \approx 10^0$$

for $Q^{\mathrm{SI}} \in [0.1,\, 50.0]\, \frac{\mu L}{s}$ equally for all R^{SI}. Our Radeon VII GPUs allow for $k = 36$ at $R^{\mathrm{sim}} = 16$ or a box size of (576, 576, 304). That is about 101 million lattice points, occupying all of the 16 *GB* video memory available. At any time, four such simulations are run in parallel on the four GPUs of SMAUG-8, each corresponding to one of the R^{SI}. Due to the additional thickness of the surface for (D), in these simulations only $k = 35$ is possible, making the box a bit smaller and slightly changing the interpolation factors as well.

10.1.5 Results

The simulation results are shown in figure 38, the qualitative force distribution is illustrated in figures 41 and 42 and the typical velocity fields are illustrated in figure 43. The force is assumed to be proportional to the flow rate, which holds true for the smaller particle radii. Only the simulations for $R^{\mathrm{SI}} = 5\, \mu m$ slightly deviate from a linear relation, which is expected due to the higher Reynolds number of $Re \approx 2.18$ for $Q^{\mathrm{SI}} = 50\, \frac{\mu L}{s}$. For a rough surface, the forces are a bit lower overall. $F(Q)$ is fitted based on equation (196), extended by a correction factor C:

$$F = C\, 36\, \pi\, \rho\, \nu\, Q\, \frac{R^2\, (L_z - R)}{L_y\, L_z^3} \tag{201}$$

This factor C itself is weakly dependent on R (because $Re > 0$) as shown in figure 39 and fitted with a parabola:

$$C(R) = \begin{cases} (0.00256 \pm 0.00016) \left(\frac{R}{10^{-6} m}\right)^2 + (0.06962 \pm 0.00095) \frac{R}{10^{-6} m} + (1.50757 \pm 0.00080) & \text{for flat surface} \\ (-0.0220 \pm 0.0074) \left(\frac{R}{10^{-6} m}\right)^2 + (0.241 \pm 0.043) \frac{R}{10^{-6} m} + (1.249 \pm 0.037) & \text{for rough surface} \end{cases} \tag{202}$$

While this fit is very accurate for the flat surface, not much trust should be put into it for the rough surface, where the particle radii are on a similar length scale compared to the surface bumps (illustrated by figure 40); there a lot of variation is present in the force acting on the particle based on where exactly the nearby surface bumps are located.

The qualitative force distribution (figure 41) shows that the force is much larger at the top of the particle than at the bottom. This creates significant torque on the particle, meaning that the force with which the particle pulls on the substrate might – depending on the contact area between particle and substrate – be much larger than the overall sideways force on the particle as displayed in figure 38.

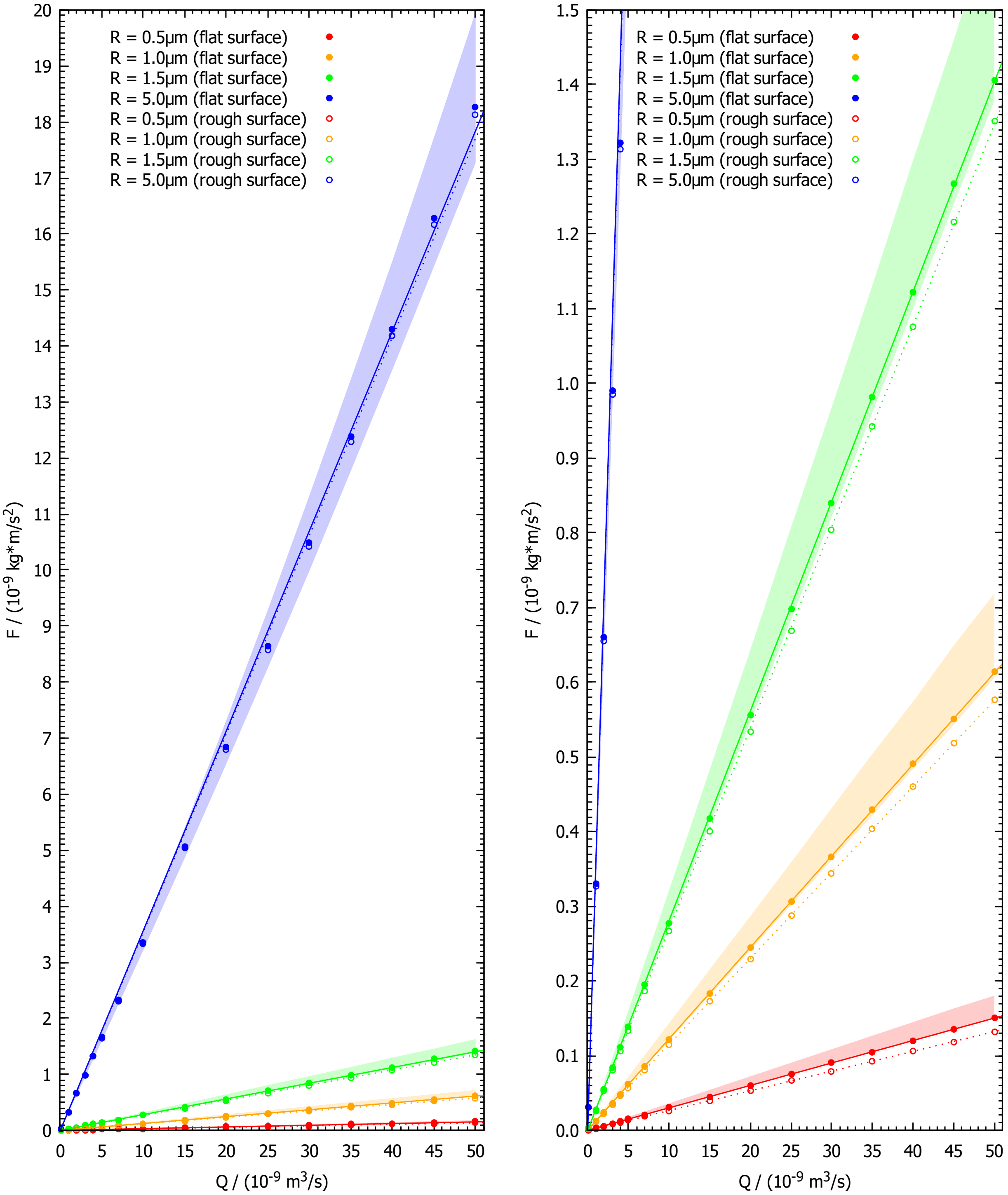


Figure 38: The force $F(Q)$ as a function of the flow rate Q plotted. The filled dots represent the simulations (C) for a spere on a flat surface and the circles (D) for the sphere on a rough surface. The lightly colored range is the corridor between (A) and (B). A linear fit (eq. (201)) is performed on every simulation row, plotted as continuous lines (C) and dotted lines (D). On the right, the plot on the left is shown for a reduced range of F. For smaller R, the force (C) is closer to the lower side of the corridor (B), which is expected as a result of the interpolation factors.

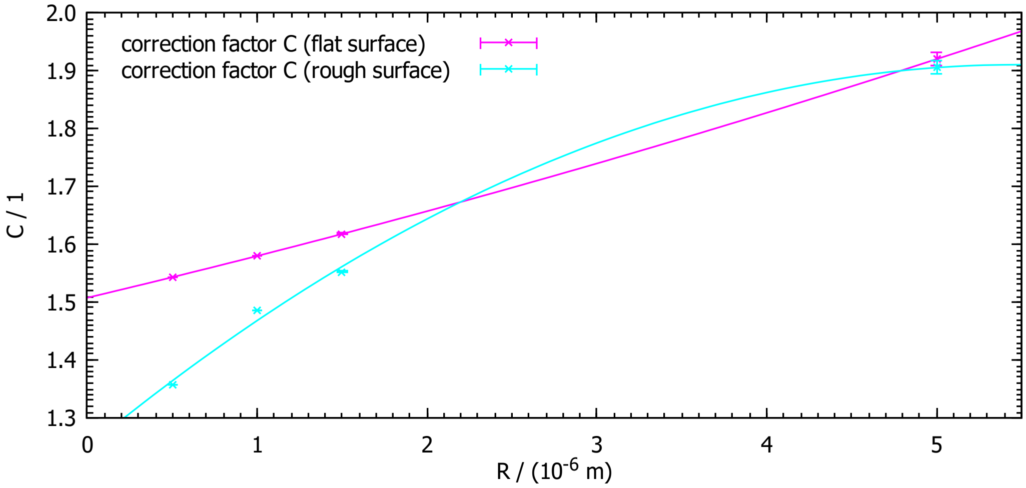


Figure 39: The correction factor C in equation (201) is weakly dependent on R. A parabolic fit (eq. (201) works well for the particle sitting on a flat surface, but for the rough surface the individual points deviate quite a lot. This deviation is explained by the surface features being on a similar length scale compared to the particle for the small particle radii (illustrated in figure 40), meaning for small R it becomes essential to know *where exactly* the surface bumps are.

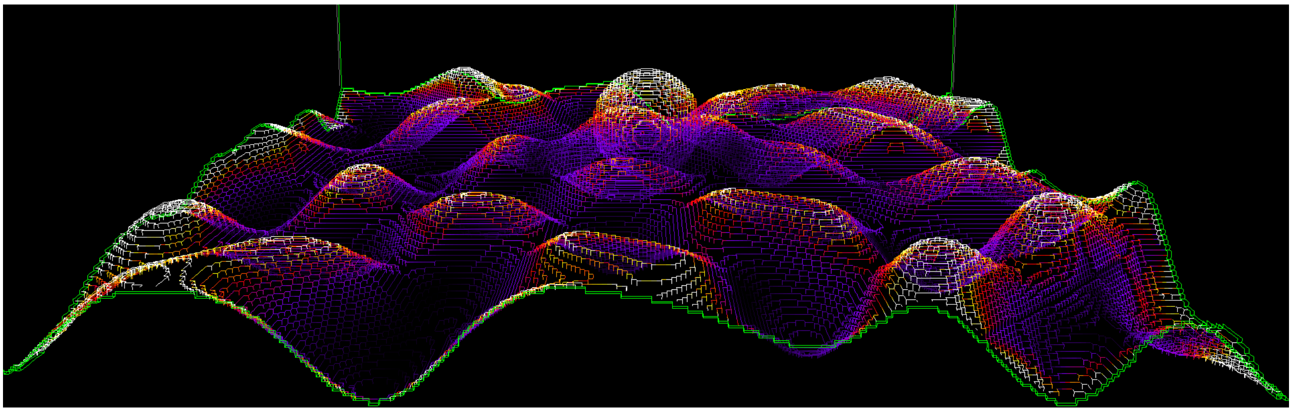


Figure 40: The qualitative force distribution on the smallest particle with $R^{\text{SI}} = 0.5\,\mu m$ ($R^{\text{sim}} = 16$) illustrated with force magnitude coloring for a simulation box size of $k = 16$. The flow direction is left-to-right. The particle diameter is comparable to the length scale of surface features, meaning that the positions of nearby surface bumps have a large impact on the force acting on the particle. Visualization is done with the real-time OpenCL graphics engine of *FluidX3D*.

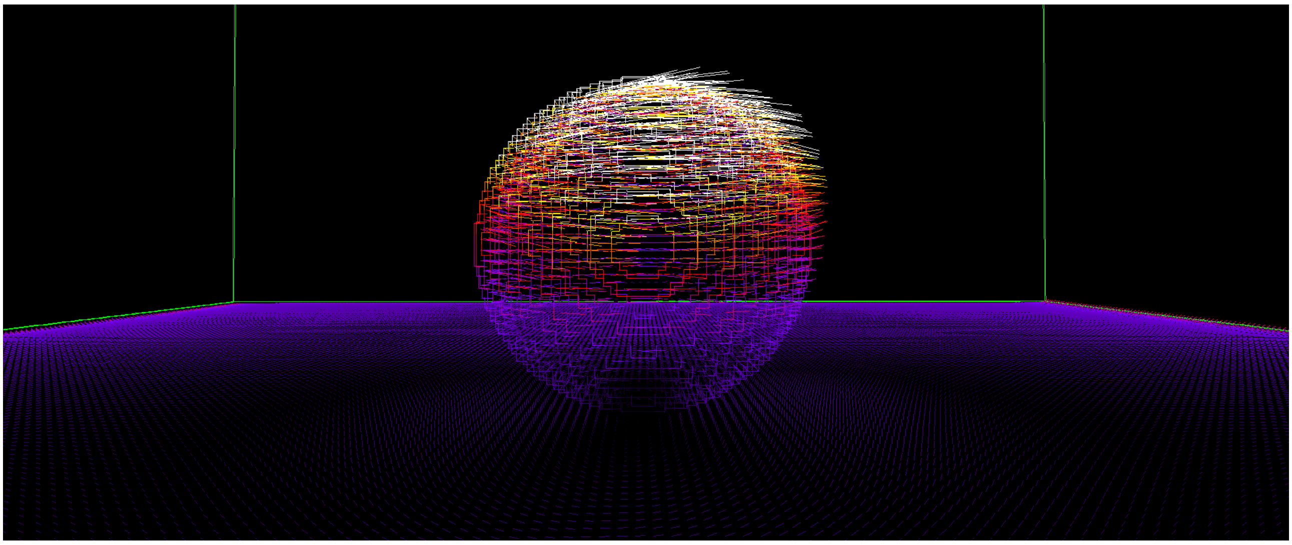


Figure 41: The qualitative force distribution on the particle with $R^{\mathrm{SI}} = 1.5\,\mu m$ ($R^{\mathrm{sim}} = 16$) sitting on a flat surface (C) illustrated with colored force vectors (color represents force magnitude) for a simulation box size of $k = 16$. The flow direction is left-to-right. Visualization is done with the real-time OpenCL graphics engine of *FluidX3D*.

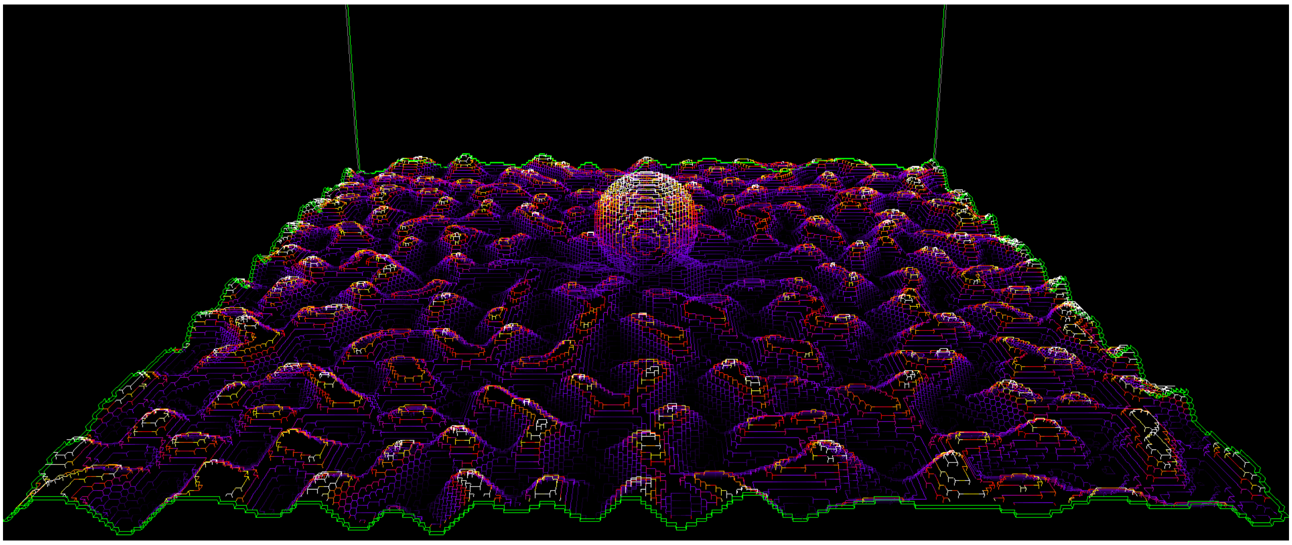


Figure 42: The qualitative force distribution on the particle with $R^{\mathrm{SI}} = 1.5\,\mu m$ ($R^{\mathrm{sim}} = 16$) sitting on a rough surface (D) illustrated with force magnitude coloring for a simulation box size of $k = 16$. The flow direction is left-to-right. Visualization is done with the real-time OpenCL graphics engine of *FluidX3D*.

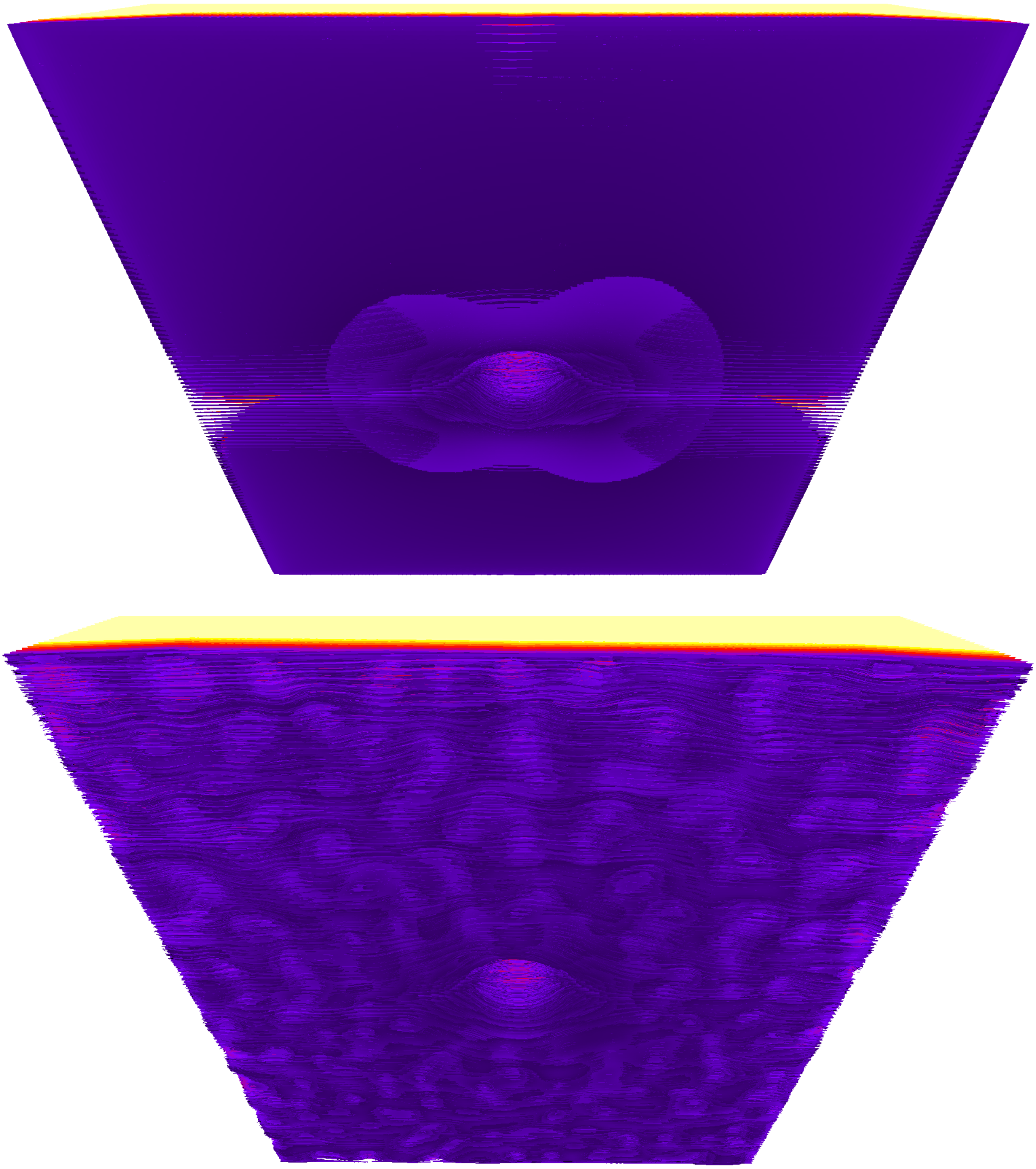


Figure 43: The flow around the particle with $R^{\mathrm{SI}} = 1.5\,\mu m$ ($R^{\mathrm{sim}} = 16$) illustrated with dense colored streamlines (color represents velocity magnitude) for a box size of $k = 16$. The flow direction is left-to-right and the camera looks at the simulation box from down below. The top image shows the velocity field on the particle sitting on a flat surface (C) while the bottom image shows the particle sitting on a rough surface (D). The real-time OpenCL graphics engine of *FluidX3D* here manages to draw 302 million lines per frame at 60 frames per second on a single Titan Xp GPU.

10.2 Oblique Drop Impact

In nature an exactly straight drop impact is more of an exception, yet the straight impact has been the major focus of previous studies in literature. Varying the angle of velocity of the incoming drop greatly changes the dynamics involved in the impact, from a symmetrical cavity and crown over asymmetry up to the incoming drop skidding across the surface for very shallow impact angles. There are countless applications in industry where oblique drop impacts matter, including ultraviolet lithography in semiconductor manufacturing and turbine blade design for use in airplanes. Here an oblique drop impact with an intermediate impact angle is investigated, because for this case there are both experimental high-speed images and previous simulation results for comparison.

10.2.1 Setup

For the oblique drop impact [98, 99] the setup consists of a cubic simulation box filled with fluid in the bottom half with a small droplet initialized just above the surface with downward velocity, not exactly straight downward but at an angle α. The drop upon impact forms an asymmetric cavity and an asymmetric crown which breaks up into small droplets.
In order to stay as close to the experimental setup as possible, these parameters are chosen:

- drop diameter $D^{\text{SI}} = 0.1\,mm$
- fluid density $\rho^{\text{SI}} = 1000\,\frac{kg}{m^3}$
- dynamic viscosity $\mu^{\text{SI}} = 8.36 \cdot 10^{-4}\,\frac{kg}{m\,s}$
- surface tension coefficient $\sigma^{\text{SI}} = 0.072\,\frac{kg}{s^2}$
- gravitational acceleration $g^{\text{SI}} = 9.81\,\frac{m}{s^2}$
- height of the pool $H^{\text{SI}} = 0.5\,mm$
- cubic simulation box with side length $L^{\text{SI}} = 1\,mm$
- Weber number $We = 416.5$
- impact angle $\alpha = 28.5°$

These values result in an impact velocity of $u^{\text{SI}} = 17.32\,\frac{m}{s}$ and a Reynolds number of $Re = 2071$, which is still stable in the simulation. In simulation units, three independent parameters are chosen for the dimensionalization procedure,

- fluid density $\rho^{\text{sim}} = 1$
- impact velocity $u^{\text{sim}} = 0.15$
- simulation box side length $L^{\text{sim}} = 384$

resulting in the remaining quantities in simulation units to be:

- kinematic shear viscosity $\nu^{\text{sim}} = 2.78 \cdot 10^{-3}$
- surface tension $\sigma^{\text{sim}} = 2.07 \cdot 10^{-3}$
- force per volume $f^{\text{sim}} = 1.92 \cdot 10^{-9}$

The simulations are done with D3Q19 and the SRT operator, which somehow is much more stable here than TRT or MRT. The volume force in simulation units is so small that it is most probably completely canceled out in equations (39) and (40) due to numeric loss of significance.

10.2.2 Difficulties arising from the Setup

The impacting drop is very small (less than a millimeter in diameter) and the impact is at very high velocity (several meters per second), so that after parametrization into lattice units the viscosity is very small and on the edge of being a stable simulation. [99] argue that the Reynolds number in this case is so large that it can just be approximated by $Re = 1000$, even though in the experimental setup $Re = 2071$. It seems to be a common practice to do simulations with slightly different parameters than in the experiment in order to get better agreement between the results [99, 100].

Furthermore, if the volume force in simulation units gets too small, it at some point is completely eliminated by numeric loss of significance. Increasing the simulation box size proportionally decreases the volume force, so better spatial resolution makes the effect even worse. However if the simulated time interval is short enough, the volume force has only very small impact on the simulation anyways.

Another issue is when the surface tension coefficient in simulation units becomes too large, which will lead to instability [101].

10.2.3 Simulation Results

The in figure 44 illustrated time frames are $t = \{0.46, 2.33, 8.22, 12.15, 18.9\} \cdot t_i$, where $t_i := \frac{D}{u}$, in lattice units equivalent to $t^{\text{sim}} = \{118, 597, 2105, 3111, 4830\} \cdot \Delta t$. Compute time is less than three minutes on the Nvidia Titan Xp.

In the last few frames, artifacts in the cavity become visible, caused by several simulation parameters being at the limit of the stable range. A full quantitative comparison is not possible because in the original image series no length scale is given and the images all have different scaling.

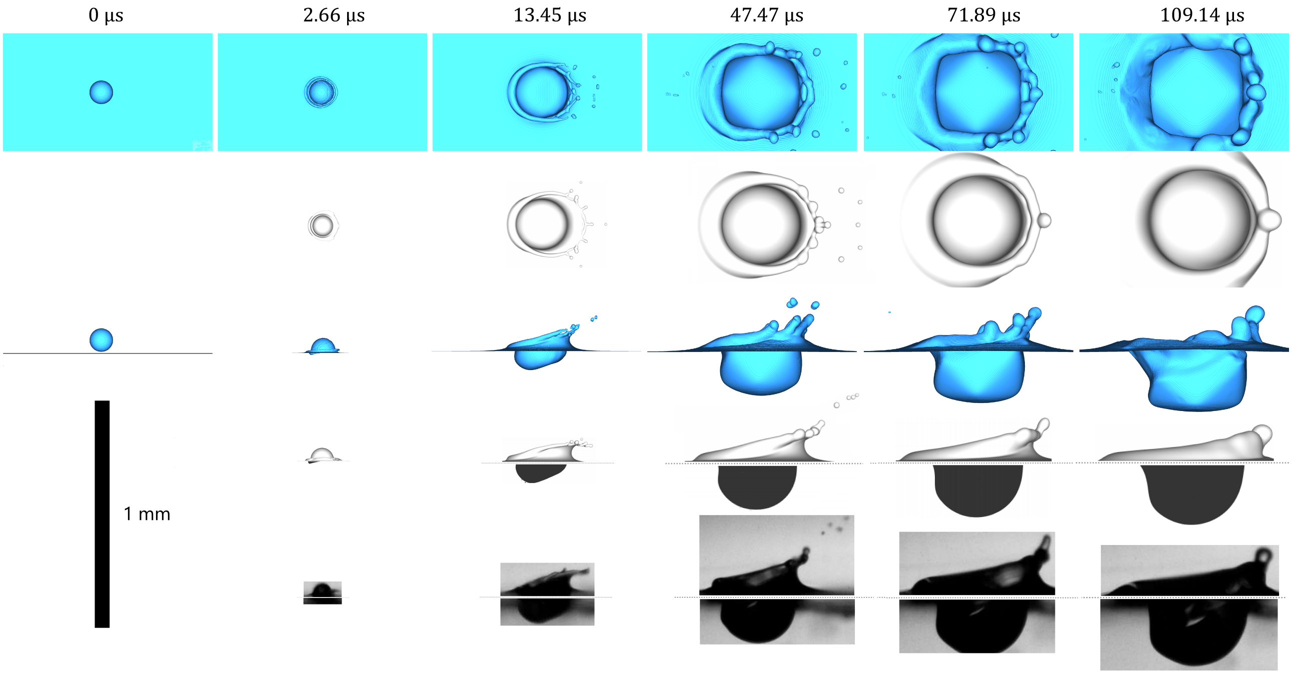


Figure 44: An attempt to recreate the setup from [99], figure 3 (new simulation in blue). The individual images from [99] had to be re-scaled, because the length scale is not kept the same in the original image series. Here, the length scale is kept constant for all images and the diameter of the drop (top left) is $D^{\text{SI}} = 0.1\,mm$.

10.3 Crown Formation by Drop Impact on a shallow Pool

Crown formation is a very fascinating phenomenon occurring at high speed drop impacts, for example in rainfall. The breakdown of the crown releases a large number of droplets into the air and is the main candidate for the process of particle transport from the in liquid suspended phase into the air, be it sea salt aerosol or microplastic particles. This makes the crown formation system interesting to test the capabilities of the free surface simulation. The tiny droplets emerging from the crown have to be sufficiently resolved while the entirety of the crown still has to fit into the simulation box, requiring large amounts of memory.

10.3.1 Setup

The crown splashing setup [102, 103] consists of a small drop impacting a shallow pool of fluid at high speed. For the drop radius only the range $D^{\text{SI}} \in \{2.0,\, 4.2\}\,mm$ is given in [103], so in the simulation the arithmetic mean is chosen. The fluid in the experiment is not pure water but 70 % glycerol in water by weight. The full list of parameters is:

- drop diameter $D^{\text{SI}} = 3.1\,mm$
- height of the pool $H^{\text{SI}} = 0.5\,D^{\text{SI}}$
- fluid density $\rho^{\text{SI}} = 1177.9\,\frac{kg}{m^3}$, assuming $T^{\text{SI}} \approx 25°C$ [104]
- surface tension coefficient $\sigma^{\text{SI}} = 0.0661\,\frac{kg}{s^2}$, assuming $T^{\text{SI}} \approx 25°C$ [105]
- gravitational acceleration $g^{\text{SI}} = 9.81\,\frac{m}{s^2}$
- simulation box dimensions $L_x^{\text{SI}} = L_y^{\text{SI}} = 30\,mm$, $L_z^{\text{SI}} = 25\,mm$
- Reynolds number $Re = 1168$
- Weber number $We = 2010$
- impact angle $\alpha = 0°$

These values result in an impact velocity of $u^{\text{SI}} = 6.03\,\frac{m}{s}$ and a kinematic shear viscosity of $\nu^{\text{SI}} = 1.60 \cdot 10^{-5}\,\frac{m^2}{s}$. In simulation units, three independent parameters are chosen for the dimensionalization procedure,

- fluid density $\rho^{\text{sim}} = 1$
- impact velocity $u^{\text{sim}} = 0.15$
- simulation box side length in x-direction $L_x^{\text{sim}} = 400$

resulting in the remaining quantities in simulation units to be:

- kinematic shear viscosity $\nu^{\text{sim}} = 5.31 \cdot 10^{-3}$
- surface tension $\sigma^{\text{sim}} = 4.63 \cdot 10^{-4}$
- force per volume $f^{\text{sim}} = 3.55 \cdot 10^{-7}$

The simulations are done with D3Q19 and the SRT operator. The volume force in simulation units is very small and might partly cancel out in equations (39) and (40) due to numeric loss of significance.

10.3.2 Simulation Results

The in figure 45 simulated time frames are $t^{\mathrm{SI}} = \{0.3,\, 1.0,\, 3.0,\, 7.5,\, 10.0\}\, ms$, whereby the starting point of the simulation offsetted by $t_0^{\mathrm{SI}} = 0.13\, ms$ in order for the first frame to be in sync with the experiment. An offset in time is fine as long as the intervals between the time frames remain the same. In lattice units these times are equivalent to $t^{\mathrm{sim}} = \{91,\, 466,\, 1539,\, 3952,\, 5292\} \cdot \Delta t$. Compute time is less than three minutes on the Nvidia Titan Xp.

The simulation shows good agreement with the experiment. In the simulation, the crown is more symmetric and later shows slight octagonal artifacts in its shape. Also in the simulation the crown appears to be a bit wider than in the experiment. The height of the crown is in good agreement with the experiment.

Possible discrepancies are caused by the volume force not being sufficiently resolved and possible deviations in the density and surface tension of the fluid as well as the sphere radius.

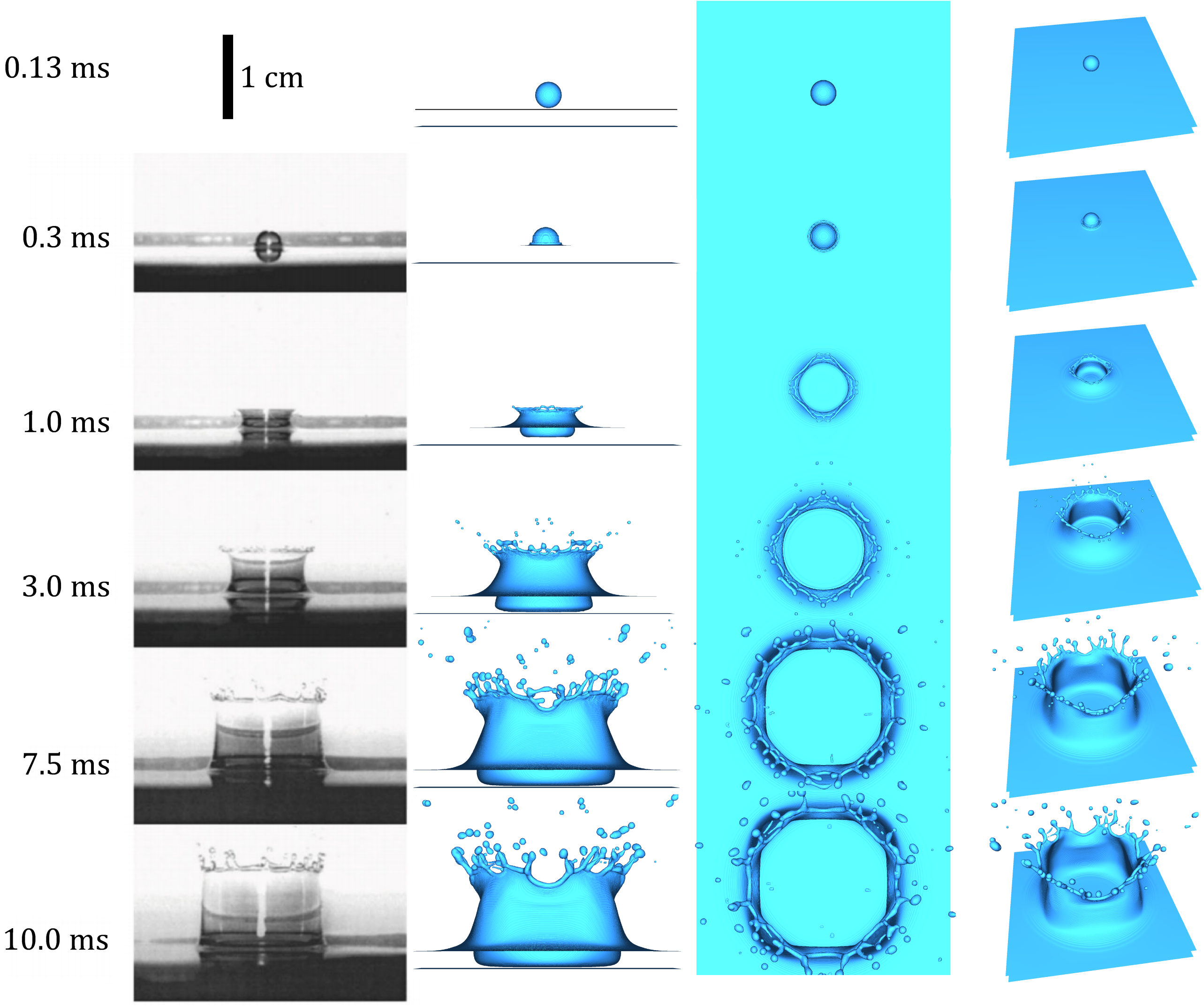


Figure 45: The simulation results are compared to the experiment from [102] as shown in figure 5 in [103]. Since in the original image no scalebars are present, scaling is done by comparing the diameter of the half sphere still visible in the frame for $0.3\, ms$ between experiment and simulation. The simulation shown in side and top view (two center columns) is to scale with the experiment. The rightmost column is not to scale and is just there for better illustration.

10.4 Simulations to demonstrate the Diversity of LBM Use-Cases

In this section, a few simulations are shown qualitatively. This should give an impression on how diversified the use-cases for LBM simulations can be and what variety of physical effects emerge from the comparatively simple algorithm. Detailed quantitative analysis of all of these setups however would be out of the scope of this thesis. Each of the simulations below takes less than half a minute of compute time on the Titan Xp.

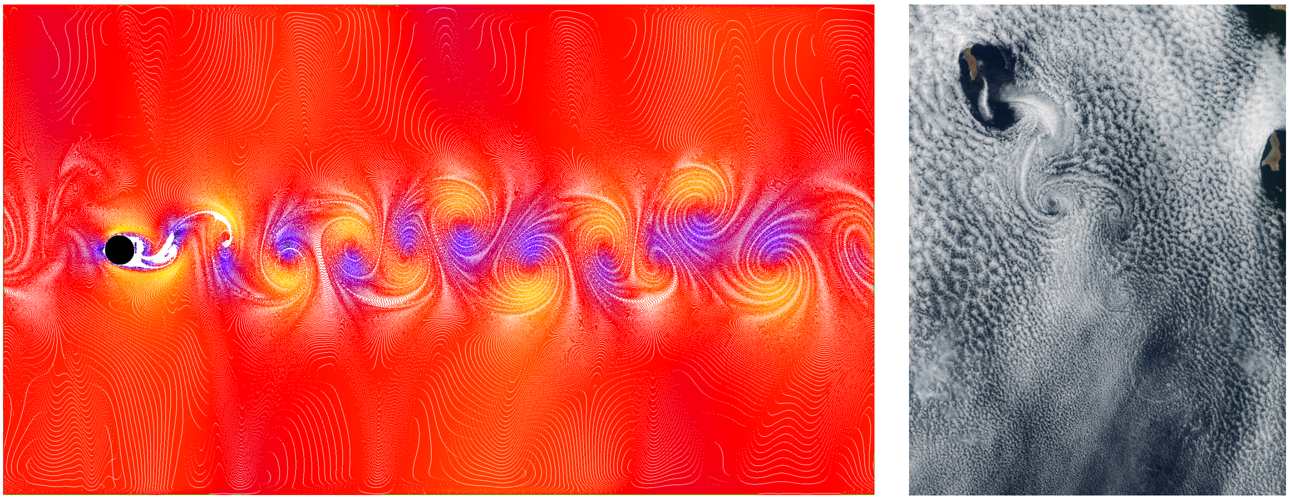


Figure 46: A 2D simulation of a von Kármán vortex street with D2Q9 TRT and $Re = 192$ on a lattice with the dimensions (1920, 1080), visualized with velocity-colored tracer particles (left). This phenomenon is frequently observed by astronauts and weather satellites in cloud patterns of wind flowing over lonely islands in the ocean such as Guadalupe Island in the Pacific (right) [106].

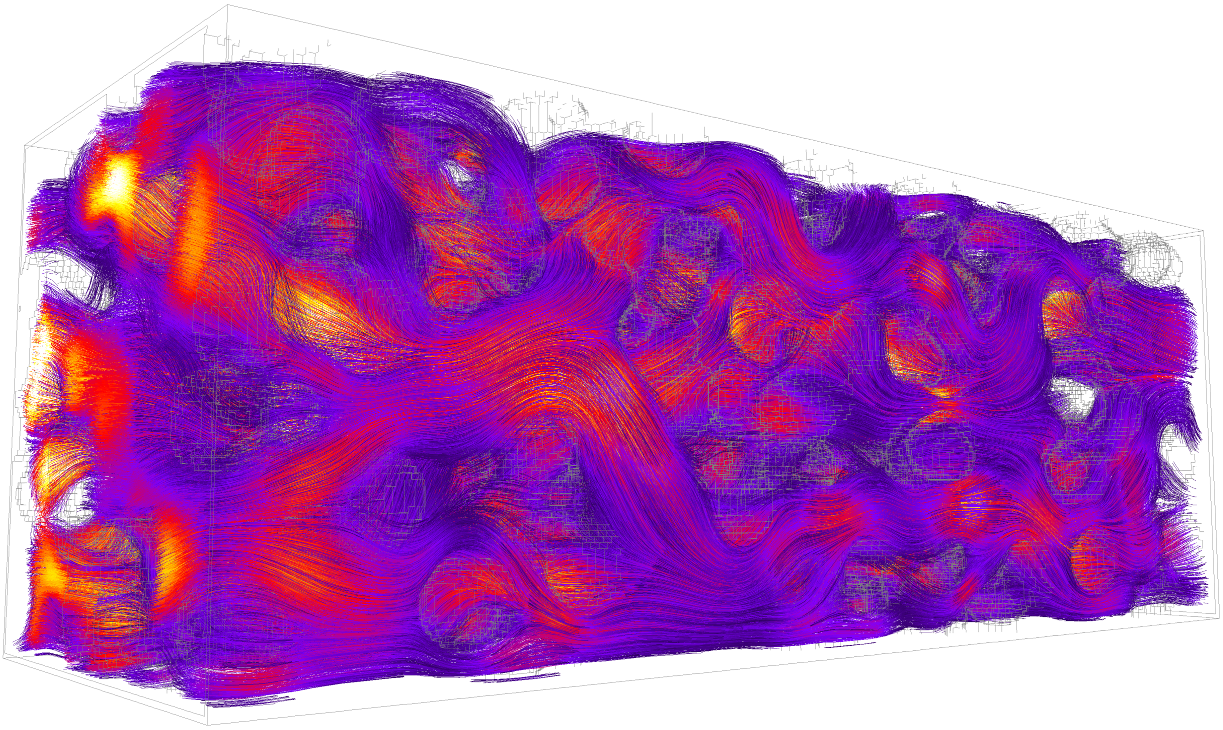


Figure 47: Flow trough a porous medium with $Re = 14$ simulated in a periodic simulation box with dimensions (96, 256, 96) visualized with streamlines (coloring visualizes velocity magnitude).

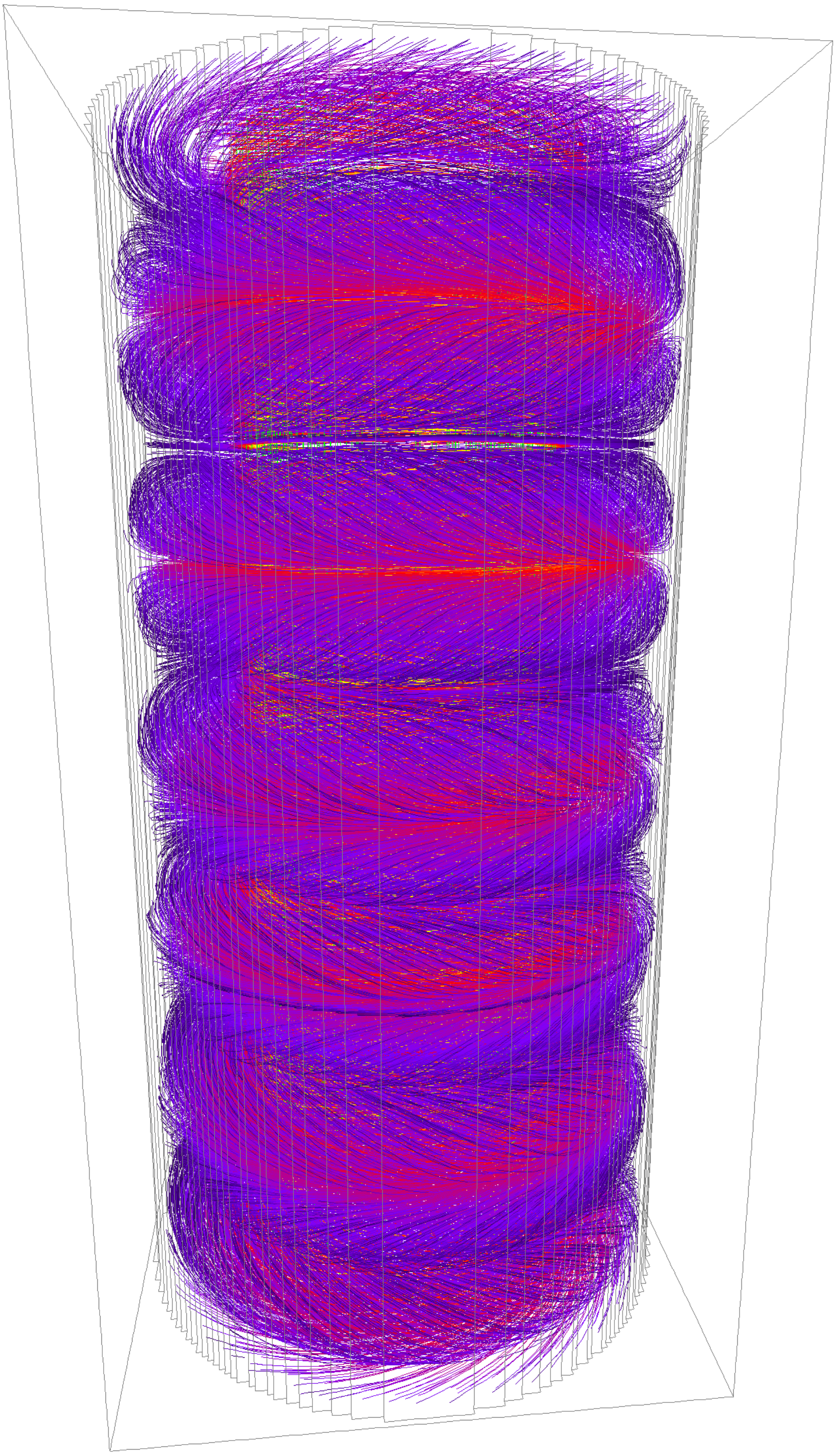


Figure 48: Taylor-Couette flow in a periodic cylinder at $Re = 288$. In the center of the cylinder there is a rotating rod with half of the diameter of the outer cylinder in the form of moving-bounce-back boundaries. The simulation box dimensions are (96, 96, 192). Very quickly after simulation startup, the distinctive convection bands are forming, here visualized with streamlines with the coloring indicating velocity magnitude.

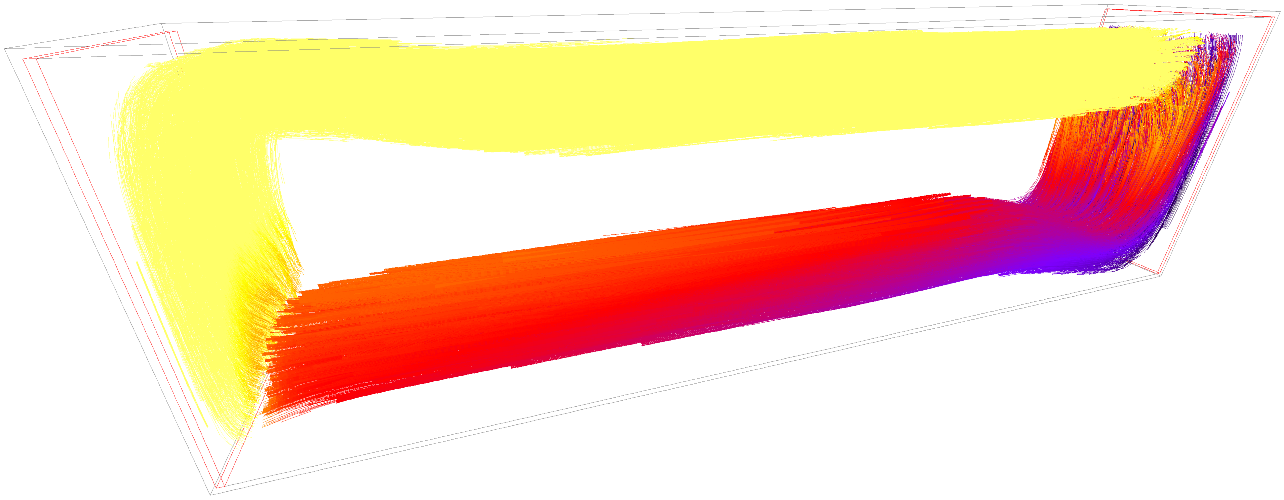


Figure 49: A simple setup for demonstrating natural thermal convection. On the left (hot) and right (cold) sides of the simulation box, temperature equilibrium boundaries enforce a temperature gradient. With gravity pointing downward, thermal convection is inevitable, visualized with streamlines, whereby the coloring indicates temperature. The simulation box dimensions are (32, 196, 60) and the parameters are $Re \approx 160$, $Fr \approx 0.559$, $\alpha = \frac{1}{12}$ and $\beta = 1$.

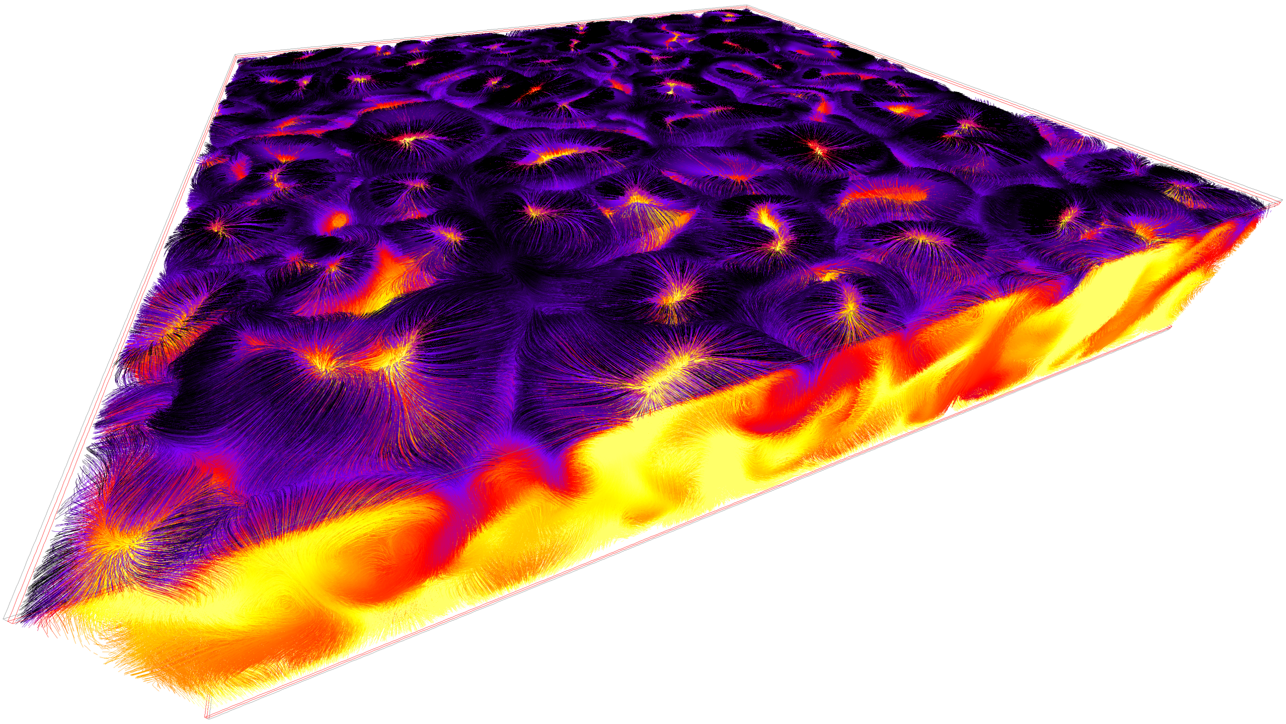


Figure 50: Between two vertical plates, a vertical temperature gradient is created, whereby the bottom plate is hot and the top plate is cold. The emerging dynamics are quite fascinating and known as Rayleigh-Bénard convection. Distinctive convection cells are dynamically forming and colliding. The flow is visualized by streamlines, whereby coloring indicates local temperature. This image consists of about 150 million individual lines. The simulation box dimensions are (384, 384, 64) and the parameters are $Re \approx 320$, $Fr \approx 0.395$, $\alpha = \frac{1}{12}$ and $\beta = 1$.

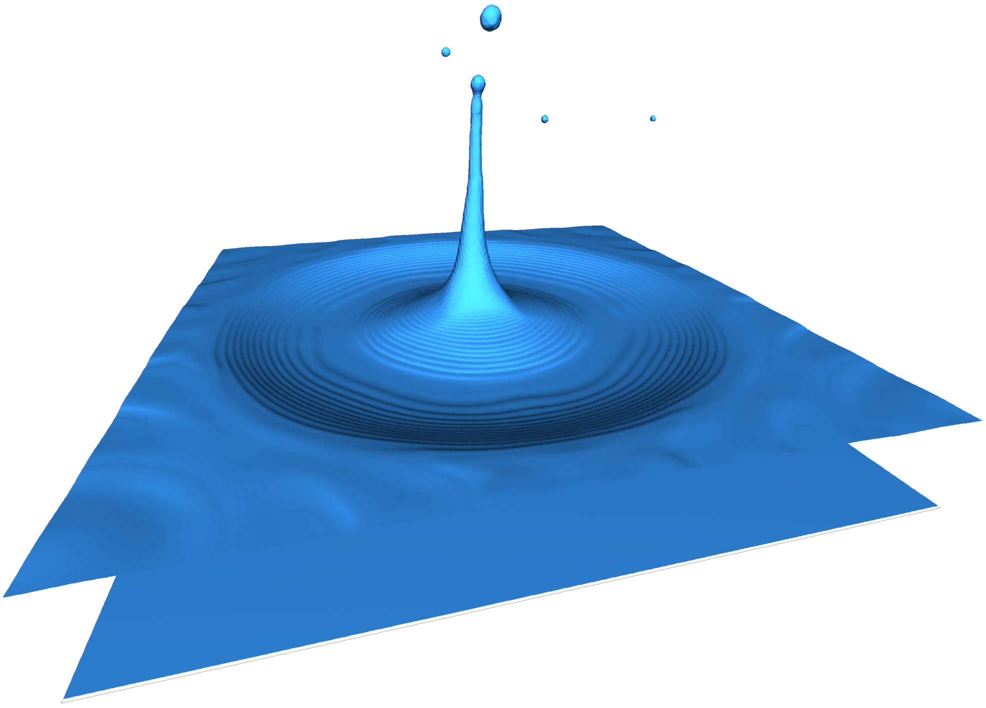


Figure 51: Simulation of a drop impact on a shallow pool with a jet coming out. The simulation box dimensions are (320, 320, 384) and the simulation parameters are $Re = 400$, $We = 320$ and $Fr = 2.236$.

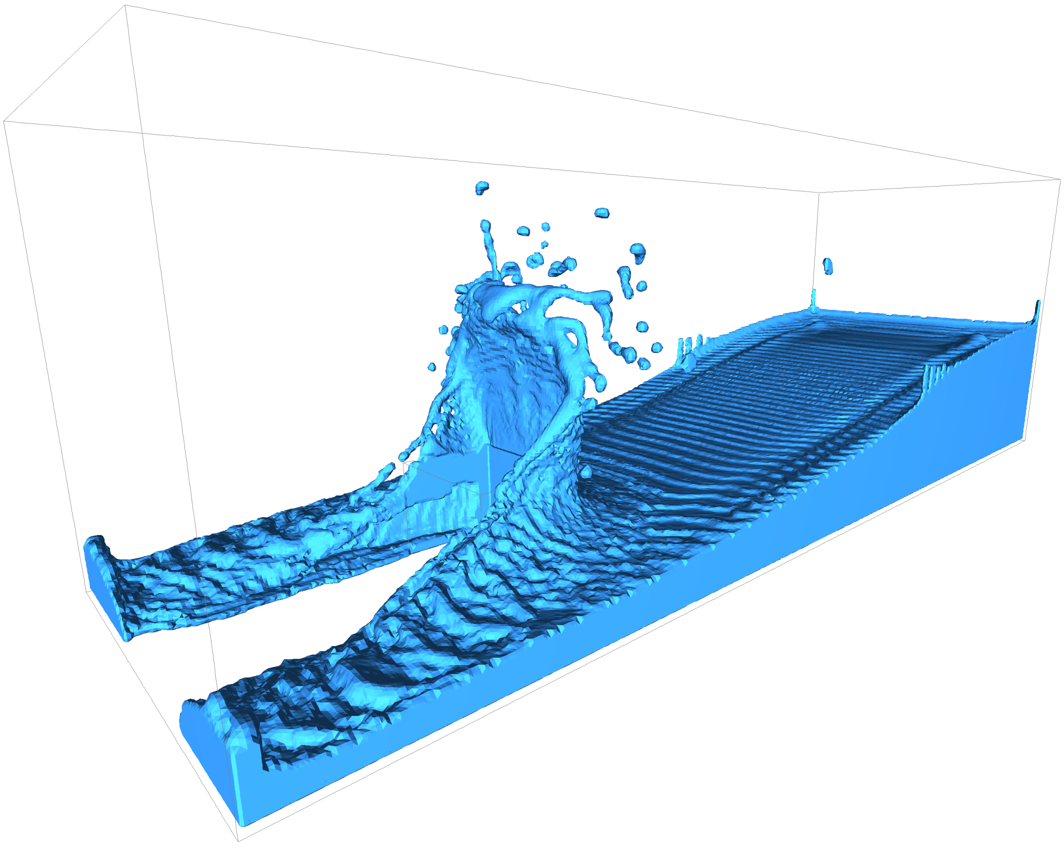


Figure 52: Simulation of a breaking dam using VoF without surface tension. A cuboid obstacle redirects the flow upwards and to the sides. The simulation box dimensions are (128, 384, 128) and the simulation parameters are $Re \approx 1067$ and $Fr \approx 3.953$.

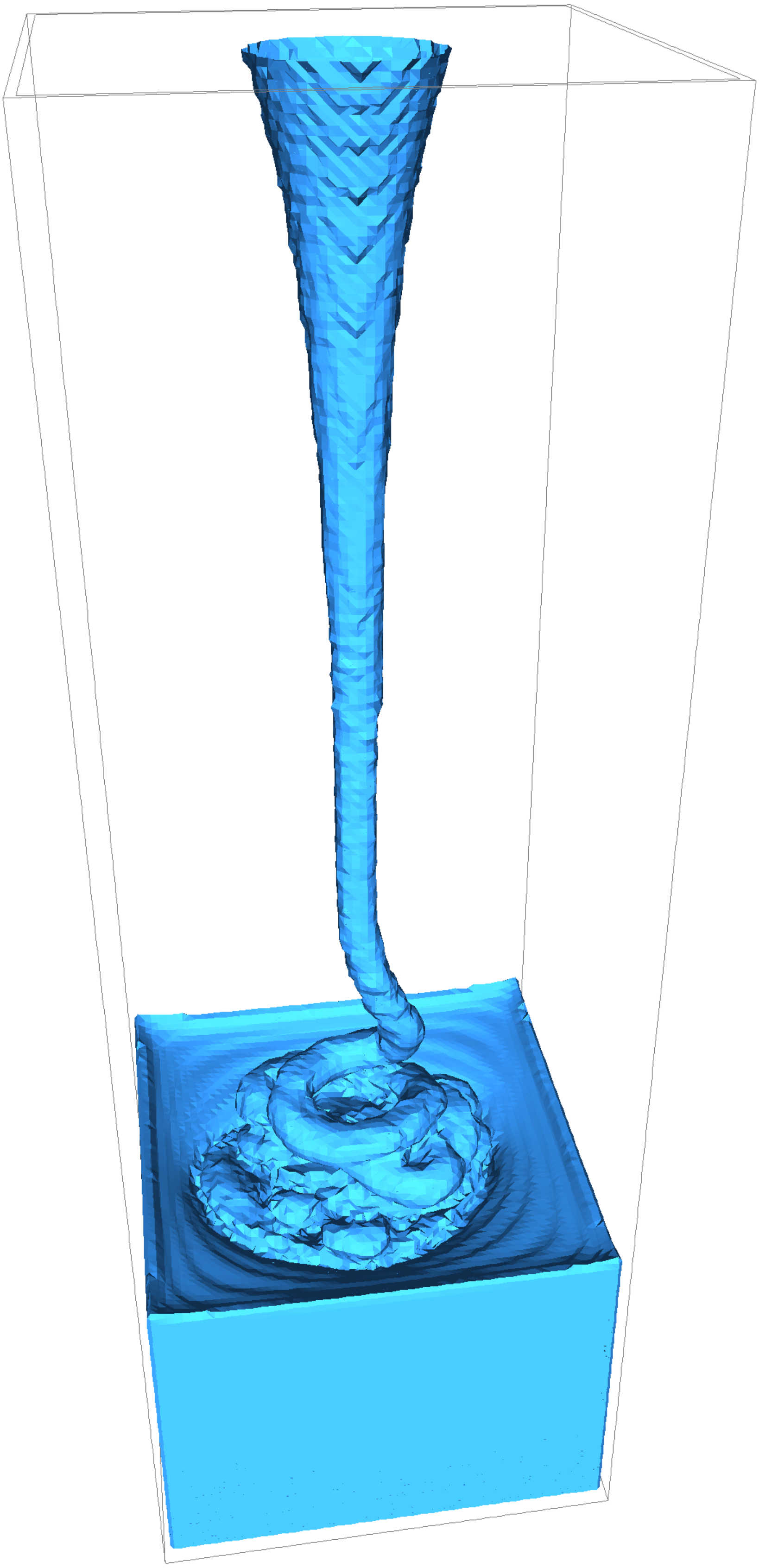


Figure 53: When a thin stream of viscous fluid impacts a surface, it coils up. This phenomenon is called honey coiling [107, 108]. No surface tension is present here and $Re \approx 0.2$ and $Fr \approx 1$. The simulation box dimensions are (64, 64, 192).

11 Outlook: Microplastic Transport Mechanisms at the Water-Air Interface

In the last decades, microplastics have become a worldwide contaminant and can nowadays even be found in the most remote places on earth. Microplastics in the food chain are a hazard for many living organisms, including humans. For this reason the DFG has decided to fund research on microplastics in a variety of different fields as illustrated in figure 54. Besides gaining fundamental understanding of the transport processes involved in spreading microplastics everywhere, also the process and degree of biological breakdown of both regular and 'bio-degradable' polymers and the influence of microplastics on living organisms is of great interest.

A combination of the Volume-of-Fluid and the immersed-boundary method will be examined in the future for simulating the transport mechanisms of how microplastic particles in waters can get from the in liquid suspended state into the air and then be transported across vast distances by wind.

MIKROPLASTIK
Verständnis der Mechanismen und Prozesse
der biologischen Effekte,
des Transports und der Bildung:
Von Modell- zu komplexen Systemen
als
Grundlage neuer Lösungsansätze
SFB
MIKROPLASTIK

Figure 54: Poster of the SFB 1357 Microplastics, in which the biological effects and transport and formation processes of microplastics are researched [109].

12 Conclusions

This work has considerably reduced the time requirements for running a CFD simulation, from days of computation down to minutes. In many cases, setting up the simulation in the first place now takes more time than running the simulation itself. A part of this speedup goes back to the excellent efficiency of the *FluidX3D* implementation and another part to choosing the right hardware platform to run the simulations on, which clearly is the GPU.

Although the Volume-of-Fluid model for simulating free surfaces in this work – aside from a better mass conservation mechanism – physically remains the same as in other state-of-the-art implementations, it has been made considerably more computationally efficient, to the point where a drop impact with the following crown formation is calculated in a matter of minutes on a single GPU. This efficiency was achieved not only by integrating VoF into LBM in a fully parallelized manner, which proved especially difficult due to the many data dependencies, but also by elaborating the analytic solution for the plane-cube intersection problem, which is one of the building blocks of the curvature calculation procedure required for including surface tension effects.

While performance and implementation efficiency have been evaluated with the roofline model, functionality of the base LBM implementation has been validated with both Poiseuille flow in a cylindrical channel and laminar Stokes flow past a sphere.
After separate validation of mass conservation in VoF across a large time period with and without surface tension and the curvature calculation accuracy for spheres of various radii as the basis for surface tension, VoF in combination with surface tension has been validated on the Plateau-Rayleigh instability of an undulated cylinder, where it could replicate the theoretically predicted stability behavior rather well.

As an application for the base LBM implementation, simulations of a microplastic particle attached to the wall of a rectangular microchannel were performed in order to predict the force acting on the particle, so that the experimental physicists at the Experimentalphysik I have reference values to compare their results to.
The VoF model has been used to replicate two drop impact experiments, one of them for an oblique impact of a tiny droplet at very high speed and the second one for the crown formation when a small drop impacts a shallow pool. Both simulations show very good agreement with the experiments, aside from some artifacts caused by a few simulation parameters getting close to the edge of their valid ranges after parametrization from SI-units to dimensionless lattice units.

While free surface simulations have been the main focus of this work, especially also to replicate some experiments, LBM should not be reduced to one particular use-case. LBM has been validated here to be capable of solving a large variety physical systems, from very low Reynolds numbers in the Stokes limit ($Re \approx 10^{-2}$) for microfluidics up to large high Reynolds numbers ($Re \approx 10^{5}$) for highly turbulent flows such as many free surface phenomena or turbulent natural convection. Applications of LBM range from astrophysics across fundamental physics to engineering and should not be understated.

13 References

[1] Robert R Schaller. “Moore’s law: past, present and future”. In: *IEEE spectrum* 34.6 (1997), pp. 52–59.

[2] David Blythe. “Rise of the graphics processor”. In: *Proceedings of the IEEE* 96.5 (2008), pp. 761–778.

[3] David Luebke et al. “GPGPU: general-purpose computation on graphics hardware”. In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM. 2006, p. 208.

[4] Timm Krüger et al. “The lattice Boltzmann method”. In: *Springer International Publishing* 10 (2017), pp. 978–3.

[5] Sydney Chapman, Thomas George Cowling, and David Burnett. *The mathematical theory of non-uniform gases: an account of the kinetic theory of viscosity, thermal conduction and diffusion in gases*. Cambridge university press, 1990.

[6] Acep Purqon et al. “Accuracy and Numerical Stabilty Analysis of Lattice Boltzmann Method with Multiple Relaxation Time for Incompressible Flows”. In: *Journal of Physics: Conference Series*. Vol. 877. 1. IOP Publishing. 2017, p. 012035.

[7] Zheng Li, Mo Yang, and Yuwen Zhang. “Lattice Boltzmann method simulation of 3-D natural convection with double MRT model”. In: *International Journal of Heat and Mass Transfer* 94 (2016), pp. 222–238.

[8] Zhaoli Guo and Chang Shu. *Lattice Boltzmann method and its applications in engineering*. Vol. 3. World Scientific, 2013.

[9] Shimpei Saito, Yutaka Abe, and Kazuya Koyama. “Lattice Boltzmann modeling and simulation of liquid jet breakup”. In: *Physical Review E* 96.1 (2017), p. 013317.

[10] Salvador Izquierdo, Paula Martínez-Lera, and Norberto Fueyo. “Analysis of open boundary effects in unsteady lattice Boltzmann simulations”. In: *Computers & Mathematics with Applications* 58.5 (2009), pp. 914–921.

[11] Zhaoli Guo, Chuguang Zheng, and Baochang Shi. “Discrete lattice effects on the forcing term in the lattice Boltzmann method”. In: *Physical Review E* 65.4 (2002), p. 046308.

[12] Xiongwei Cui et al. “A Coupled Two-relaxation-time Lattice Boltzmann-Volume Penalization method for Flows Past Obstacles”. In: *arXiv preprint arXiv:1901.08766* (2019).

[13] Anthony JC Ladd. “Numerical simulations of particulate suspensions via a discretized Boltzmann equation. Part 2. Numerical results”. In: *Journal of fluid mechanics* 271 (1994), pp. 311–339.

[14] Alfonso Caiazzo and Michael Junk. “Boundary forces in lattice Boltzmann: Analysis of momentum exchange algorithm”. In: *Computers & Mathematics with Applications* 55.7 (2008), pp. 1415–1423.

[15] AA Mohamad and A Kuzmin. “A critical evaluation of force term in lattice Boltzmann method, natural convection problem”. In: *International Journal of Heat and Mass Transfer* 53.5-6 (2010), pp. 990–996.

[16] Timm Krüger. “Introduction to the immersed boundary method”. In: *LBM Workshop, Edmonton*. 2011.

[17] Paul Bourke. “Interpolation methods”. In: *Miscellaneous: projection, modelling, rendering*. 1 (1999).

[18] Anca Hamuraru. *Atomic operations for floats in OpenCL – improved*. 2016. URL: https://streamhpc.com/blog/2016-02-09/atomic-operations-for-floats-in-opencl-improved/ (visited on 11/15/2019).

[19] Xiaowen Shan and Hudong Chen. “Lattice Boltzmann model for simulating flows with multiple phases and components”. In: *Physical Review E* 47.3 (1993), p. 1815.

[20] Amin Rahmani et al. “Evaluation of Shan-Chen Lattice Boltzmann model ability on simulation of multiphase and multicomponent flows”. In: *Conf. Semnan. Ac. Ir.* Vol. 19. 2014.

[21] Lattice Boltzmann Research Group. *OpenLB Performance*. 2019. URL: https://www.openlb.net/performance/ (visited on 11/27/2019).

[22] Pawsey Computing Centre. *Magnus petascale supercomputer*. 2019. URL: https://pawsey.org.au/systems/magnus/ (visited on 12/05/2019).

[23] Intel Corporation. *Intel Xeon E5-2690 v3 Processor*. 2019. URL: https://ark.intel.com/content/www/us/en/ark/products/81713/intel-xeon-processor-e5-2690-v3-30m-cache-2-60-ghz.html (visited on 12/05/2019).

[24] Intel Corporation. *Intel Xeon Platinum 9282 Processor*. 2019. URL: https://ark.intel.com/content/www/us/en/ark/products/194146/intel-xeon-platinum-9282-processor-77m-cache-2-60-ghz.html (visited on 11/19/2019).

[25] NVIDIA Corporation. *NVIDIA TESLA V100 GPU ACCELERATOR*. 2019. URL: https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf (visited on 12/05/2019).

[26] Inc Advanced Micro Devices. *AMD Radeon VII Graphics Card*. 2019. URL: https://www.amd.com/en/products/graphics/amd-radeon-vii (visited on 11/19/2019).

[27] The Khronos Group Inc. *OpenCL Restrictions*. 2009. URL: https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/restrictions.html (visited on 11/27/2019).

[28] WikiChip. *Core i7-8700K – Intel*. 2019. URL: https://en.wikichip.org/w/index.php?title=intel/core_i7/i7-8700k&oldid=92154 (visited on 11/13/2019).

[29] NVIDIA Corporation. *NVIDIA GeForce GTX 1080 Whitepaper*. 2016. URL: https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf (visited on 11/13/2019).

[30] NVIDIA Corporation. *NVIDIA Tesla P100 Whitepaper*. 2016. URL: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf (visited on 11/13/2019).

[31] Shucai Xiao and Wu-chun Feng. "Inter-block GPU communication via fast barrier synchronization". In: *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE. 2010, pp. 1–12.

[32] Khronos Group. *OpenCL API 1.2 Reference Guide*. 2011. URL: https://www.khronos.org/files/opencl-1-2-quick-reference-card.pdf (visited on 10/21/2019).

[33] NVIDIA Corporation. "OpenCL Programming Guide for the CUDA Architecture, Version 4.2". In: *NVIDIA Corporation* (2012).

[34] Christian Obrecht et al. "A new approach to the lattice Boltzmann method for graphics processing units". In: *Computers & Mathematics with Applications* 61.12 (2011), pp. 3628–3638.

[35] NVIDIA Corporation. *OpenCL Best Practices Guide*. 2010. URL: https://www.cs.cmu.edu/afs/cs/academic/class/15668-s11/www/cuda-doc/OpenCL_Best_Practices_Guide.pdf (visited on 10/21/2019).

[36] Peng Wang (NVIDIA). *OpenCL Optimization*. 2009. URL: https://www.nvidia.com/content/GTC/documents/1068_GTC09.pdf (visited on 10/21/2019).

[37] Aaftab Munshi et al. *OpenCL programming guide*. Pearson Education, 2011.

[38] NVIDIA Corporation. *Parallel Thread Execution ISA Version 6.4*. 2019. URL: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html (visited on 10/25/2019).

[39] NVIDIA Corporation. *NVIDIA QUADRO RTX 8000*. 2019. URL: https://www.nvidia.com/en-us/design-visualization/quadro/rtx-8000/ (visited on 11/27/2019).

[40] Fabian Häusl. *MPI-based multi-GPU extension of the Lattice Boltzmann Method*. 2019.

[41] Markus Wittmann. "Hardware-effiziente, hochparallele Implementierungen von Lattice-Boltzmann-Verfahren für komplexe Geometrien". In: (2016).

[42] Nicolas Delbosc et al. "Optimized implementation of the Lattice Boltzmann Method on a graphics processing unit towards real-time fluid simulation". In: *Computers & Mathematics with Applications* 67.2 (2014), pp. 462–475.

[43] Gregory Herschlag et al. "Gpu data access on complex geometries for d3q19 lattice boltzmann method". In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2018, pp. 825–834.

[44] Mark J Mawson and Alistair J Revell. "Memory transfer optimization for a lattice Boltzmann solver on Kepler architecture nVidia GPUs". In: *Computer Physics Communications* 185.10 (2014), pp. 2566–2574.

[45] Tim Schroeder (NVIDIA). *Memory Bandwidth Limited Kernels*. 2011. URL: http://developer.download.nvidia.com/CUDA/training/bandwidthlimitedkernels_webinar.pdf (visited on 10/22/2019).

[46] Frédéric Kuznik et al. “LBM based flow simulation using GPU computing processor”. In: *Computers & Mathematics with Applications* 59.7 (2010), pp. 2380–2392.

[47] Martin Geier and Martin Schönherr. “Esoteric twist: an efficient in-place streaming algorithmus for the lattice Boltzmann method on massively parallel hardware”. In: *Computation* 5.2 (2017), p. 19.

[48] Peter Bailey et al. “Accelerating lattice Boltzmann fluid flow simulations using graphics processors”. In: *2009 International Conference on Parallel Processing*. IEEE. 2009, pp. 550–557.

[49] Markus Mohrhard et al. “Auto-vectorization friendly parallel lattice Boltzmann streaming scheme for direct addressing”. In: *Computers & Fluids* 181 (2019), pp. 1–7.

[50] Markus Wittmann et al. “Comparison of different propagation steps for lattice Boltzmann methods”. In: *Computers & Mathematics with Applications* 65.6 (2013), pp. 924–935.

[51] Keijo Mattila et al. “An efficient swap algorithm for the lattice Boltzmann method”. In: *Computer Physics Communications* 176.3 (2007), pp. 200–210.

[52] J Latt. “How to implement your DdQq dynamics with only q variables per node (instead of 2q), Technical Report”. In: (2007).

[53] Thomas Pohl et al. “Optimization and profiling of the cache performance of parallel lattice Boltzmann codes”. In: *Parallel Processing Letters* 13.04 (2003), pp. 549–560.

[54] Monica D Lam, Edward E Rothberg, and Michael E Wolf. “The cache performance and optimizations of blocked algorithms”. In: *ACM SIGARCH Computer Architecture News*. Vol. 19. 2. ACM. 1991, pp. 63–74.

[55] Institute for Computational Physics, Universität Stuttgart. *ESPResSo User’s Guide*. `http://espressomd.org/wordpress/wp-content/uploads/2016/07/ug_07_2016.pdf`. Accessed: 2018-06-15. 2016.

[56] J Latt and JM Krause. “OpenLB user guide”. In: *Institute of Mechanical Engineering, Ecole Polytechnique Federale de Lausanne (EPFL)* (2008).

[57] Vincent Heuveline and Jonas Latt. “The OpenLB project: an open source and object oriented implementation of lattice Boltzmann methods”. In: *International Journal of Modern Physics C* 18.04 (2007), pp. 627–634.

[58] Vincent Heuveline and Mathias J Krause. “OpenLB: towards an efficient parallel open source library for lattice Boltzmann fluid flow simulations”. In: *International Workshop on State-of-the-Art in Scientific and Parallel Computing. PARA*. Vol. 9. 2010.

[59] Aaftab Munshi. *The OpenCL specification version: 1.2 document revision: 19*. 2012.

[60] Kamran Karimi, Neil G Dickson, and Firas Hamze. “A performance comparison of CUDA and OpenCL”. In: *arXiv preprint arXiv:1005.2581* (2010).

[61] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. “A comprehensive performance comparison of CUDA and OpenCL”. In: *2011 International Conference on Parallel Processing*. IEEE. 2011, pp. 216–225.

[62] Salman Zaidi. “Performance of OpenCL in MultiCore Processors”. In: (2017).

[63] Intel Corporation. *OpenCL Developer Guide for Intel Core and Intel Xeon Processors*. 2018. URL: `https://software.intel.com/en-us/iocl-tec-opg-vectorization-simd-processing-within-a-work-group` (visited on 06/30/2019).

[64] Samuel Williams, Andrew Waterman, and David Patterson. *Roofline: An insightful visual performance model for floating-point programs and multicore architectures*. Tech. rep. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2009.

[65] NVIDIA Corporation. *Tuning CUDA Applications for Pascal*. 2019. URL: `https://docs.nvidia.com/cuda/pascal-tuning-guide/index.html#sm-occupancy` (visited on 11/18/2019).

[66] NVIDIA Corporation. *Tuning CUDA Applications for Volta*. 2019. URL: `https://docs.nvidia.com/cuda/volta-tuning-guide/index.html#sm-occupancy` (visited on 11/18/2019).

[67] Simon Bogner, Ulrich Rüde, and Jens Harting. “Curvature estimation from a volume-of-fluid indicator function for the simulation of surface tension and wetting with a free-surface lattice Boltzmann method”. In: *Physical Review E* 93.4 (2016), p. 043302.

[68] Carolin Körner et al. “Lattice Boltzmann model for free surface flow for modeling foaming”. In: *Journal of Statistical Physics* 121.1-2 (2005), pp. 179–196.

[69] Nils Thürey, C Körner, and U Rüde. “Interactive free surface fluids with the lattice Boltzmann method”. In: *Technical Report05-4. University of Erlangen-Nuremberg, Germany* (2005).

[70] Thomas Pohl. *High performance simulation of free surface flows using the lattice Boltzmann method.* Verlag Dr. Hut, 2008.

[71] Martin Schreiber and DTMP Neumann. “GPU based simulation and visualization of fluids with free surfaces”. PhD thesis. Diploma Thesis, Technische Universität München, 2010.

[72] Andrew N Pressley. *Elementary differential geometry.* Springer Science & Business Media, 2010.

[73] Elsa Abbena, Simon Salamon, and Alfred Gray. *Modern differential geometry of curves and surfaces with Mathematica.* Chapman and Hall/CRC, 2017.

[74] Jingyi Yu et al. “Focal surfaces of discrete geometry”. In: *ACM International Conference Proceeding Series.* Vol. 257. 2007, pp. 23–32.

[75] Zvi Har’el. “Curvature of curves and surfaces–a parabolic approach”. In: *Department of Mathematics, Technion–Israel Institute of Technology* (1995).

[76] Yan-Bin Jia. “Gaussian and Mean Curvatures”. In: (2018).

[77] Stéphane Popinet. “An accurate adaptive solver for surface-tension-driven interfacial flows”. In: *Journal of Computational Physics* 228.16 (2009), pp. 5838–5866.

[78] BJ Parker and DL Youngs. *Two and three dimensional Eulerian simulation of fluid flow with material interfaces.* Atomic Weapons Establishment, 1992.

[79] David Eberly. “Least squares fitting of data”. In: *Chapel Hill, NC: Magic Software* (2000).

[80] Paul Bourke. *Polygonising a scalar field.* 1994.

[81] David L Youngs. “Time-dependent multi-material flow with large fluid distortion”. In: *Numerical methods for fluid dynamics* (1982).

[82] David L Youngs. “An interface tracking method for a 3D Eulerian hydrodynamics code”. In: *Atomic Weapons Research Establishment (AWRE) Technical Report* 44.92 (1984), p. 35.

[83] Christian F Janßen, Stephan T Grilli, and Manfred Krafczyk. “On enhanced non-linear free surface flow simulations with a hybrid LBM–VOF model”. In: *Computers & Mathematics with Applications* 65.2 (2013), pp. 211–229.

[84] Maciej Skarysz, Andrew Garmory, and Mehriar Dianat. “An iterative interface reconstruction method for PLIC in general convex grids as part of a Coupled Level Set Volume of Fluid solver”. In: *Journal of Computational Physics* 368 (2018), pp. 254–276.

[85] D Kothe et al. “Volume tracking of interfaces having surface tension in two and three dimensions”. In: *34th Aerospace Sciences Meeting and Exhibit.* 1996, p. 859.

[86] Osborne Reynolds. “XXIX. An experimental investigation of the circumstances which determine whether the motion of water shall be direct or sinuous, and of the law of resistance in parallel channels”. In: *Philosophical Transactions of the Royal society of London* 174 (1883), pp. 935–982.

[87] Timm Krüger. “Unit conversion in LBM”. In: *LBM Workshop. Dostupné z: http://lbmworkshop. com/wp-content/uploads/2011/08/2011-08-22_Edmonton_scaling. pdf.* 2011.

[88] Cx K Batchelor and GK Batchelor. *An introduction to fluid dynamics.* Cambridge university press, 2000.

[89] George Gabriel Stokes. *On the effect of the internal friction of fluids on the motion of pendulums.* Vol. 9. Pitt Press Cambridge, 1851.

[90] Frank M White and Isla Corfield. *Viscous fluid flow.* Vol. 3. McGraw-Hill New York, 2006.

[91] John Southard. *Introduction to Fluid Motions, Sediment Transport and Current-Generated Sedimentary Structures.* 2006.

[92] Oren Breslouer. “Rayleigh-Plateau Instability: Falling Jet”. In: *Project Report* (2010).

[93] *Engineering ToolBox.* 2001. URL: `https://www.engineeringtoolbox.com` (visited on 11/04/2019).

[94] Henrik Bruus. *Theoretical microfluidics.* Vol. 18. Oxford university press Oxford, 2008.

[95] Niels Asger Mortensen, Fridolin Okkels, and Henrik Bruus. “Reexamination of Hagen-Poiseuille flow: Shape dependence of the hydraulic resistance in microchannels”. In: *Physical Review E* 71.5 (2005), p. 057301.

[96] Stefan Gustavson. “Simplex noise demystified”. In: *Linköping University, Linköping, Sweden, Research Report* (2005).

[97] John C Hart. “Perlin noise pixel shaders”. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*. ACM. 2001, pp. 87–94.

[98] Marise V Gielen et al. “Oblique drop impact onto a deep liquid pool”. In: *Physical review fluids* 2.8 (2017), p. 083602.

[99] Sten A Reijers et al. “Oblique droplet impact onto a deep liquid pool”. In: *arXiv preprint arXiv:1903.08978* (2019).

[100] Huimin Ma et al. “Deformation characteristics and energy conversion during droplet impact on a water surface”. In: *Physics of Fluids* 31.6 (2019), p. 062108.

[101] Ruben Scardovelli and Stéphane Zaleski. “Direct numerical simulation of free-surface and interfacial flow”. In: *Annual review of fluid mechanics* 31.1 (1999), pp. 567–603.

[102] An-Bang Wang and Chi-Chang Chen. “Splashing impact of a single drop onto very thin liquid films”. In: *Physics of fluids* 12.9 (2000), pp. 2155–2158.

[103] Gangtao Liang et al. “Crown behavior and bubble entrainment during a drop impact on a liquid film”. In: *Theoretical and Computational Fluid Dynamics* 28.2 (2014), pp. 159–170.

[104] Kähler Nian-Sheng Cheng Volk. *Calculate density and viscosity of glycerol/water mixtures*. 2018. URL: `http://www.met.reading.ac.uk/~sws04cdw/viscosity_calc.html` (visited on 11/23/2019).

[105] Glycerine Producers’ Association et al. *Physical properties of glycerine and its solutions*. Glycerine Producers’ Association, 1963.

[106] NASA. *Two Views of Von Karman Vortices*. 2017. URL: `https://eoimages.gsfc.nasa.gov/images/imagerecords/90000/90734/guadalupe_vir_2017144_lrg.jpg` (visited on 11/29/2019).

[107] Brendan Fry, Luke McGuire, and Aalok Shah. “An experimental study of frequency regimes of honey coiling”. In: *The University of Arizona. Available at:¡ http://math. arizona. edu/˜ bfry/2008* 20 (2008).

[108] Neil M Ribe et al. “Multiple coexisting states of liquid rope coiling”. In: *Journal of Fluid Mechanics* 555 (2006), pp. 275–297.

[109] University of Bayreuth. *Collaborative Research Centre 1357 Microplastics*. 2019. URL: `https://www.sfb-mikroplastik.uni-bayreuth.de/en/index.html` (visited on 11/29/2019).

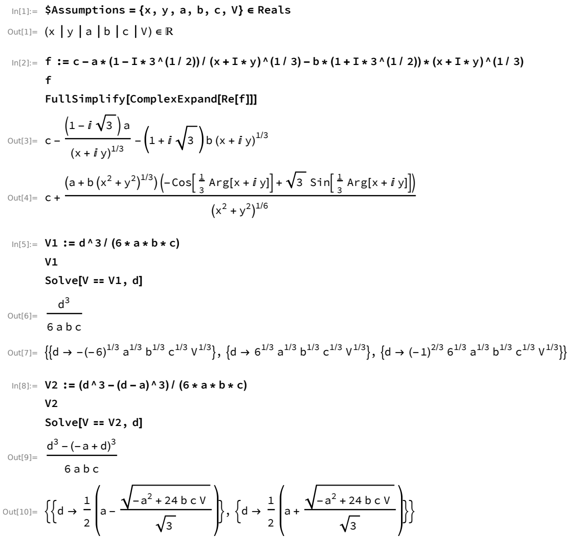
14 Acknowledgements

First of all I would like to thank Stephan Gekle for his support and advice during the last years. Also I'd like to thank Fabian Häusl for collaborating with me on the code, for very successfully extending it to run on multiple GPUs and also for tracking down one or the other implementation mistake from my side.
I thank Markus Hilt for making upgrades to and our local SMAUG compute cluster, greatly extending its capability from 50 $TFLOPs/s$ to 188 $TFLOPs/s$.
Furthermore I would like to thank Konstantin Luft, Sanwardhini Pantawane, Simon Streit, Lukas Weihmayr, Sebastian Müller, Axel Bourdick and all the others in the research group for a nice time in the office.

The Titan Xp GPU used for the majority of the development of this code was kindly donated by the NVIDIA Corporation and saved me countless hours.

Last but not least I want to thank my grandpa, Anton Haas, for supporting me over the years more than anyone else. He passed away in September at the consequences of a stroke and I am glad for every minute I could spend with him.

15 Appendix: PLIC Inversion with Mathematica



```
In[1]:= $Assumptions = {x, y, a, b, c, V} ∈ Reals
```

Out[1]= $(x \mid y \mid a \mid b \mid c \mid V) \in \mathbb{R}$

```
In[2]:= f := c - a * (1 - I * 3 ^ (1 / 2)) / (x + I * y) ^ (1 / 3) - b * (1 + I * 3 ^ (1 / 2)) * (x + I * y) ^ (1 / 3)
        f
        FullSimplify[ComplexExpand[Re[f]]]
```

Out[3]=
$$c - \frac{\left(1 - i\sqrt{3}\right) a}{(x + i\,y)^{1/3}} - \left(1 + i\sqrt{3}\right) b\,(x + i\,y)^{1/3}$$

Out[4]=
$$c + \frac{\left(a + b\left(x^2 + y^2\right)^{1/3}\right)\left(-\mathrm{Cos}\left[\frac{1}{3}\,\mathrm{Arg}[x + i\,y]\right] + \sqrt{3}\ \mathrm{Sin}\left[\frac{1}{3}\,\mathrm{Arg}[x + i\,y]\right]\right)}{\left(x^2 + y^2\right)^{1/6}}$$

```
In[5]:= V1 := d ^ 3 / (6 * a * b * c)
        V1
        Solve[V == V1, d]
```

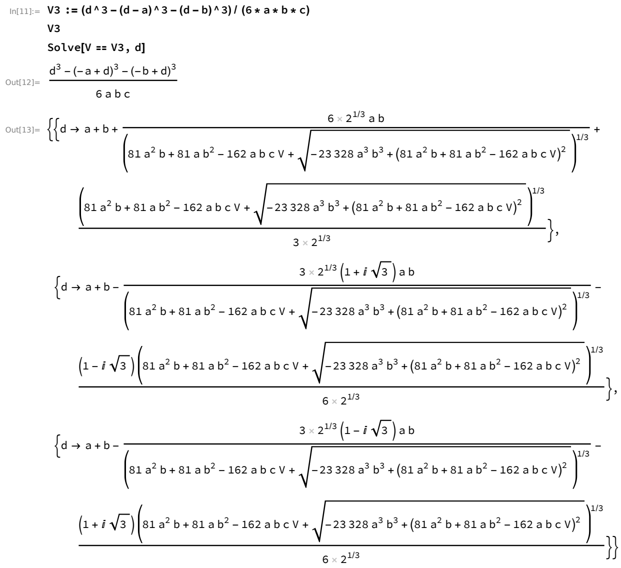
Out[6]=
$$\frac{d^3}{6\,a\,b\,c}$$

Out[7]=
$$\left\{\left\{d \to -(-6)^{1/3}\,a^{1/3}\,b^{1/3}\,c^{1/3}\,V^{1/3}\right\}, \left\{d \to 6^{1/3}\,a^{1/3}\,b^{1/3}\,c^{1/3}\,V^{1/3}\right\}, \left\{d \to (-1)^{2/3}\,6^{1/3}\,a^{1/3}\,b^{1/3}\,c^{1/3}\,V^{1/3}\right\}\right\}$$

```
In[8]:= V2 := (d ^ 3 - (d - a) ^ 3) / (6 * a * b * c)
        V2
        Solve[V == V2, d]
```

Out[9]=
$$\frac{d^3 - (-a + d)^3}{6\,a\,b\,c}$$

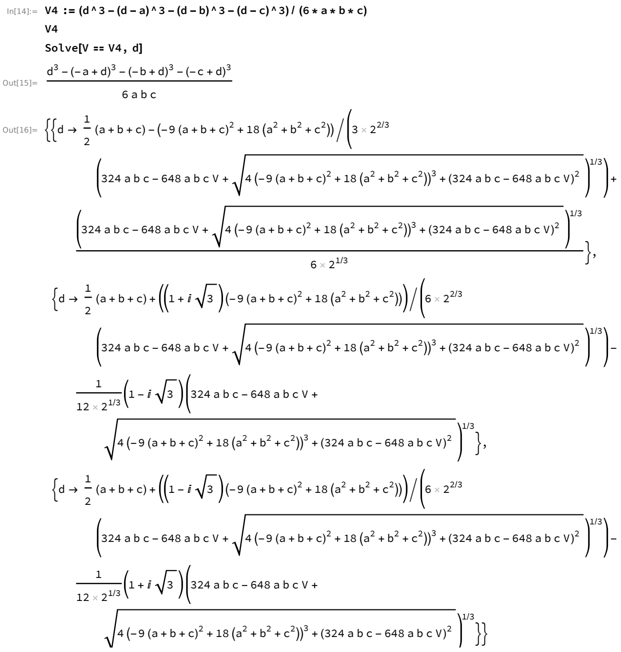
Out[10]=
$$\left\{\left\{d \to \frac{1}{2}\left(a - \frac{\sqrt{-a^2 + 24\,b\,c\,V}}{\sqrt{3}}\right)\right\}, \left\{d \to \frac{1}{2}\left(a + \frac{\sqrt{-a^2 + 24\,b\,c\,V}}{\sqrt{3}}\right)\right\}\right\}$$



```
In[11]:= V3 := (d ^ 3 - (d - a) ^ 3 - (d - b) ^ 3) / (6 * a * b * c)
         V3
         Solve[V == V3, d]
```

Out[12]=
$$\frac{d^3 - (-a + d)^3 - (-b + d)^3}{6\,a\,b\,c}$$

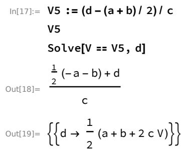
Out[13]=
$$\left\{\left\{d \to a + b + \frac{6 \times 2^{1/3}\,a\,b}{\left(81\,a^2\,b + 81\,a\,b^2 - 162\,a\,b\,c\,V + \sqrt{-23\,328\,a^3\,b^3 + \left(81\,a^2\,b + 81\,a\,b^2 - 162\,a\,b\,c\,V\right)^2}\right)^{1/3}} + \frac{\left(81\,a^2\,b + 81\,a\,b^2 - 162\,a\,b\,c\,V + \sqrt{-23\,328\,a^3\,b^3 + \left(81\,a^2\,b + 81\,a\,b^2 - 162\,a\,b\,c\,V\right)^2}\right)^{1/3}}{3 \times 2^{1/3}}\right\},\right.$$
$$\left\{d \to a + b - \frac{3 \times 2^{1/3}\left(1 + i\sqrt{3}\right) a\,b}{\left(81\,a^2\,b + 81\,a\,b^2 - 162\,a\,b\,c\,V + \sqrt{-23\,328\,a^3\,b^3 + \left(81\,a^2\,b + 81\,a\,b^2 - 162\,a\,b\,c\,V\right)^2}\right)^{1/3}} - \frac{\left(1 - i\sqrt{3}\right)\left(81\,a^2\,b + 81\,a\,b^2 - 162\,a\,b\,c\,V + \sqrt{-23\,328\,a^3\,b^3 + \left(81\,a^2\,b + 81\,a\,b^2 - 162\,a\,b\,c\,V\right)^2}\right)^{1/3}}{6 \times 2^{1/3}}\right\},$$
$$\left.\left\{d \to a + b - \frac{3 \times 2^{1/3}\left(1 - i\sqrt{3}\right) a\,b}{\left(81\,a^2\,b + 81\,a\,b^2 - 162\,a\,b\,c\,V + \sqrt{-23\,328\,a^3\,b^3 + \left(81\,a^2\,b + 81\,a\,b^2 - 162\,a\,b\,c\,V\right)^2}\right)^{1/3}} - \frac{\left(1 + i\sqrt{3}\right)\left(81\,a^2\,b + 81\,a\,b^2 - 162\,a\,b\,c\,V + \sqrt{-23\,328\,a^3\,b^3 + \left(81\,a^2\,b + 81\,a\,b^2 - 162\,a\,b\,c\,V\right)^2}\right)^{1/3}}{6 \times 2^{1/3}}\right\}\right\}$$



```
In[14]:= V4 := (d ^ 3 - (d - a) ^ 3 - (d - b) ^ 3 - (d - c) ^ 3) / (6 * a * b * c)
         V4
         Solve[V == V4, d]
```

Out[15]=
$$\frac{d^3 - (-a + d)^3 - (-b + d)^3 - (-c + d)^3}{6\,a\,b\,c}$$

Out[16]=
$$\left\{\left\{d \to \frac{1}{2}(a + b + c) - \left(-9\,(a + b + c)^2 + 18\left(a^2 + b^2 + c^2\right)\right) \Big/ \left(3 \times 2^{2/3}\left(324\,a\,b\,c - 648\,a\,b\,c\,V + \sqrt{4\left(-9\,(a + b + c)^2 + 18\left(a^2 + b^2 + c^2\right)\right)^3 + (324\,a\,b\,c - 648\,a\,b\,c\,V)^2}\right)^{1/3}\right) + \frac{\left(324\,a\,b\,c - 648\,a\,b\,c\,V + \sqrt{4\left(-9\,(a + b + c)^2 + 18\left(a^2 + b^2 + c^2\right)\right)^3 + (324\,a\,b\,c - 648\,a\,b\,c\,V)^2}\right)^{1/3}}{6 \times 2^{1/3}}\right\},\right.$$
$$\left\{d \to \frac{1}{2}(a + b + c) + \left(\left(1 + i\sqrt{3}\right)\left(-9\,(a + b + c)^2 + 18\left(a^2 + b^2 + c^2\right)\right)\right) \Big/ \left(6 \times 2^{2/3}\left(324\,a\,b\,c - 648\,a\,b\,c\,V + \sqrt{4\left(-9\,(a + b + c)^2 + 18\left(a^2 + b^2 + c^2\right)\right)^3 + (324\,a\,b\,c - 648\,a\,b\,c\,V)^2}\right)^{1/3}\right) - \frac{1}{12 \times 2^{1/3}}\left(1 - i\sqrt{3}\right)\left(324\,a\,b\,c - 648\,a\,b\,c\,V + \sqrt{4\left(-9\,(a + b + c)^2 + 18\left(a^2 + b^2 + c^2\right)\right)^3 + (324\,a\,b\,c - 648\,a\,b\,c\,V)^2}\right)^{1/3}\right\},$$
$$\left.\left\{d \to \frac{1}{2}(a + b + c) + \left(\left(1 - i\sqrt{3}\right)\left(-9\,(a + b + c)^2 + 18\left(a^2 + b^2 + c^2\right)\right)\right) \Big/ \left(6 \times 2^{2/3}\left(324\,a\,b\,c - 648\,a\,b\,c\,V + \sqrt{4\left(-9\,(a + b + c)^2 + 18\left(a^2 + b^2 + c^2\right)\right)^3 + (324\,a\,b\,c - 648\,a\,b\,c\,V)^2}\right)^{1/3}\right) - \frac{1}{12 \times 2^{1/3}}\left(1 + i\sqrt{3}\right)\left(324\,a\,b\,c - 648\,a\,b\,c\,V + \sqrt{4\left(-9\,(a + b + c)^2 + 18\left(a^2 + b^2 + c^2\right)\right)^3 + (324\,a\,b\,c - 648\,a\,b\,c\,V)^2}\right)^{1/3}\right\}\right\}$$



```
In[17]:= V5 := (d - (a + b) / 2) / c
         V5
         Solve[V == V5, d]
```

Out[18]=
$$\frac{\frac{1}{2}(-a - b) + d}{c}$$

Out[19]=
$$\left\{\left\{d \to \frac{1}{2}(a + b + 2\,c\,V)\right\}\right\}$$

16 Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und alle Zitate deutlich kenntlich gemacht zu haben. Die Arbeit wurde nicht bereits in gleicher oder vergleichbarer Form zur Erlangung eines akademischen Grades eingereicht. Des Weiteren versichere ich, dass die digitale und die gedruckte Version inhaltlich identisch sind.

Bayreuth, den 09.12.2019

Moritz Lehmann