Technical University of Košice

Faculty of Mining, Ecology, Process Control and Geotechnologies

Design and Implementation of Modern Methods for Modeling and Control of Technological Objects and Processes

Dissertation Thesis

2021 **Michal Takáč**

Technical University of Košice

Faculty of Mining, Ecology, Process Control and Geotechnologies

Design and Implementation of Modern Methods for Modeling and Control of Technological Objects and Processes

Dissertation Thesis

Study Programme:	Cybernetics
Field of study:	Process Control
Department:	Institute of Process Control and Informatization (URIVP)
Supervisor:	prof. Ing. Ivo Petráš, DrSc.
Consultant(s):	

Košice 2021 **Michal Takáč**

Abstract

Text abstraktu v svetovom jazyku je potrebný pre integráciu do medzinárodných informačných systémov. Ak nie je možné cudzojazyčnú verziu abstraktu umiestniť na jednej strane so slovenským abstraktom, je potrebné umiestniť ju na samostatnú stranu (cudzojazyčný abstrakt nemožno deliť a uvádzať na dvoch strabách).

Keywords

Steelmaking, Mathematical modeling, Visualization, Virtual Reality

Abstrakt

Abstrakt je povinnou súčasťou každej práce. Je výstižnou charakteristikou obsahu dokumentu. Nevyjadruje hodnotiace stanovisko autora. Má byť taký informatívny, ako to povoľuje podstata práce. Text abstraktu sa píše ako jeden odstavec. Abstrakt neobsahuje odkazy na samotný text práce. Mal by mať rozsah 250 až 500 slov. Pri štylizácii sa používajú celé vety, slovesá v činnom rode a tretej osobe. Používa sa odborná terminológia, menej zvyčajné termíny, skratky a symboly sa pri prvom výskyte v texte definujú.

Kľúčové slová

Oceliarstvo, Matematické modelovanie, Vizualizácia, Virtuálna realita

Assign Thesis

Namiesto tejto strany vložte naskenované zadanie úlohy. Odporúčame skenovať s rozlíšením 200 až 300 dpi, čierno-bielo! V jednej vytlačenej ZP musí byť vložený originál zadávacieho listu!

Declaration

I hereby declare that this thesis is my own work and effort. Where other sources of information have been used, they have been acknowledged.

Košice, April 1, 2021 . . . . . .

Signature

Acknowledgement

I would like to express my sincere thanks to my supervisor Prof. Ing. Ivo Petráš, DrSc., the main Supervisor, for his constant, and constructive guidance throughout the study. To all other who gave a hand, I say thank you very much.

Preface

Predhovor (*Preface*) je povinnou náležitosťou záverečnej práce, pozri (**?**). V predhovore autor uvedie základné charakteristiky svojej záverečnej práce a okolnosti jej vzniku. Vysvetlí dôvody, ktoré ho viedli k voľbe témy, cieľ a účel práce a stručne informuje o hlavných metódach, ktoré pri spracovaní záverečnej práce použil.

Contents

1 Introduction **14**

2 Modeling and Simulation **16**

2.1 Mathematical Modeling 16

2.2 Numerical Simulation 16

2.3 Computational Fluid Dynamics 17

2.4 Integrated Modeling and Simulation 18

3 Lattice Boltzmann Method **18**

3.1 Kinetic Theory 21

3.2 Bhatnagar-Gross-Krook Model 23

3.3 Multiple Relaxation Time 23

3.4 Boundary Conditions 25

3.5 Lattice Boltzmann In Context of Computational Fluid Dynamics 25

3.6 Turbulence Modeling 25

3.6.1 Fluid Turbulence 25

3.6.2 Sub-grid Scale Modeling 26

3.7 Multiphase Flows 26

3.8 Adaptive Mesh Refinement 28

3.9 Complex Fluids and Beyond 28

4 Visualization Methods **28**

4.1 Human-Computer Interface 28

4.1.1 SCADA 28

4.1.2 Principles of UI Design 29

4.1.3 From 2D to 3D User Interfaces 29

4.2 Post Hoc Visualization 29

4.3 Real-time Visualization 30

4.4 Interactive Visualization 32

4.4.1 Steering the Running Simulation . . . 34
4.4.2 Time Manipulation . . . 34
4.5 Virtual and Augmented Reality . . . 34
4.5.1 Virtual Reality in Steelmaking . . . 37

5 GPU Computing 38
5.1 Parallel Programming . . . 39
5.2 Heterogeneous Computing . . . 42
5.3 CUDA . . . 42
5.4 OpenCL . . . 43
5.5 Cross-platform GPU Programming . . . 43
5.6 Accelerating Lattice Boltzmann Simulations with GPUs . . . 47

6 Implementation of a Simulation Software 49
6.1 Technology Stack . . . 49
6.1.1 Cross-platform Development with ArrayFire . . . 49
6.1.2 Rust as C/C++ Alternative . . . 51
6.1.3 Hardware . . . 53
6.2 Implementation of Lattice Boltzmann Method for GPUs . . . 53
6.2.1 Initialization . . . 54
6.2.2 Boundary Conditions . . . 54
6.2.3 Initial Conditions . . . 55
6.2.4 Streaming . . . 56
6.2.5 Collision . . . 58
6.2.6 External Force . . . 59
6.3 GPU Optimizations . . . 59
6.3.1 Data Organization . . . 61
6.3.2 Removing Branch Divergence . . . 63
6.3.3 Pull vs Push Scheme . . . 65
6.3.4 Load Balancing in Multi-GPU Setups . . . 66

6.4 Interactive Simulation 66
6.4.1 Unity and Rust Interop 66
6.4.2 Setting Initial Conditions 69
6.4.3 Visualizing Simulation Output in Real-Time 70
6.4.4 Updating Boundary Conditions 70
6.4.5 Time Manipulation 70
6.5 Virtual Reality User Interface 72
6.5.1 Cross-platform Development 72
6.5.2 VRTK 72
6.5.3 Interacting with 3D Data 72
6.5.4 Plotting in VR 72
6.6 Performance Analysis 72
6.6.1 Hardware 73
6.6.2 Results 74

7 Conclusions **75**

8 Discussion **77**
8.1 Future Work 77
8.1.1 Streaming Visualized Pixels Over Network 77
8.1.2 Simulation as an Educational Tool 77

List of Figures

3 – 1 D2Q9 lattice node scheme. 20

3 – 2 Three-dimensional lattice node schemes. 22

4 – 1 Differences between CPU-bound visualization with expensive copying bottleneck and high-performance GPU-bound visualization keeping full speeds of high-bandwidth of PCIe interface. 31

4 – 2 Lid-driven cavity test case at 128×128 resolution after 5000 iterations. . . 32

4 – 3 Kármán vortex street (channel flow past circle-shaped obstacle) test case at 1000×300 resolution after 5000 iterations. 33

5 – 1 CUDA memory model 42

6 – 1 Unity Editor with script interface. 70

6 – 2 Interface for controlling the state of simulation. 71

6 – 3 Interface for controlling the state of simulation. 72

6 – 4 Single-precision performance analysis of 2D Kármán vortex on D2Q9 stencil. 76

6 – 5 Double-precision performance analysis of 2D Kármán vortex on D2Q9 stencil. 77

6 – 6 Single-precision performance analysis of 2D Kármán vortex on D2Q9 stencil. 79

6 – 7 Double-precision performance analysis of 2D Kármán vortex on D2Q9 stencil. 80

6 – 8 Peak performance of single-precision LBM simulations on D2Q9 stencil. 80

8 – 1 Interactive, educational simulation of basic oxygen steelmaking by steeluniversity.org. 81

List of Tables

5 – 1 Functions for working with OpenCL context in ArrayFire-OpenCL interoperability scenario. 46

6 – 1 Hardware specifications for GPUs used for testing the simulation software described in this paper (taken from https://www.techpowerup.com/gpu-specs/). SM - streaming multiprocessor, CU - computing units. 74

6 – 2 Average MLUPS of lid-driven cavity test case with single-precision floating point computation (taking only the steady performance after initial warm-up and removal of sudden spikes). 75

6 – 3 Average MLUPS of Kármán vortex street test case with single-precision floating point computation (taking only the steady performance after initial warm-up and removal of sudden spikes). 78

List of Terms

Dizertácia je rozsiahla vedecká rozprava, v ktorej sa na základe vedeckého výskumu a s použitím (využitím) bohatého dokladového materiálu ako i vedeckých metód rieši zložitý odborný problém.

Font je súbor, obsahujúci predpisy na zobrazenie textu v danom písme, napr. na tlačiarni. To čo vidíme je písmo; font je súbor a nevidíme ho.

Kritika je odborne vyhrotený, prísny pohľad na hodnotenú vec. Medzi recenziou a kritikou je taký pomer ako medzi diskusiou a polemikou. Pri kritike treba prísnosť chápať v tom zmysle, že sa v nej okrem iného navrhuje, ako hodnotené dielo skvalitniť.

Meter (m) je vzdialenosť, ktorú svetlo vo vákuu prejde za časový interval $1/299\,792\,458$ sekundy.

Písmom rozumieme vlastný vzhľad znakov.

Problém termín používaný vo všeobecnom zmysle vo vzťahu k akejkoľvek duševnej aktivite, ktorá má nejaký rozoznateľný cieľ. Samotný cieľ nemusí byť v dohľadne. Problémy možno charakterizovať tromi rozmermi – oblasťou, obtiažnosťou a veľkosťou.

Proces je postupnonosť či rad časovo usporiadaných udalostí tak, že každá predchádzajúca udalosť sa zúčastňuje na determinácii nasledujúcej udalosti.

1 Introduction

The objective of process control is to keep key process-operating parameters within narrow bounds of the reference value or setpoint. Controllers are used to automate a human function in an effort to control a variable. A basic controller can keep an individual loop on an even point, so long as there is not too much disruption. Complex processes like ones in metallurgy might employ dozens or even hundreds of such controllers, but keeping an eye on the big picture was, until not so long ago, a human process.

Although a device was used to automate a human function in an effort to control a variable, there was no sense of what the process was doing overall. A basic controller could keep an individual loop on an even keel, more or less, so long as there was not too much disruption. Complex processes might employ dozens or even hundreds of such controllers, each with its performance displayed on a panel board, but keeping an eye on the big picture was still a human process.

When distributed control system (DCS) platforms were introduced in the 1970s, they simplified the mechanics of the panel board, but did not do much to improve its capabilities. Big-picture analysis was still largely a human responsibility. Sure, getting beyond the technical constraints of pneumatic field devices with their troublesome compressed air tubing made it easier to install more instruments and actuators, but the basic control concepts did not really change. Any movement to advanced process control (APC) and other forms of control optimization were still in their infancy. Process automation capable of supporting APC had to encompass many technologies and techniques. It was characterized by incorporating many more input data points into algorithms and orchestrating more complex sequences.

The transition to process automation and advanced process control (APC) was empowered by being able to create an all-encompassing platform capable of coordinating more than single loops or small cascade groups. One major advantage of newer platforms is the ability to optimize a process to suit the owner's specific economic goals based on

any number of desired outcomes. The process automation system can operate the plant to minimize energy consumption, maximize output, and deliver specific product quality attributes.

Implementing such systems is challenging. During the initial design phase of a control system upgrade or a new installation, it is far too easy to focus just on process fundamentals, and never get beyond considering desired steady-state conditions. Automation system upgrades and new installations can therefore miss opportunities to engage with process and automation technology experts capable of uncovering better ways of doing things. Many capabilities of modern process automation systems are still underutilized in most process plants. Far fewer companies use APC as effectively as they could, even though basic APC technologies have been around for decades.

Modeling, together with simulation, are important parts of engineering. Integrating them into engineer's toolbelt yields many benefits. Before actual testing in physical reality, one can do many different virtual tests by simulating the modeled phenomena with lower overhead of financial requirements. There's also close to zero risk of hurting anybody (most notable risk being ill-treatment of electricity). Still, coming up with and refining actual mathematical or physical model, to the point of it being accurate enough for the job at hand, takes a lot of time. Also, actual simulation part can take hours or days to converge. However, it can be argued that, in some industrial applications, broad parameter space mapping is sometimes more valuable than higher order analysis of fewer design points (**?**).

With the continuation of expanding computational power available for engineers that can tap into it for their simulations, the time for simulation to converge decreases accordingly. it becomes clear enough that ...

We'll discuss the path towards interactive simulation in section 4.4.

2 Modeling and Simulation

"The purpose of computing is insight, not numbers"

– R. W. Hamming, *The Art of Doing Science and Engineering*

Why simulation?

Investigate what cannot be measured

Reduce need for testing

Design optimization: narrow design space

Proactive instead of reactionary design

2.1 Mathematical Modeling

2.2 Numerical Simulation

A numerical simulation is a calculation that is run on a computer following a program that implements a mathematical model for a physical system. Numerical simulations are required to study the behaviour of systems whose mathematical models are too complex to provide analytical solutions, as in most nonlinear systems.

The motivation for using computer simulations to investigate complex processes is two-fold. , it enables design changes to be tested before building a prototype, which naturally leads to a lower total design cost. Second, it makes it possible to investigate phenomena that cannot easily be measured or observed in the process. Even a seemingly simple operation such as the continuous measurement of the temperature during the decarburization process is difficult due to the very high temperatures in the process and generally harsh conditions prevailing in the steel plants

2.3 Computational Fluid Dynamics

Modern fluid mechanics problems would be impossible to solve without use of Computational Fluid Dynamics (CFD), since the scope of analytical solutions to fundamental equations of fluid mechanics is very limited and, once a more difficult geometry is encountered, we usually have to choose a given numerical method for obtaining a solution. CFD encompasses a wide spectrum of numerical methods used for solving complex three-dimensional (3D) and time-dependent flow problems (Rapp; 2017). Since early pioneering work in the metallurgical field done by Szekely et al. (1977), the cost of performing computer simulations has decreased over the last few decades, while the available processing power has increased. Most of the processors and processing units that are currently developed and produced have several cores that can execute instructions in parallel. Thus, the processing power available to a CFD software also depends on the capability of the software to execute in parallel. A study by Ersson and Tilliander (2018) of the last two decades of metallurgical CFD simulations reveals huge improvements on the type of phenomena that can be explored and we will see this trend is continuing thanks to improvements in both the available processing power and the avail- able algorithms. Therefore CFD found its way into numerous studies in steelmaking, where these methods proved useful in demonstrating the hidden and significant proper- ties. However, its use in the steel industry may not be as integrated as in the aero and automotive industries, in which the development of new designs is of key importance. The major difference between aero and metallurgical industries is that the metallurgical industries almost always deal with multiphase systems at elevated temperatures and that the motivation of modeling is mainly process optimization. With a continuing development in multiphase models as well as in reacting flow modeling, the continued usefulness of CFD in metallurgy remains clear.

2.4 Integrated Modeling and Simulation

3 Lattice Boltzmann Method

LBM formulated in 1988 by McNamara and Zanetti

– 1859: Maxwell's distribution function

– 1868: Boltzmann transport equation

– 1954: Bhatnagar, Gross, and Krook (BGK) collision operator

– 1956: FEM by Turner

– 1973,76: Hardy, Pomeau, and de Pazzis (HPP) model/Lattice Gas Automata (LGA)

– 1980: Finite volume method (FVM) at Imperial College

LB method has witnessed an astonishing growth in its methodology development and application over the past quarter of a century. It fills a vital gap between the macroscopic continuum approaches such as the Navier–Stokes solvers and the particle-based microscopic approaches such as molecular dynamics. Such a mesoscopic approach has found applications in almost all areas of energy and combustion Li et al. (2016).

The lattice Boltzmann method (LBM) originally grew out of the lattice gas automata Succi (2001). It is positioned in the middle between Eulerian and Lagrangian methods for solving fluid flow problems. Instead of calculating the properties of individual particles, the particle distribution function (PDF) is used for describing the distribution of particles that is computed for each node in the discretized domain. As we mentioned earlier, each node needs only its neighbours for the actual computation, allowing for good parallelization. A collision of particle distributions is described by Ω operator, that states the rate of change of PDF (denoted as f) is equal to the rate of collision in the limit of $dt \to 0$:

$$\frac{df}{dt} = \Omega(f). \tag{3.1}$$

The collision operator Ω is difficult to solve. It's been simplified by the work of Bhatnagar, Gross and Crook Bhatnagar et al. (1954), that introduced the BGK operator

$$\Omega_i = \frac{1}{\tau}(f_i^{eq} - f_i), \tag{3.2}$$

where f_i^{eq} is an particle distribution function in an equilibrium state of the system obtained by Taylor expansion of the Maxwell-Boltzmann equilibrium function. The relaxation parameter τ is the reciprocal that presents a time in which the systems relaxes towards the equilibrium.

The Lattice Boltzmann Equation (LBE) in its discrete form, the fundamental part of the lattice Boltzmann method, is obtained by discretization of the velocity space of the Boltzmann equation into a finite number of discrete velocities e_i, $i = 0, 1, ..., 8$. It can be stated as

$$f_i(\boldsymbol{x} + \boldsymbol{e}_i\Delta t, t + \Delta t) = f_i(\boldsymbol{x}, t) - \frac{1}{\tau}\Big[f_i(\boldsymbol{x}, t) - f_i^{eq}(\boldsymbol{x}, t)\Big], \tag{3.3}$$

where $f_i(\boldsymbol{x}, t)$ denotes the individual direction of the PDF at each lattice point in particular time and $f_i(\boldsymbol{x} + \boldsymbol{e}_i\Delta t, t + \Delta t)$ is equal to resulting PDF for all neighbouring nodes in the next iteration step. Necessary criterion for stability is that physical information should not travel faster than fastest speed supported by the lattice Succi (2001).

Macroscopic quantities are obtained from hydrodynamic moments of the distribution function. They are computed with 3.16 and momentum flux 3.17 (from which the velocity can be obtained by simply dividing the equation by ρ)

$$\rho = \sum_{i=0}^{9} f_i, \tag{3.4}$$

$$\rho\boldsymbol{u} = \sum_{i=0}^{9} \boldsymbol{e}_i f_i. \tag{3.5}$$

Equilibrium distribution function f^{eq} can be expressed by performing a Hermite expansion of the Boltzmann equilibrium function as

$$f_i^{eq} = \omega_i \rho \left(1 + 3\frac{\boldsymbol{e}_i \cdot \boldsymbol{u}}{c_s} + \frac{9}{2}\frac{(\boldsymbol{e}_i \cdot \boldsymbol{u})^2}{c_s^2} - \frac{3}{2}\frac{\boldsymbol{u}^2}{c_s^2}\right), \tag{3.6}$$

where c_s is the speed of sound within the lattice, usually set to $c_s = \frac{1}{\sqrt{3}}$ and ω denotes different weights for discrete velocity in D2Q9 stencil

$$\omega_0 = 4/9, \tag{3.7}$$

$$\omega_{1,2,3,4} = 1/9, \tag{3.8}$$

$$\omega_{5,6,7,8} = 1/36. \tag{3.9}$$

$$\tag{3.10}$$

With a proper set of discrete velocities, the LBE recovers the incompressible Navier–Stokes equations by the Chapman–Enskog expansion. For the flows within the incompressible limit, assumptions such as low-Mach number and variations in density of order $\mathcal{O}(M^2)$ has to be made.

The present study focuses on the D2Q9 discrete velocity model which is illustrated in Fig. 3 – 1.

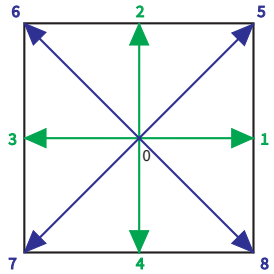


Figure 3 – 1: D2Q9 lattice node scheme.

Discrete velocities $\boldsymbol{e}_i$ are expressed as

$$e_i = \begin{bmatrix} 0 & 1 & 0 & -1 & 0 & 1 & -1 & -1 & 1 \\ 0 & 0 & 1 & 0 & -1 & 1 & 1 & -1 & -1 \end{bmatrix} \tag{3.11}$$

Two general steps of the LBM solver involve solving Eq. 3.12 for collision and Eq. 3.13 for streaming in each iteration.

$$f_i(\boldsymbol{x}, t + \Delta t) = f_i(\boldsymbol{x}, t) - \frac{1}{\tau}\Big[f_i(\boldsymbol{x}, t) - f_i^{eq}(\boldsymbol{x}, t)\Big]. \tag{3.12}$$

$$f_i(\boldsymbol{x} + \boldsymbol{e}_i \Delta t, t + \Delta t) = f_i(\boldsymbol{x}, t + \Delta t). \tag{3.13}$$

For the boundary condition, we implemented a simple no-slip boundary known as bounce-back, which effectively reverses the direction of f_i **?**. In places like input and output of simulated pipe for Kármán vortex test case, we implemented periodic boundary conditions **?**.

During the last three decades, the mesoscopic lattice Boltzmann method (LBM), based on the kinetic theory, has become an increasingly important method for numerical simulations of multiphase flows, mainly on account of its meso-scale features, easy the numerical stability compared with the SRT-LBM. The corresponding non-orthogonal MRT-LBM has been extended to sim

3.1 Kinetic Theory

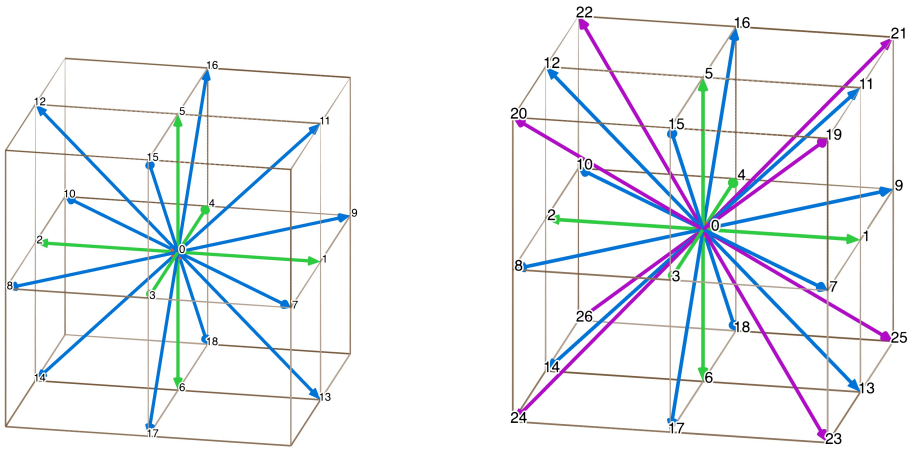
The main properties of any numerical scheme can be classified as follows (Succi, 2001):

- causality,
- accuracy,
- stability,
- consistency,

- efficiency,
- flexibility.

With respect to above properties, lattice Boltzmann equation (LBE) can be classified as an explicit, Lagrangian, finite-hyperbolicity approximation of Navier-Stokes equations.

The LBE is obtained by discretizing the velocity space of the Boltzmann equation into a finite number of discrete velocities e_α $\{\alpha = 0,1,...,26\}$. With a proper set of discrete velocities, the LBE recovers the continuum Navier–Stokes equations by the Chapman–Enskog expansion.



(a) D3Q19 (b) D3Q27

Figure 3 – 2: Three-dimensional lattice node schemes.

Caution have to be taken when working within lattice's discrete world. For simulating the interface of blown oxygen with melted fluid slag, we're working with higher Mach speeds. The basic notion is that the lattice can only support signals with a finite propagation speed. Necessary criterion for stability is that physical information should not travel faster than fastest speed supported by the lattice Succi (2001).

We can calculate the error to $\varepsilon(Ma^3)$ in space and proportional to $\varepsilon(Ma \cdot dt)$ in time, where $Ma = \frac{u}{c_s}$ is the Mach number of the system.

$p = c_s^2 \rho$ is the pressure, $c_s = \frac{c}{\sqrt{3}}$ is the speed of sound, and the kinematic and viscosity ν is related to the relaxation time rates for the second-order moments by $\nu = \left(\frac{1}{s_\nu - 0.5}\right) c_s^2 \Delta t$ and $\xi = \frac{2}{3}\left(\frac{1}{s_b - 0.5}\right)$ respectively.

3.2 Bhatnagar-Gross-Krook Model

- TODO: napisat aj o klasickom BGK modeli (Bhatnagar et al., 1954)

The collision operator Ω is difficult to solve. It's been simplified by the work of Bhatnagar, Gross and Crook Bhatnagar et al. (1954), that introduced the BGK operator

$$f_i(\boldsymbol{x} + \boldsymbol{e}_i \Delta t, t + \Delta t) = f_i(\boldsymbol{x}, t) - \frac{1}{\tau}\Big(f_i(\boldsymbol{x}, t) - f_i^{eq}(\boldsymbol{x}, t)\Big) \tag{3.14}$$

In the presence of a body force density $\boldsymbol{F} = \rho \boldsymbol{g}$, where $\boldsymbol{g}$ is the acceleration due to $\boldsymbol{F}$, the LBE must be modified to account for the force Guo et al. (2002).

$$f_i^{eq} = \omega_i \rho \left(1 + 3\frac{\boldsymbol{e}_i \cdot \boldsymbol{u}}{c} + \frac{9}{2}\frac{(\boldsymbol{e}_i \cdot \boldsymbol{u})^2}{c^2} - \frac{3}{2}\frac{\boldsymbol{u}^2}{c^2}\right) \tag{3.15}$$

$$\rho = \sum_{i=0}^{27} f_i \tag{3.16}$$

$$\rho \boldsymbol{u} = \sum_{i=0}^{27} f_i \boldsymbol{e}_i + \frac{\Delta t \boldsymbol{F}}{2} \tag{3.17}$$

3.3 Multiple Relaxation Time

To overcome difficulties of numerical instability in applying the LBM method, the multiple-relaxation-time (MRT) scheme is useful to stabilize the solution and to obtain satisfactory results because the MRT model allows the usage of an independently optimized relaxation-time for each physical process Suga et al. (2015).

A general collision step in MRT can be written as Fei et al. (2019)

$$f_i^*(\boldsymbol{x},t) = f_i(\boldsymbol{x},t) - \Lambda_{i,k}\left[f_k - f_k^{eq}\right]_{(\boldsymbol{x},t)} + \frac{\Delta t}{2}\left[F_i(\boldsymbol{x},t) + F_i(\boldsymbol{x}+\boldsymbol{e}_i\Delta t, t+\Delta t)\right] \quad (3.18)$$

where $\boldsymbol{x}$ is the spatial position in the 3D grid, t is time, F_i are the forcing terms in discrete velocity space, and collision operator $\Lambda_{i,k}$ computed as Fei et al. (2019)

Although many schemes to discretize the velocity space have been proposed, for three-dimensional (3-D) flows, the present study focuses on so called the three- dimensional twenty-seven (D3Q27) discrete velocity model which is illustrated in Fig. 1. Table 1 lists the sound speed c_S , the discrete velocity e_α and the weight parameter w_α in the model with $c = \delta x/\delta t$ where δx and δt are the lattice spacing and the time step, respectively. The MRT LBM transforms the distribution function in the velocity space to the moment space by the transformation matrix M . Transformation matrix $\boldsymbol{M}$ can be obtained from Eqs. 3.24.

$$M = ...doplnit \quad (3.19)$$

Since the moments of the distribution function correspond directly to flow quantities, the moment representation allows us to perform the relaxation processes with different relaxation-times according to different time-scales of various physical processes. The evolution equation for the particle distribution function f is thus written as

$$|f(x_i + e_\alpha\delta t, t+\delta t)\rangle - |f(x_i,t)\rangle = -M^{-1}S[|m(x_i,t)\rangle - |m^{eq}(x_i,t)\rangle] + |F(x_i,t)\rangle. \quad (3.20)$$

where x_i is the position vector of node i, $\vec{S}$ is the collision matrix, m is the moment, F represents an external body force and the notation for the column vector (known as the ket vector) such as $|f\rangle$ represents

$$\Lambda_{i,k} = (\boldsymbol{M}^{-1}\boldsymbol{S}\boldsymbol{M})_{i,k} \quad (3.21)$$

in which S is a diagonal relaxation matrix.

$$f_i^{eq} = \omega_i \rho \left(1 + 3\frac{\boldsymbol{e}_i \cdot \boldsymbol{u}}{c} + \frac{9}{2}\frac{(\boldsymbol{e}_i \cdot \boldsymbol{u})^2}{c^2} - \frac{3}{2}\frac{\boldsymbol{u}^2}{c^2}\right) \tag{3.22}$$

$$\boldsymbol{m}^* = \boldsymbol{m} - \boldsymbol{S}(\boldsymbol{m} - \boldsymbol{m}^{eq}) + \left(\boldsymbol{I} - \frac{\boldsymbol{S}}{2}\right)\Delta t \boldsymbol{F} \tag{3.23}$$

where $\boldsymbol{m} = \boldsymbol{M}\boldsymbol{f}$, $\boldsymbol{m}^{eq} = \boldsymbol{M}\boldsymbol{f}^{eq}$, and $\boldsymbol{F} = \boldsymbol{M}\boldsymbol{F}$.

$$\boldsymbol{m} = [m_0, m_1, ..., m_{26}]^T \tag{3.24}$$

For better numerical stability, Multiple Relaxation Time (MRT) scheme is used. It allows for more degrees of freedom and better tunability of relaxation parameters. Stability is a key property in any numerical scheme Succi (2001). It helps to protect against cumulative error build-up or other sources of inaccuracies.

3.4 Boundary Conditions

3.5 Lattice Boltzmann In Context of Computational Fluid Dynamics

3.6 Turbulence Modeling

3.6.1 Fluid Turbulence

turbulence modelling using the Smagorinsky model in LBM for the simulation of high Reynolds number flow and the coupling of two LBM simulations to simulate thermal flows under the Boussinesq approximation.

3.6.2 Sub-grid Scale Modeling

3.7 Multiphase Flows

Interfaces between different phases and/or components are ubiquitous in multiphase flows and energy applications, such as rain dynamics, plant spraying, water boiling, and gas turbine blade cooling, to name but a few. A deeper understanding of the fundamental physics of such complex interfaces is of great importance in many natural and industrial processes. The dynamics of the interfaces is difficult to investigate because typical interfaces are extremely thin, complex in shape, and deforming at short time scales. In addition, the density ratio and Weber and Reynolds numbers involved in many practical multiphase flows, such as binary droplet collisions and melt-jet breakup, are usually very high, which further increases the complexity of the phenomena involved. Therefore, development of robust and accurate computational methods to capture the complex interfacial phenomena is crucial in the study of multiphase flows Fei et al. (2019).

Non-ideal fluids and multiphase flows. A major area of LB application is the simulation of a variety of multiphase and multicomponent flows [24–26]. Here, the main asset is the flexibility of the source term and/or local equilibria towards the inclusion of non-ideal interactions. A particularly popular expression is the one proposed by Shan and Chen,

$$\vec{F}(\vec{x}) = \psi(\vec{x}) \sum_{i=0}^{b} G(\vec{x}, \vec{x}+\vec{c}_i)\vec{c}_i \psi(\vec{x}+\vec{c}_i), \tag{3.25}$$

where $\psi(\vec{x})$ is a local functional of the fluid density $\rho(\vec{x})$.

By proper choice of this functional, the main features of non-ideal fluids, namely a non-monotonic equation of state supporting phase transitions and non-zero surface tension can be incorporated at a minimum programming effort. This simple variant opens up a vast scenario of applications involving multiphase and multicomponent flows, including foams and emulsions (see fig. 2). Needless to say, this variant comes with a number of limitations, such as spurious interface currents, which severly constrain the accessible range of

density ratios between the liquid and vapor phase. Yet, owing to its simplicity and efficiency, the method has gained increasing popularity over the years. Subsequent developments have improved significantly over the original version, but much remains to be done to gain further accuracy, especially in terms of multigrid/multiscale procedures at complex fluid interfaces. Another important issue is the incorporation of finite-temperature fluctuations for nanoscale flows.

existing multiphase LB models can be classified into four categories: the color-gradient model, the pseudopotential model, the it was shown by Li et al. that a non-orthogonal MRT-LBM free-energy model, and the mean-field model.

Among them, can retain the numerical accuracy while simplifying the implementation of its orthogonal counterpart. In parallel, the CLBM which can be viewed as a non-orthogonal MRT-LBM in the co-moving frame, has been shown to possess very good numerical stability for high Rayleigh number thermal flows,39 as well the pseudopotential model is considered in the present work due to its simplicity and computational efficiency. In this model, the interactions among populations of molecules are modeled by a density-dependent pseudopotential. Through interactions among the particles on the nearest-neighboring sites, phase separation and breakup and/or merging of phase interfaces can be achieved automatically. For further details about the multiphase LB models, interested readers are directed to some comprehensive review

The volume of fluid (VOF) model was used in this simulation. By tracking the volume fraction of each control unit, the VOF model can solve a single momentum equation. Thus, it can be used to simulate the fluid flow of two phase or multiphase, and it is typically applied to track the steady-state or transient gas–liquid interface. Each phase in the model has its own volume fraction a. The sum of the volume fraction of each phase in an arbitrary calculation area is 1 **?**.

3.8 Adaptive Mesh Refinement

3.9 Complex Fluids and Beyond

4 Visualization Methods

Visualization facilitates insight into data across many disciplines. It's an essential tool for displaying trends in data. These can be in a form of plots, graphs or colorful patterns drawn on screen. The target audience doesn't need to be just scientists, but also general public.

Scientific visualization is the use of computer graphics to create visual images that aid in the understanding of complex numerical representations of scientific concepts or results. Computational fluid dynamics (CFD) based numerical simulations often output massive amounts of data. These simulations often contain high-dimensional data in a three- dimensional volume. The display of phenomena associated with this data may involve complex three-dimensional structures.

Much of computer science is about transforming and representing information that enables or supports information processing, either by machines or humans. Graphics, visualization and human-computer interaction comprises the science, engineering and design of graphical, visual, informational and interactive representations for use by humans. At UC Davis, we study theories and principles fundamental to the construction and optimization of graphics, visualization and interactive systems, as well as applications and use of these technologies in a broader, interdisciplinary context.

4.1 Human-Computer Interface

4.1.1 SCADA

- ako sa scada systemy pouzivaju

- neexistuje prepojenie s virtualnou realitou

- future work co by sa dalo spravit

4.1.2 Principles of UI Design

4.1.3 From 2D to 3D User Interfaces

4.2 Post Hoc Visualization

Scientific visualization is usually performed as a post-processing task (**?**). The output from simulation is saved to disk and then the data are loaded into visualization software for further processing. The goal is to gather the insights from data, to communicate insights to other scientists, professionals or general public.

This approach has the benefit that the visualization software has access to all of the data from every step all at once, making algorithms and visualization workflows easier to develop.

Most of the parallelism in current scientific visualization tools relies on not just distributed memory parallelism, but specifically the message passing interface (MPI). MPI is heavy-weight, and requires a whole copy of the visualization program per process. As we transition our visualization codes to higher and higher concurrencies on the march to exascale, this overhead can exceed the system memory and disk space before any data is even loaded [19]. This revelation is important to consider when running a visualization tool at scales approaching those the size of the scientific simulations themselves. In order to achieve parallel scalability for massive threading, visualization algorithms will have to be redesigned [23]. The key in this redesign will be to focus on data model, data interdependencies, and portable performance.

Scientific visualization for exascale computing is very likely to require in situ processing. Traditional simulation checkpointing and post hoc visualization will likely be unsustainable on future systems at current trends due to the growing gap between I/O bandwidth and FLOPS. As a result, the majority of simulation data may be lost if in situ visualization techniques are not deployed. In situ visualization in this paradigm will be given

unprecedented access to simulation output, potentially being able to process all relevant simulation output at every simulation time step, allowing for very high temporal fidelity compared to traditional post hoc visualization. However, this access poses many challenges in terms of data management, resource management and sharing, algorithm development and design, and implementation approaches. Currently, the community primarily relies on two in situ techniques: tightly coupled (on the same resource as the simulation) and loosely coupled (not sharing resources with the simulation). Each of these approaches have positive and negative factors which affect their performance under different simulation, resource, and visualization type constraints. Meaning, that for every given visualization task, it is not generally known which method would give the best performance on every data type, architecture, and set of resource constraints. Due to the lack of research and development on this topic it is still an open research problem requiring future research **?**

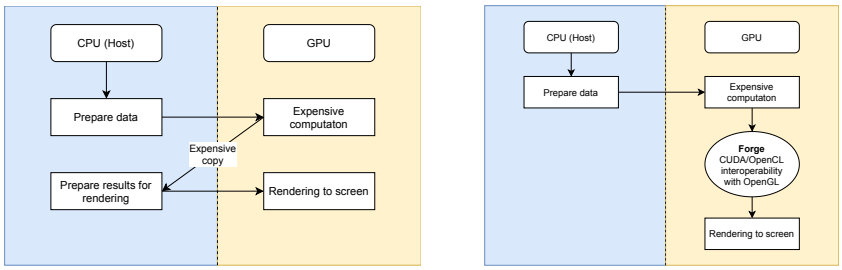
- The Visualization Toolkit
- ParaView

4.3 Real-time Visualization

Real-time fluid simulation, i.e. the ability to simulate a virtual system as fast as the real system would evolve, can benefit to many engineering application such as the optimisation of the ventilation system design in data centres or the simulation of pollutant transport in hospitals. And although real-time fluid simulation is an active field of research in computer graphics, these are generally focused on creating visually appealing animation rather than aiming for physical accuracy. The approach taken for this thesis is different as it starts from a physics based model, the lattice Boltzmann method, and takes advantage of the computational power of a graphics processing unit (GPU) to achieve real-time compute capability while maintaining good physical accuracy. Delbosc (n.d.).

The progress of scientific computations can be viewed in real-time thanks to the high-

performance OpenGL visualization library called Forge. It is developed by the same team behind ArrayFire and distributed together with their library for high-performance parallel computing. It is written specifically for use with GPU-accelerated applications as it doesn't require the expensive copying from GPU to CPU and back to GPU for rendering, but instead it builds on CUDA/OpenCL interoperability with OpenGL and allows for direct reading of the data on GPU **?**. Forge provides various plotting and visualization functions for 2D and 3D domains.



(a) High-performance visualization pipeline.

(b) High-performance visualization pipeline.

Figure 4 – 1: Differences between CPU-bound visualization with expensive copying bottleneck and high-performance GPU-bound visualization keeping full speeds of high-bandwidth of PCIe interface.

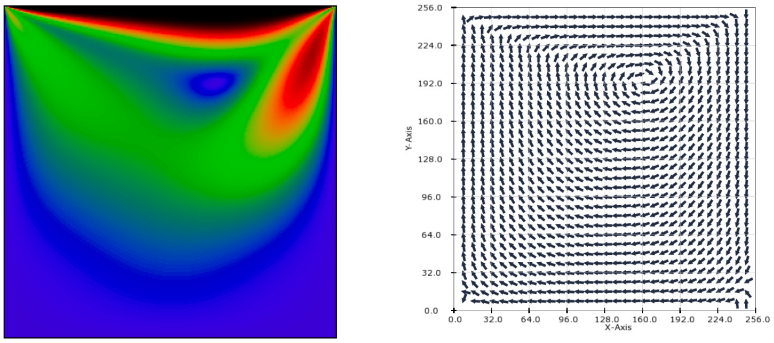
Practical scientific simulations for in-depth study of complex physical phenomena from real world, e.g. direct numerical simulation of cellular blood flow Kotsalos et al. (2019), requires higher accuracy. Instead of single-precision floating-point type (`f32`) computation, double-precision floating-point (`f64` has to be chosen. In LBM context, this practically doubles the amount of memory needed. For real-time visualizations of results with this type of precision, they have to be converted to a single-precision floating-point for Forge to effectively work with the data. In ArrayFire (and generally in programming), function for this operation is called `cast` (Alg. 1).

```
1    // C
2    af_array A_f64 = af_randu(100,100);
3    af_array B_f32;
4    cast(*B_f32, A_f64, f32);
```

```
// C++
array A_f64 = randu(100,100);
array B_f32 = A_f64.as(f32);
// Rust
let dims = af::Dim4::new(&[100, 100,  1, 1]);
let A_f64 = af::randu::<f64>(dims);
let B_f32 = A_f64.cast::<f32>();
```

Listing 1: Converting to single precision floating point for Forge visualization in C, C++ and Rust

In the current implementation, we used `image()` and `vector_field_2()` functions for visualizing density and velocity of on both lid-driven cavity and Kármán vortex street test cases (Fig. 4 – 2 and Fig. 4 – 3).



(a) Visualization of lid-driven cavity simulation.

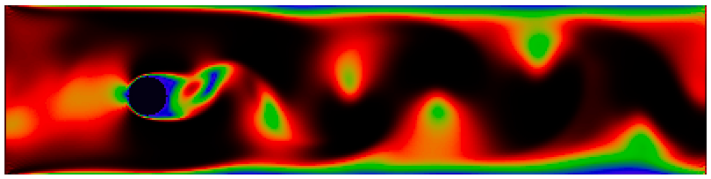
(b) Velocity vector field.

Figure 4 – 2: Lid-driven cavity test case at 128×128 resolution after 5000 iterations.

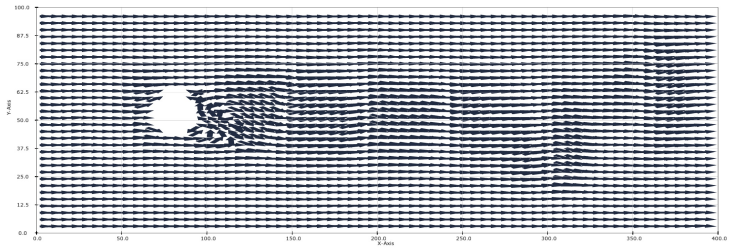
Visualizations can be zoomed, panned and rotated with a mouse.

4.4 Interactive Visualization

A real-time simulation platform is a manifestation of this concept, allowing run-time manipulation of geometrical and physical simulation variables. This enables users to rapidly and intuitively investigate different scenarios and design configurations. Ultimately, a complete interactive simulation package would be capable of simulating a multi-physics 3D environment in real-time to an application-appropriate degree of accuracy. However,



(a) Visualization of Kármán vortex street simulation.



(b) Velocity vector field (filtered for better presentation).

Figure 4 – 3: Kármán vortex street (channel flow past circle-shaped obstacle) test case at 1000×300 resolution after 5000 iterations.

significant inter-disciplinary research is required in reduced-order physical modelling, numerical methods, and software integration to realise a solution (**?**).

In order to achieve real-time flow simulation, numerical methods need to be selected carefully such that they can make full use of the capabilities of accelerated computing hardware. Our work focuses on the use of the lattice-Boltzmann Method (LBM):2 a CFD method ideally suited to acceleration on GPUs due to its spatial and temporal locality. GPU-LBM simulations have extremely high computational throughput compared with traditional CFD methods.

Recently, the increase of computational power and advances in general-purpose computing on GPUs (GPGPU) opened the door for real-time and interactive CFD simulations Delbosc (n.d.); Delbosc et al. (2014); Glessmer and Janßen (2017); Harwood and Revell (2017); **?**. Together with the performance and speed of the LBM method, it's now possible to compute several hundreds of iterations per second which makes an interaction with the simulation in progress possible Wang et al. (2019). Getting instant feedback

according to the change of various parameters in simulation gives researchers the ability to iterate faster toward the creation of accurate model, better understanding of underlying phenomena, or employing simulation within the control of industrial systems. It is therefore desirable to push the limits of execution speed of LBM simulations.

Main goal to implement LBM algorithm for our simulation software is the ability to high performance computation. Various optimization techniques exist to parallelize and optimize LBM algorithms Harwood and Revell (2017); Körner et al. (2006); Tran et al. (2017); Wang et al. (2019); Wittmann (2018); **?**. Wang et. al. Wang et al. (2019) was able to leverage performance and speed of the LBM method to reach interactive simulation time, i.e. providing several timestep per second to see the impact on interaction of user during a simulation in progress.

4.4.1 Steering the Running Simulation

4.4.2 Time Manipulation

4.5 Virtual and Augmented Reality

Non-immersive interactive visualization systems implemented for the conventional desktop and mouse are effective for moderately complex problems. Immersive virtual environments, by comparison, lie at the other end of the spectrum and permit looking around an object by moving one's head position.

Therefore, a fundamental difference between desktop-and-mouse virtual realia and immersive VR is that the latter is a true 3D representation that may be either viewer or object-centered while the first is exclusively viewer-centered. In other words, changes in the relative positions of a 2D object's components result from shifts in the viewer's perspective. The same may be true for objects viewed in a three dimensional environment, whether real or vir- tual. However, in such an environment, an object may also appear to change shape (e.g., through foreshortening), not due to an altered position of the viewer, but because the ob- ject itself has moved to a different position. Immersive virtual re-

ality displays aid in the unambiguous display of these structures by providing a rich set of spatial and depth cues. Virtual reality interface concepts allow the rapid and intuitive exploration of the volume containing the data, enabling the phenomena at various places in the volume to be ex- plored, as well as provide simple control of the visualization environment through inter- faces integrated into the environment (Bryson; 1996).

Desktop-and-mouse interfaces for 3D visualizations make it difficult to specify positions in three dimensions and do not provide unambiguous display of 3D structure. Virtual reality interfaces attempt to provide the most anthropomorphic interfaces possible - that means they must be human-conforming and should be designed to allow the most natural, unambiguous way of scientific exploration. They must include two components: display and user control. Scientific visualization makes particular demands on virtual reality displays. The phenomena to be displayed in a scientific visualization application often involve delicate and detailed structure, requiring high-quality, high-resolution full-color displays. A wide field of view is often desirable, because it allows the researcher to view how detailed structures are related to larger, more global phenomena.

Historically, the early attempts at using head-mounted virtual reality technologies started with CRT-based Binocular Omni-Oriented Monitor (BOOM) created by Fakespace Systems Inc. BOOM was a stereoscopic display device with screens and optical system housed in a box that is attached to a multi-link arm. Head tracking was accomplished via sensors in the links of the arm that holds the box.

Advent of commodity-level VR hardware like HTC Vive or Oculus Touch has made this technology accessible for meaningful applications. These headset utilize lasers and photosensitive sensors (HTC Vive) or cameras (Oculus Touch) for head and hands tracking and provide six degrees of freedom (6DoF) for movement in virtual environment. By immersing the user into the simulation itself, virtual reality reveals the spatially complex structures in computational science in a way that makes them easy to understand and study. But beyond adding a 3D interface, virtual reality also means greater computational complexity (Bryson; 1996). The ability to provide real-time interaction can

provide strong depth cues, either through allowing interactive rotations or through the use of head-tracked rendering. Applications and techniques are being developed to discern how immersive technology benefits visualization. The medical field provides an especially promising context for this development, as medical practitioners require a thorough understanding of specific 3D structures: human anatomy. Users may interact simultaneously with high resolution computed tomography (CT) scans and their corresponding, 3D anatomical structures.

Another frequently used type of immersive, interactive display technology nowadays is projection-screen-based Cave Automatic Virtual Environment (CAVE). These systems consists of 3 to 6 large displays positioned into a room-sized cube around the observer. The walls of a CAVE are typically made up of rear-projection screens, but recently the flat panel displays are commonly used. The floor can be a downward-projection screen, a bottom projected screen or a flat panel display. The projection systems are very high-resolution due to the near distance viewing which requires very small pixel sizes to retain the illusion of reality. The user wears 3D glasses inside the CAVE to see 3D graphics generated by the CAVE. People using the CAVE can see objects apparently floating in the air, and can walk around them, getting a proper view of what they would look like in reality. This is made possible by infrared cameras. Movement of the observer in the CAVE is tracked by the sensors typically attached to the 3D glasses and the video continually adjusts to retain the viewers perspective.

Many universities and engineering companies own and use CAVE systems. Researchers can use these systems to conduct their research topic in a more effective and accessible method. Engineers have found them useful in enhancing of a product development through prototyping and testing phases.

In field of mathematics, VR application named Cal (2019) is making serious progress. It is developed by a company Nanome, Inc. started by students from University of California San Diego. Team behind Calcflow is using VR to help students grasp the biggest ideas in vector calculus. Its features include visualizations of vector addition, cross prod-

uct, parametrized functions, spherical coordinate mapping, double and surface integrals. Beside Calcflow, they are implementing a VR platform specialized for atomic, molecular and protein visualization, built for researchers and scientists (Nan; 2019).

One can say that virtual reality established itself in many disciplines of human activities, as a medium that allows easier perception of data or natural phenomena appearance. In fact, theme of this dissertation was influenced by my previous work with using virtual reality for mathematics education at the university TODO: CITOVAT MATHWORLDVR CLANOK!!!

4.5.1 Virtual Reality in Steelmaking

Substantial amount of work in applying 3D visualizations and virtual reality for solving technological issues and bringing new trends into steelmaking industry is currently happening at Center for Innovation through Visualization and Simulation (CIVS) at Purdue University Northwest (located in Indiana, USA). CIVS has been globally recognized for its integrated and application-driven approaches through state-of-the-art simulation and virtual reality visualization technologies for providing innovative solutions to solve various university research problems, industry issues, as well as education. More than 350 projects that have been completed at the center from its inception in 2014 until today provided substantial educational and economic impact, resulting in more than 40 million US dollars (more than 36.1 million e at the time of this writing) in savings for companies. In collaboration with other universities and companies from steelmaking industry, they focus on research regarding integration of virtual reality with simulation technologies and high performance computing; application of simulation and visualization technologies to industrial processes for process design trouble-shooting and optimization to address the issues of productivity, energy, environment, and quality; and last but not least, development of advanced learning environments in virtual reality for training and education. They launched novel, industry-led association of steel manufacturers and stakeholders called Steel Manufacturing Simulation and Visualization Consortium (SMSVC).

Interesting application of combining 3D CFD smulation and virtual reality for visual inspection is pulverized coal injection (PCI) and coke combustion model. Research efforts between the Canadian government (CANMET), CIVS and the American Iron and Steel Institute (AISI) were conducted and resulted in modeling of the blowpipe and tuyere of the blast furnace. Combination of aforementioned technologies turned out to be powerful and provided detailed information of flow streams that were previously very difficult to measure. The CFD model shown in Fig. 2 – 12 was used to simulate PCI with natural gas co-injection in the lance, blowpipe and tuyere TODO: DOPLNIT CITACIU!!!.

Another very interesting project conducted at CIVS involved development of comprehensive package of modules for simulating multiple processes in blast furnace. 3D CFD model shown in Fig. 2 – 13 has been developed by Zheng and Hu (2014) specifically for simulating the blast furnace hearth. The campaign life of a blast furnace is highly dependent on residual thickness of refractory lining in the hearth. The progress of hearth lining erosion is greatly affected by hot metal flow patterns and heat transfer in refractory under different operating conditions. CFD model incorporates both the hot metal flow and conjugate heat transfer through the refractories. They achieved consistency of results between measured and calculated refractory temperature profiles, as the model has been extensively validated using measurement data from industry blast furnace. The virtual reality (VR) visualization technology has been used to analyze the velocity and temperature distributions and wear patterns of different furnaces and operating conditions. This interactive 3D visualization is shown in Fig. 2 – 14. Based on the results, it was possible to predict the inner profile of hearth and provide guidance to protecting the blast furnace hearth.

5 GPU Computing

Innovations in GPUs over the last decades was driven mainly by video games. They advanced from merely displaying pixels to being capable of doing mathematical com-

putations. After the introduction of programmable shaders and floating-point support on graphics processors, the dynamics has changed though. At first it opened the door for using GPUs in complex physics calculations like wind blowing, fluid flows, cloth movement, etc. in games but also for scientific simulation. This movement to leveraging GPU capabilities in serious engineering work became dubbed as general-purpose computing on GPUs (GPGPU), a term coined by NVIDIA's Distinguished Engineer Mark Harris.

GPU-based parallel computing reduces the time for heavy computing tasks like training a machine learning system on large data set, climate modeling, protein folding, drug discovery, or data analysis, by orders of magnitude (Pacheco, 2011; Storti and Yurtoglu, 2016). The massive parallelism achievable with GPUs pushed the developers to invest more time in creating programs that could use it for their advantage. These gains can be achieved at very reasonable costs in terms of both developer effort and the hardware required. It's now possible to speed-up scientific computations in a way that what took minutes or hours can now be done in seconds or even milliseconds.

Interesting case of engineering problem for which GPU computing is showing tremendous potential is solving differential equations while changing the initial or boundary conditions in real-time. More computationally intensive tasks like Monte Carlo simulations of molecular dynamics are still hard to speed up to an order of real-time simulation.

The main challenge of optimizing the GPU code is reducing the amount of copying between CPU and GPU.

5.1 Parallel Programming

For a long time, performance of computers was determined by the amount of transistors that could be fitted to a dense integrated circuit. By following the Moore's law, software developers could just wait for a predictable increase in transistors count. As speed of transistors increased, their power consumption also increased. This power is dissipated as heat, which poises a problem with reliability of the integrated circuit when it gets too hot.

And with transistors getting smaller, packing more of them together makes it approach the limits of integrated circuit's ability to dissipate heat.

Rather then building more complex monolithic processors, the industry has decided to put many simpler processors on a single chip, becoming multicore processors (Pacheco, 2011). Nowadays practically all processor have multiple cores. Unfortunately, most conventional programs were written for single-core systems that couldn't exploit the multiple cores. To effectively us them, programs have to ..

we need to either rewrite our serial programs so that they're parallel, so that they can make use of multiple cores, or write translation programs, that is, programs that will automatically convert serial programs into parallel programs

When we talk about parallel programming, we have to distinguish between CPU and GPU parallel programming.

When it comes to heavy computation, leveraging multiple cores that are nowadays ubiquitous in processing units or other hardware accelerators is necessary for achieving better performance.

Quickly skim through:
Distributed memory parallelism
shared memory parallelism
Message-passing, MPI, Pthreads, OpenMP
Rust multithreading with memory-safety guarantees (Rc, Arc)

Venovat sa skor GPU parallelism
Ako maju GPU rozlozenu memory a ako pracuju s threads vseobecne

We can consider matrix multiplication as a great exercise to showcase how code can be transformed from serial to parallel programming and differences between CPU and GPU implementation. Lets consider the problem of computing the product of two large, $N \times N$,

dense matrices (represented in row-major order). The naive CPU algorithm can look like in Alg. 2.

```
1 for (i=0;i<N;i++) {
2   for (j=0;j<N;j++) {
3     C[i,j] = 0;
4     for (k=0;k<N;k++) {
5       C[i,j] += A[i,k]*B[k,j];
6     }
7   }
8 }
```

Listing 2: Pseudocode with serial loop.

This example suffers from poor locality. Elements of B are accessed column-wise and therefore they are not in sequential order in memory. The row i could be removed from cache by the time the inner-most loop of j completes.

Algorithm is bandwidth limited with poor performance and low efficiency (time spent computation versus loading data).

On the GPU, we can write a program, that computes the matrix multiplication in a single pass. GPU texture will store 2×2 blocks of matrix in 4-component texels. The program will read 2 rows from matrix A and 2 columns of B to compute 4 elements of the output matrix C at once (Alg. 3).

```
1 for (k=0;k<N/2;k++) {
2   C[i,j].xyzw += A[i,k].xxzz*B[k,j].xyxy + A[i,k].yyww*B[k,j].zwzw;
3 }
```

Listing 3: Pseudocode with serial loop.

5.2 Heterogeneous Computing

5.3 CUDA

CUDA is a proprietary hardware and software platform for parallel computing on GPUs created by NVIDIA. It provides access to hardware-specific capabilities of graphics cards equipped with CUDA-enabled graphics processing units Storti and Yurtoglu (2016). The software platform provides a development kit (SDK) and application programming interface (API), build as a superset of C programming language. Since its launch it became a dominant proprietary framework for programming NVIDIA GPUs.

The GPU-based approach to massively parallel computing used by CUDA is also the core technology used in many of the world's fastest and most energy-efficient supercomputers. The key criterion has evolved from FLOPS (floating point operations per second) to FLOPS/watt (i.e., the ratio of computing productivity to energy consumption), and GPU-based parallelism competes well in terms of FLOPS/watt (Storti and Yurtoglu, 2016).

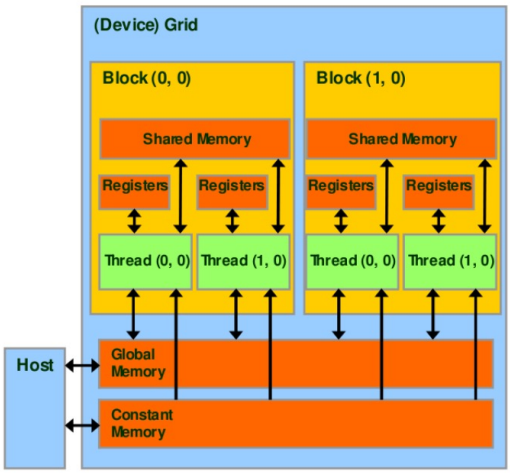


Figure 5 – 1: CUDA memory model

5.4 OpenCL

OpenCL (Open Computing Language) is a free and open-source standard for cross-platform, parallel programming of diverse accelerators like CPUs, GPUs, FPGAs, TPUs, etc. found in devices like personal computers or smartphones, embedded platforms, but also high-performance computing systems and supercomputers. In contrast to proprietary nature of CUDA, OpenCL is not tied to any specific hardware manufacturer, which makes its application good for plethora of different hardware.

5.5 Cross-platform GPU Programming

Despite the growing list of success stories, GPU software development adoption has had a slow rise. The slowness of the rise is attributable to the difficulty in programming GPUs (Malcolm et al., 2012).

(Karimi, n.d.)

CUDA and OpenCL APIs differ from each other. They can be considered as extensions of the C/C++ language and require significant experience in low-level C/C++ programming. To write optimized, parallel software, developers need to employ different techniques, specific to the API they choose, which adds substantial overhead when trying to port one to another in case of a need. In heterogeneous computing systems, trying to write optimized cross-platform code for different GPUs means writing multiple hardware-specific kernels in CUDA and OpenCL.

In this work, a high-performance, parallel computing library ArrayFire has been used to significantly simplify programming for GPUs. Its easy-to-use API provides high-level abstractions in the form of hundreds of functions. They are automatically converted to optimized, fast GPU kernels, utilizing just-in-time (JIT) compilation and lazy evaluation Chrzeszczyk (n.d.). ArrayFire's high-level object construct called Array is a data structure that acts as a container that represents memory stored on the device. On top of it, ArrayFire provides abstractions in the form of Unified Backend for working with differ-

ent computational backends. This way it is possible to switch to CPU, GPU, FPGA, or another type of accelerator at runtime **?** (Alg. 14), and permits programmers to write massively parallel applications in a high-level language with a much lower number of lines of code (LoC). Arrays (or matrices) of up to 4 dimensions can be created.

```
1 #include <arrayfire.h>
2 int main()
3 {
4   af::setBackend(AF_BACKEND_CUDA);
5   // af::setBackend(AF_BACKEND_OPENCL);
6   // af::setBackend(AF_BACKEND_CPU);
7   return 0;
8 }
```

Listing 4: C++ code for setting different computing backends.

Although ArrayFire is quite extensive, there remain many cases in which you may want to write custom kernels in CUDA or OpenCL. For example, you may wish to add ArrayFire to an existing code base to increase your productivity, or you may need to supplement ArrayFire's functionality with your own custom implementation of specific algorithms.

Since I'm targeting different types of GPUs in current work, I'll add examples of how to do the interoperability with OpenCL only. Working with CUDA looks similar in principle, but differs in API implementation (different naming conventions, slightly different semantics when launching kernels). ArrayFire manages its own context, queue, memory, and creates custom IDs for devices. As such, most of the interoperability functions focus on reducing potential synchronization conflicts between ArrayFire and OpenCL. There is some bookkeeping that needs to be done to integrate custom OpenCL kernel. If your kernels can share operate in the same queue as ArrayFire, you should:

1. obtain the OpenCL context, device, and queue used by ArrayFire,
2. obtain cl_mem references to Array objects,
3. load, build, and use your kernels,

4. return control of Array memory to ArrayFire.

Note, ArrayFire uses an in-order queue, thus when ArrayFire and your kernels are operating in the same queue, there is no need to perform any synchronization operations.

If your kernels needs to operate in their own OpenCL queue, the process is essentially identical, except you need to instruct ArrayFire to complete its computations using the sync function prior to launching your own kernel and ensure your kernels are complete using clFinish (or similar) commands prior to returning control of the memory to ArrayFire:

1. obtain the OpenCL context, device, and queue used by ArrayFire,
2. obtain cl_mem references to Array objects,
3. instruct ArrayFire to finish operations using sync,
4. load, build, and use your kernels,
5. instruct OpenCL to finish operations using `clFinish()` or similar commands,
6. return control of Array memory to ArrayFire.

Adding ArrayFire to an existing application is slightly more involved and can be somewhat tricky due to several optimizations we implement. The most important are as follows:

- ArrayFire assumes control of all memory provided to it.
- ArrayFire does not (in general) support in-place memory transactions.

To add ArrayFire to existing code you need to:

1. instruct OpenCL to complete its operations using clFinish (or similar),
2. instruct ArrayFire to use the user-created OpenCL Context,
3. create ArrayFire arrays from OpenCL memory objects,

4. perform ArrayFire operations on the Arrays,
5. instruct ArrayFire to finish operations using sync,
6. obtain cl_mem references for important memory,
7. continue your OpenCL application.

ArrayFire's memory manager automatically assumes responsibility for any memory provided to it. If you are creating an array from another RAII style object, you should retain it to ensure your memory is not deallocated if your RAII object were to go out of scope.

If you do not wish for ArrayFire to manage your memory, you may call the `Array::unlock` function and manage the memory yourself; however, if you do so, please be cautious not to call `clReleaseMemObj` on a `cl_mem` when ArrayFire might be using it!

It is fairly straightforward to interface ArrayFire with your own custom code. ArrayFire provides several functions to ease this process. The pointer returned by Array::device_ptr should be cast to cl_mem before using it with OpenCL opaque types. The pointer is a cl_mem internally that is force casted to pointer type by ArrayFire before returning the value to caller.

Additionally, the OpenCL backend permits the programmer to add and remove custom devices from the ArrayFire device manager (Table 5 – 1). These permit you to attach ArrayFire directly to the OpenCL queue used by other portions of your application.

Function	Purpose
add_device_context	Add a new device to ArrayFire's device manager
set_device_context	Set ArrayFire's device from cl_device_id & cl_context
delete_device_context	Remove a device from ArrayFire's device manager

Table 5 – 1: Functions for working with OpenCL context in ArrayFire-OpenCL interoperability scenario.

5.6 Accelerating Lattice Boltzmann Simulations with GPUs

Nowadays, the limiting factor is the cost of accessing data. It must be watched carefully in LBM applications, because it's going to be more and more demanding as the size of the problems to be simulated increases. A common LBM simulation program needs roughly 200 FLOPS per node and requires about 20 arrays. The amount of Bytes to be accessed in memory for one floating-point operation is in the order of 0.5 Bytes/FLOP.

To reduce the amount of GPU memory accesses, the data is loaded into extremely fast memory called *cache* that is designed to keep up with the CPU. Useful simulations need large amounts of data to be processed, but loading all of them into cache is impossible as they tend to be limited to few Megabytes. Lattice's computational domain is usually represented by 2D or 3D grid of nodes, each carrying multiple data. Computer memory is one dimensional, which means that the element of 3D array of size N^3 lies $4 \times N^2$ bytes away from physically contiguous element. If we consider x, y and z axis of 3D domain, it is fine when searching for neighboring nodes in x direction, which, as an innermost index, runs first. But searching for neighbors in z direction for element $f(x,y,z+1)$ means that the distance in 1D memory (called *stride*) would be large. To be able to quickly load such neighbor in z direction, we would need to have whole stride loaded in cache, which would require tens of Megabytes for large simulations of $N \sim 1000$. Optimizing memory access is one of the most critical issues in accelerating large-scale LBM simulations.

Developers have to be careful with the memory limitations, even though GPUs provide high memory bandwidth, as LBM algorithms tend to consume large amounts of memory for storing the data. GPU architecture is designed for high data throughput thanks to the combination of Single Instruction, Multiple Data (SIMD) execution model and multithreading, called Single Instruction, Multiple Threads (SIMT). With the parallel nature of the LBM algorithm, CFD simulations that use it can achieve high speeds not just on HPC systems but also PC workstations with a single GPU. For applications of real-time or online interactive visualization of the running simulation, getting to high frame rates is easier to achieve on such workstation PCs because of the high bandwidth of the PCIe

slot. On networked HPC systems, the transfer speeds are limited by network throughput and higher latency. Therefore, transferring data from GPU on the HPC system back to the PC client for visualization is much slower Linxweiler et al. (2010). In this study, we use GPUs that have between 70 and 900 GB/s theoretical maximum memory bandwidth.

To get more computational power from GPUs, algorithm optimization techniques for parallel computing need to be considered. Compiled code needs to be vectorized and multi-threaded to leverage parallel capabilities in massively parallel architectures Delbosc et al. (2014). This is typically done by using specific compilation commands to automatically vectorize code (NVC++ compiler from NVIDIA with `stdpar`), writing GPU-specific code (compute kernels) with frameworks like CUDA and OpenCL, or compiling both CUDA and OpenCL from the same code without specialized compiler directives using cross-platform library like ArrayFire. In multi-GPU setups, Message Passing Interface (e.g. MPI, OpenMP) is usually employed.

There has been an increase in studies focused on optimizing the execution speed of LBM algorithms, after the idea of using GPGPUs for CFD simulations started gaining traction more than a decade ago. It's been bolstered by the LBM's advantage in the locality of computations since only the values from neighbouring nodes are needed.

In this space, the most used APIs for programming GPUs are CUDA and OpenCL. CUDA (Compute Unified Device Architecture) is a proprietary API used to program NVIDIA GPUs Storti and Yurtoglu (2016). OpenCL (Open Computing Language) is an open standard that supports different hardware from various vendors on the market **?**.

Recently, multiple projects regarding 2D and 3D LBM simulations used CUDA or OpenCL for their parallel implementation targeting GPUs Delbosc (n.d.); Delbosc et al. (2014); *FULL GPU Implementation of Lattice-Boltzmann Methods with Immersed Boundary Conditions for Fast Fluid Simulations* (2017); Harwood et al. (2018); Harwood and Revell (2017); Januszewski and Kostur (2014); Kotsalos et al. (2019); Szőke et al. (2017); **?**. There is increasing amount of studies on memory access efficiency and optimization tech-

niques Herschlag et al. (2018); Tran et al. (2017). However, to accomplish near-optimal performance, it's extraordinary amount of work. Programming software for GPGPU is still very difficult **?**. Developers need to optimize for specific hardware and therefore have to know each architecture thoroughly. For this reason, such hardware-specific implementations in cross-platform code tend to get very complex.

ArrayFire library can help by automatically leveraging the best hardware features available on multiple architectures, hiding the hardware-specific optimizations. Developers can write code in a high-level language like C++, Rust, or Python (for which the ArrayFire library wrapper exists) and have it compiled for CPU, GPU, or other accelerators like FPGA.

6 Implementation of a Simulation Software

6.1 Technology Stack

The ArrayFire library is used for high-performance, cross-platform implementation of Lattice Boltzmann algorithm.

6.1.1 Cross-platform Development with ArrayFire

Programmers and Data Scientists want to take advantage of fast and parallel computational devices. Writing vectorized code is necessary to get the best performance out of the current generation parallel hardware and scientific computing software. However, writing vectorized code may not be immediately intuitive. ArrayFire provides many ways to vectorize a given code segment. In this chapter, we present several methods to vectorize code using ArrayFire and discuss the benefits and drawbacks associated with each method.

ArrayFire is a vectorized library. Most functions operate on Arrays as a whole i.e. on all elements in parallel.

ArrayFire provides several different methods for manipulating arrays and matrices. The functionality includes:

- `moddims()` - change the dimensions of an array without changing the data
- `flat()` - flatten an array to one dimension
- `flip()` - flip an array along a dimension
- `join()` - join up to 4 arrays
- `reorder()` - changes the dimension order within the array
- `shift()` - shifts data along a dimension
- `tile()` - repeats an array along a dimension
- `transpose()` - performs a matrix transpose

The simulation part of the software solution presented in this thesis was built using ArrayFire, a cross-platform library for developing parallel algorithms. I picked it after considering other approaches like using bunch of different helper libraries together for vectorization and cross-compilation of C++ code for different GPU platforms. C++ is battle-tested language with plethora of available libraries and has been extensively used in CFD simulation software development. Although, I would have to optimize the code for specific GPUs and their distinctive features myself, writing extensive amount of code in the process. This is where the ArrayFire library shines the most: it provides hundreds of hand-tuned, low-level optimized functions for various domains including vector algorithms, image processing, computer vision, signal processing, linear algebra, statistics, and more. Writing complex algorithms that are automatically optimized for all kinds of hardware accelerators (GPUs, CPUs, FPGAs etc.) can be done in few lines of code instead of writing hundreds or thousands of lines of kernel code.

ArrayFire main API is written in C with additional wrappers for other languages like C++, Python, Rust and JavaScript. It provides hundreds of hand-tuned, low-level optimized

functions for various domains including vector algorithms, image processing, computer vision, signal processing, linear algebra, statistics, and more.

It helped with simplifying the overall implementation and reducing lines of code (LoC).

To start developing cross-platform software with ArrayFire, first it has to be installed on a development machine. It can be installed by either using binary installer for Windows, Linux or OSX, or built from source.

The high-performance visualization library called Forge is bundled with the installer. It can be disabled in the installation process if user doesn't want to install it. I used it in throughout the development process to test early protypes, check if the output is visually correct and also benchmark the simulation software for what computational domain the visualization stays fast enough to show real-time output as the simulation is running.

6.1.2 Rust as C/C++ Alternative

As an alternative to C++ programming language, I considered Rust. It has been chanted around the programming world s a language that could one day replace C or C++ as a go-to languages for writing performant low-level code. The problem with going down the road of using Rust is that it's still considered a new language with developing ecosystem of libraries, often lacking the ones that are readily available in C/C++ ecosystem. Good thing is that it's growing everyday. After reading through tens of articles about pros and cons of the language, I had to test it first and try developing some prototypes to find out for myself if it's worth developing such a complex application.

The main proposition of Rust language is that it helps you write faster, more reliable software. Traditionally, developers who were looking for the best raw performance as they can get to squeeze out of their programs by aggressively optimizing it, were mostly reaching out for C++ as it's possible to get down to low-level if needed. The object-oriented nature of C++ on the other hand allows for constructing code that is easier to reason about. But still, the high-level ergonomics and low-level control are often at odds in pro-

gramming language design **?**. Developers are still responsible for managing the memory in C++ applications. This can lead to problems if application is not well architected or developers not being disciplined about memory cleanup after use.

Rust challenges the status quo of high-level ergonomics with low-level performance in one language. Its unique feature called ownership allows the compiler to make memory safety guarantees without the need of an garbage collector. When building low-latency software, garbage collection adds unwanted pauses to the execution time.

Through balancing powerful technical capacity and a great developer experience, Rust gives you the option to control low-level details (such as memory usage) without all the hassle traditionally associated with such control. **?**

zero-cost abstractions, borrowing, borrow-checker, memory safety, easy and memory-safe multi-threading, parallel iterators with rayon (par_iter) generics (compare with templates in C++), pattern matching, macros, foreign function interface FFI with C, can use C libraries if needed

Rust is for people who crave speed and stability in a language. By speed, we mean the speed of the programs that you can create with Rust and the speed at which Rust lets you write them. The Rust compiler's checks ensure stability through feature additions and refactoring. This is in contrast to the brittle legacy code in languages without these checks, which developers are often afraid to modify. By striving for zero-cost abstractions, higher-level features that compile to lower-level code as fast as code written manually, Rust endeavors to make safe code be fast code as well. **?**

Overall, Rust's greatest ambition is to eliminate the trade-offs that programmers have accepted for decades by providing safety and productivity, speed and ergonomics. **?**

It wasn't always so clear, but the Rust programming language is fundamentally about empowerment: no matter what kind of code you are writing now, Rust empowers you to reach farther, to program with confidence in a wider variety of domains than you did before.

Take, for example, “systems-level” work that deals with low-level details of memory management, data representation, and concurrency. Traditionally, this realm of programming is seen as arcane, accessible only to a select few who have devoted the necessary years learning to avoid its infamous pitfalls. And even those who practice it do so with caution, lest their code be open to exploits, crashes, or corruption.

Rust breaks down these barriers by eliminating the old pitfalls and providing a friendly, polished set of tools to help you along the way. Programmers who need to “dip down” into lower-level control can do so with Rust, without taking on the customary risk of crashes or security holes, and without having to learn the fine points of a fickle toolchain. Better yet, the language is designed to guide you naturally towards reliable code that is efficient in terms of speed and memory usage.

Programmers who are already working with low-level code can use Rust to raise their ambitions. For example, introducing parallelism in Rust is a relatively low-risk operation: the compiler will catch the classical mistakes for you. And you can tackle more aggressive optimizations in your code with the confidence that you won’t accidentally introduce crashes or vulnerabilities.

But Rust isn’t limited to low-level systems programming. It’s expressive and ergonomic enough to make CLI apps, web servers, and many other kinds of code quite pleasant to write — you’ll find simple examples of both later in the book. Working with Rust allows you to build skills that transfer from one domain to another; you can learn Rust by writing a web app, then apply those same skills to target your Raspberry Pi.

?

6.1.3 Hardware

6.2 Implementation of Lattice Boltzmann Method for GPUs

Implementation of D2Q9 stencil which mean 2 dimensional computation domain with each node having 9 discrete speeds pointing in the directions of where the particle distri-

bution can move.

In this section I'll take you through each step in the program lifecycle and show how different parts are constructed so they can be called from the Unity game engine (as a Native Plugin).

The code is constructed in a way that it is easy to interface with the Unity.

6.2.1 Initialization

The simulation program needs to be initialized at the start with correct parameters.

```
1 #include <arrayfire.h>
2 int main()
3 {
4   af::setBackend(AF_BACKEND_CUDA);
5   // af::setBackend(AF_BACKEND_OPENCL);
6   // af::setBackend(AF_BACKEND_CPU);
7   return 0;
8 }
```

Listing 5: C++ code for setting different computing backends.

6.2.2 Boundary Conditions

```
1 #include <arrayfire.h>
2 int main()
3 {
4   af::setBackend(AF_BACKEND_CUDA);
5   // af::setBackend(AF_BACKEND_OPENCL);
6   // af::setBackend(AF_BACKEND_CPU);
7   return 0;
8 }
```

Listing 6: C++ code for setting different computing backends.

6.2.3 Initial Conditions

Physical and lattice parameters that are supplied to the simulation at the start so it can begin with something.

There is difference between initial conditions in BGK and MRT code.

- BGK

```
1 #include <arrayfire.h>
2 int main()
3 {
4  af::setBackend(AF_BACKEND_CUDA);
5  // af::setBackend(AF_BACKEND_OPENCL);
6  // af::setBackend(AF_BACKEND_CPU);
7  return 0;
8 }
```

Listing 7: C++ code for setting different computing backends.

- MRT

```
1 #include <arrayfire.h>
2 int main()
3 {
4  af::setBackend(AF_BACKEND_CUDA);
5  // af::setBackend(AF_BACKEND_OPENCL);
6  // af::setBackend(AF_BACKEND_CPU);
7  return 0;
8 }
```

Listing 8: C++ code for setting different computing backends.

6.2.4 Streaming

Streaming in generally implemented as a separate step, implemented as a individual kernel, but also can be bundled together with the collision kernel to save memory, therefore boost speed.

In out program, the streaming is implemented as a shift of indices of neighbouring nodes. In this fashion, the index is pre-computed at the beginning and then used throughout the simulation lifetime to index into the particle distributions of neighbouring nodes.

```
1     let ci: Array<u64> = (range::<u64>(dim4!(1, 8), 1) + 1) * total_nodes;
2     // Indices ordered to reflect a direction to the neighbouring nodes
3     let nbidx = Array::new(&[2, 3, 0, 1, 6, 7, 4, 5], dim4!(8));
4     let span = seq!();
5     let nbi: Array<u64> = view!(ci[span, nbidx]);
6
7     let main_index = moddims(&range(dim4!(total_nodes * 9), 0), dim4!(nx,
        ny, 9));
8     let nb_index = flat(&stream(&main_index));
```

Listing 9: Pre-computed streaming step in the way of shifting indices during the program intialization phase.

2D

```
1   fn stream(f: &Array<FloatNum>) -> Array<FloatNum> {
2     let mut pdf = f.clone();
3     eval!(pdf[1:1:0, 1:1:0, 1:1:1] = shift(&view!(f[1:1:0, 1:1:0, 1:1:1]),
        &[1, 0, 0, 0]));
4     eval!(pdf[1:1:0, 1:1:0, 2:2:1] = shift(&view!(f[1:1:0, 1:1:0, 2:2:1]),
        &[0, 1, 0, 0]));
5     eval!(pdf[1:1:0, 1:1:0, 3:3:1] = shift(&view!(f[1:1:0, 1:1:0, 3:3:1]),
        &[-1, 0, 0, 0]));
6     eval!(pdf[1:1:0, 1:1:0, 4:4:1] = shift(&view!(f[1:1:0, 1:1:0, 4:4:1]),
        &[0, -1, 0, 0]));
7     eval!(pdf[1:1:0, 1:1:0, 5:5:1] = shift(&view!(f[1:1:0, 1:1:0, 5:5:1]),
        &[1, 1, 0, 0]));
8     eval!(pdf[1:1:0, 1:1:0, 6:6:1] = shift(&view!(f[1:1:0, 1:1:0, 6:6:1]),
        &[-1, 1, 0, 0]));
9     eval!(pdf[1:1:0, 1:1:0, 7:7:1] = shift(&view!(f[1:1:0, 1:1:0, 7:7:1]),
        &[-1, -1, 0, 0]));
10    eval!(pdf[1:1:0, 1:1:0, 8:8:1] = shift(&view!(f[1:1:0, 1:1:0, 8:8:1]),
        &[1, -1, 0, 0]));
11    pdf
12  }
```

Listing 10: Streaming with `shift` function for two dimensions with 9 discrete speeds (D2Q9).

3D

```
fn stream(f: &Array<FloatNum>) -> Array<FloatNum> {
  let mut pdf = f.clone();
  // nearest-neighbours
  eval!(pdf[1:1:0, 1:1:0, 1:1:0, 1:1:1] = shift(&view!(f[1:1:0, 1:1:0, 1:1:0, 1:1:1]), &[ 1, 0, 0, 0]));
  eval!(pdf[1:1:0, 1:1:0, 1:1:0, 2:2:1] = shift(&view!(f[1:1:0, 1:1:0, 1:1:0, 2:2:1]), &[ -1, 0, 0, 0]));
  eval!(pdf[1:1:0, 1:1:0, 1:1:0, 3:3:1] = shift(&view!(f[1:1:0, 1:1:0, 1:1:0, 3:3:1]), &[ 0, 1, 0, 0]));
  eval!(pdf[1:1:0, 1:1:0, 1:1:0, 4:4:1] = shift(&view!(f[1:1:0, 1:1:0, 1:1:0, 4:4:1]), &[ 0, -1, 0, 0]));
  eval!(pdf[1:1:0, 1:1:0, 1:1:0, 5:5:1] = shift(&view!(f[1:1:0, 1:1:0, 1:1:0, 5:5:1]), &[ 0, 0, 1, 0]));
  eval!(pdf[1:1:0, 1:1:0, 1:1:0, 6:6:1] = shift(&view!(f[1:1:0, 1:1:0, 1:1:0, 6:6:1]), &[ 0, 0, -1, 0]));
  // next-nearest neighbours
  // xy plane
  eval!(pdf[1:1:0, 1:1:0, 1:1:0, 7:7:1] = shift(&view!(f[1:1:0, 1:1:0, 1:1:0, 7:7:1]), &[ 1, 1, 0, 0]));
  eval!(pdf[1:1:0, 1:1:0, 1:1:0, 8:8:1] = shift(&view!(f[1:1:0, 1:1:0, 1:1:0, 8:8:1]), &[ -1, 1, 0, 0]));
  eval!(pdf[1:1:0, 1:1:0, 1:1:0, 9:9:1] = shift(&view!(f[1:1:0, 1:1:0, 1:1:0, 9:9:1]), &[ 1, -1, 0, 0]));
  eval!(pdf[1:1:0, 1:1:0, 1:1:0, 10:10:1] = shift(&view!(f[1:1:0, 1:1:0, 1:1:0, 10:10:1]), &[ -1, -1, 0, 0]));
  // xz plane
  eval!(pdf[1:1:0, 1:1:0, 1:1:0, 11:11:1] = shift(&view!(f[1:1:0, 1:1:0, 1:1:0, 11:11:1]), &[ 1, 0, 1, 0]));
  eval!(pdf[1:1:0, 1:1:0, 1:1:0, 12:12:1] = shift(&view!(f[1:1:0, 1:1:0, 1:1:0, 12:12:1]), &[ -1, 0, 1, 0]));
  eval!(pdf[1:1:0, 1:1:0, 1:1:0, 13:13:1] = shift(&view!(f[1:1:0, 1:1:0, 1:1:0, 13:13:1]), &[ 1, 0, -1, 0]));
  eval!(pdf[1:1:0, 1:1:0, 1:1:0, 14:14:1] = shift(&view!(f[1:1:0, 1:1:0, 1:1:0, 14:14:1]), &[ -1, 0, -1, 0]));
  // yz plane
  eval!(pdf[1:1:0, 1:1:0, 1:1:0, 15:15:1] = shift(&view!(f[1:1:0, 1:1:0, 1:1:0, 15:15:1]), &[ 0, 1, 1, 0]));
  eval!(pdf[1:1:0, 1:1:0, 1:1:0, 16:16:1] = shift(&view!(f[1:1:0, 1:1:0, 1:1:0, 16:16:1]), &[ 0, -1, 1, 0]));
  eval!(pdf[1:1:0, 1:1:0, 1:1:0, 17:17:1] = shift(&view!(f[1:1:0, 1:1:0, 1:1:0, 17:17:1]), &[ 0, 1, -1, 0]));
  eval!(pdf[1:1:0, 1:1:0, 1:1:0, 18:18:1] = shift(&view!(f[1:1:0, 1:1:0, 1:1:0, 18:18:1]), &[ 0, -1, -1, 0]));
  // next next-nearest neighbours
  eval!(pdf[1:1:0, 1:1:0, 1:1:0, 19:19:1] = shift(&view!(f[1:1:0, 1:1:0, 1:1:0, 19:19:1]), &[ 1, 1, 1, 0]));
  eval!(pdf[1:1:0, 1:1:0, 1:1:0, 20:20:1] = shift(&view!(f[1:1:0, 1:1:0, 1:1:0, 20:20:1]), &[ -1, 1, 1, 0]));
  eval!(pdf[1:1:0, 1:1:0, 1:1:0, 21:21:1] = shift(&view!(f[1:1:0, 1:1:0, 1:1:0, 21:21:1]), &[ 1, -1, 1, 0]));
  eval!(pdf[1:1:0, 1:1:0, 1:1:0, 22:22:1] = shift(&view!(f[1:1:0, 1:1:0, 1:1:0, 22:22:1]), &[ -1, -1, 1, 0]));
  eval!(pdf[1:1:0, 1:1:0, 1:1:0, 23:23:1] = shift(&view!(f[1:1:0, 1:1:0, 1:1:0, 23:23:1]), &[ 1, 1, -1, 0]));
```

```
32      eval!(pdf[1:1:0, 1:1:0, 1:1:0, 24:24:1] = shift(&view!(f[1:1:0, 1:1:0,
          1:1:0, 24:24:1]), &[ -1, 1, -1, 0]));
33      eval!(pdf[1:1:0, 1:1:0, 1:1:0, 25:25:1] = shift(&view!(f[1:1:0, 1:1:0,
          1:1:0, 25:25:1]), &[ 1, -1, -1, 0]));
34      eval!(pdf[1:1:0, 1:1:0, 1:1:0, 26:26:1] = shift(&view!(f[1:1:0, 1:1:0,
          1:1:0, 26:26:1]), &[ -1, -1, -1, 0]));
35
36      pdf
37    }
```

Listing 11: Streaming with `shift` function for three dimensions with 27 discrete speeds (D3Q27).

6.2.5 Collision

- BGK

```
1  #include <arrayfire.h>
2  int main()
3  {
4   af::setBackend(AF_BACKEND_CUDA);
5   // af::setBackend(AF_BACKEND_OPENCL);
6   // af::setBackend(AF_BACKEND_CPU);
7   return 0;
8  }
```

Listing 12: C++ code for setting different computing backends.

- MRT

```
1  #include <arrayfire.h>
2  int main()
3  {
4   af::setBackend(AF_BACKEND_CUDA);
5   // af::setBackend(AF_BACKEND_OPENCL);
6   // af::setBackend(AF_BACKEND_CPU);
7   return 0;
8  }
```

Listing 13: C++ code for setting different computing backends.

6.2.6 External Force

The implementation of external force on the fluid simulation is straightforward - it's a simple addition to the ...

```
1 #include <arrayfire.h>
2 int main()
3 {
4   af::setBackend(AF_BACKEND_CUDA);
5   // af::setBackend(AF_BACKEND_OPENCL);
6   // af::setBackend(AF_BACKEND_CPU);
7   return 0;
8 }
```

Listing 14: C++ code for setting different computing backends.

6.3 GPU Optimizations

In the following section, we describe simple optimizations employed in our LBM-based solver. To achieve high throughput on different parallel architectures, a large portion of time-consuming, hardware-specific, low-level optimizations are done automatically by ArrayFire. The underlying JIT compilation engine converts expressions into the smallest number of CUDA or OpenCL kernels, but to decrease the number of kernel calls and unnecessary global memory operations, it tries to merge cooperating expressions into a single kernel. However, not everything can be optimized automatically. There are some specifics between LBM algorithms where some optimizations have a large impact on the performance of the simulation, namely coalesced memory writes resulting from better data organization scheme, removing branch divergence, improvement of cache locality, and thread parallelism.

ArrayFire is a high performance software library for parallel computing with an easy-to-use API. ArrayFire abstracts away much of the details of programming parallel architectures by providing a high-level container object, the Array, that represents data stored on a CPU, GPU, FPGA, or other type of accelerator. This abstraction permits developers to write massively parallel applications in a high-level language where they need not be concerned about low-level optimizations that are frequently required to achieve high throughput on most parallel architectures.

(**?**)

Going on from ArrayFire's version 3.2., the Unified backend was introduced. It allows developers to change between different backends CUDA, OpenCL and CPU

ArrayFire provides the only GPU-enabled data-parallel loop: $gfor$. Use $gfor$ in your code in place of standard for-loops to batch process data parallel operations (Malcolm et al., 2012). You can think of gfor as performing auto-vectorization of your code, e.g. you write a gfor-loop that increments every element of a vector but behind the scenes ArrayFire rewrites it to operate on the entire vector in parallel (**?**). For example, the following for-loop calculates several matrix multiplications serially:

```
1     for (int i = 0; i < n; ++i) {
2       A(i) = A(i) + 1;
3     }
```

Listing 15: Pseudo-code with imerative for loop

The same three matrix multiplications can be carried out in one pass instead of three by utilizing a gfor-loop,

```
1     A = constant(1,n,n);
2     gfor (seq i, n) {
3       A(i) = A(i) + 1;
4     }
```

Listing 16: Pseudo-code with if-statement removed

Whereas the former loop serially computes each matrix multiplication, the latter loop computes all matrix multiplications in one pass. Similarly, gfor can be used for other

such embarrassingly parallel codes in a straightforward fashion. A good way of thinking about gfor is to consider it as syntactic sugar: gfor serves as an iterative style of writing otherwise vectorized algorithms (Malcolm et al., 2012).

6.3.1 Data Organization

To ensure the most efficient memory throughput when programming for GPU is to ensure that memory access is coalesced Tran et al. (2017). There are two common patterns for data organization: Array of Structures (AoS) and Structure of Arrays (SoA). Therefore if we consider a 1D array, in AoS, 9 distributions of each node occupy 9 consecutive elements of the array, and in SoA, the value of one distribution of all nodes is arranged consecutively in memory, then next distribution in all nodes and so on. In a GPU-based LBM algorithm, we need to store values of distribution functions for each node in the computational domain represented by a grid. Using Structure of Arrays (SoA) is significantly faster than Array of Structures (AoS) Delbosc et al. (2014); Tran et al. (2017). In order to represent the 2-dimensional grid of cells, we use the 1-dimensional array which has $N_x * N_y * Q$ elements, where N_x, N_y, are width and height of the grid and Q is the number of directions of each node Tran et al. (2017). This way the cache use is considerably improved **?**. For example, in lid-driven cavity of lattice dimensions $N_x = 64$, $N_y = 64$, employing 9 velocity distributions (D2Q9), we have the 1D array of $64 \times 64 \times 9 = 36864$ elements.

To create a SoA structure with 1D array to store particle distributions in all directions along whole computational domain, we can set first dimension of Array construct straightforwardly to the $n_x * n_y * Q$ number of elements, but for easier index creation further down the line, it's better to create 3D array initially and flatten it when used for heavy computation of actual simulation:

```
1 unsigned nx = 64, ny = 64, dirs = 9;
2 // SoA 1D array
3 array f_1d_soa = constant(0, nx * ny * dirs);
4 // 3D array, flattened to SoA 1D array
```

```
5  array f_3d = constant(0, nx, ny, dirs);
6  array f_1d_soa = flat(f_3d)
```

Listing 17: Creating SoA structure representation of D2Q9 lattice with ArrayFire in C++.

For streaming operation in LBM, we use ArrayFire's `shift` function for each direction of particle distributions. Instead of running this function on every iteration, we create two indices for access to "current" nodes and their neighbouring nodes at the initialization phase of the solver and store them for the whole lifetime of the simulation. Streaming is then as easy as accessing particle distributions in a 1D array with neighbours index. In ArrayFire, indexing is also executed in parallel, but is not a part of a JIT compilation. Instead, it is a handwritten optimized kernel. Any JIT code that is fed to indexing is evaluated in a single kernel if possible.

```
1  unsigned nx = 64, ny = 64, dirs = 9;
2  unsigned total_nodes = x * ny;
3
4  // Index of all directions except center point 0
5  array CI = (range(dim4(1,8),1)+1) * total_nodes;
6  // Neighboring nodes index
7  unsigned int nb_index_arr[8] = {2,3,0,1,6,7,4,5};
8  array nbidx(8, nb_index_arr);
9  array NBI = CI(span,nbidx);
10
11 dim4 dims = dim4(total_nodes*dirs);
12 array main_index = moddims(range(dims),nx,ny,dirs);
13 array nb_index = flat(stream(main_index));
```

Listing 18: Creating "current" (or main) index and neighboring index.

The result of streamed distribution functions is stored temporarily without overwriting the previous ones (Alg. 19):

```
1  array F_streamed = F(nb_index);
```

Listing 19: Streaming step.

Targeting any of the 9 direction in 1D array can be done by computing $[node\_position] + [total\_nodes * directions]$.

6.3.2 Removing Branch Divergence

When writing classical, imperative code, handling control flow is usually done by using `if-else` blocks, creating different possible branches. In multithreaded execution model like SIMT used in GPUs, the processor's threads execute different paths of the control flow, leading to poor utilisation due to thread-specific control flow using masking Delbosc et al. (2014). Branch divergence is a major cause for performance degradation in GPGPU applications **?**. To keep the flow coherent for the processing threads, it's recommended to remove `if-else` blocks from the code (Alg. 21).

```
1 // With branch divergence
2 if (cell_type == ”solid”) {
3   x = a;
4 } else {
5   x = b;
6 }
7 // Without branch divergence
8 let is_solid = cell_type == ”solid”;
9 x = a * is_solid + b * (!is_solid);
```

Listing 20: Pseudo-code showcasing the removal of branch divergence by removing if statement.

In practical LBM application, branch divergence occurs when doing different computations on different types of the nodes in computational domain, e.g. "`if fluid node, do computation, else do nothing`". Branch removal in the C++ version of ArrayFire applications is shown in Alg. 21.

```
1 // Node types (0 = solid, 1 = fluid)
2 unsigned types[] = {0,0,0,1,1,1};
3 array T(3, 3, types);
4
```

```
5  // original array
6  array A = randu(3, 3);
7  // part of the domain to be replaced
8  array FLUID = constant(1, 3, 3);
9  // new values
10 array B = randu(3, 3);
11
12 array cond = FLUID == T;
13 array out = A * (1 - cond) + cond * B;
```

Listing 21: Example C++ code of removing branch divergence using ArrayFire.

In Rust version, the branching removal is achieved in the same manner, but at the end to use the condition, function `select` can be used:

```
1      let out = af::select(&A, &cond, &B);
```

Listing 22: Example Rust code of removing branch divergence using ArrayFire.

In LBM simulations, we're also concerned with setting up boundary conditions. It's necessary to tell the solver which cells are solid (e.g. for doing bounce-back in some step down the line) and which are other types of fluids. For the simplest case, let's consider only two types of cells - solid and fluid. Boolean mask, in this case represented as integers (fluid as 0 and solid as 1), is instantiated in `mask` variable. Indexes of the solid nodes within computational domain can be easily found with `where` function:

```
1  #include <stdio.h>
2  #include <arrayfire.h>
3  using namespace af;
4  int main(){
5    int nx = 400, ny = 100;
6    array mask = constant(0,nx,ny);
7    // Rectangle obstacle of size 2x20 cells
8    mask(seq(100,102),seq(40,60))) = 1;
9    mask(span,0) = 1; // Top wall
10   mask(span,end) = 1; // Bottom wall
```

```
11    // Get the indices of each solid cell
12    array solids = where(mask);
13    // ... rest of the code ...
14    return 0;
15  }
```

Listing 23: C++ code for constructiong the index of all solid cells using ArrayFire.

With the solid indices, it's very easy to set the boundary conditions at solid nodes back to zero after the streaming step:

```
1  UX(solids) = 0; // velocity in X−direction
2  UY(solids) = 0; // velocity in Y−direction
3  DENSITY(solids) = 0;
```

Listing 24: Boundary conditions at solid nodes.

6.3.3 Pull vs Push Scheme

Most common algorithms for the streaming phase in LBM solvers use push and pull scheme Herschlag et al. (2018); Tran et al. (2017). In the push scheme, the streaming step occurs after the collision step, at which point the particle distribution values are written to neighbouring nodes. This presents a misalignment of the memory locations, resulting in an uncoalesced writes, degrading the performance significantly. On the other hand, in pull scheme, streaming step occurs before collision step, at which point the neighbouring particle distribution values are gathered to the current nodes and then used for computations ending with collision step, after which the results are written directly to the current nodes. This way, the writes are coalesced in memory.

The idea behind preferring coalesced writes to GPU device memory is that the requests for values that are stored at memory addresses within 128-byte range are combined into one, which saves memory bandwidth. It's generally accepted that the cost of the uncoalesced reading is smaller than the cost of the uncoalesced writing Tran et al. (2017).

6.3.4 Load Balancing in Multi-GPU Setups

With heterogenous GPU computing constraints (that means, group of GPUS each with different processing power, memory bandwidth or memory layout), load balancing is critical in this setting, therefore it has to be considered. To avoid wasteful delays, all computational units should do the same amount of work.

Focus of this study was to test performance of ArrayFire implementation of LBM codes on a single GPU. Some literature mentions simple extension to algorithms written with ArrayFire to add multiple GPU support in 4 lines of code Malcolm et al. (2012). Implementation like this is great for simple usecases and systems with the same type of GPU to prevent load-balancing issues. Proper multi-GPU support is planned in future versions of ArrayFire library that will be compatible with its Unified Backend convention. Therefore, we're eager to integrate multi-GPU support in future and test performance on heterogenous HPC systems when explicit mutli-GPU will be ready.

6.4 Interactive Simulation

6.4.1 Unity and Rust Interop

Note that OpenCL-GL sharing works in a very particular way. The OpenCL context that needs to be shared with an OpenGL context has to be created using the OS specific OpenGL context handle as one of it's context properties. In your current code, the OpenGL context used by ArrayFire's GL-CL shared context is totally different from the OpenGL context the rust window toolkit is showing.

Error executing function: clCreateFromGLTexture2D Status error code: CL_INVALID_-CONTEXT (-34)

What you need is the following

- Pick the device you want to do CL-GL sharing on and create a ocl_device_id handle using the ocl create

- Now, for this device from step (1) create a new OpenGL context using the OpenGL context and passing the opengl context properties specific to your OS
- Create an OpenCL queue for this device in this CL-GL shared context
- Now add this context to ArrayFire device manager using afcl::add_device_context
- Finally, set this device/context queue as your ArrayFire device using set_device_-context()

All of these has to be completed before any ArrayFire computations are carried out so that you don't have buffers that are created on a totally different OpenCL context.

Now, you can create a shared CL-GL buffer using the texture using this texture like you are doing now. Note that once you pass the cl_mem/device-ptr to ArrayFire, it shall assume control of that buffer, so take care of ownership of that object using methods like the following Array.lock() and Array.unlock().

Use references when you can, use pointers when you must. If you're not doing FFI or memory management beyond what the compiler can validate, you don't need to use pointers.

Both references and pointers exist in two variants. There are shared references & and mutable references &mut. There are const pointers *const and mut pointers *mut (which map to const and non-const pointers in C). However, the semantics of references is completely different from the semantics of pointers.

References are generic over a type and over a lifetime. Shared references are written &'a T in long form (where 'a and T are parameters). The lifetime parameter can be omitted in many situations. The lifetime parameter is used by the compiler to ensure that a reference doesn't live longer than the borrow is valid for.

Pointers have no lifetime parameter. Therefore, the compiler cannot check that a particular pointer is valid to use. That's why dereferencing a pointer is considered unsafe.

When you create a shared reference to an object, that freezes the object (i.e. the object becomes immutable while the shared reference exists), unless the object uses some form of interior mutability (e.g. using Cell, RefCell, Mutex or RwLock). However, when you have a const pointer to an object, that object may still change while the pointer is alive.

When you have a mutable reference to an object, you are guaranteed to have exclusive access to that object through this reference. Any other way to access the object is either disabled temporarily or impossible to achieve. For example:

```
let mut x = 0;
{
        let y = &mut x;
        let z = &mut x; // ERROR: x is already borrowed mutably
        *y = 1; // OK
        x = 2; // ERROR: x is borrowed
}
x = 3; // OK, y went out of scope
```

Mut pointers have no such guarantee.

A reference cannot be null (much like C++ references). A pointer can be null.

Pointers may contain any numerical value that could fit in a usize. Initializing a pointer is not unsafe; only dereferencing it is.

If you have a *const T, you can freely cast it to a *const U or to a *mut T using as. You can't do that with references. However, you can cast a reference to a pointer using as, and you can "upgrade" a pointer to a reference by dereferencing the pointer (which, again, is unsafe) and then borrowing the place using & or &mut. For example:

```
use std::ffi::OsStr;
use std::path::Path;
```

```
pub fn os_str_to_path(s: &OsStr) -> &Path {
        unsafe { &*(s as *const OsStr as *const Path) }
}
```

n C++, references are ”automatically dereferenced pointers”. In Rust, you often still need to dereference references explicitly. The exception is when you use the . operator: if the left side is a reference, the compiler will automatically dereference it (recursively if necessary!). Pointers, however, are not automatically dereferenced. This means that if you want to dereference and access a field or a method, you need to write (*pointer).field or (*pointer).method(). There is no -¿ operator in Rust.

6.4.2 Setting Initial Conditions

The physics and lattice parameters can be set in Unity Editor and also inside the virtual reality environment.

In development phase, these parameters are available as a part of the simulation script interface. The inputs are generated automatically for each public variable in script's class (Listing 25).

```
1   public class Sim : MonoBehaviour
2   {
3     public UInt32 width = 0;
4     public UInt32 height = 0;
5     private Texture2D image;
6
7     void Start()
8     {
9       image = new Texture2D(width, height, TextureFormat.RGBA32, false);
10      GetComponent<Renderer>().material.mainTexture = image;
11    }
12
13    // ... rest of the code
14  }
```

Listing 25: Boundary conditions at solid nodes.

When looking into the Unity Editor, the script interface now has two numerical inputs, `width` and `height` (Fig. 6 – 1).

DOPLNIŤ GRAF/DÁTA

Figure 6 – 1: Unity Editor with script interface.

6.4.3 Visualizing Simulation Output in Real-Time

Unity Game Engine as a Rendering Platform

- CPU-bound (using Marshal.Copy)
- GPU-bound (Render to texture, fast)

6.4.4 Updating Boundary Conditions

6.4.5 Time Manipulation

Visualization is running continuously throughout the simulation in parallel to the actual computations. It's sometimes good to pause the running simulation when there's something interesting to investigate or explore in more detail.

For simple pausing and continuing the simulation, the VR interface contains a GUI for

that (Fig. 6 – 2).

DOPLNIŤ GRAF/DÁTA

Figure 6 – 2: Interface for controlling the state of simulation.

Implementation of the co-routine starting, pausing and stopping is implemented in Listing 26.

```
1   public class Sim : MonoBehaviour
2   {
3     void Start()
4     {
5       // ...
6       StartCoroutine("StartSimulation")
7     }
8
9     IEnumerate StartSimulation() {
10      // ... rest of the code
11    }
12  }
```

Listing 26: ”Starting pausing and stopping the co-routines in Unity.”

To manipulate time and go back in history, the simulation data has to be stored to the disk. Each second, XY GB of data is stored.

To go back in time, user can use the slider and move the knob backward or forward (Fig.

6 – 3).

DOPLNIŤ GRAF/DÁTA

Figure 6 – 3: Interface for controlling the state of simulation.

6.5 Virtual Reality User Interface

6.5.1 Cross-platform Development

6.5.2 VRTK

6.5.3 Interacting with 3D Data

6.5.4 Plotting in VR

6.6 Performance Analysis

The performance of computers used for scientific applications are commonly measure in floating point operations per second (FLOPS), which represents the time it takes for multiplying two 32 or 64 bit floating-point numbers.

At the time of writing, the fastest supercomputer in the world runs at roughly 440 PetaFLOPS

(440×10^{15}). The next milestone in computer engineering is to build a supercomputer capable of running at speeds exceeding 1 ExaFLOPS (10^{18}). In contrast to HPC systems, the fastest GPU on consumer market, at the time of writing, is NVIDIA's GeForce RTX 3090 that peaks at 35 TeraFLOPS.

Performance of LBM simulations is measured in "Million Lattice Updates Per Second" (MLUPS), which is a standard unit of measurement within the LBM research community. It states that the simulation code updates a computational domain of million cells in lattice during one CPU second. The same metric is used for both single and double-precision floating-point operations in benchmarked simulations.

6.6.1 Hardware

Since ArrayFire allows for using not only GPU backends but also CPU, we added a CPU benchmarks executed on Intel Core i7 6800K running at 3.40GHz. Performance peaked at around 19 MLUPS and stayed the same between various domain sizes across both academic test cases.

Most of the GPU benchmarks were done using both CUDA and OpenCL backends, although differences between them were minimal. Therefore in following tables and graphical representations of the data, we show MLUPS numbers solely from OpenCL benchmarks, since testing on this platform allowed us to perform benchmarks not only on NVIDIA hardware, but also on AMD GPU.

We tested the solver performance on 5 different GPUs. The AMD Radeon R9 M370X is of mobile GPU type installed in laptops. In the current study, the AMD GPU was tested on the higher-end Macbook Pro 2015. The NVIDIA GTX 1070 is the average desktop GPU and its price at the time of writing this article is $443.78 USD according to PassMark G3D Mark (a GPU benchmarking website). The NVIDIA RTX 3090 is a top-of-the-line, consumer-grade, enthusiast-level GPU with the price of $2139.99 USD according to PassMark G3D Mark at the time of writing. We've included two other GPUs to test across multiple architectures, namely NVIDIA Tesla K20c and NVIDIA GeForce

RTX 2080 Ti.

	R9 M370X	Tesla K20c	GTX 1070	RTX 2080 Ti	RTX 3090
Architecture	GCN 1.0	Kepler	Pascal	Turing	Ampere
Number of cores (CU/SM)	640 (10 CU)	2496 (13 SM)	1920 (15 SM)	4352 (68 SM)	10496 (82 SM)
Peak f32 performance (TFLOPS)	1.024	3.524	6.463	13.45	35.58
Memory clock (MHz)	1125	1300	2002	1750	1219
Memory bandwidth (GB/s)	72.00	208.0	256.3	616.0	936.2
L1 cache size (KB)	16	16	48	64	128

Table 6 – 1: Hardware specifications for GPUs used for testing the simulation software described in this paper (taken from https://www.techpowerup.com/gpu-specs/). SM - streaming multiprocessor, CU - computing units.

6.6.2 Results

For the 2D lid-driven cavity test, benchmarks showed great results (Table 6 – 2) with significant speedup compared to CPU backend.

Single floating-point calculations perform significantly faster than double-precision (Fig. 6 – 4 and Fig. 6 – 5 for lid-driven cavity tests, Fig. 6 – 6 and Fig. 6 – 7 for Kármán vortex tests).

For the 2D Kármán vortex test case, benchmarks showed similar results (6 – 3). The performance spikes in so-called ”warm-up” phase at the start of simulation were less significant in double-precision benchmarks than in lid-driven cavity test.

Maximum MLUPS of the GPUs for single-precision calculations for 2D test cases are slightly higher than the study reported by Boroni et. al *FULL GPU Implementation*

!ht]

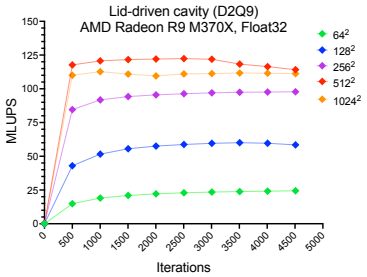
Domain	R9 M370X	K20c	GTX 1070	RTX 2080 Ti	RTX 3090
64×64	23	40	40	11	55
128×128	59	157	226	50	225
256×256	97	468	675	186	868
512×512	120	531	1006	600	2890
1024×1024	111	559	1143	1401	4620
2048×2048	8	566	1200	2195	5465
4096×4096	6	-	-	2394	5730

Table 6 – 2: Average MLUPS of lid-driven cavity test case with single-precision floating point computation (taking only the steady performance after initial warm-up and removal of sudden spikes).

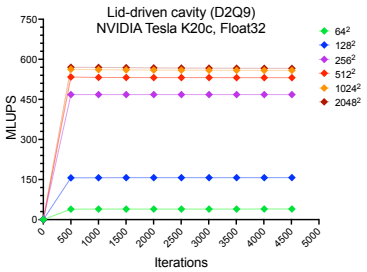
of Lattice-Boltzmann Methods with Immersed Boundary Conditions for Fast Fluid Simulations (2017), in which they reported 80 MLUPS at peak performance achieved on NVIDIA GeForce GTX 580 GPU. The AMD Radeon R9 M370X GPU used in our work can be seen as similarly performing card.

7 Conclusions

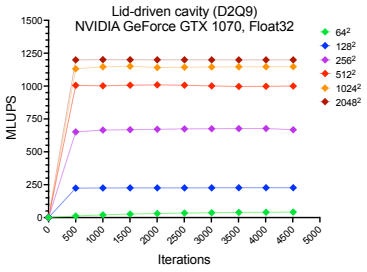
For this thesis, the cross-platform LBM solver that can be used on variety of parallel accelerators (e.g. GPUs, CPUs or FPGAs) was implemented. It uses ArrayFire, a high-performance parallel computing library (version 3.8.0 was used for this work). We created reference implementation in C++ and ported the same code to Rust. We chose Rust because it has modern capabilities, is memory safe and its performance is comparable to C/C++. We were able to produce sufficient LBM solver in under 150 lines of code, including real-time visualization code. Benchmarks were concluded for both C++ and Rust versions, resulting in identical performance. Data reported in this study are taken



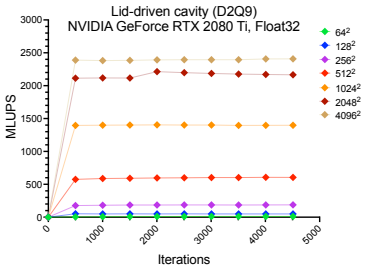
(a) Performance on Radeon R9 M370X GPU.



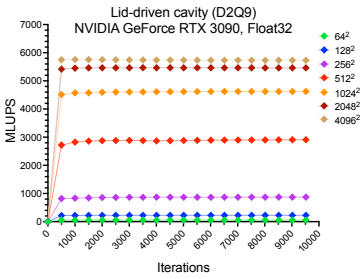
(b) Performance on Tesla K20c GPU.



(c) Performance on GTX 1070 GPU.



(d) Performance on RTX 2080 Ti GPU.

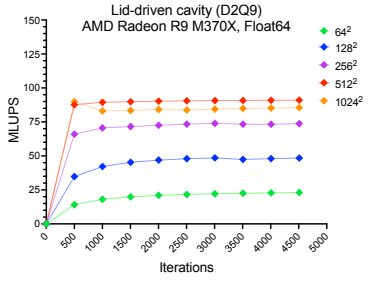


(e) Performance on RTX 3090 GPU.

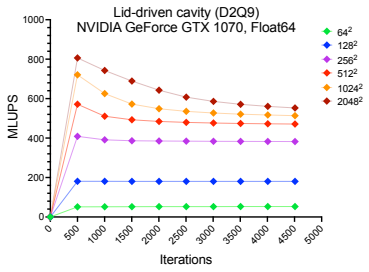
Figure 6 – 4: Single-precision performance analysis of 2D Kármán vortex on D2Q9 stencil.

from the Rust version.

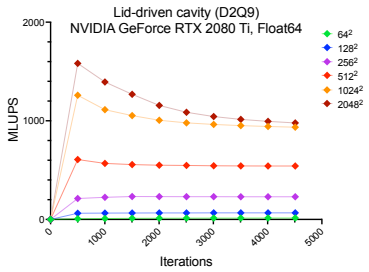
For the benchmarks, we analyzed two classical, academic test cases, the lid-driven cavity and Kármán vortex street (flow around the obstacle in pipe). We employed commonly used metric for measuring speed of LBM implementations, the MLUPS. We benchmarked



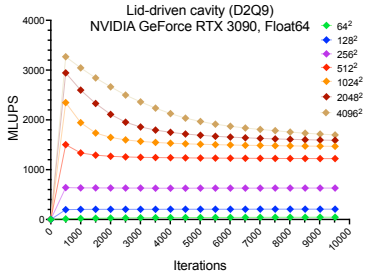
(a) Performance on Radeon R9 M370X GPU.



(b) Performance on GTX 1070 GPU.



(c) Performance on RTX 2080 Ti GPU.



(d) Performance on RTX 3090 GPU.

Figure 6 – 5: Double-precision performance analysis of 2D Kármán vortex on D2Q9 stencil.

our LBM implementation on 5 different GPUs and one CPU. Between CPU and GPU, we saw from 4 to more than 300 times speedup on various GPUs.

8 Discussion

8.1 Future Work

8.1.1 Streaming Visualized Pixels Over Network

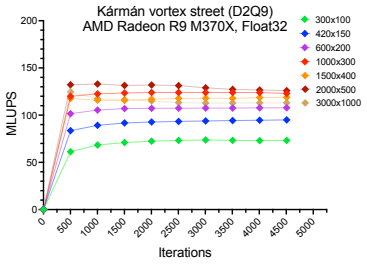
8.1.2 Simulation as an Educational Tool

In some simplified form, combinations of numerical simulations and visualizations of steel- making processes can be used as a educational tool in process control courses at

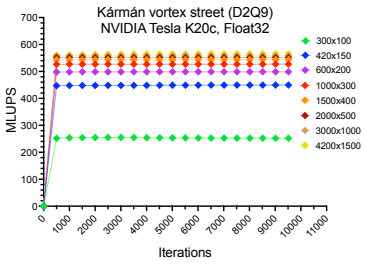
Domain	R9 M370X	K20c	GTX 1070	RTX 2080 Ti	RTX 3090
300×100	73	253	340	361	361
420×150	93	448	627	122	740
600×200	107	498	810	180	1343
1000×300	123	527	1007	325	3000
1500×400	117	543	1100	731	4055
2000×500	130	552	1140	1010	4510
3000×1000	113	557	1175	1311	5313
4200×1500	-	565	1194	2016	5557
6000×2000	-	-	-	2522	5670
10000×3000	-	-	-	-	5720

Table 6 – 3: Average MLUPS of Kármán vortex street test case with single-precision floating point computation (taking only the steady performance after initial warm-up and removal of sudden spikes).

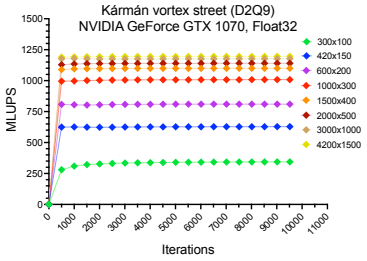
technical universities. The aim of the online, web-based interactive simulation of basic oxygen steelmaking at steeluniversity.org shown in Fig. **??** is to introduce students to this process in a more fun and engaging way.



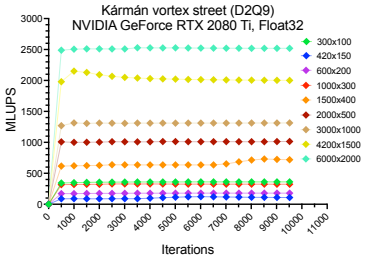
(a) Performance on Radeon R9 M370X GPU.



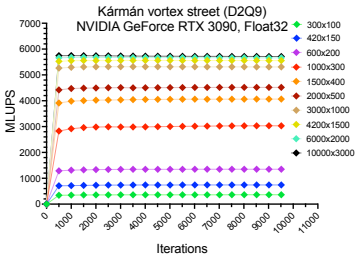
(b) Performance on Tesla K20c GPU.



(c) Performance on GTX 1070 GPU.

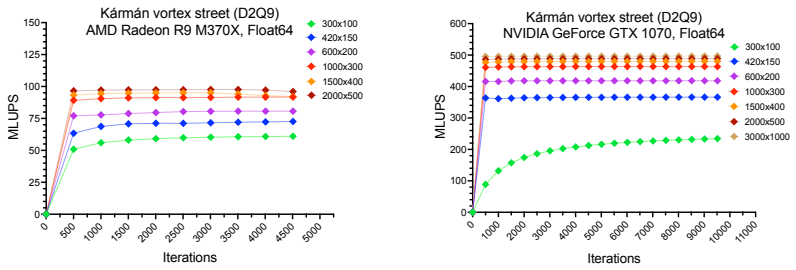


(d) Performance on RTX 2080 Ti GPU.



(e) Performance on RTX 3090 GPU.

Figure 6 – 6: Single-precision performance analysis of 2D Kármán vortex on D2Q9 stencil.



(a) Performance on Radeon R9 M370X GPU.

(b) Performance on GTX 1070 GPU.

(c) Performance on RTX 2080 Ti GPU.

(d) Performance on RTX 3090 GPU.

Figure 6 – 7: Double-precision performance analysis of 2D Kármán vortex on D2Q9 stencil.

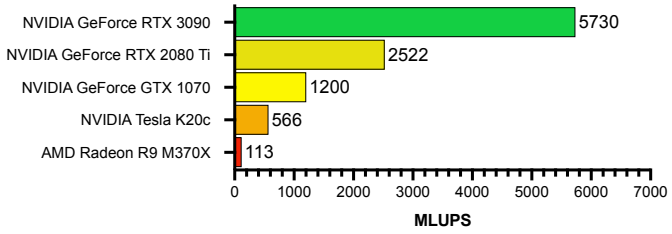


Figure 6 – 8: Peak performance of single-precision LBM simulations on D2Q9 stencil.

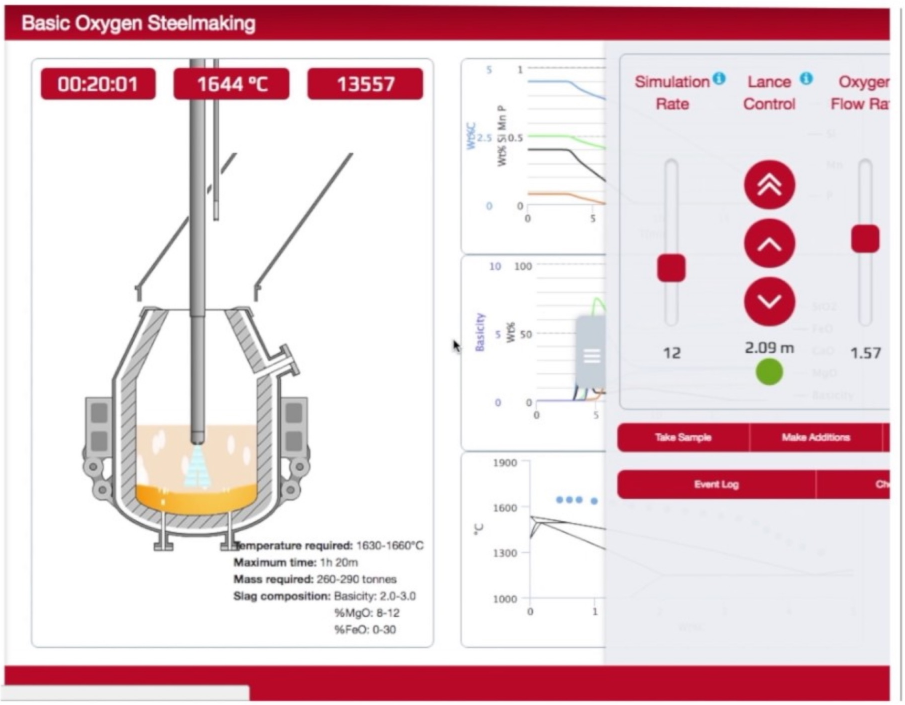


Figure 8 – 1: Interactive, educational simulation of basic oxygen steelmaking by steeluniversity.org.

References

Bhatnagar, P. L., Gross, E. P. and Krook, M. (1954). A Model for Collision Processes in Gases. I. Small Amplitude Processes in Charged and Neutral One-Component Systems, *Physical Review* **94**(3): 511–525.
URL: *https://link.aps.org/doi/10.1103/PhysRev.94.511*

Chrzeszczyk, A. (n.d.). Matrix Computations on GPU with ArrayFire - Python and ArrayFire - C/C++, p. 88.

Delbosc, N. (n.d.). Real-Time Simulation of Indoor Air Flow Using the Lattice Boltzmann Method on Graphics Processing Unit, p. 261.

Delbosc, N., Summers, J., Khan, A., Kapur, N. and Noakes, C. (2014). Optimized implementation of the Lattice Boltzmann Method on a graphics processing unit towards real-time fluid simulation, *Computers & Mathematics with Applications* **67**(2): 462–475.
URL: *https://linkinghub.elsevier.com/retrieve/pii/S0898122113006068*

Fei, L., Du, J., Luo, K. H., Succi, S., Lauricella, M., Montessori, A. and Wang, Q. (2019). Modeling realistic multiphase flows using a non-orthogonal multiple-relaxation-time lattice Boltzmann method, *Physics of Fluids* **31**(4): 042105.
URL: *http://aip.scitation.org/doi/10.1063/1.5087266*

FULL GPU Implementation of Lattice-Boltzmann Methods with Immersed Boundary Conditions for Fast Fluid Simulations (2017). *The International Journal of Multiphysics* **11**(1).
URL: *http://www.journal.multiphysics.org/index.php/IJM/article/view/11-1-1*

Glessmer, M. and Janßen, C. (2017). Using an Interactive Lattice Boltzmann Solver in Fluid Mechanics Instruction, *Computation* **5**(4): 35.
URL: *http://www.mdpi.com/2079-3197/5/3/35*

Guo, Z., Zheng, C. and Shi, B. (2002). Discrete lattice effects on the forcing term in the

lattice Boltzmann method, *Physical Review E* **65**(4): 046308.
URL: *https://link.aps.org/doi/10.1103/PhysRevE.65.046308*

Harwood, A. R., O'Connor, J., Sanchez Muñoz, J., Camps Santasmasas, M. and Revell, A. J. (2018). LUMA: A many-core, Fluid–Structure Interaction solver based on the Lattice-Boltzmann Method, *SoftwareX* **7**: 88–94.
URL: *https://linkinghub.elsevier.com/retrieve/pii/S2352711018300219*

Harwood, A. R. and Revell, A. J. (2017). Parallelisation of an interactive lattice-Boltzmann method on an Android-powered mobile device, *Advances in Engineering Software* **104**: 38–50.
URL: *https://linkinghub.elsevier.com/retrieve/pii/S0965997816301855*

Herschlag, G., Lee, S., Vetter, J. S. and Randles, A. (2018). GPU Data Access on Complex Geometries for D3Q19 Lattice Boltzmann Method, *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, Vancouver, BC, pp. 825–834.
URL: *https://ieeexplore.ieee.org/document/8425236/*

Januszewski, M. and Kostur, M. (2014). Sailfish: a flexible multi-GPU implementation of the lattice Boltzmann method, *Computer Physics Communications* **185**(9): 2350–2368. arXiv: 1311.2404.
URL: *http://arxiv.org/abs/1311.2404*

Karimi, K. (n.d.). A Performance Comparison of CUDA and OpenCL, p. 10.

Kotsalos, C., Latt, J., Beny, J. and Chopard, B. (2019). Digital Blood in Massively Parallel CPU/GPU Systems for the Study of Platelet Transport, *arXiv:1911.03062 [physics]* . arXiv: 1911.03062.
URL: *http://arxiv.org/abs/1911.03062*

Körner, C., Pohl, T., Rüde, U., Thürey, N. and Zeiser, T. (2006). Parallel Lattice Boltzmann Methods for CFD Applications, *in* A. M. Bruaset and A. Tveito (eds), *Numerical Solution of Partial Differential Equations on Parallel Computers*, Vol. 51, Springer-

Verlag, Berlin/Heidelberg, pp. 439–466. Series Title: Lecture Notes in Computational Science and Engineering.
URL: *http://link.springer.com/10.1007/3-540-31619-1_13*

Li, Q., Luo, K., Kang, Q., He, Y., Chen, Q. and Liu, Q. (2016). Lattice Boltzmann methods for multiphase flow and phase-change heat transfer, *Progress in Energy and Combustion Science* **52**: 62–105.
URL: *https://linkinghub.elsevier.com/retrieve/pii/S0360128515300162*

Linxweiler, J., Krafczyk, M. and Tölke, J. (2010). Highly interactive computational steering for coupled 3D flow problems utilizing multiple GPUs: Towards intuitive desktop environments for interactive 3D fluid structure interaction, *Computing and Visualization in Science* **13**(7): 299–314.
URL: *http://link.springer.com/10.1007/s00791-010-0151-3*

Malcolm, J., Yalamanchili, P., McClanahan, C., Venugopalakrishnan, V., Patel, K. and Melonakos, J. (2012). ArrayFire: a GPU acceleration platform, Baltimore, Maryland, USA, p. 84030A.
URL: *http://proceedings.spiedigitallibrary.org/proceeding.aspx?doi=10.1117/12.921122*

Pacheco, P. S. (2011). Chapter 1 - why parallel computing?, *in* P. S. Pacheco (ed.), *An introduction to parallel programming*, Morgan Kaufmann, Boston, pp. 1–14.
URL: *https://www.sciencedirect.com/science/article/pii/B9780123742605000014*

Storti, D. and Yurtoglu, M. (2016). *CUDA for engineers: an introduction to high-performance parallel computing*, Addison-Wesley, New York.

Succi, S. (2001). *The lattice boltzmann equation: For fluid dynamics and beyond*, Numerical mathematics and scientific computation, Clarendon Press. tex.lccn: 2001036220.
URL: *https://books.google.hu/books?id=OC0Sj_xgnhAC*

Suga, K., Kuwata, Y., Takashima, K. and Chikasue, R. (2015). A D3Q27 multiple-relaxation-time lattice Boltzmann method for turbulent flows, *Computers & Mathe-*

matics with Applications **69**(6): 518–529.
URL: *https://linkinghub.elsevier.com/retrieve/pii/S0898122115000346*

Szőke, M., Józsa, T. I., Koleszár, A., Moulitsas, I. and Könözsy, L. (2017). Performance Evaluation of a Two-Dimensional Lattice Boltzmann Solver Using CUDA and PGAS UPC Based Parallelisation, *ACM Transactions on Mathematical Software* **44**(1): 1–22.
URL: *https://dl.acm.org/doi/10.1145/3085590*

Tran, N.-P., Lee, M. and Hong, S. (2017). Performance Optimization of 3D Lattice Boltzmann Flow Solver on a GPU, *Scientific Programming* **2017**: 1–16.
URL: *https://www.hindawi.com/journals/sp/2017/1205892/*

Wang, M., Ferey, N., Bourdot, P. and Magoules, F. (2019). Interactive 3D Fluid Simulation: Steering the Simulation in Progress Using Lattice Boltzmann Method, *2019 18th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES)*, IEEE, Wuhan, China, pp. 72–75.
URL: *https://ieeexplore.ieee.org/document/8921356/*

Wittmann, M. (2018). Lattice Boltzmann benchmark kernels as a testbed for performance analysis, p. 11.