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:	Process Control
Field of study:	Cybernetics
Department:	Institute of Control and Informatization of Production Processes (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

Mathematical modeling, Simulation, 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

Matematické modelovanie, Simulácia, 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

List of Figures

List of Tables

List of Terms

LBM - Lattice Boltzmann Method.

LBE - Lattice Boltzmann Equation.

DF - Distribution Function.

GPU - Graphics Processing Unit.

GPGPU - General-Purpose Computing on Graphics Processing Unit.

D2Q9 - Two-dimensional lattice stencil with 9 discrete velocity directions in each node.

D3Q27 - Three-dimensional lattice stencil with 27 discrete velocity directions in each node.

JIT - Just-In-Time compilation.

API - Application Programming Interface.

1 Introduction

In this day and age, we humans live in a world where things which were unimaginable just few decades ago are becoming reality, thanks to the huge progress made possible by computers. The increase in computational power and ubiquity of wide range of computing devices has opened the doors to practically implement new methods that were previously discussed only on a theoretical level and intractable in analytical forms. Computer-based techniques like numerical simulations allows us to, at least approximately, evaluate challenging and complex phenomena occurring in nature. In this thesis, I will be mostly dealing with the one of such complex phenomena, namely fluid flows.

We encounter flowing fluids everywhere on our known human scale, but when the world around us is investigated very closely under a microscope, or when we zoom out and look far beyond our Earth, much more complex fluid flows can be encountered. Again, computers allowed us to find out about the world too small or parts of the universe too far away to see with the naked eye. Understanding such physical phenomena across different scales then opens the door for developing novel technologies.

What drives the hardware computing tools is the software. It's what we use to instruct information to flow in a way that provides value and drives us towards aforementioned goal of deep understanding.

Scientific and engineering software is different from the one that is used by ordinary people who want to consume arbitrary information, read articles, order meals, watch videos, or connect with their social circle through social media applications. In a scientific software, user is typically presented with large amounts of data resulting from complex physical experiments or numerical simulations. This puts importance on an interface for proper visualization of that data, in a way that the user is able to assess and analyze it. This type of software is a medium for *visual communication* and has to provide means to effectively *communicate*.

The primary design challenge for scientific software is to provide affordances that make

these visualizations correct, understandable, and reduce the complexity into the simpler, more approachable form. In this thesis, I will be exploring modern methods and approaches for visualizing fluid simulations and how to seamlessly interact with the running simulation from virtual reality application.

The usage of emerging technologies like virtual or augmented reality, thanks to their highest exposure in gaming industry, was mostly just an afterthought within scientific computing. These technologies have remained absent from science and engineering workflows due to the custom-built nature of the hardware and software (Su et al.; 2020). But as we approach the exascale computing capabilities, the volume and complexity of data from high-energy physics experiments, modeling of complex engineering designs and large-scale multi-phase simulations, is growing dramatically. VR and AR can provide advanced visualization approaches to perform data assessment in data-intensive scientific and engineering applications. Many software solutions that focus on data analysis of large-scale scientific simulations have integrated support for some of the consumer-grade VR headsets, available off-the-shelf, but also CAVE (Cave Automatic Virtual Environment) systems. CAVE system is a room-sized cube with virtual environment projected to its walls (Ohno and Kageyama; n.d.). These systems typically require big investments in tens of thousands of euros. On the other hand, VR headsets, or head-mounted displays (HMDs) are much cheaper (their price is usually in low hundreds of euros). In this thesis I've focused on leveraging the HMDs with their appropriate hand controllers or built-in hand tracking, as the target audience should not be limited to the enterprises and institutions with big budgets, but anybody who can get their hands on the average VR-ready hardware, starting with individual researchers. Therefore to evaluate other software solutions for scientific visualization and data analysis, I'm targeting those that support commonly used HMDs made by HTC, Oculus (HTC Vive, Oculus Rift), or various Windows Mixed Reality headsets, usually through a higher-level API like OpenVR.

One of such software is ParaView (Ahrens et al.; 2005). It's a open-source, multi-platform data analysis and visualization application extensively used within scientific community

(Ahrens et al.; 2008; Marion et al.; 2012). I'ts built as a graphical interface on top of Visualization Toolkit (VTK), another open-source software that sits at the hearth of not just ParaView, but many more software applications for scientific data analysis and visualization. VTK plays central role in VR support in these types of software, because it provides the basic building blocks for talking to HMD hardware APIs through higher-level interface such as OpenVR. Rendering of the datasets to be analyzed to the HMD is handled very easily. More complex interactions with the data and 3D imagery has to be implemented additionally. In ParaView, basic interactions are already implemented into the software, but it doesn't provide any user interface (UI) specifically for immersive exploration of the data in VR. Shetty et al. (2011) extended the ParaView to add physical interaction through external touch display while preserving the data streaming and data parallel mechanisms for large data visualization. Their approach could be labeled as mixed, but the way of working with the software through indirect interaction outside of HMD makes it very hard to operate it towards the task of effectively exploring the large-scale datasets.

As the work with complex datasets from scientific or engineering applications is very challenging, tools for working with their visualization have to be designed appropriately. If we want VR to serve its purpose in science and engineering, then it must add value instead of slowing us down or complicating things even more than they already are. The company SciVista that is behind their SummitVR application addresses these concerns with their approach to extending VR capabilities of ParaView beyond just a fancy 3D viewer for of the scientific data. Their VR software provides an intuitive, 3D UI built purposely for handling datasets that extend to petabytes of data per day coming in from the largest fusion experiment(ITER). Ability to work inside the immersive virtual environment with data assessment tools seamlessly accessible through hand-controller-driven UI, choosing from data filtering options and ability to move 3D data representations around the user is very effective strategy towards bringing value for researchers (SciVista; 2021).

Aforementioned cases where visualization of scientific data for further analysis is impor-

tant aspect of research work, are just some of the examples how data from experiments or post-simulation can be analysed. This is the classic type of data visualization and it's called "post hoc".

In field of fluid dynamics, we are interested in different properties of how fluid flows and behaves under various circumstances. With advent of powerful computers, fluid flows can be studied much more effectively with computational fluid dynamics (CFD) methods and tools. Visualization of different aspects of fluid flows is very important part of CFD analysis workflow. Classic approach to data analysis in such cases through post hoc visualization, although very helpful, is insufficient and not effective enough. With current power of consumer-grade PC workstations and tremendous power of supercomputers gearing towards exascale computational power, we can do much better.

During the last two decades, we can see the move from sequential computing and specialized programming of many-core systems towards parallel computing. This was mainly pushed by the fact that power consumption of many-core CPUs with hundreds of cores and resulting heat generation was reaching the physical limit after which the chip could become unstable and potentially melt. CPU manufacturers transferred to the multi-core architectures with single-digit number of cores and parallel computing has become the dominant paradigm.

Alongside the multi-core and many-core CPUs, graphical processing units (GPUs) started gaining traction in computer science community for accelerating scientific applications after they became increasingly programmable throughout the years of 1999-2000. GPU consists of hundreds or thousands of smaller cores. This massively parallel architecture is what gives the GPU its high compute performance. Back then it was extremely hard to write programs for GPUs, but nowadays various computing platforms and frameworks exist to ease the development. This democratized the use of GPU for larger scientific community and and spearheaded the movement called GPGPU, or General Purpose GPU computing.

Since then, GPU computing has been intensively used for CFD simulations and data visualization. Thanks to the GPUs' high performance computation, a parallel algorithms of modern methods for CFD simulations like Lattice Boltzmann Method (LBM) developed for simulating complex fluid flows outperform classic ones in speed by more than hundred times or more (Boroni et al.; 2017; Harwood et al.; 2018; Januszewski and Kostur; 2014; Qian et al.; 1992; Szőke et al.; 2017; Tran et al.; 2017; Wittmann; 2018). Such speedups opened the door to real-time simulation and interactive visualization (Delbosc; n.d.; Koliha et al.; 2015; Kress; n.d.; Linxweiler et al.; 2010; Mawson; 2014; Thürey et al.; n.d.; Wang et al.; 2019). Beside the great parallelizability, LBM is relatively easy to implement and it's also possible to additionally implement needed physics as the problem at hand requires. LBM is therefore perfectly suited as an underlying method for developing CFD simulation software that will be integrated with the interactive virtual reality visualization software.

The important aspect of such software that will be presented in this thesis is not crunching numbers in the shortest time possible, but instead, it's the synthesis of VR application design with simulation software fast enough that it is possible to amplify human reasoning through data visualization and automated analysis. The CFD researcher who would potentially use this application must see the effects of any manipulation of input data or change of any option immediately. Thanks to this ability to steer the running simulation, the exploration phase should be an order of magnitude quicker than with conventional CFD simulation and visualization methods.

This thesis is structured as follows. Chapter 2 lays forward an introduction to and an explanation why mathematical modeling and simulation are important in modern approach to science and engineering; how they fit into the framework of computational fluid dynamics (CFD); and finally why combining them together is important. In chapter 3, the basic theory of fluid flows will be presented; this will be expanded to mesoscopic approach to fluid dynamics based on Lattice Boltzmann Method (LBM); different aspects of LBM will be discussed, including boundary conditions, multiphase flows, adaptive

mesh refinement, and finally this chapter will conclude with the look ahead beyond Newtonian fluids. Chapter 4 will deal with the technical aspects of GPU computing; how parallel programming is different from sequential programming; how CPUs and GPUs can work alongside each other in symbiosis, often called heterogeneous computing; what GPU computing platforms and frameworks exist; how they can be targeted from single codebase; and in the end this chapter will connect with the previous one, showing how LBM simulations can be accelerated through GPU computing. In chapter 5, various visualization methods will be briefly discussed. Chapter 6 deals with the implementation of integrated simulation software based on LBM and interactive virtual reality visualization software. In closing, chapter 7 is meant to serve as a review of the work done within the context of thesis theme and it ends with the discussion of potential next steps and future work that can be done in chapter 8.

2 Modeling and Simulation

"The purpose of computing is insight, not numbers"

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

Nowadays, in the times of ubiquitous computing, it is hard to imagine modern scientist or engineer that doesn't use computers in some way for their work. They pose a tremendous help in constructing our understanding of real-life phenomena, mostly in those which cannot be measured by conventional tools or the experimentation phase to get important insights is very expensive.

This section will explain how mathematical modeling and numerical simulation coupled with modern computing help with building our understanding of the world.

2.1 Mathematical Modeling

Mathematical modeling is an indispensable tool in science and engineering. The main idea behind using mathematical tools for scientific description of real-life phenomena is to help with their thorough understanding and provide insight, answers and guidance for the originating applications through theoretical and numerical analysis.

Every model consists of a set of variable parameters, relations between them and rules for their evolution. By applying those rules to the model's parameters, it's possible to get important insight into the modeled phenomenon, based on which we can then make informed decision or make an optimal choice (Neumaier; 2004). Most of the decision-making and choice-picking is done by computers in the way of control algorithms acting upon results from numerical simulation.

In field of engineering, especially in the branch of fluid dynamics, mathematical modeling in conjunction with numerical simulation is a powerful duo, playing important role in studying phenomena like turbulence, chemical reactions, etc. The efficiency of modern computing allows for simulating better and more complex models of phenomena seen around us in our everyday lives, including blood circulation, weather and climate prediction, and many more.

As a matter of fact, the process of mathematical modeling alone was and still is the domain of mathematicians and physicians, trained for years, expanding their skillset of mathematical methods for constructing and refining mathematical models. Generally speaking, success in this process requires experience, skill, and familiarity with the relevant literature. There is lack of software tools to equip people that didn't devote years into mathematical training to actually develop models and refine them themselves. By pulling this process into tighter loop with simulation that is fast enough to speed up the numerical analysis, there's an opportunity to design software applications that allows engineers and researchers to work in a different and more efficient way by constructing and refining mathematical within the graphical user interface. Thanks to this combination of model-

ing and simulation, together with an application interface design that brings mathematical tools coupled with visual experimentation and interactive analysis together into a cohesive platform, I hypothesise that productivity of individual researchers and research teams of all sizes could increase by an order of magnitude. Additional details about the approach of integrating the modeling and simulation together are provided in section 2.4.

2.2 Numerical Simulation

A numerical simulation is a computer calculation of a mathematical model for a physical system. It can provide answers about the studied phenomena in matter of seconds or minutes up to hours or days, depending on the model complexity, its algorithmic representation in computer code, and hardware infrastructure they are performed on. The final results of a numerical simulation can be stored to computer memory for further post-processing and represented graphically to screen. Its notable advantage is removing the need for costly real-life experiments. Together with mathematical modeling, they are important parts of engineering. Integrating them into engineer's toolbelt yields many benefits. The motivation for using computer simulations to investigate complex physical phenomena is two-fold: (1) it enables design changes to be tested before building a prototype, saving precious financial resources, and (2) it allows for investigation of complex phenomena unmeasurable by classic measurement tools, often involving extremely small sizes of the investigated environment in which the process evolves (Rago; 2015).

The mathematical models for which numerical simulation is employed usually consists of ordinary differential equations (ODEs) and partial differential equations (PDEs) (Yang; 2017). Coming up and refining actual model to needed accuracy is often very laborious and time consuming. Also, the actual simulation part can take hours or days to complete. Thanks to modern computing capabilities, the effectiveness of various accelerators like GPUs, FPGAs, etc. and recently developed optimizations techniques allowed the simulation times to dramatically shorten - nowadays towards minutes, seconds or even to the point of interactive and real-time simulations (Harwood et al.; n.d.).

2.3 Computational Fluid Dynamics

Fluid dynamics is the branch of fluid mechanics, that is concerned with the study of the effect of forces on fluids and fluid motion. It models matter as a macroscopic quantity rather than microscopic. Computational Fluid Dynamics (CFD) is the analysis of fluid flow, heat transfer and additional phenomena like chemical reactions with aid of computer-based simulations. Since its inception in 1960s until today, it has been heavily applied in the multitude of industries around the world. Some examples of the applications areas include:

- aerodnamics of aircraft and vehicles,
- wind flow around the buildings,
- hydrodynamics of ships,
- hydrology and oceanography,
- combustion and gas turbines,
- turbomachinery,
- heating and ventilation,
- distribution of pollutants,
- meteorology,
- biomedical engineering.

Modern fluid mechanics problems would be impossible to solve without use of CFD, since the analytical solutions to fundamental equations of fluid mechanics is very limited. CFD encompasses a wide spectrum of numerical methods used for solving complex three-dimensional (3D) and time-dependent flow problems Rapp (2017).

CFD simulations allows huge improvements on the type of phenomena that can be explored. This trend will continue accelerating thanks to improvements in both the avail-

able processing power and active research in algorithms for CFD simulations. 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.

2.4 Integrated Modeling and Simulation

Combination of mathematical modeling and the visualization of numerical simulation into single package, usually in a form of monolithic application with graphical user interface, poises an interesting benefit for shortening of the discovery and optimization of the correct and accurate mathematical or physical model of investigated phenomena. Integration of the two has been dubbed "MODSIM" by the engineering industry. Computer-aided Design (CAD) and Computer-aided Engineering (CAE) technology evolved separately over decades and because of legacy processes, they been usually employed separately, often sequentially one after another. Together, they provide essential means to guide the design of complex engineering systems and to study dynamics of internal or external forces on those systems (Mahmood et al.; 2019). One of significant proponents of this combined approach is Dassault Systems. They are leading the transformation of MODSIM research into practial application in their 3DEXPERIENCE platform (Dassault Systems; 2019).

In terms of fluid dynamics and specifically CFD, conventional modeling intent was focused on geometric and functional aspects. Simulation results were used for checking and guiding the process from initial design towards optimized one (Li, Ma and Lange; 2016). For complex, non-linear fluid flows like microfluidics or flows with chemical reactions, the MODSIM intent in CFD is different (Zhang; 2011). The goal is to optimize the mathematical model to be as accurate as possible, but in discretized to be algorithmically feasible to implement on computers and simulated as efficiently as possible, catering to accuracy requirements and driving towards fast convergence speeds. In this

regard, numerical simulations based on Lattice-Boltzmann method for CFD, especially algorithms implementing the single-relaxation time collision operator, are great candidates for MODSIM implementations. The method originates from a molecular description of a fluid, which means the physical terms that arise from a knowledge of the interaction between molecules can be directly incorporated. Thanks to this, it can be used as a tool in fundamental research, as it keeps the cycle between the elaboration of a theory and the formulation of a corresponding numerical model short.

3 Mesoscopic Approach to Fluid Dynamics

> *"Available energy is the main object at stake in the struggle for existence and the evolution of the world."*
>
> – Ludwig Boltzmann

In the search for fast solver for simulating fluid flows while allowing for complex representation of underlying dynamics, one doesn't have to wait for computers to become powerful enough to just compute direct numerical simulation. Fast enough in a sense that the resulting simulation would allow for interactivity and real-time visualization. For this become feasible, still large progress in computing power and new efficient algorithms has to come.

Although, there is an intermediate solution until those days. In fact, quite literally, an intermediate solution: the Lattice-Boltzmann method (LBM). It is a method for solving equations of continuum fluid mechanics in the realm between macroscopic and microscopic scales - the mesoscopic scale. It's simplicity and great locality of the algorithm presents interesting possibilities for achieving very high simulation speeds. Executing the parallel LBM algorithms on graphic processing units (GPUs) allows for real-time and interactive CFD simulations. That is the reason why this method was picked for the development of simulation backend in this work.

This section describes the Lattice-Boltzmann method, how Boltzmann's kinetic theory behind LBM transfers to the continuum dynamics of Navier-Stokes equations, and how real-world phenomena can be modeled and simulated.

3.1 Navier-Stokes Equations

The famous equations governing the motion of fluid was described by Claude-Luis Navier (1785-1836) and Gabriel Stokes (1819-1903). They are collectively called Navier-Stokes (N-S) equations.

Let's consider Euclidean space $\mathbb{R}^d$, with d equal to 2 or 3 (representing dimensions). At position $x = (x_1, ..., x_d) \in \mathbb{R}^d$ and at time $t \in \mathbb{R}$, the fluid is moving with a velocity vector $\boldsymbol{u}(x,t) = (\boldsymbol{u}_1(x,t), ..., \boldsymbol{u}_d(x,t)) \in \mathbb{R}^d$ and the fluid pressure is $p(x,t) \in \mathbb{R}$. We can state the Euler equation as

$$\left(\frac{\partial}{\partial t} + \sum_{j=1}^{d} u_j \frac{\partial}{\partial x_j}\right) u_i(x,t) = -\frac{\partial p}{\partial x_i}(x,t) \quad i = 1, ...d \tag{3.1}$$

for all (x,t). Or in modern notation, the Euler's equation can be stated as

$$\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u} \cdot \nabla \boldsymbol{u} = -\frac{\nabla p}{\rho} \tag{3.2}$$

The Navier-Stokes equation is stated as

$$\left(\frac{\partial}{\partial t} + \sum_{j=1}^{d} u_j \frac{\partial}{\partial x_j}\right) u_i(x,t) \tag{3.3}$$

$$= \nu \left(\sum_{j=1}^{d} \frac{\partial^2}{\partial x_j^2}\right) u_i(x,t) - \frac{\partial p}{\partial x_i}(x,t) \quad i = 1, ...d \tag{3.4}$$

for all (x,t), or in modern form

$$\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u} \cdot \nabla \boldsymbol{u} = -\frac{\nabla p}{\rho} + \nu \nabla^2 \boldsymbol{u}. \tag{3.5}$$

The viscosity ν is a coefficient of friction and is $\nu > 0$.

To not over-complicate things, we'll stay within the realm of incompressible fluids, which means that the velocity field doesn't diverge and can be stated as

$$\nabla \cdot \boldsymbol{u} \equiv \sum_{j=1}^{d} u_j \frac{\partial u_j}{\partial x_j} = 0 \tag{3.6}$$

for all (x,t).

Fluid flows can be described from different viewpoints. One is called Eulerian and second is called Lagrangian approach. The qualitative difference is in how the observer looks at the physics of the fluid. In the Eulerian approach, the observer stands still at a given space location and watches the fluid flow through defined control space (usually a grid). In the Lagrangian approach, observer moves with the fluid and tracks the infinitesimal elements or "patches" of fluid.

The Eulerian form (3.1) can be also put as "conservation form". Simply put, it's the limiting case $\nu = 0$ of Navier-Stokes. It emphasizes the mathematical interpretation of the equation as conservation equations with the time evolution through control volume fixed in space. The equations are adjustable according to the problem at hand and are expressed based on principles of mass, momentum and energy conservation. The conservation of mass is defined by continuity equation

$$\frac{\partial \rho}{\partial t} + \nabla \cdot j = 0, \tag{3.7}$$

where j defines the flux of total amount of the quantity in the fluid volume. It implies that the conserved quantity cannot be crated or destroyed. In the integral form of continuity equation, the surface integral is defined for any closed surface that fully encloses a

volume.

The Lagrangian approach studies the configuration of the underlying particles, namely the solution of the equation

$$\frac{d}{dt}g(t) = u(t, g(t)) \tag{3.8}$$

To re-cast the the N-S equations to Lagrangian form, which emphasizes the transport along the fluid lines whose tangent identifies with the fluid velocity itself, we can rewrite the above N-S equations to

$$\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u} \cdot \nabla \boldsymbol{u} = -\frac{\nabla p}{\rho} + \nu \nabla^2 \boldsymbol{u}. \tag{3.9}$$

The N-S equations describe the notion of ideal, incompressible fluid. They agree well with real experiments of real fluids under different circumstances. The importance of these equations are of very high order within science and engineering fields, since the presence of fluid flows is prevalent across most human activities and daily life. Engineers are concerned with finding the solutions to N-S equations numerically with the accuracy and effectiveness that conventional computers allows them to achieve. Mathematicians are more interested in knowing if actual solution exists and whether if there is only one solution. Although the N-S equations are known for nearly two centuries (with the Euler equation being discovered more than 250 years ago), they are still very poorly understood, which presents a major challenge. There is no consensus if N-S or Euler solutions exist for all time or if they break down at a finite time. Any mathematically rigorous proof was not yet accepted. In fact, solving the problem of Navier-Stokes existence and uniqueness bares a prize of 1,000,000 US dollars put forward by Clay Mathematics Institute. Having such proof would help with fundamental way of understanding the physical world we live in.

3.2 Real-Life Fluid Flows

Numerical treatment of the N-S equations with use of computer simulation, also called Computational Fluid Dynamics (CFD, discussed in section 2.3), is leading a forefront of computational science. Along with the major progress, CFD analysis also shown many times that despite the harmless look of the N-S equations, they prove exceedingly hard to solve on digital computers due to the chaotic behaviour of turbulence at higher Reynolds numbers. Turbulence results from acting of large-scale advection, the term $\boldsymbol{u} \cdot \nabla \boldsymbol{u}$ and small-scale dissipation (Succi; 2018). This ratio is measured in Reynolds number, i.e.

$$Re = \frac{UL}{\nu} \tag{3.10}$$

In most of the real-life fluid flows we encounter, the Reynolds number easily exceeds millions. The reason behind the overwhelming non-linearity over dissipation is best highlighted by re-casting the Reynolds number to Von Kármán ratio

$$Re = \frac{U}{c_s}\frac{L}{l_\mu} = \frac{Ma}{Kn} \tag{3.11}$$

where c_s is the speed of sound and $l_\mu = \nu/c_s$ is the molecular mean free path (the average distance travelled by a molecule before colliding with another molecule). The Ma stands for Mach number, representing the ratio of fluid to sound speed

$$Ma = \frac{U}{c_s} \tag{3.12}$$

and Kn is a Knudsen number, representing the ratio of molecular mean free path length to a physical length scale

$$Kn = \frac{L}{l_\mu}. \tag{3.13}$$

In air, molecules travel about 0.1μ before collision happens. If the physical scale length is $L = 1m$, the resulting ratio L/l_μ is 10 million. Reynolds number is huge because it measures the physical length in units of mean free path. The advection acts at macroscopic scales, but dissipation in much smaller scales, also called Kolmogorov scales. According to the scaling theory formulated by Kolmogorov, the smallest active scale in turbulent flow a given Reynolds number is given by

$$l_d = \frac{L}{RE^{3/4}}, \tag{3.14}$$

and thus the number of degrees of freedom in a turbulent flow of size L is

$$N_{dof} = \left(\frac{L}{l_d}\right)^3 \approx Re^{9/4}. \tag{3.15}$$

In realistic flows with Reynolds number in millions, or $Re \approx 10^6$, this results in about $N_{dof} \approx 10^{14}$ degrees of freedom. Current state-of-the-art CFD solvers that simulate turbulent fluid flows of such scale can handle orders of tens of billions of degrees of freedom on conventional supercomputers. Adding to this, the fluids of practical interest usually move in complex geometries. Because of this combined difficulty, simulating close to real physics with N-S equations in classical CFD solvers is extremely difficult (Succi; 2018).

3.3 Lattice Boltzmann Method

Majority of traditional CFD methods focus on various discretization of N-S equations, as a set of non-linear partial differential equations, starting with their continuum form and transforming them for the use on discrete grid (Eulerian form) or moving along the with the fluid (Lagrangian form). For simulation to be spatially resolved, the grid spacing must be smaller than the Kolmogorov scale ($\delta x < l_d$). According to this, the number of grid points must exceed the number of physical degrees of freedom dictated by physics

of turbulence, hence they do not depend on the specific discretization procedure (Succi; 2018).

Conversely, alternative method like Lattice-Boltzmann (LBM) doesn't go down the path of discretizing partial differential equations in N-S. Instead, the main idea is to track distributions of fictitious particles between each node of discrete lattice and extract the macroscopic fluid behavior as an emergent phenomenon (Succi; 2018). This dynamics is the heritage of a LBM predecessor, the Lattice Gas Automata (LGCA) (Frisch et al.; 1986; Hardy et al.; 1973).

The big advantage is that the underlying dynamics of LBM is much simpler than the dynamics of hydrodynamic fields. As a result, the popularity of LBM has steadily grown since its inception from LGCA. It 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 (Li, Luo, Kang, He, Chen and Liu; 2016).

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. 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 $dt \to 0$:

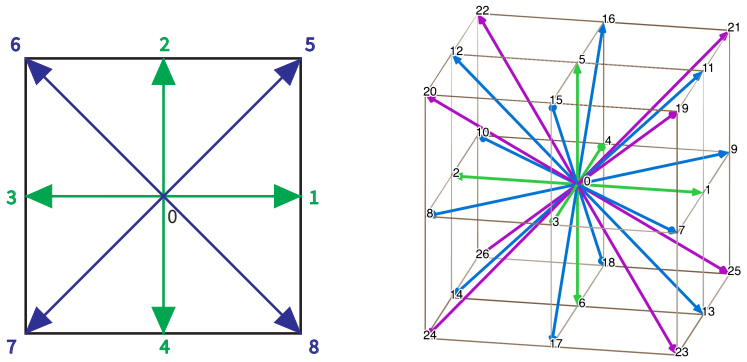
$$\frac{df}{dt} = \Omega(f). \tag{3.16}$$

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

$$\Omega_i = \frac{1}{\tau}(f_i^{eq} - f_i), \tag{3.17}$$

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, $i = 0, 1, ..., Q$, with $Q = 9$ for D2Q9 and $Q = 27$ for D3Q27 stencil respectively (Figure 3 – 1).



(a) D2Q9. (b) D3Q27.

Figure 3 – 1: Lattice stencils.

The LBE can be stated as

$$f_i(\boldsymbol{x} + \boldsymbol{e}_i\Delta t, t + \Delta t) = f_i(\boldsymbol{x}, t) - \frac{1}{\tau}\Big[f_i(\boldsymbol{x}, t) - f_i^{eq}(\boldsymbol{x}, t)\Big], \tag{3.18}$$

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). Discrete velocities $\boldsymbol{e}_i$ in D2Q9 model can be expressed in an array as

$$\boldsymbol{e}_i = \begin{bmatrix} 0 & 1 & 0 & -1 & 0 & 1 & -1 & -1 & 1 \\ 0 & 0 & 1 & 0 & -1 & 1 & 1 & -1 & -1 \end{bmatrix} \tag{3.19}$$

for the single node in 2D grid and for D3Q27 the array is extended to accommodate 27 velocities moving in 3D grid

$$e_i = \begin{bmatrix} 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & 1 & -1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \end{bmatrix} \tag{3.20}$$

Macroscopic quantities are obtained from hydrodynamic moments of the distribution function (Eq. 3.21, 3.22)

$$\rho = \sum_{i=0}^{Q} f_i, \tag{3.21}$$

$$\rho \boldsymbol{u} = \sum_{i=0}^{Q} \boldsymbol{e}_i f_i. \tag{3.22}$$

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

$$f_i^{eq} = \omega_i \rho \left(1 + 3\frac{\boldsymbol{e}_i \cdot \boldsymbol{u}}{c_s} + \frac{9}{2}\frac{(\boldsymbol{e}_i \cdot \boldsymbol{u})^2}{c_s^2} - \frac{3}{2}\frac{\boldsymbol{u}^2}{c_s^2} \right), \tag{3.23}$$

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

$$\omega_0 = 4/9, \tag{3.24}$$

$$\omega_{1,2,3,4} = 1/9, \tag{3.25}$$

$$\omega_{5,6,7,8} = 1/36. \tag{3.26}$$

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 $\mathscr{O}(M^2)$ has to be made.

Two general steps of the LBM solver involve solving Eq. 3.27 for collision and Eq. 3.28 for streaming in each iteration.

$$f_i(\boldsymbol{x}, t+\Delta t) = f_i(\boldsymbol{x}, t) - \frac{1}{\tau}\Big[f_i(\boldsymbol{x}, t) - f_i^{eq}(\boldsymbol{x}, t)\Big]. \tag{3.27}$$

$$f_i(\boldsymbol{x} + \boldsymbol{e}_i\Delta t, t+\Delta t) = f_i(\boldsymbol{x}, t+\Delta t). \tag{3.28}$$

To overcome difficulties of numerical instability in applying the 3D LBM method, the multiple-relaxation-time (MRT) scheme is useful to stabilize the solution and to obtain satisfactory results. The MRT model allows for independent tuning of the relaxation times for each physical process (Suga et al.; 2015). It's therefore natural to extend current work to include MRT. In this study we implemented non-orthogonal MRT-LBM in D3Q27 because it simplifies the transformation between the discrete velocity space and the moment space (Fei et al.; 2019).

The collision operator in MRT-LBM is defined as

$$\Omega_{MRT} = M^{-1}SM, \tag{3.29}$$

where S is a diagonal relaxation matrix and M being the transformation matrix (Fei et al.; 2019). The collision step is performed in moment space. Formally it can be defined as

$$m^* = m - S(m - m^{eq}), \tag{3.30}$$

where $m = Mf$ and $m^{eq} = Mf^{eq}$. After the collision step, the distribution function is be reconstructed by $f^* = M^{-1}m^*$ and used in the streaming step as usual

$$f_i(\boldsymbol{x} + \boldsymbol{e}_i\Delta t, t + \Delta t) = f_i^*(\boldsymbol{x}, t + \Delta t). \tag{3.31}$$

3.4 Boundary Conditions

Dynamics of the fluid flow is dependent on the surrounding environment described mathematically with suitable boundary conditions. They play a crucial role as they select solutions which are compatible with external constraints. Their overall treatment makes the difference in the quality of CFD simulation (Succi; 2018).

At the start point during the simulation initialization phase, an initial conditions have to be set up. They are basically a boundary conditions at $t = 0$. Popular practice is to set the PDF to local equilibrium according to the prescribed value of the hydrodynamic field and initial density with initial velocity to zero

$$f_i(\boldsymbol{x}, t_0) = f_i^{eq}(\boldsymbol{x}, t_0), \tag{3.32}$$

$$\rho_0(\boldsymbol{x}) = \rho(\boldsymbol{x}, t_0), \tag{3.33}$$

$$\boldsymbol{u}_0(\boldsymbol{x}) = \boldsymbol{u}(\boldsymbol{x}, t_0). \tag{3.34}$$

Geometric boundary conditions have multiple forms, namely:

- periodic,
- no-slip,
- free-slip,
- frictional slip,
- sliding walls,
- open inlet and outlet.

Various LBM solvers implement different types of listed boundary conditions, but usually for the sake of simplicity and lower computational cost, periodic and no-slip are used. Periodic boundary conditions are one of the simplest to implement. They are typically used to isolate bulk fluid phenomena from boundaries of physical system. No-slip boundary-condition, also known as bounce-back, refers to solid surfaces with zero fluid velocities. They expect that the solid walls have enough rugosity to prevent any additional motion of the fluid relative to the wall (Succi; 2018). Effectively, this results in a reversal of the discrete velocities direction of f_i (Mawson; 2014).

In current work, the simulation software implements a simple no-slip boundary. In places like inflow and outflow of simulated pipe for Kármán vortex test case, we implemented periodic boundary conditions (Succi; 2001).

3.5 Multiphase Flows

An important area of LBM applications is multiphase and multicomponent simulations. Such phenomena can be observed when fluids of different densities meet at some interface, usually between gases and liquids. A deeper understanding of the fundamental physics of such complex interfaces is of great importance in many natural and industrial processes.

Dynamics of multicomponent flows are difficult to investigate due to thinness and complexness of the interface between them. The problem with multiphase flows, in turn, can

be very short times of change between phases. Additionally, 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 (Fei et al.; 2019).

Development of accurate and robust multiphase models to investigate complex processes at the interfaces is crucial. Such models mostly fall into the following categories:

- color-gradient methods,
- pseudopotential methods,
- free-energy methods,
- phase-field methods.

All of aforementioned methods were successfully applied to dynamic multiphase flows at large density ratios ($\rho_l/\rho_g \approx 10^3$) and high Reynolds numbers (Li, Luo, Kang, He, Chen and Liu; 2016). Practical applications include droplet splashing and droplet collision, water-gas two-phase transport in fuel cells, electrolyte transport dynamics in batteries, and phase-change heat transfer like boiling or evaporation.

Although multiphase flows are interesting use-case for LBM simulation, their support was not implemented in current solver implementation described in this thesis. But at the same time, some parts of the research that is being done at the Institute of Control and Informatization of Production Processes at Technical University of Košice, where I currently work, focuses on steelmaking and underground gasification of coal. Simulating involved processes with LBM and multiphase methods can pave a new way towards better understanding of the phenomena in those areas.

3.6 Adaptive Mesh Refinement

Turbulent flows or multiphase flows involves different densities, like gas bubbles, liquid droplets or creation of foam. Turbulence is a complex, non-linear, multi-scale phe-

nomenon, which is very difficult to simulate accurately for smaller parts of the turbulent flows. Problem with bubbles is that the gas-liquid interface typically spans to a thickness of 3-5 grid spacing. Both of these problems need denser grid to accurately represent them, thus requiring more computational power to simulate them effectively. Typically, not everything in the computational domain needs to be represented with the same amount of accuracy.

Effective technique for such problems can be an Adaptive Mesh Refinement. With this technique, the mesh resolution is not constant, but adapts according to the accuracy requirements in specific part of the domain. Fine mesh resolution is used for problematic parts where high accuracy is needed and coarse mesh is applied to the bulk fluid away from the interface. This gives a tremendous benefit in reducing the computation cost (Yuan et al.; 2017).

3.7 Complex Fluids and Beyond

In modern science and technology, we often deal with complex flows such as flows with chemical reactions, phase transitions, suspended particles, etc. Broad range of physical phenomena go beyond the classic Navier-Stokes equations. For instance, it's possible to implement a electromagnetic forces within the Lattice-Boltzmann formalism, opening the door to magnetohydrodynamics. As a result, this allows extending LBM to be used for studying properties and dynamics of plasmas, liquid metals, salt water or electrolytes.

Another interesting frontier of modern physics is soft matter, in which LBM is also trying to establish itself as a viable method for studying polymers, foams, gels, granular materials, liquid crystals and some biological materials.

It doesn't stay there. Lattice-Boltzmann formalism doesn't restrict itself only to classical Newtonian mechanics, but allows to be extended to the case of relativistic as well as quantum fluids in a comparatively simple manner. The kinetic theory harnessed in LBM can serve as a very effective functional generator of a variety of linear and non-linear

partial differential equations of mathematical physics (Succi; 2018).

4 GPU Computing

> *"It's hardware that makes a machine fast. It's software that makes a fast machine slow."*
>
> – Craig Bruce

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 computations. 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 computation 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. Although, more computationally intensive tasks like Monte Carlo simulations of molecular dynamics are still hard to speed up to an order of real-time simulation.

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. In following sections, various techniques on how to tap into the power of GPU computing will be presented.

4.1 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, which 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 use them, software have to employ parallel programming model.

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. 1.

```
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 1: 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.

We can write a program that executes on the GPU, which 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. 2).

```
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 2: Pseudocode with serial loop.

Considering the small size of this toy problem, the difference between CPU and GPU version in computation speed won't be noticeable. But with larger problems in terms of bigger and complex physics simulations, amount of operations can grow to extreme numbers. At that stage, speed differences between slower CPU and faster GPU computation can reach orders of hundreds.

In this work we compared benchmarks of simulation software, developed with parallel algorithms, on CPU and different GPUs. Performance analysis is described in section 6.6, showing noticeable differences.

4.2 Heterogeneous Computing

Scaling of the computational power in high-performance accelerators and supercomputers was historically done by adding more and more CPUs together to a multi-processor

distributed system, also known as many-core system. When single CPU started to have multiple cores, scaling the power of those systems was done by switching older CPUs to multi-core ones. But adding more and more cores is not infinite process. With increasing number of transistors stuffed into the same size of the chip, the power and heat starts to rise again. Physical limitations were starting to manifest themselves.

More than a decade ago, the GPUs were starting to supplement CPUs for computational work. The general processor (the CPU) with one type of architecture was being helped by the other processor with different architecture type (the GPU). This was the start of moving from homogenous computing towards heterogeneous computing, for which the coordination of two or more different processors is needed.

4.3 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).

4.4 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

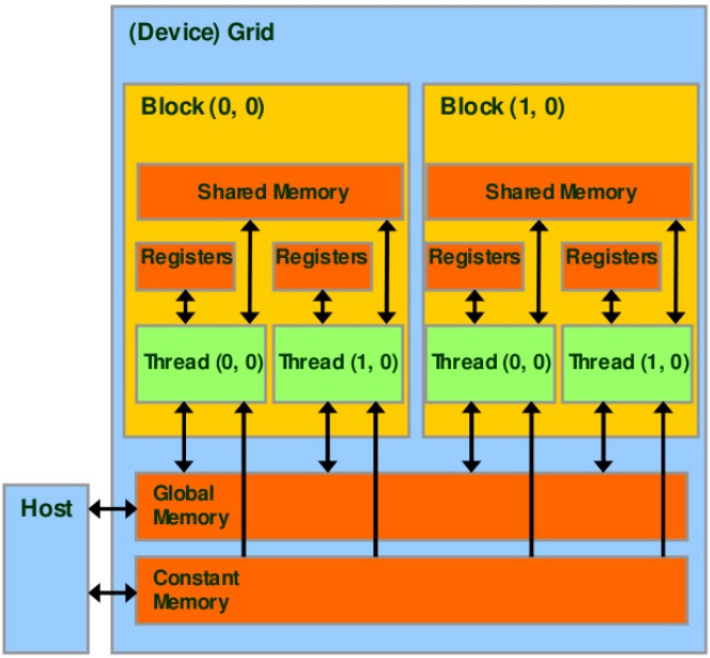


Figure 4 – 1: CUDA memory model

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.

4.5 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 different computational backends. This way it is possible to switch to CPU, GPU, FPGA, or another type of accelerator at runtime **?** (Alg. 3), 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 3: 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 4 – 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 4 – 1: Functions for working with OpenCL context in ArrayFire-OpenCL interoperability scenario.

4.6 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 multithreaded 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 Boroni et al. (2017); Delbosc (n.d.); Delbosc et al. (2014); Harwood et al. (2018); Harwood and Revell (2017); Januszewski and Kostur (2014); Koliha et al. (2015); Kotsalos et al. (2019); Szőke et al. (2017). There is increasing amount of studies on memory access efficiency and optimization techniques 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.

5 Visualization Methods

"Errors are the portals to discovery."

– James Joyce, *Ulysses*

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 can be not only scientists, but also general public.

But let's keep focus on a scientific visualization in the course of this work.

Numerical simulations of Computational Fluid Dynamics (CFD) often output massive amounts of high-dimensional data. Visualizing them is usually done as a separate, independent step after the data are generated and stored. These types of visualizations can be classified as post-hoc.

With current performance of conventional computers, it's no longer necessary to separate these steps. It's now possible to have a "live" visualization running alongside the simulation. It all depends on the complexity of investigated fluid flow phenomena and the power of hardware at hand. Visualization like this is called real-time.

This section will explore differences between post-hoc and real-time visualization in sections 5.1 and 5.2. From there on, it is just a small step from real-time visualization towards an interactive one. That step will be taken in section 5.3, and another beyond simple interactivity on 2D screen. With the emerging virtual reality technology, section 5.4 will explore how immersive user interface can take things further.

5.1 Post Hoc Visualization

Scientific visualization is usually performed as a post-processing task (Kress; n.d.). The output from simulation is saved to disk and then the data are loaded into visualization software for further processing. The benefit to this approach is that the majority of computation load is focused on graphics rendering to the screen. Final products of the post-hoc visualizations are high-quality pictures of 2D or 3D post-processed simulation data.

When doing a post-hoc visualization of CFD simulation, user's main concern with this approach is the post-processing. Three-dimensional rendering is done by marching squares algorithm by the program automatically. For fluid flows around complex geometries, the geometric structures should generally be visible, with filtered data shown as isolines. Commercial software like ANSYS, SimScale or COMSOL Multiphysics have integrated visualization software into their platforms (Fig. 5 – 1 with the example visualization).

They provide plethora of options.

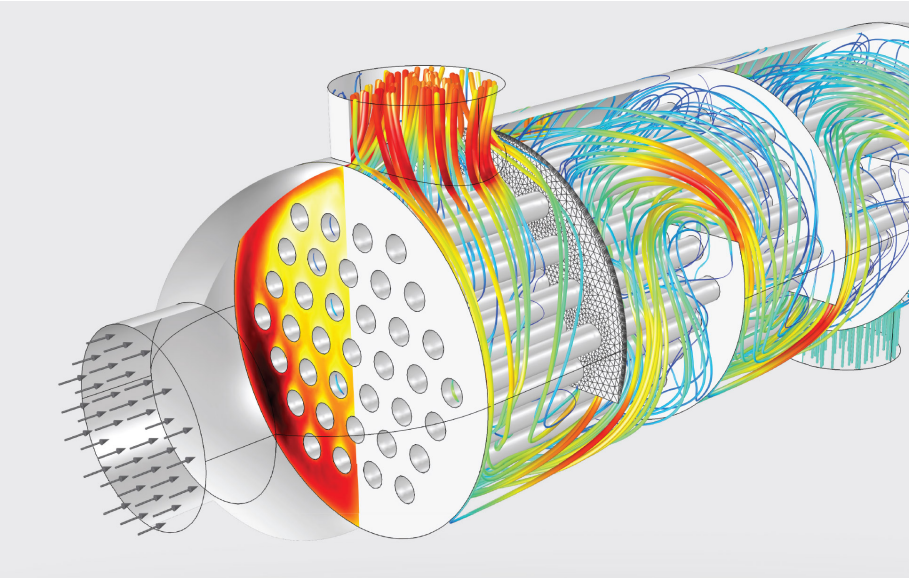


Figure 5 – 1: Example of post-hoc visualization with applied post-processing techniques.

A Visualization Toolkit (VTK) exists as an open-source alternative to proprietary visualization software. It's fully programmable and it can be deployed on highly parallelized architectures with large core counts. It implements polygonal, glyphing and volume rendering for 3D data and plotting capabilities for creating 2D charts. API is provided for C++ with Python and Java wrappers (Hanwell et al.; 2015).

VTK has been implemented and integrated into large number of applications for scientific data visualizations, like ParaView, VisIt, 3D Slicer, Medical Imaging Interaction Toolkit and so on. The aforementioned ParaView is worth a more detailed mention. It is developed by the team behind VTK (Kitware, Inc.) and integrates all the programmable parts of it into the cohesive experience packaged as desktop client application with GUI. Besides all of the classic tools for preparing the quality visualization output, it can render 3D content into virtual reality headsets like Oculus Rift or HTC Vive (Ahrens et al.; 2005). Both VTK and ParaView can leverage parallel architectures with built-in Message-Passing Interface (MPI) support.

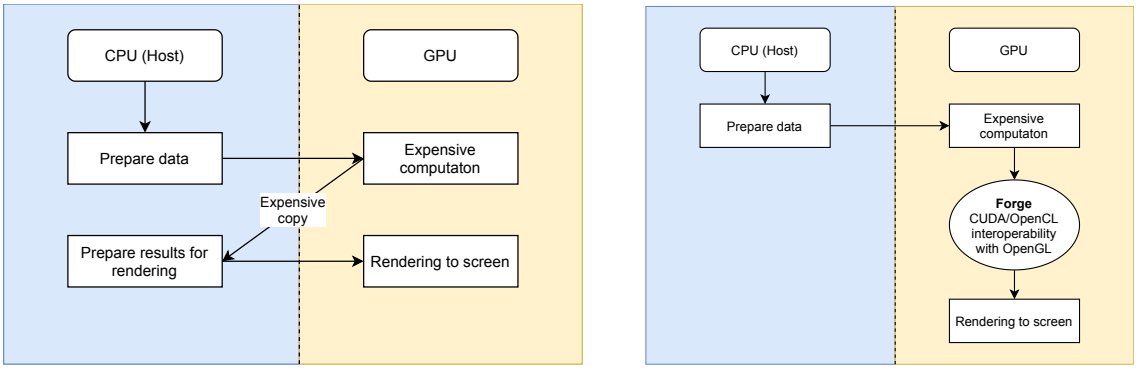
5.2 Real-time Visualization

Real-time simulation, i.e. the ability to simulate a virtual system as fast as the real system would evolve, can be beneficial to many engineering applications. To achieve real-time fluid flow simulation, numerical methods need to be selected carefully to make full use of the hardware capabilities. Generally, these were often simplified for use in games with lower accuracy, as the focus in gaming environments is more on creating visually appealing animation rather than aiming for physical accuracy (Delbosc; n.d.).

The Lattice-Boltzmann method in context of CFD simulations is ideally suited to acceleration on GPUs due to its spatial and temporal locality. Thanks to this they have extremely high computational throughput compared with traditional CFD methods. Therefore, with such method as LBM, it is now possible to achieve sound physical accuracy and good enough speed to reach real-time simulation capabilities.

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

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



(a) Common visualization pipeline.

(b) High-performance visualization pipeline.

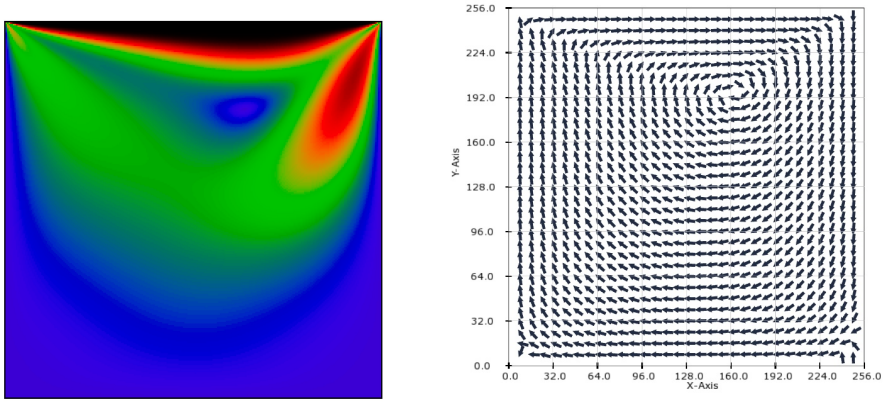
Figure 5 – 2: Differences between CPU-bound visualization with expensive copying bottleneck and high-performance GPU-bound visualization keeping full speeds of high-bandwidth of PCIe interface.

Forge to effectively work with the data. In ArrayFire (and generally in programming), function for this operation is called `cast` (Alg. 4).

```
// C
af_array A_f64 = af_randu(100,100);
af_array B_f32;
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 4: Converting to single precision floating point for Forge visualization in C, C++ and Rust

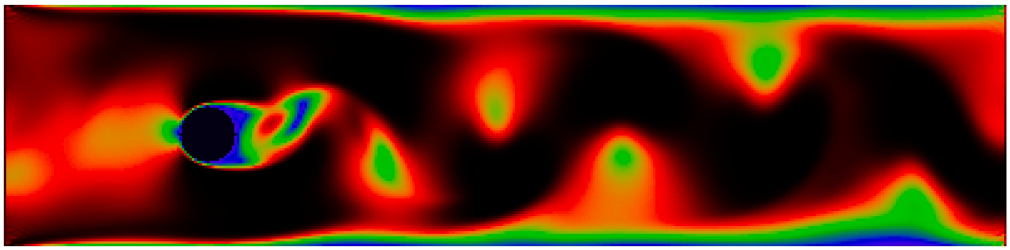
While developing the C++ and Rust implementations of LBM simulation backend, we used `image()` and `vector_field_2()` functions from Forge library for visualizing density and velocity of both lid-driven cavity and Kármán vortex street test cases in 2D (Figure 5 – 3 and Figure 5 – 4).



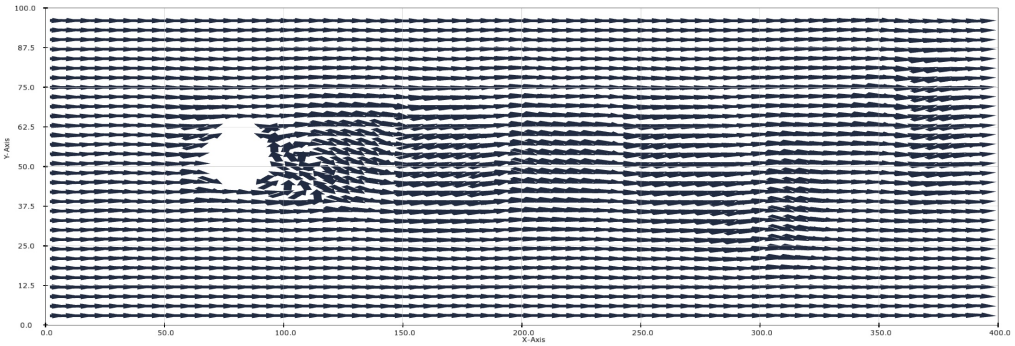
(a) Visualization of lid-driven cavity simulation.

(b) Velocity vector field.

Figure 5 – 3: Lid-driven cavity test case at 128×128 resolution after 5000 iterations.



(a) Visualization of Kármán vortex street simulation.



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

Figure 5 – 4: Kármán vortex street (channel flow past circle-shaped obstacle) test case at 1000×300 resolution after 5000 iterations.

5.3 Interactive Visualization

Interpreting simulation results is often a laborious process. Already few decades ago, it was clear that to understand fluid phenomena in depth, we should move towards interactive simulation (Frisch et al.; 1986). This would allow run-time manipulation of geometrical and physical simulation variables, opening ways to rapidly and intuitively investigate different scenarios and design configurations.

Recently, the increase of computational power and advances in general-purpose computing on GPUs (GPGPU) made this possible (Delbosc; n.d.; Glessmer and Janßen; 2017; Harwood and Revell; 2017; Koliha et al.; 2015). 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 algorithms for our simulation software is the ability to perform high performance computation and still keep the CPU free to do the additional work. The reason for it is that the visualization part is performed in virtual reality, which needs most of the CPU resources for processing the user tracking. Thanks to the fact that ArrayFire library was used, many hardware-specific optimizations were done behind-the-scenes. Although, there is growing body of work on optimizing LBM simulations (Harwood et al.; n.d.; Harwood and Revell; 2017; Körner et al.; 2006; Tran et al.; 2017; Wang et al.; 2019; Wittmann; 2018). In the simulation software implementation described in this thesis, I've used some of the methods described in these articles. They are described in more detail in section 6.3.

5.3.1 Steering the Running Simulation

When the simulation together with visualization are fast enough to stream live data to the screen while the simulation is running, it's convenient to send signals to the simulation to change itself with updated parameters in real-time. Thanks to this ability, the process of optimizing the physical model employed in the CFD analysis, or experimentation phase towards end result specified by the researcher who is using it, can happen much faster, thus becoming an order of magnitude more efficient (Kreylos et al.; 2002).

In some applications where the flowing fluid has to deform into certain shape, steering the running simulation can be made fully autonomous by applying the concepts from control theory. Such application is usually employed in visual effects or gaming industry. Fluid shape is difficult to control manually by hand. For controlling the shape as a whole, the forward gradient computation method is usually applied. Recently, a more efficient method, called adjoint method, has been applied to this problem, which improved the efficiency by an order of magnitude, allowing for fully controlled 3D simulations in the fraction of a time (McNamara et al.; 2004).

However, the current focus of this work is on design and implementation of the environment where humans can interact with running computer simulations and interfere with them towards deeper understanding or constructing better models. During the design process, I was inspired by following an approach from gaming industry, where designing an interactivity and proper instant feedback is of paramount importance - without it the game would not make sense, would be easy for the player to get stuck or the game itself would become boring very quickly (Victor; 2006). Rather than following a traditional paradigm in human-computer interface (HCI) design, in which the focus is based on efficiency, I've chosen to design towards helping user with experimentation phase within the UI (Spadafora et al.; 2016).

5.3.2 Time Manipulation

The process of modeling in the context where simulation is integrated into tighter loop to help with creating a better model, the typical approach of iterating towards the model improvement is still slow and tedious. It involves trying to come up with a mathematical representation as close to the real world phenomenon as possible at the initial phase, solving it numerically with a simulation, analyzing the outcome, then afterwards going back to the "drawing board" and work towards more accurate model by changing the parameters of a model and re-running the simulation again and again. With speedups provided by GPU computing and efficiently parallel implementations of methods like LBM, the experimental phase of the model's optimization process, based on numerical analysis by simulation, can be significantly improved. The novel approach that could help is ability to "see" through time and manipulate it within the recorded history of many simulation checkpoints.

One of the best example that illustrates the benefit of time manipulation is from a software tool prototype created by Bret Victor. In his presentation "Inventing on Principle", he highlights the time manipulation approach through player's recorded history of jump across obstacle, with his shadow following a trajectory specified by physics parameters (gravity, velocity). Those parameters can be changed on-thy-fly during the interactive game development process, thereby inciting a quick and efficient experimentation instead of classic and tedious update-and-compile approach which takes number of blind attempts to hit the correct value. Screenshot highlighting the time manipulation interface is shown in Figure 5 – 5.

5.4 Virtual Reality

Typical non-immersive visualization systems implemented for the conventional desktop PCs with 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 user's head position.

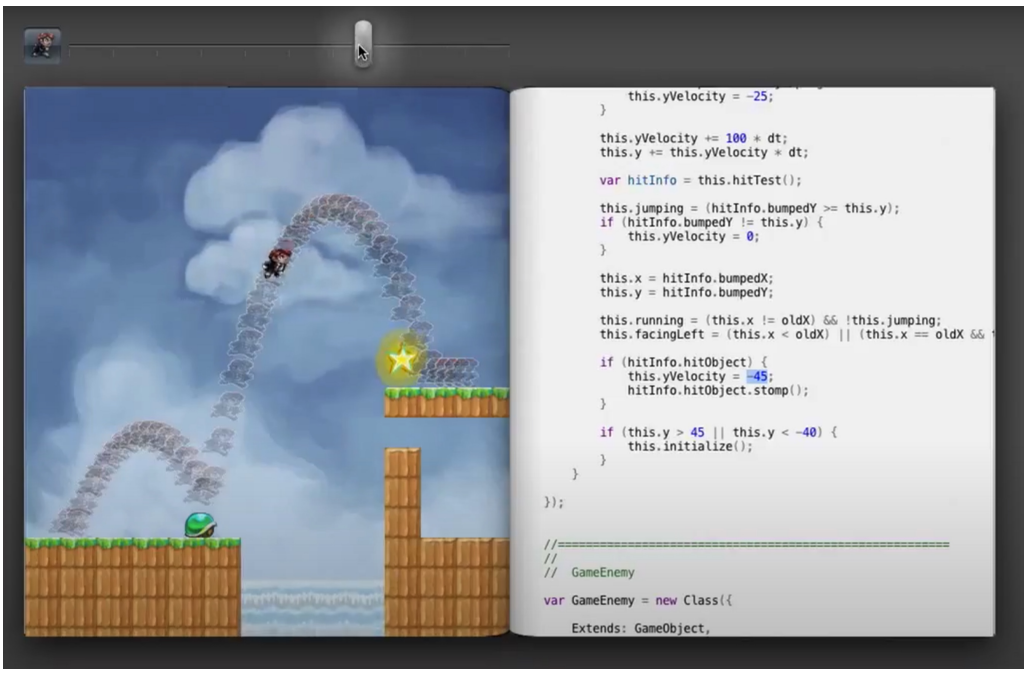


Figure 5 – 5: Screenshot from Bret Victor's presentation "Inventing on Principle" showcasing time manipulation interface. It allows developer to see the game object's trajectory through history of recorded states along specified time range and step through them. Changing the objects' physical parameters triggers the re-computation of history of states, resulting in immediate feedback of trajectory change (Victor; 2018).

A fundamental difference between three-dimensional (3D) virtual environments accessible through 2D PC monitor 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 virtual. However, in such an environment, an object may also appear to change shape not due to an altered position of the viewer, but because the object itself has moved to a different position. Immersive virtual reality 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 vol-

ume to be explored, as well as provide simple control of the visualization environment through interfaces integrated into the environment (Bryson; n.d.).

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.

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; n.d.). 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.

With the addition of inside-out tracking in VR headsets like Widows Mixed Reality and Oculus Quest, it brought the hand tracking support into the software development kits for each platform. More and more VR developers started to leverage the hand tracking in the interfaces of their applications. Thanks to the growth of gestures usage in virtual reality and embodied cognition, there have been various new technologies developed to either improve the modelling efficiency, or to provide more nature intuitive experience to the

users (Dangeti et al.; 2016).

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.

6 Implementation of a Simulation and Visualization Software

As the previous chapters laid the theoretic framework regarding the methods employed in modeling, simulation and visualization of fluid dynamics, together with approaches to writing efficient software for multicore hardware accelerators, from now on the focus of this work will change to description of an actual implementation of simulation and visualization software integrated into a virtual reality platform.

The following chapter, section 6.1 will deal with the description of technology stack; in section 6.2, each part of the Lattice Boltzmann algorithm implementation will be discussed; section 6.3 describes various optimization techniques employed in the implementation of LBM simulation software that runs on GPUs; section 6.4 will show how the virtual reality user interface (VRUI) was built; in section 6.5, the simulation and visualization software interconnection will be described; and, in closing, section 6.6 will provide a performance analysis of the simulation software implementation.

6.1 Technology Stack

From a high-level perspective, presented software application consists of three parts, namely: (1) the LBM solver for CFD simulation, (2) the interactive visualization part, and (3) the virtual reality user interface for immersive human-computer interaction.

The first part, a LBM solver, is written as separate program in Rust programming language leveraging a high-performance, parallel computation library, ArrayFire. It exposes a Foreign Function Interface (FFI) for external programs to work with the solver by calling functions for simulation initializations, performing the computation, and getting the output from it.

Both second and third parts are handled within Unity game engine.

The second part, an interactive visualization, is implemented as two separate C# scripts called `LBM.cs` and `Sim.cs`. The `LBM.cs` represents the interconnection between Unity and external Rust program, the LBM solver, as a Native Plugin. The `Sim.cs` scripts implements an interaction and control interface for simulation lifecycle and a 2D texture to visualize simulation results.

The third part, a virtual reality user interface (VRUI), is implemented with Unity's XR Interaction Toolkit, which drastically simplifies cross-platform development of VR experiences and provides many useful elements for quick UI prototyping.

6.1.1 Cross-platform Development with ArrayFire

The simulation part of the application 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. However, 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. It 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 (**?**). Developers can use several different methods for manipulating arrays and matrices implemented as a `Array` container object. The functionality includes:

- `moddims()` - lazily 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

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. Most of them operate on `Array`s as a whole i.e. on all elements in parallel, as the library is mostly vectorized.

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 prototypes, 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 as a language that could one day replace C or C++ as a go-to language for writing performant low-level applications and systems. 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 daily. 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 soft-

ware. 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 programming language design (Klabnik and Nichols; 2018). 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.

The simulation backend (i.e. actual implementation of LBM solver) was written in both C++ and Rust programming languages to compare the resulting performance and overall development experience. Both of those versions perform very similarly, but since Rust is being very interesting to work with in term of memory-safety, it was picked as a winner. Therefore the simulation backend that is used in tandem with VR visualization frontend is actually implemented in Rust.

6.1.3 Hardware

The focus of this work was to build a high-performance simulation software with integrated VR visualization that can be used on variety of platforms. VR is readily supported on Windows systems, with bit of an additional tuning here-and-there on Linux distributions. MacOS is not supported from any platform as of today, but the hardware it runs on is not powerful enough anyway (this can change in not-so-distant future, though!). Hardware requirement for running the VR application that is described in this work is clearly displayed in Table 8 – 1 in section 8.1.5 of Appendix A. The performance of simulation backend was thoroughly benchmarked. List of GPUs that was used throughout the

development is clearly displayed in Table 6 – 1 in section 6.6.1.

6.2 Implementation of Lattice Boltzmann Method for GPUs

The simulation part of the virtual reality software was implemented in both C++ and Rust to assess performance of ArrayFire library when used for developing GPU-based solver for LBM. Resulting speeds for both lid-driven cavity and Kármán vortex street problems were nearly identical across multiple grid densities (performance analysis is discussed in more detail in section 6.6). Therefore, as discussed before in section 6.1.2, Rust programming language was chosen for implementing the simulation part of the software platform, as it adds some interesting properties to the software development process, making up for the current lack of libraries.

In this section, each step in the simulation program lifecycle is presented, showing how different parts are constructed so they can be called from the Unity game engine (as a Native Plugin).

6.2.1 Initialization

In Rust, the common data structure for composing various types of data under one roof is `struct`. In the simulation implementation, `struct Sim {...}` stores the all parameters and other data structures that are needed during the program lifecycle.

Structs can be initialized in different ways. Most common way is via the constructor method `new()`. For proper constructing of `Sim` and its properties, the struct implementation has to be defined with `impl Sim {...}` (Listing 5). Additional directive `#[repr(C)]` is added on top of `Sim` struct to specify fairly simple intent: do what C does. That means the order, size, and alignment of fields is exactly what would be expected from C or C++. Any type that is expected to be passed through an FFI boundary should have `#[repr(C)]`, as C is the lingua-franca of the programming world. This is necessary to do more elaborate manipulation with data layout such as reinterpreting values as a different type.

```rust
#[repr(C)]
pub struct Sim {
  // ... all the fields are defined here
}

impl Sim {
  pub fn new(nx: u64, ny: u64, initial_density: f32, initial_ux: f32,
    omega: f32, obstacle_x: u64, obstacle_y: u64, obstacle_r: u64
  ) -> Result<Sim, String> {
    // ... rest of the implementation
  }
}
```

Listing 5: Implementation of Sim struct with constructor method.

The simulation program needs to be initialized at the start with correct parameters. These are passed into the `new()` constructor method as its arguments and then used for initialization in ArrayFire Arrays. There are number of independent Arrays being initialized in this method, such as discrete velocities, weights for computing particle distributions in equilibrium, etc. (Listing 6). During this phase, initial (physical) conditions and boundary (geometrical) conditions has to be set so the simulation can be correctly started. All of the conditions are also stored in `Sim` struct. The actual construction of `Sim` is done at the end of `new()` method.

```rust
let t1: f32 = 4. / 9.;
let t2: f32 = 1. / 9.;
let t3: f32 = 1. / 36.;
// Discrete velocities
let ex = Array::<f32>::new(&[0., 1., 0., -1., 0., 1., -1., -1., 1.], dim4
    !(9));
let ey = Array::<f32>::new(&[0., 0., 1., 0., -1., 1., 1., -1., -1.], dim4
    !(9));
// weights
let w = Array::new(&[t1, t2, t2, t2, t2, t3, t3, t3, t3], dim4!(9));
// Initial physical parameters
let mut density = constant::<f32>(initial_density, dim4!(nx, ny));
let mut ux = constant::<f32>(initial_ux, dim4!(nx, ny));
let mut uy = constant::<f32>(0.0, dim4!(nx, ny));

// ...
// At the end of new(), construct the Sim and return itself
Sim { ex, ey, w, density, ux, uy, /* rest of the params...*/ }
```

Listing 6: Setting initial parameters that are independent from constructor method's input arguments.

6.2.2 Boundary Conditions

The position of solid boundary conditions, also referred to as physical boundary conditions, are implemented with ArrayFire Arrays as a mask on 2D or 3D grids. Instead of using boolean mask of true and false values, this implementation works with numerical constants - 1 representing a “bound”, or solid node (true), and 0 a fluid node (false).

For example, in Kármán vortex street problem, we have to specify where a solid geometry—a circle, around which the fluid flows—resides. Additionally, simulation have to know which walls of the computational domain (in context of Euclidean space) are solid and which are open. The mask describing solid nodes along whole Euclidean domain is stored in `bound` variable.

The circle geometry implementation is done by mathematically describing a relation of each point within the circle radius (Listing 7).

```
let mut bound = constant::<f32>(1.0, dim4!(nx, ny));
// circle
let r = constant::<f32>(obstacle_r as f32, dim4!(nx, ny));
let r_sq = &r * &r;
let circle = moddims(
  &le(
    &(pow(
      &(flat(&x) - obstacle_x as f32),
      &(2.0 as f32), false)
    + pow(
      &(flat(&y) - obstacle_y as f32),
      &(2.0 as f32), false)
    ),
    &flat(&r_sq),
    false
  ),
  dim4!(nx, ny),
);
bound = selectr(&bound, &circle, 0.0 as f64);
```

Listing 7: Setting the boundary conditions of a solid circle geometry in 2D Karmán vortex street.

For the top and bottom solid walls, the implementation is very simple and is done by calling `set_col` function to mutate `bound` Array and set whole column to constant 1.0 (Listing 8).

```
// top
set_col(&mut bound, &constant::<f32>(1.0, dim4!(nx)), 0);
// bottom
set_col(
  &mut bound,
  &constant::<f32>(1.0, dim4!(nx)),
  ny as i64 - 1,
);
```

Listing 8: Setting the boundary conditions of a solid walls.

For each solid boundary node, a physical boundary conditions have to be specified. In this work, we use no-slip boundary condition, which is easy to implement. With no-slip boundaries, fluid adheres to the stationary wall and maintains zero velocity relative to the solid wall. Implementation of this boundary condition is shown in Listing 9).

```
// matrix offset of each solid node
let on = locate(&bound);

let zeroed_on = constant::<f32>(0.0, on.dims());

eval!(ux[on] = zeroed_on);
eval!(uy[on] = zeroed_on);
eval!(density[on] = zeroed_on);
```

Listing 9: No-slip boundary conditions.

6.2.3 Collision

In collision phase of the LBM algorithm, after fictitious particles collide, the particle distribution functions relax towards the equilibrium state in each node. A collision operator Ω (implemented as a variable `omega`) states the rate of this change. As described in section 3.3, it's often called BGK operator.

Setting the omega initially can be based on findings of other researchers in the field of LBM, or can be computed from physical description of the modeled fluid according to the Equation 3.17. This is translated into pseudocode (Listing 10) to show the essence of how it's done within the `Sim.cs` script in Unity. To experiment with the different behaviours of fluid flows based on different Reynolds number, the interactive slider in VR user interface is hooked to this value and user can experiment with it.

```
// get the Reynolds number from the user input through VR interface
re = reynoldsNumberSlider.value;
// kinematic viscosity
nu = inflow_speed * 2.0f * obstacle_radius / re;
// relaxation time
tau = 3.0f * nu + 0.5f;
// BGK collision operator
omega = 1.0f / tau;
```

Listing 10: Pseudocode for computing the BGK collision operator omega on Unity's side.

Actual collision phase implementation consists of two steps, namely computing the equilibrium particle distribution function (Equation 3.23) across whole domain (each node at once thanks to the GPUs parallel architecture) and then relaxing towards them according to the Equation 3.27 (Listing 11).

```
let u_sq = flat(
  &(pow(&ux, &(2.0 as f32), false)
  + pow(&uy, &(2.0 as f32), false))
);

let eu = flat(
  &(&mul(&transpose(&self.ex, false), &flat(&ux), true)
  + &mul(&transpose(&self.ey, false), &flat(&uy), true)),
);

let feq = flat(
  &mul(
    &transpose(&self.w, false), &flat(&density), true))
    * ((1.0 as f32)
    + (3.0 as f32) * &eu
    + (4.5 as f32)
    * (&pow(&eu, &(2.0 as f32), false))
    - (1.5 as f32)
    * (&tile(&flat(&u_sq), dim4!(9))
  )
);

// Relaxation with pre-computed BGK operator (omega)
f = omega * &feq + (1.0 - omega) * &f_streamed;
```

Listing 11: BGK collision phase.

6.2.4 Streaming

In streaming phase, collided particles from each lattice node are advected to the neighboring nodes for each direction (in D2Q9 stencil, this means advecting them in 9 directions, from which ones in the center stays at the same place). This process is illustrated in Figure 6 – 1.

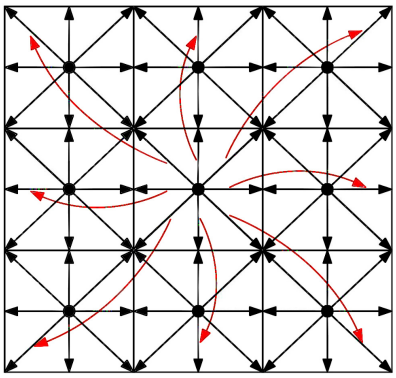


Figure 6 – 1: Illustration of the streaming step.

Streaming is implemented as a `shift` function acting upon an Array that represents a particle distributions across each node in computational domain (Listing 12). This way the indices of the Array are shifted in a direction represented by 4×1 vector consisting of a combination of 0, 1 and -1 values, describing the direction and a dimension along which they move.

```
1   fn stream(f: &Array<f32>) -> Array<f32> {
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 12: Streaming with `shift` function for two dimensions with 9 discrete speeds (D2Q9).

In general, streaming is usually treated 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. Another way to speed things up, which is used in the current implementation, is to save the computation time by streaming the indices of neighboring nodes instead of streaming particle distribution functions on every iteration. In this fashion, the index is pre-computed once when the simulation is instantiated and then used throughout the simulation lifetime to index into the particle distributions of neighboring nodes (Listing 13).

```
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
9     // Streaming of the neighboring indices is pre-computed here
10    let nb_index = flat(&stream(&main_index));
```

Listing 13: Pre-computed streaming step in the way of shifting indices during the program intialization phase.

This has an additional benefit when dealing with the nodes near the solid walls at the boundaries. From these streamed neighboring indices, simple "reflective" boundary conditions treatment (also called bounce-back) can be handled very easily by constructing two additional indices at the simulation initialization phase: (1) `to_reflect`, and (2) `relfected` (Listing 14).

```
1   let to_reflect = flat(&tile(&on, dim4!(ci.elements() as u64)))
2   + flat(&tile(&ci, dim4!(on.elements() as u64)));
3
4   let reflected = flat(&tile(&on, dim4!(nbi.elements() as u64)))
5   + flat(&tile(&nbi, dim4!(on.elements() as u64)));
```

Listing 14: Bounce-back indices.

To tie things together—the value of all particle distributions f_i are streamed over to the neighboring lattice nodes along the discrete velocity directions by simply indexing to the pre-computed neighboring nodes index; then, the nodes from which particle distributions should be reflected are flagged and stored in `bounceback` variable; at the end of LBM algorithm, after the collision phase, relaxed particle distributions are updated with the bounced-back values (Listing 15).

```
// Streaming step
let f_streamed = view!(f[self.nb_index]);

// Storing the streamed PDFs bouncing back at next timestep
let bouncedback = view!(f_streamed[self.to_reflect]);

// ... rest of the LBM algorithm ...

// After the collision phase, PDFs relax towards equilibrium
f = omega * &feq + (1.0 - omega) * &f_streamed;

// Relaxed PDFs are updated with the bounced-back values
eval!(f[self.reflected] = bouncedback);
```

Listing 15: Streaming and boundaries treatment by relfection using the bounce-back indices.

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.

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, all discrete velocities in each node occupy consecutive elements of the array, and in SoA, the value of one discrete velocity direction from all nodes is arranged consecutively in memory, then the next direction in all nodes and so on. In a GPU-based LBM algorithm, it's beneficial to store values of distribution functions for each node in the computational domain represented by a grid in 1-dimensional array. For D2Q9 and D3Q27 stencils, this translates into storing single velocity direction for 2D grid represented by $N_x * N_y$ or 3D grid represented by $N_x * N_y * N_z$ nodes at a time, consecutively 9-times for D2Q9 and 27-times for D3Q27 respectively. Using Structure of Arrays (SoA) is significantly faster than Array of Structures (AoS) Delbosc et al. (2014); Tran et al. (2017). This way the cache use is considerably improved Mawson (2014).

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 $n_x * n_y * n_z * Q$ number of elements, but for easier index creation further down the line, it's better to create 3D (or 4D in case of D3Q27) array initially and flatten it when used for heavy computation of actual simulation:

```
1     let nx = 64;
2     let ny = 64;
3     let dirs = 9;
4     // SoA 1D array
5     let f_1d_soa = constant::<f32>(0.0, nx * ny * dirs);
6     // Array flattened to SoA 1D array
7     let f = constant::<f32>(0.0, nx, ny, dirs);
8     let f_1d_soa = flat(f);
```

Listing 16: Creating SoA structure representation of D2Q9 lattice with ArrayFire in Rust.

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 neighboring 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. Implementation of the streaming phase is described in more detail in section 6.2.4. 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. If needed, targeting any of the discrete speeds directions 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. 18).

```
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 17: 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. 18.

```
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 18: Pseudocode 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 19: 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:

```
int main(){
  int nx = 400, ny = 100;
  array mask = constant(0,nx,ny);
  // Rectangle obstacle of size 2x20 cells
  mask(seq(100,102), seq(40,60))) = 1;
  mask(span,0) = 1; // Top wall
  mask(span,end) = 1; // Bottom wall
  // Get the indices of each solid cell
  array solids = where(mask);
  // ... rest of the code ...
}
```

Listing 20: Pseudocode 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:

```
UX(solids) = 0; // velocity in X-direction
UY(solids) = 0; // velocity in Y-direction
DENSITY(solids) = 0;
```

Listing 21: Pseudocode for boundary conditions at solid nodes.

6.3.3 Pull 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 single write operation, 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).

6.4 Virtual Reality User Interface

The immersive virtual reality user interface (VRUI) is designed to accommodate an experimentation phase when testing different physical parameters or flow regimes (Figure 6 – 2). Current VR prototype is constrained to 2D simulations. On the side of simulation software, additional support for simulating fluid flows in 3D on D3Q27 lattice with multiple-relaxation time was developed, but not incorporated into the VR environment. It should be straightforward to implement a 3D volume rendering with 3D textures in Unity, but this will be addressed in future work beyond this dissertation.

VRUI provides a way of interacting with the running simulation through interactive sliders, which are hooked to the simulation parameters. This interaction interface sits on the grabbable 3D rectangle that is reminiscent of digital "tablet". User interacts with sliders through hand controllers or with their own hands that are tracked with VR hardware like Oculus Quest headset. Changing the slider value updates simulation, which results in immediate feedback displayed through real-time visualization.

Following sections will describe how the individual parts were implemented with help of Unity game engine.

6.4.1 Cross-Platform Development

The goal for the usability of VRUI was to provide the same experience across multiple VR hardware and their means for interaction with a VR environment. A great way to create cross-platform VR applications is to use a game engine. One of such engine, Unity, provides many useful tools for creating VR applications efficiently.

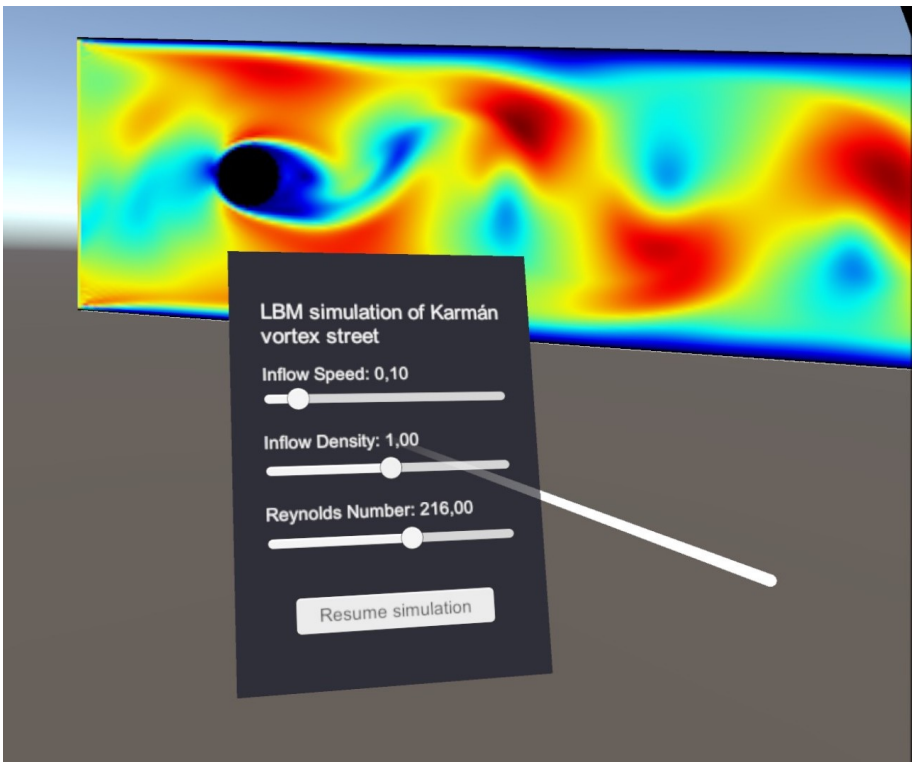


Figure 6 – 2: Immersive interface in virtual reality.

Unity engine can build apps for multiple VR hardware platforms:

- Oculus Quest
- Oculus Rift
- HTC Vive
- Windows Mixed Reality
- Microsoft HoloLens
- Magic Leap

Until recently, a cross-platform VR support in Unity was achieved by building on top of OpenVR standard. However, with an increasing interest in augmented reality (AR) applications and better AR hardware, new standard called OpenXR was developed by Khronos group. It is a royalty-free, open standard that provides high-performance access to both

VR and AR hardware. XR denotes a term “extended reality”, which was coined to better describe this new paradigm in application development. Without a cross-platform standard, VR and AR applications and engines must use each platform’s proprietary APIs. New input devices need customized driver integration. OpenXR provides cross-platform, high-performance access directly into diverse XR device runtimes across multiple platforms. OpenXR enables applications and engines, including WebXR, to run on any system that exposes the OpenXR APIs (Figure 6 – 3).

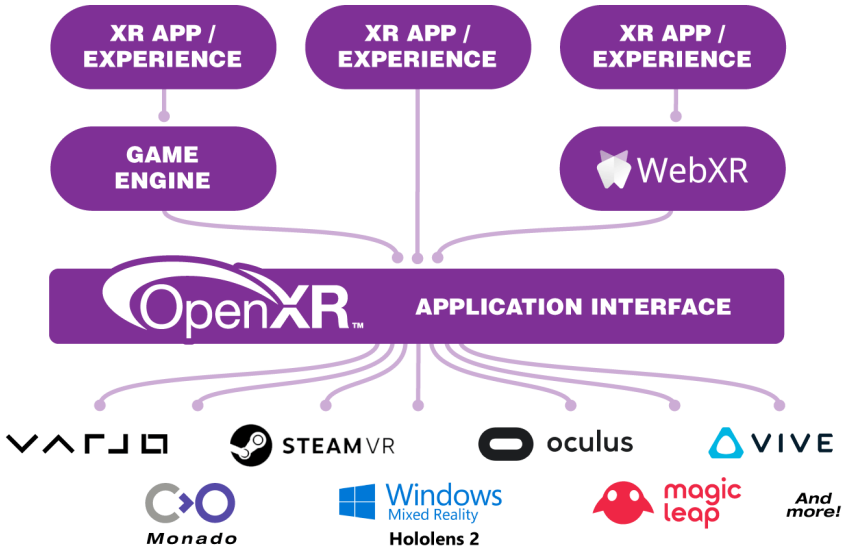


Figure 6 – 3: OpenXR provides cross-platform, high-performance access directly into diverse XR device runtimes across multiple platform.

Unity engine provides a streamlined software development kit (SDK) on top of OpenXR (Figure 6 – 4)

As a result of Unity’s decision to build a future-proof SDK on top of OpenXR, this opens the door for supporting not just VR hardware, but also powerful AR devices like Microsoft HoloLens or Magic Leap. Since we don’t possess the AR hardware for testing, current work is focused solely on creating VR experience for CFD simulation. The issue of extending current prototype to the realm of AR is further discussed in chapter Discussion

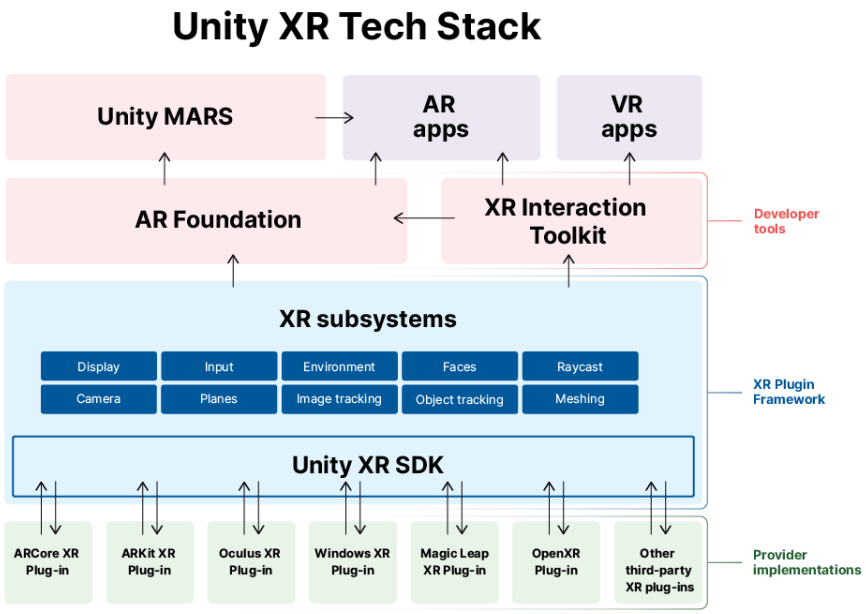


Figure 6 – 4: Unity's software development kit (SDK) for XR (allowing to target both VR and AR devices).

under section Augmented Reality 8.1.4.

To enable the VR support, developers have to select all of the platforms they want their application to be built for. Figure 6 – 5 shows the menu where they can select them, which is accessible in Unity from *Edit* > *Project Settings* > *XR Plug-in Management*.

VR hardware like Oculus Rift, HTC Vive, and all Windows Mixed Reality headsets can be used for the high-performance visualization software, as the heavy computation is done by GPU on dedicated PC workstation. Current implementation was tested on Oculus Rift, HTC Vive and Oculus Quest with Oculus Link cable.

At the start of the development process, main objects that are needed to add support for VR into Unity scene are XR Rig and XR Interaction Manager. They can be accessed from Unity menu shown in Figure 6 – 6. Subsequent section will describe how the interactivity is implemented.

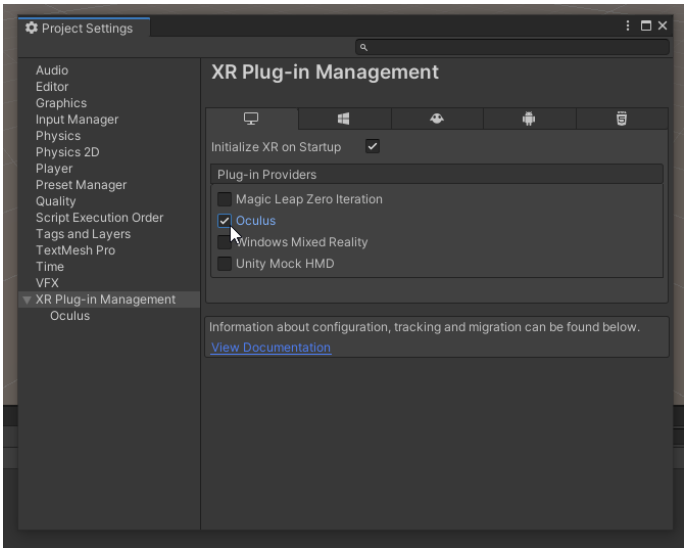


Figure 6 – 5: Setting up VR support in Unity.

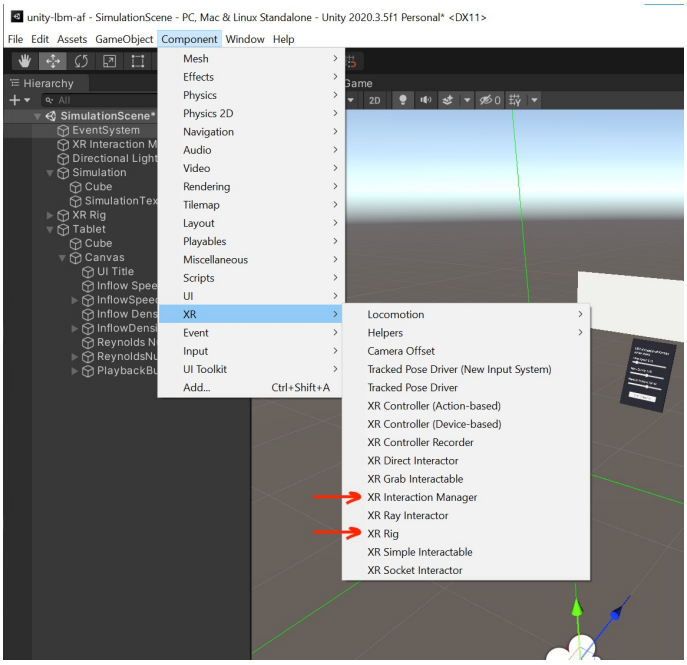


Figure 6 – 6: Fundamental parts of the VR scene in Unity is XR Rig (VR camera) and XR Interaction Manager (for user interactions).

6.4.2 User Interactions in VR

The simulation script that is hooked to the 2D rectangle object floating in 3D space awaits the selection of interactive Slider components for each of the editable variables (Figure 6 – 7). Clicking on buttons that red arrows point to will open a pop-up window, presenting the available `Slider` components in current Unity scene.

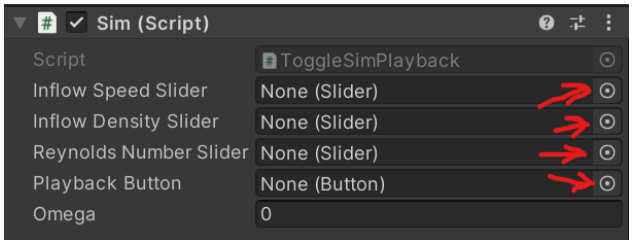


Figure 6 – 7: Initial state of `Sim.cs` script attached to the simulation panel with `Texture2D`.

Within the VR environment, an interface for interaction with running simulation is designed as a table-like 3D shape that can be grabbed with any hand controller or left floating as-is. It provides list of sliders for changing simulation parameters and button for starting or stopping the simulation. Its hierarchy is presented in Figure 6 – 8.

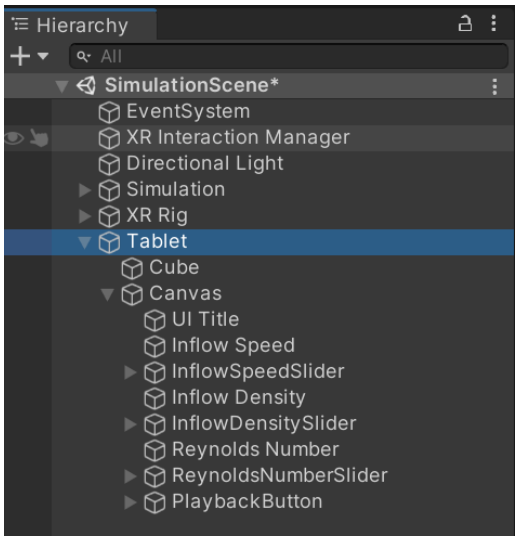


Figure 6 – 8: Unity scene hierarchy.

To connect the simulation with tablet UI, each `Slider` component from this hierarchy has to be connected to the correct slider input and simulation playback button to the button input in `Sim.cs` script options from Figure 6 – 7. Correctly initialized `Sim.cs` script is shown in Figure 6 – 9.

Figure 6 – 9: Initialized `Sim.cs` script with appropriate components connected.

All of the Unity UI elements like `Sliders`, `Buttons` etc. are accessible with mouse by default. However, for VR application, the interaction device is hand controller, or possibly a hand itself (if VR hardware supports hand tracking), instead of a mouse. Unity provides easy-to-use interface called XR Interaction Manager that handles interaction support for multiple kinds of VR hardware devices. Through XR Interaction Manager, all kinds of interaction types can be channeled. It has to be connected to all scripts that specify how certain object within VR environment, to which this script is attached, should be interacted with.

For example, the tablet UI also with the simulation texture can be made grabbable which adds to the immersivity and allows user to better control the virtual environment. The `XR Grab Interactable` script is attached to the tablet object and XR Interaction Manager is then connected to the Interaction Manager input (Figure 6 – 10).

To not interfere with any other additional objects in scene that can be possibly added later, the “grabbable” layer has to be created (Figure 6 – 11).

All objects that will benefit from grabbing interaction within Unity scene should be added to this grabbable layer (Figure 6 – 12).

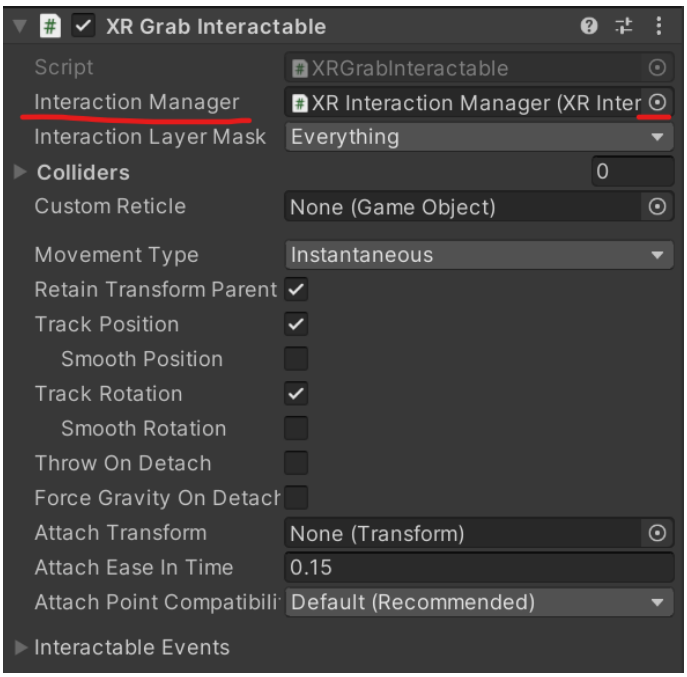


Figure 6 – 10: Enabling XR Interaction Manager.

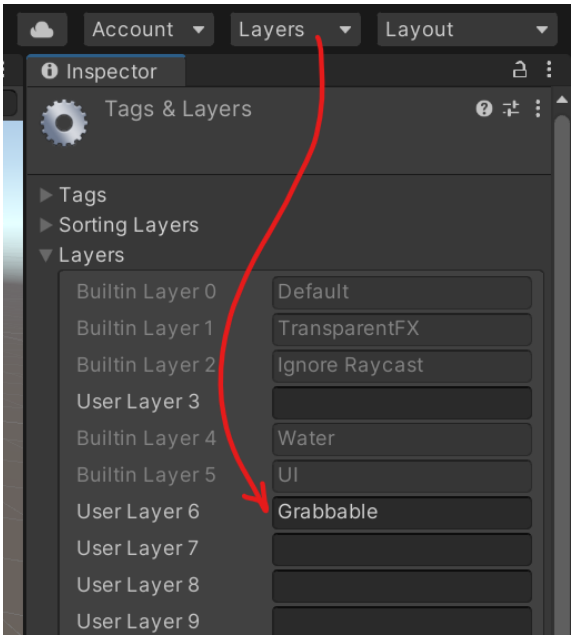


Figure 6 – 11: Creating new layer with the name `Grabbable`.

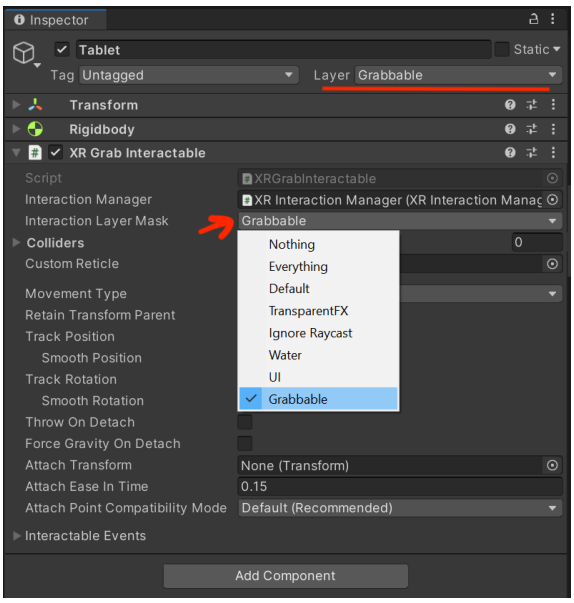


Figure 6 – 12: Selecting the “grabbable” layer.

6.5 Interactive Simulation

In previous sections, individual parts of the software application was presented and their implementation described. However, these parts cannot function in vacuum. An interconnecting layer has to be provided between them, through which simulation, visualization and interaction parts can properly “talk” to each other by sending data and calling particular functions.

Each part has been created with different tools and different programming languages. Simulation part is written from scratch in Rust programming language, while visualization and interaction parts are implemented with the help of Unity engine, where C# language is used.

Therefore, most of the following chapters would jump back-and-forth between Rust and C# code samples. But first, let’s set the ground for Unity and Rust interoperability.

6.5.1 Unity and Rust Interoperability

The simulation program has to communicate with foreign interface of Unity game engine. Unity applications and games are programmed in C#, therefore these two won't understand each other. For this we have to go with the strategy of implementing Foreign Function Interface (FFI) between Rust and Unity's C# environment.

On Rust side, the simulation program will have functions prepared to be called from Unity, which will control the simulation lifecycle. First, to actually use the `Sim` struct this way, the pointer to the place in memory where it is stored has to be passed to Unity, where it will be stored in a local variable for further use. Note: working with pointers in Rust is marked as unsafe and has to be specifically tagged with `unsafe` keyword before a function, or the part of the code that deals with unsafe behavior should be wrapped in `unsafe` $\{\ldots\}$ block (Listing 22, some additional error-checking has been removed from code listing to enhance readability). In the same way, pointer to the memory where output of a simulation will be stored has to be sent to Unity, where it will be stored in another local variable.

```
#[no_mangle]
pub extern "C" fn init_sim(
ptr: *mut *mut Sim, // pointer to Sim struct
data_ptr: *mut *mut u8, // pointer to simulation results
width: u32,
height: u32,
initial_density: f32,
initial_ux: f32,
omega: f32,
// ... rest of the parameters ...
) {
  let sim = Sim::new(
  width.into(),
  height.into(),
  initial_density,
  initial_ux,
  omega,
  // ... rest of the parameters ...
  );

  unsafe {
    *ptr = sim.to_ptr();
    *data_ptr = Sim::from_ptr(*ptr).results_ptr();
  }
}
```

Listing 22: The “external” function `init_sim` for instantiating `Sim` struct by calling the function from external program through FFI.

To read from correct memory location, two additional methods are added to `Sim` to transfer itself into pointer to memory on the heap and to get the actual `Sim` struct back from memory back if the pointer to it is passed to the method (Listing 23).

```
impl<'a> Sim {
  pub fn to_ptr(self) -> *mut Sim {
    let sim_boxed = Box::new(self);
    Box::into_raw(sim_boxed)
  }

  pub fn from_ptr(ptr: *mut Sim) -> &'a mut Sim {
    unsafe { &mut *ptr }
  }

  // ... rest of the methods
}
```

Listing 23: Storing and restoring `Sim` struct with pointers.

On Unity side, external functions from Rust FFI are loaded from compiled dynamic library through `DllImport(...)` directive. The external simulation program is regarded as a Native Plugin. For convenience, additional C# interface is created for cleaner API when working with exposed Rust functions and for passing in correct arguments with correct types (Listing 24).

```
public class LBMAF
{
  private static IntPtr _sim_handle;
  private static IntPtr _data_handle;

  [DllImport("lbmaf")]
  private static extern bool init_sim(
  out IntPtr sim_handle,
  out IntPtr data_handle,
  /* ...rest of the arguments... */
  );

  // C#/Unity interface with the Rust FFI
  public static bool InitSimulation(/* ...rest of the arguments... */)
  {
    return init_sim(out _sim_handle, out _data_handle, /* ... */);
  }
  // ... rest of the methods
```

```
19      }
```

Listing 24: Creating a C# interface for the Rust functions from simulation plugin and storing the pointers to Rust `Sim` struct and results data.

6.5.2 Visualizing Simulation Output in Real-Time

First, on the Rust side, the data output from the LBM simulation has to be normalized to the range within minimum and maximum values (Listing 25).

```
1   fn normalize(a: &Array<f32>) -> Array<f32> {
2     let min = min_all(a).0;
3     let max = max_all(a).0;
4     (a - min) / (max - min) as f32
5   }
```

Listing 25: Normalization of the simulation output.

This range is then color-coded to the rainbow colormap. Blue-ish colors denote slower velocities and red-ish colors faster velocities (Listing 26). Data representing information about color and opacity (alpha) of each pixel is ordered in r, g, b and a sub-arrays, then joined together and shuffled around to be correctly represented as Unity's Texture2D RGBA32 texture format.

```
1   pub fn simulate(&mut self, inflow_density: f32, inflow_ux: f32, omega:
        f32) {
2     // ... code for actual computation
3     results = normalize(&results);
4     // Colormap for Unity's Texture2D (RGBA32)
5     let r = flat(&((1.5f32 - abs(&(1.0f32 - 4.0f32 * (&results - 0.5f32))))
          * 255));
6     let g = flat(&((1.5f32 - abs(&(1.0f32 - 4.0f32 * (&results - 0.25f32)))
        ) * 255));
7     let b = flat(&((1.5f32 - abs(&(1.0f32 - 4.0f32 * &results))) * 255));
8     let a = flat(&constant::<f32>(1.0 as f32, self.dims));
9     // Joining Arrays together to follow Unity's Texture2D RGBA32 format
10    self.colors = flat(&transpose(&join_many(1, vec![&r, &g, &b, &a]),
        false)).cast::<u8>();
11    // ... rest of the code
12  }
13
14  // Method for copying ArrayFire data to Rust's Vec
15  pub fn copy_colors_host(&mut self) {
16    self.colors.host(self.results.as_mut_slice());
17  }
```

Listing 26: Preparing the data output from Rust side of LBM simulation.

Second, on the Unity side, a simple C# interface is created for getting the pointer to simulation output data (Listing 27).

```
[DllImport("lbmaf")]
private static extern void get_sim_data(IntPtr handle);
public static void GetSimData()
{
  get_sim_data(_sim_handle);
}
```

Listing 27: Method for getting the pointer to the memory where simulation output is stored.

Next, additional method is implemented for copying the simulation output data from Rust side to a buffer created in Unity (Listing 28).

```
public static void CopyResultsToBuffer(byte[] buffer, Int32 size) {
  Marshal.Copy((IntPtr)_data_handle, buffer, 0, size);
}
```

Listing 28: Method for copying simulation output to Unity buffer.

Finally, to render a simulation output data to 2D texture initialized in `Start` method, a co-routine `GetData()` is called by pressing the "Start simulation" button through the VRUI (Listing 29). It calls the methods for getting the output data, copying them from Rust to Unity buffer, loading them to 2D texture as raw data and rendering them to screen. Additional functionality regarding the starting and stopping the running simulation is described in section 6.5.4.

```
void Start()
{
  // .. rest of the code ...
  size = domain_width * domain_height * 4;
  buffer = new byte[size];
  image = new Texture2D(domain_width, domain_height, TextureFormat.RGBA32
    , false);
  GetComponent<Renderer>().material.mainTexture = image;
  // ... rest of the code ...
}

public IEnumerator GetData()
{
  while (true)
  {
    // .. rest of the code ...
    LBMAF.GetSimData();
```

```
17        LBMAF.CopyResultsToBuffer(buffer, size);
18        image.LoadRawTextureData(buffer);
19        image.Apply();
20        yield return null;
21      }
22    }
```

Listing 29: Rendering the simulation output to 2D texture by additionally copying the data.

The faster alternative to CPU-bound data copying is to work only within context of the GPU, since we already have the data residing there after doing computations for LBM simulation. The easiest way to do this is to leverage OpenCL and OpenGL interoperability (Malcolm et al.; 2012).

OpenCL-OpenGL sharing works in a very particular way. The OpenCL context that needs to be shared with an OpenGL context has to be created using an OS specific OpenGL context handle as one of it's context properties. The OpenGL context used by ArrayFire's OpenCL-OpenGL shared context is completely different from the OpenGL context that Unity uses.

As for such attempts trying to use ArrayFire's OpenGL context for copying data from OpenCL Arrays to Unity's OpenGL texture, these can fail due to the common error that usually pops up: `CL_INVALID_CONTEXT`.

First, a workaround has to be implemented prior to doing any work with external OpenGL context. Simply put, ArrayFire has to switch to this context and use it. Workaround procedure consists of the following:

1. Pick the device you want to do OpenCL-OpenGL sharing on and create a `ocl_device_id` handle using fuctions from the`ocl` Rust create.
2. Now, for this device from step (1), create a new OpenGL context, passing the properties specific to your OS as an arguments for the creation function.
3. Create an OpenCL queue for this device in newly created OpenCL-OpenGL shared context.

4. Now add this context to ArrayFire device manager using `afcl::add_device_context`.

5. 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 different OpenCL context. Only after this initial procedure, the creation of a shared OpenCL-OpenGL buffer that is then used for direct memory copying to the Unity's OpenGL texture is then possible. ArrayFire shall assume control of shared OpenCL-OpenGL buffer to take care of ownership, only this way it can operate on shared OpenCL memory. For such case ArrayFire provides `Array.lock()` and `Array.unlock()` functions.

Implementing OpenGL-OpenCL interoperability with ArrayFire and Unity isn't straightforward and involves additional help from the ArrayFire developers. Although it is documented for simple windowing platforms that use OpenGL textures for drawing to the screen, it's much more complex within the setting of this work, such as implementing interoperability with Unity engine. On the other note, OpenGL support in virtual reality headsets that use cable connection to PC workstations (which is currently needed for having access to powerful GPUs) is very limited. Therefore, it was not successfully implemented into the current work.

Still, it is of very high importance to implement GPU-bound visualization techniques into the future versions of LBM solver, though. When dealing with complex 3D simulation, multiple-relaxation time algorithm and more directions of particle distribution function within the lattice node, CPU-bound visualization could easily become a bottleneck.

6.5.3 Updating Boundary Conditions and Simulation Parameters

Manipulating various parts of the simulation is done by sending the updated parameters to the `simulate(...)` function from Rust code (Listing 30).

```rust
#[no_mangle]
pub extern "C" fn simulate(
  ptr: *mut Sim,
  inflow_density: f32,
  inflow_ux: f32,
  omega: f32
) {
  if !ptr.is_null() {
    Sim::from_ptr(ptr).simulate(inflow_density, inflow_ux, omega);
  }
}
```

Listing 30: `simulate` function that computes one iteration of a simulation.

An actual change to the various parameters is done by user input by means of interacting with `Slider` components in VRUI (Figure 6 – 13).

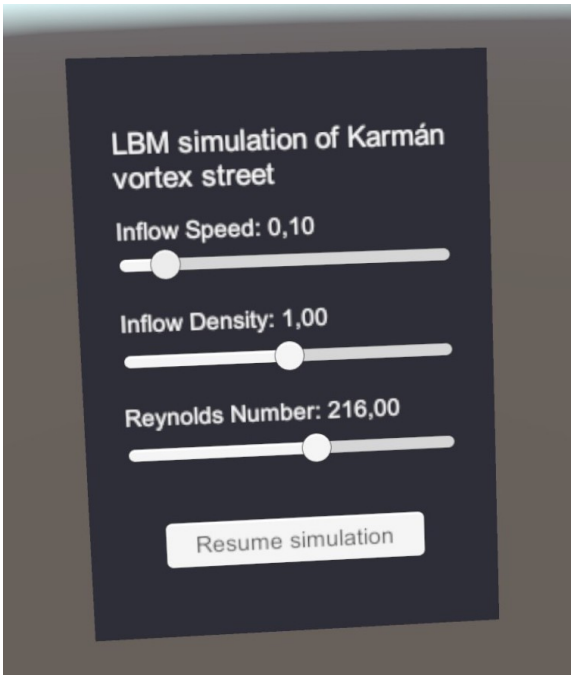


Figure 6 – 13: VRUI in the form of a grabbable digital tablet.

Sliding the movable button inside a slider changes the value, which is then propagated to the `GetData()` co-routine and subsequently used as an updated parameter of (Listing 31).

```
public IEnumerator GetData()
{
```

```
3       while (true)
4       {
5         inflow_speed = inflowSpeedSlider.value;
6         inflow_density = inflowDensitySlider.value;
7         re = reynoldsNumberSlider.value;
8         nu = inflow_speed * 2.0f * obstacle_r / re;
9         tau = 3.0f * nu + 0.5f;
10        omega = 1.0f / tau;
11
12        LBMAF.SimulateNextIteration(inflow_density, inflow_speed, omega);
13
14        // ... rest of the code for getting the data and rendering them ...
15        yield return null;
16      }
17    }
```

Listing 31: Implementation of the re-computing of the simulation each frame based on values from user input through VRUI (sliders).

After the change in slider input, re-computation happens immediately and updated simulation output is visualized in real-time on the screen. Thanks to the speed of LBM simulation, by changing for example inflow speed variable, a wave is introduced which propagates through the computational domain. Its propagation can be seen in real-time as it happens (Figure 6 – 14).

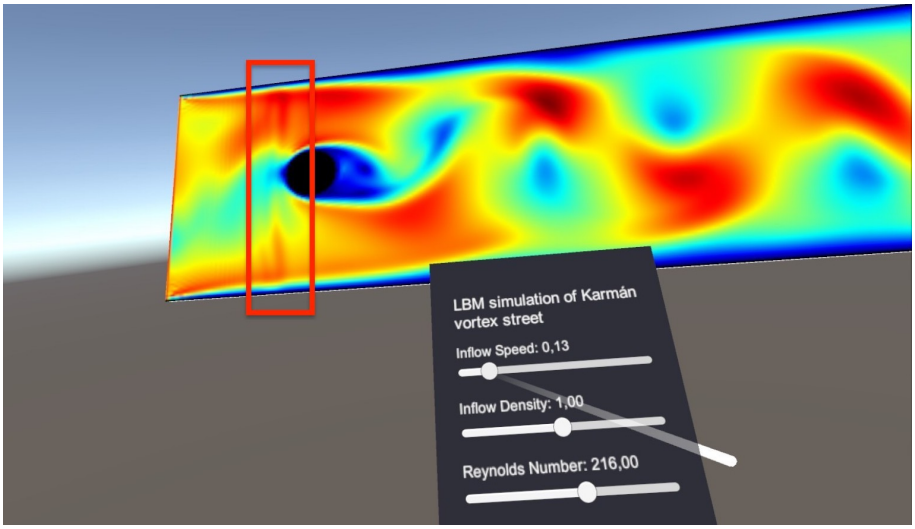


Figure 6 – 14: Propagating wave that resulted from changing the inflow speed.

6.5.4 Simulation Playback

Visualization is running continuously throughout the simulation in parallel to the actual computations since it's implemented as co-routine in Unity. However, it's a good idea to at some point pause the running simulation when there's something interesting to investigate or explore in more detail.

For simple simulation playback control, the VRUI contains a button for that (Figure 6 – 15).

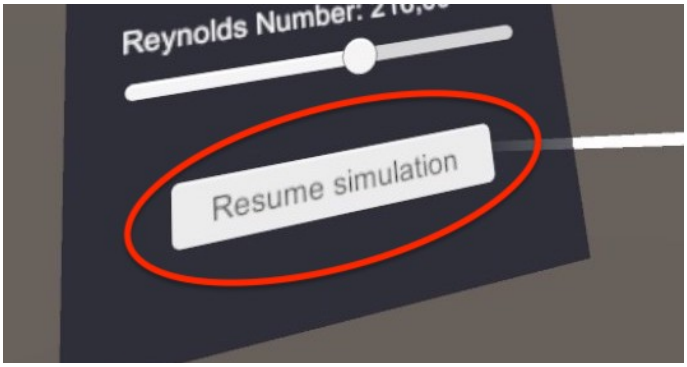


Figure 6 – 15: Simulation can be paused or resumed with push of a button.

Implementation of the co-routine for starting and pausing is implemented in Listing 32.

```
1   void Start()
2   {
3     // ... rest of the code ...
4     Button btn = playbackButton.GetComponent<Button>();
5     btn.onClick.AddListener(ToggleSimPlayback);
6   }
7
8   void ToggleSimPlayback() {
9     paused = !paused;
10    if (paused) {
11      StopCoroutine("GetData");
12      playbackButton.GetComponentInChildren<Text>().text = "Resume
        simulation";
13    } else {
14      StartCoroutine("GetData");
15      playbackButton.GetComponentInChildren<Text>().text = "Pause
        simulation";
16    }
17  }
```

Listing 32: "Starting pausing and stopping the co-routines in Unity."

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 4 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. Also, to test the performance across multiple architectures of NVIDIA GPU cards, we added a NVIDIA GeForce RTX 2080 Ti to the suite of benchmarks.

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

Table 6 – 1: GPU hardware specifications. These were used for benchmarking the LBM simulation software described in this work (taken from https://www.techpowerup.com/gpu-specs/). SM - streaming multiprocessor, CU - computing units.

6.6.2 Results

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

Single floating-point (f32) calculations perform significantly faster than double-precision (f64). The difference between f32 and f64 computation is the doubling of the problem data size. ArrayFire's Array objects are typed, i.e. syntactic construction of a f32 type array looks different from f64 array. It's not hidden in any fashion. In fact, the programmer needs to know what type they are using as performance on accelerators (GPUs) is

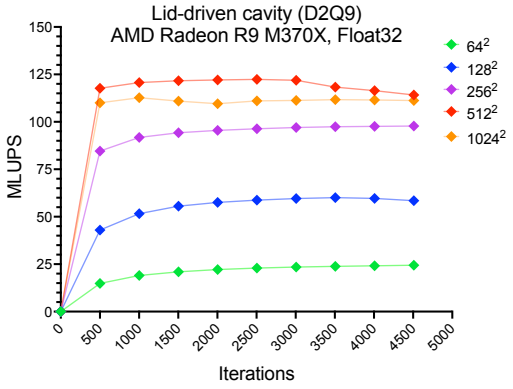
Domain	R9 M370X	GTX 1070	RTX 2080 Ti	RTX 3090
64×64	23	40	11	55
128×128	59	226	50	225
256×256	97	675	186	868
512×512	120	1006	600	2890
1024×1024	111	1143	1401	4620
2048×2048	-	1200	2195	5465
4096×4096	-	-	2394	5730

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

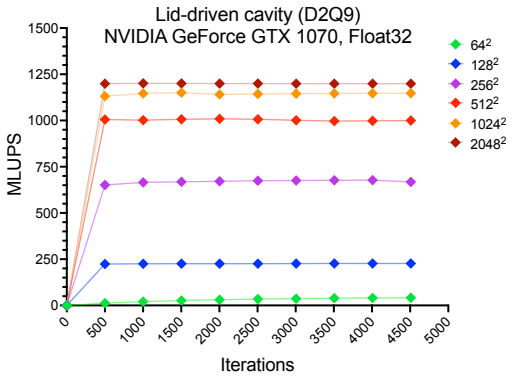
not the same for all types, scalar or vector types. Type dictates the size of memory on the GPU that will be used, and hence the amount of bandwidth that can be utilized during data transfers. Usually similar sized types have similar data transfer characteristics. Doubles are theoretically expected to have half the performance given by Floats. This difference can be seen in D2Q9 lid-driven cavity and Kármán vortex test cases (Figure 6 – 16 ,6 – 17, 6 – 18, 6 – 19, but it's manifesting much better in D3Q27-MRT lid-driven cavity benchmarks (Figure 6 – 21 ,6 – 22).

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.

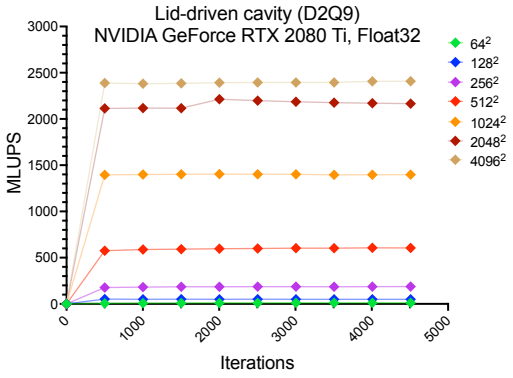
Maximum MLUPS of the GPUs for single-precision calculations for 2D test cases are slightly higher than the study reported by Boroni et. al Boroni et al. (2017), in which they reported 80 MLUPS at peak performance achieved on NVIDIA GeForce GTX 580 GPU. The AMD Radeon R9 M370X GPU used in our work can be seen as similarly performing card.



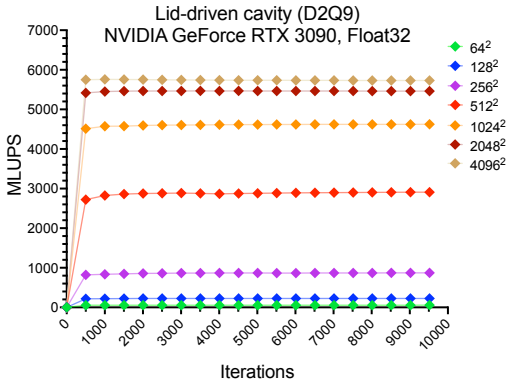
(a) Performance on Radeon R9 M370X GPU.



(b) Performance on GTX 1070 GPU.



(c) Performance on RTX 2080 Ti GPU.

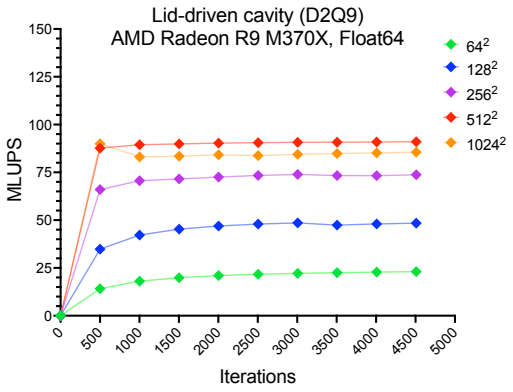


(d) Performance on RTX 3090 GPU.

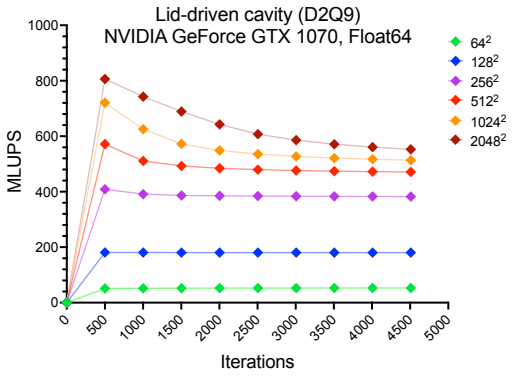
Figure 6 – 16: Single-precision performance analysis of 2D lid-driven cavity on D2Q9 stencil.

For the 3D LBM simulation with D3Q27 stencil and multiple-relaxation time (MRT), results showed the computation speeds reaching up to 1500 MLUPS in single-precision (Figure 6 – 21) and up to 490 MLUPS in double-precision benchmarks Figure 6 – 22. The MRT is much more computationally intensive than BGK and with the 27 particle distributions to track, the difference between D2Q9 and D3Q27 is significant. The memory bandwidth for D3Q27 lid-driven cavity tests is much higher, therefore we were limited to grids under 128^3 in current implementation.

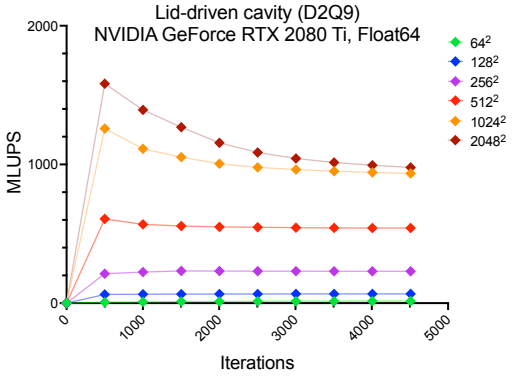
In 3D lid-driven cavity test, benchmarks showed inconsistencies with expectations and actual results (Table 6 – 4). The performance is lower in coarse grids (16^3 and 32^3) with



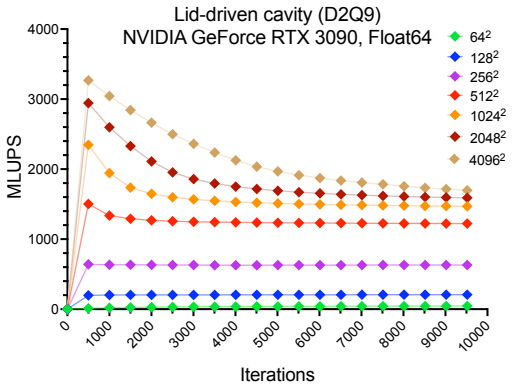
(a) Performance on Radeon R9 M370X GPU.



(b) Performance on GTX 1070 GPU.



(c) Performance on RTX 2080 Ti GPU.



(d) Performance on RTX 3090 GPU.

Figure 6 – 17: Double-precision performance analysis of 2D lid-driven cavity on D2Q9 stencil.

more powerful RTX 2080 Ti conversely to higher performance reported with weaker GTX 1070 (Figure 6 – 21b and Figure 6 – 21c). This points to the well-known problem of representing multi-dimensional data in the most optimal way stop GPU from cache misses. It means the data organization of 27 individual distribution functions in D3Q27 lattice nodes along 3D grid has to be rethought and reimplemented to address this issue.

When comparing the 3D performance of LBM solvers, and specifically those using D3Q27 stencil with multiple-relaxation times, our LBM-MRT solver has still lot of room for improvement. One of the fastest solvers on the market for such 3D LBM smulation is Sailfish Januszewski and Kostur (2014). They take advantage of multi-GPU and many-core CPU

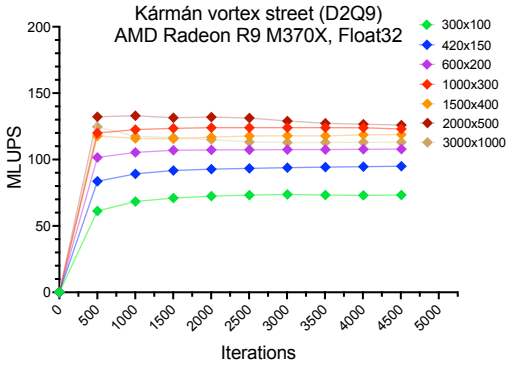
Domain	R9 M370X	GTX 1070	RTX 2080 Ti	RTX 3090
300×100	73	340	361	361
420×150	93	627	122	740
600×200	107	810	180	1343
1000×300	123	1007	325	3000
1500×400	117	1100	731	4055
2000×500	130	1140	1010	4510
3000×1000	113	1175	1311	5313
4200×1500	-	1194	2016	5557
6000×2000	-	-	2522	5670
10000×3000	-	-	-	5720

Table 6 – 3: Peak MLUPS of 2D Kármán vortex street test case with BGK collision operator. The data represents single-precision floating point computation (taking only the steady performance after initial warm-up and removal of sudden spikes).

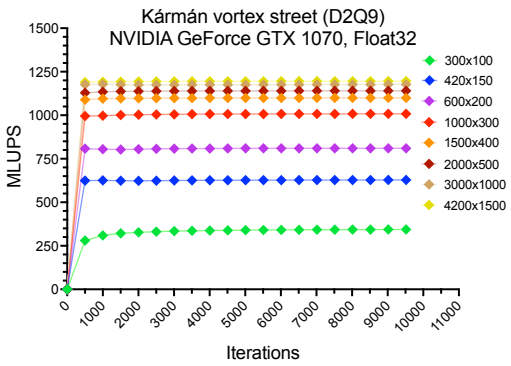
Domain	R9 M370X	GTX 1070	RTX 2080 Ti	RTX 3090
16×16×16	7	25	28	40
32×32×32	18	155	70	225
64×64×64	97	675	186	868
128×128×128	120	1006	600	2890

Table 6 – 4: Peak MLUPS of lid-driven cavity test case in 3D. The data represents single-precision floating point computation (taking only the steady performance after initial warm-up and removal of sudden spikes).

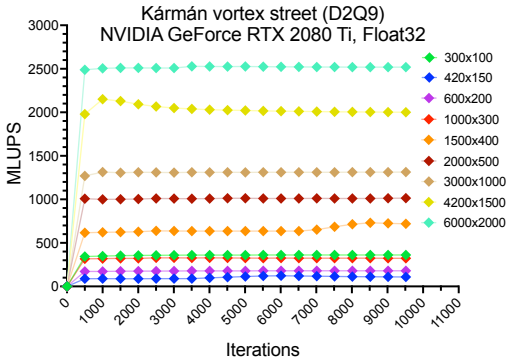
support in their software, but even for single-GPU, our solver doesn't reach the satisfactory speeds comparable to Sailfish. In D3Q27 solver described in this work, maximum MLUPS achieved for single-precision calculations in 3D test cases on similar hardware is



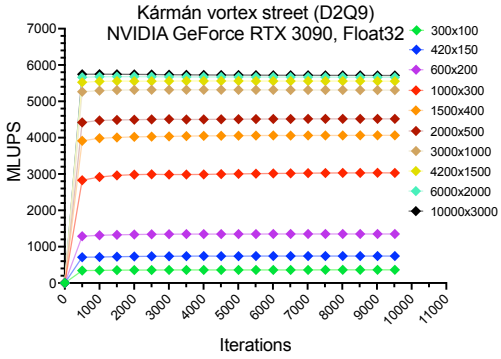
(a) Performance on Radeon R9 M370X GPU.



(b) Performance on GTX 1070 GPU.



(c) Performance on RTX 2080 Ti GPU.



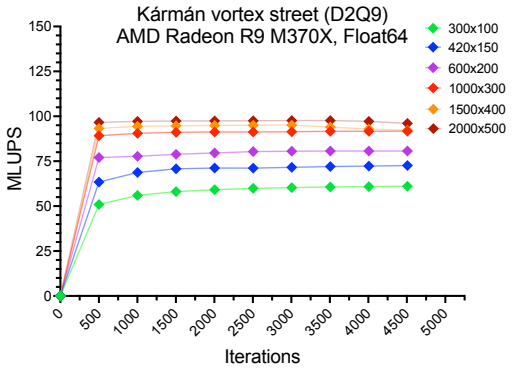
(d) Performance on RTX 3090 GPU.

Figure 6 – 18: Single-precision performance analysis of 2D Kármán vortex on D2Q9 stencil.

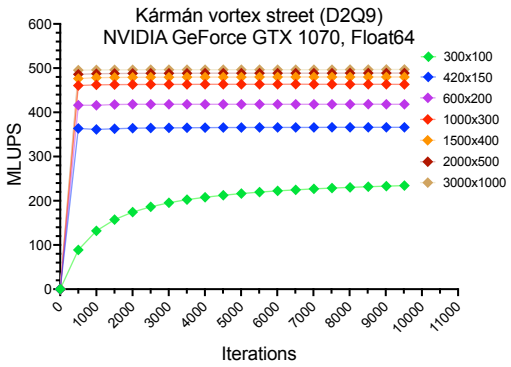
shown in Figure 6 – 23.

7 Conclusions

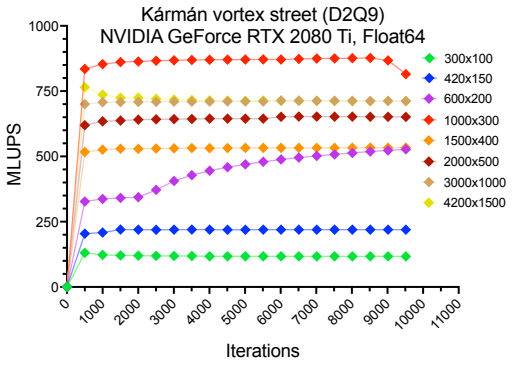
For this thesis, the cross-platform LBM solver that can be used on variety of parallel accelerators (e.g. GPUs, CPUs or FPGAs) was implemented. It uses ArrayFire, a high-performance parallel computing library (version 3.8.0 was used for this work). We created reference implementation in C++ and ported 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,



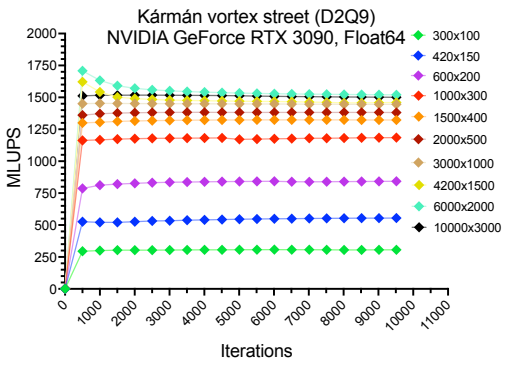
(a) Performance on Radeon R9 M370X GPU.



(b) Performance on GTX 1070 GPU.



(c) Performance on RTX 2080 Ti GPU.



(d) Performance on RTX 3090 GPU.

Figure 6 – 19: Double-precision performance analysis of 2D Kármán vortex on D2Q9 stencil.

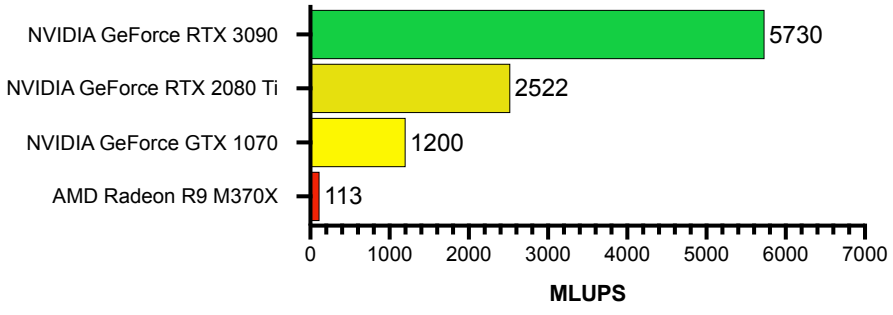
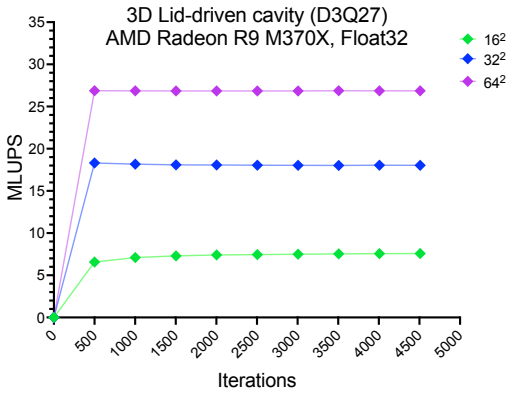
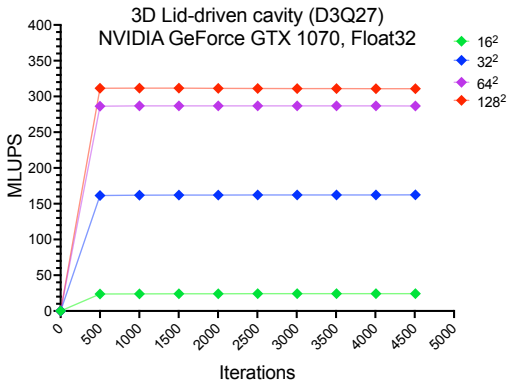


Figure 6 – 20: Peak performance of single-precision LBM simulations on D2Q9 stencil.

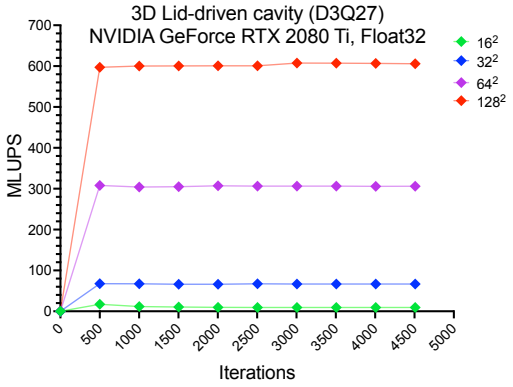
including real-time visualization code. Benchmarks were concluded for both C++ and Rust versions, resulting in identical performance. Data reported in this study are taken



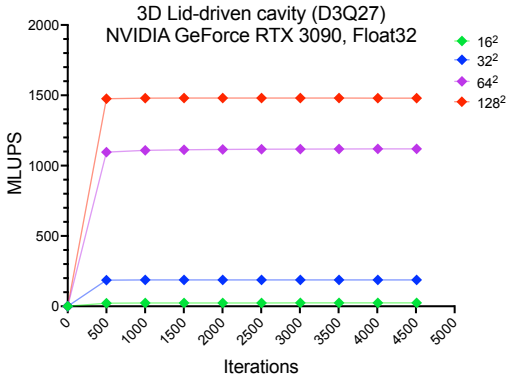
(a) Performance on Radeon R9 M370X GPU.



(b) Performance on GTX 1070 GPU.



(c) Performance on RTX 2080 Ti GPU.



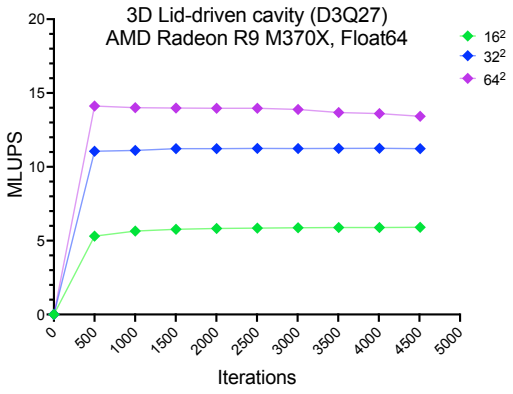
(d) Performance on RTX 3090 GPU.

Figure 6 – 21: Single-precision performance analysis of 3D lid-driven cavity on D3Q27 stencil with multiple-relaxation time.

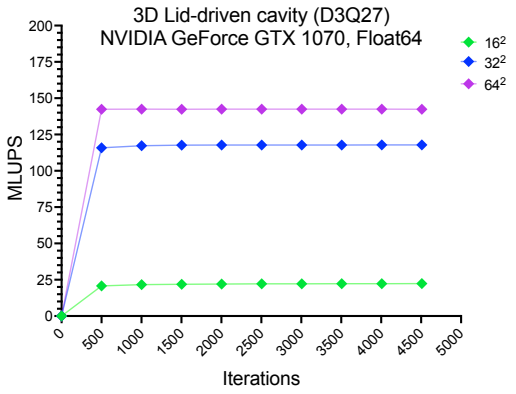
from the Rust version.

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 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.

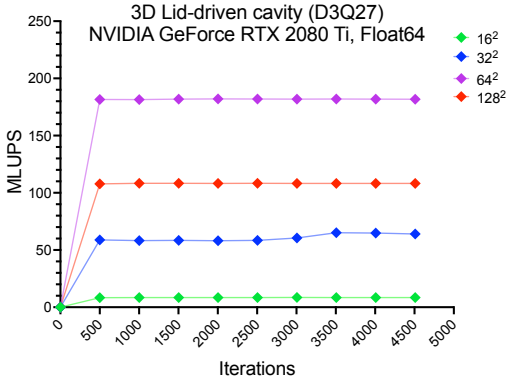
8 Discussion



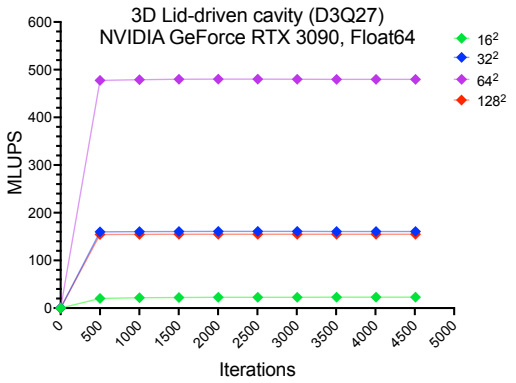
(a) Performance on Radeon R9 M370X GPU.



(b) Performance on GTX 1070 GPU.



(c) Performance on RTX 2080 Ti GPU.



(d) Performance on RTX 3090 GPU.

Figure 6 – 22: Double-precision performance analysis of 3D lid-driven cavity on D3Q27 stencil with multiple-relaxation time..

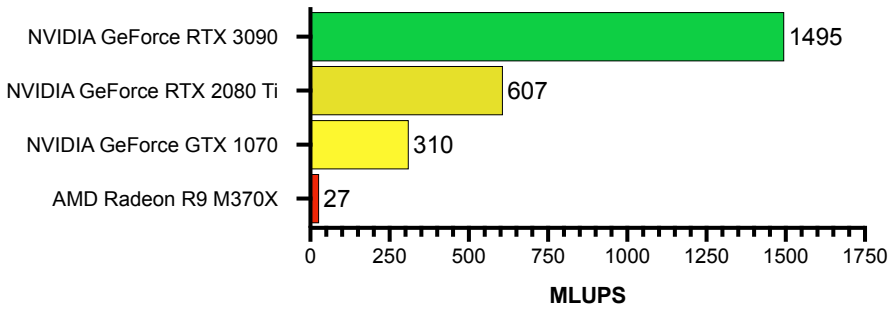


Figure 6 – 23: Peak performance of single-precision LBM simulations on D3Q27 stencil with multiple-relaxation time.

8.1 Future Work

As with work such as these

8.1.1 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.

8.1.2 Application to Real World

Biology
Steelmaking
Non-newtonian fluids

8.1.3 Streaming Visualized Pixels Over Network

Oculus Quest as a standalone VR is great device for games and applications that need only baseline VR performance... When used for scientific visualization in standalone mode, the communication would have to be wireless and implemented with client-server architecture, which means the speed of sending data would deteriorate the usefulness of

such application. It's still possible to use Oculus Quest for scientific visualization in VR, but it has to be connected to the PC workstation by USB-C or an equivalent proprietary cable called Oculus Link, which is not part of the standard package. It has to be bought separately.

8.1.4 Augmented Reality

Also, the augmented reality (AR) apps can be build from single codebase with Unity. Developers need to plan ahead and decide whether they want to target VR users, AR users or have the app be used by both device types (VR and AR). Usually, the term Mixed Reality is used to describe apps that target both types of devices. In this work, the focus was only on VR. Although in future, the AR looks as promising technology for specific usecases across scientific visualization, simulation and modeling industry. By leveraging Unity engine for the development of the visualization software for this thesis, extending the software to support AR will be straightforward process.

8.1.5 Simulation as an Educational Tool

References

Ahrens, J., Li-Ta Lo, Nouanesengsy, B., Patchett, J. and McPherson, A. (2008). Petascale visualization: Approaches and initial results, *2008 Workshop on Ultrascale Visualization*, IEEE, Austin, TX, USA, pp. 24–28.
URL: *http://ieeexplore.ieee.org/document/5154060/*

Ahrens, J. P., Geveci, B. and Law, C. C. (2005). ParaView: An end-user tool for large-data visualization, *in* C. D. Hansen and C. R. Johnson (eds), *The visualization handbook*, Academic Press / Elsevier, pp. 717–731. tex.bibsource: dblp computer science bibliography, https://dblp.org tex.biburl: https://dblp.org/rec/books/el/05/AhrensGL05.bib tex.timestamp: Fri, 28 Jun 2019 10:05:27 +0200.
URL: *https://doi.org/10.1016/b978-012387582-2/50038-1*

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*

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

Bryson, S. (n.d.). Virtual reality in scientific visualization, *COMMUNICATIONS OF THE ACM* **39**(5): 10.

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

Dangeti, S., Chen, Y. V. and Zheng, C. (2016). Comparing bare-hand-in-air Gesture and Object-in-hand Tangible User Interaction for Navigation of 3D Objects in Modeling, *Proceedings of the TEI '16: Tenth International Conference on Tangible, Embedded,*

and Embodied Interaction - TEI '16, ACM Press, Eindhoven, Netherlands, pp. 417–421.
URL: *http://dl.acm.org/citation.cfm?doid=2839462.2856555*

Dassault Systems (2019). Experience your design with Integrated Modeling and Simulation.
URL: *https://ifwe.3ds.com/modeling-and-simulation*

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*

Frisch, U., Hasslacher, B. and Pomeau, Y. (1986). Lattice-Gas Automata for the Navier-Stokes Equation, *Physical Review Letters* **56**(14): 1505–1508.
URL: *https://link.aps.org/doi/10.1103/PhysRevLett.56.1505*

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*

Hanwell, M. D., Martin, K. M., Chaudhary, A. and Avila, L. S. (2015). The Visualization Toolkit (VTK): Rewriting the rendering code for modern graphics cards, *SoftwareX* **1-**

2: 9–12.
URL: *https://linkinghub.elsevier.com/retrieve/pii/S2352711015000035*

Hardy, J., Pomeau, Y. and de Pazzis, O. (1973). Time evolution of a two-dimensional classical lattice system, *Physical Review Letters* **31**(5): 276–279. Number of pages: 0 Publisher: American Physical Society.
URL: *https://link.aps.org/doi/10.1103/PhysRevLett.31.276*

Harwood, A. R. G., Wenisch, P. and Revell, A. J. (n.d.). A REAL-TIME MODELLING AND SIMULATION PLATFORM FOR VIRTUAL ENGINEERING DESIGN AND ANALYSIS, p. 11.

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.

Klabnik, S. and Nichols, C. (2018). *The rust programming language*, No Starch Press, USA.

Koliha, N., Janßen, C. and Rung, T. (2015). Towards Online Visualization and Interactive Monitoring of Real-Time CFD Simulations on Commodity Hardware, *Computation* **3**(3): 444–478.
URL: *http://www.mdpi.com/2079-3197/3/3/444*

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*

Kress, J. (n.d.). In Situ Visualization Techniques for High Performance Computing, p. 26.

Kreylos, O., Tesdall, A. M., Hamanny, B., Hunter, J. K. and Joy, K. I. (2002). Interactive visualization and steering of CFD simulations, *in* D. Ebert, P. Brunet and I. Navazo (eds), *Eurographics / IEEE VGTC symposium on visualization*, The Eurographics Association. ISSN: 1727-5296.

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, L., Ma, Y. and Lange, C. F. (2016). Association of Design and Simulation Intent in CAD/CFD Integration, *Procedia CIRP* **56**: 1–6.
URL: *https://linkinghub.elsevier.com/retrieve/pii/S2212827116309982*

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*

Mahmood, I., Kausar, T., Sarjoughian, H. S., Malik, A. W. and Riaz, N. (2019). An Integrated Modeling, Simulation and Analysis Framework for Engineering Complex Systems, *IEEE Access* **7**: 67497–67514.
URL: *https://ieeexplore.ieee.org/document/8718666/*

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*

Marion, P., Kwitt, R., Davis, B. and Gschwandtner, M. (2012). PCL and ParaView — Connecting the dots, *2012 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, IEEE, Providence, RI, USA, pp. 80–85.
URL: *http://ieeexplore.ieee.org/document/6238918/*

Mawson, M. (2014). *Interactive fluid-structure interaction with many-core accelerators*, PhD thesis, University of Manchester, UK. tex.bibsource: dblp computer science bibliography, https://dblp.org tex.biburl: https://dblp.org/rec/phd/ethos/Mawson14.bib tex.timestamp: Mon, 15 Aug 2016 18:48:06 +0200.
URL: *http://www.manchester.ac.uk/escholar/uk-ac-man-scw:219513*

McNamara, A., Treuille, A., Popović, Z. and Stam, J. (2004). Fluid control using the adjoint method, *ACM Trans. Graph.* **23**(3): 449–456. Number of pages: 8 Publisher: Association for Computing Machinery tex.address: New York, NY, USA tex.issue_-

date: August 2004.
URL: *https://doi.org/10.1145/1015706.1015744*

Neumaier, A. (2004). Mathematical model building, *in* J. Kallrath (ed.), *Modeling languages in mathematical optimization*, Springer US, Boston, MA, pp. 37–43.
URL: *https://doi.org/10.1007/978-1-4613-0215-5*

Ohno, N. and Kageyama, A. (n.d.). Introduction to Virtual Reality Visualization by the CAVE system, p. 41.

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*

Qian, Y. H., D'Humières, D. and Lallemand, P. (1992). Lattice BGK Models for Navier-Stokes Equation, *Europhysics Letters (EPL)* **17**(6): 479–484.
URL: *https://iopscience.iop.org/article/10.1209/0295-5075/17/6/001*

Rago, G. (2015). *Numerical simulation: Theory and analysis*, Clanrye International.
URL: *https://books.google.sk/books?id=nRwdrgEACAAJ*

Rapp, B. (2017). Computational fluid dynamics, pp. 609–622.

SciVista (2021). SummitVR.
URL: *https://www.scivista.com/summitvr*

Shetty, N., Chaudhary, A., Coming, D., Sherman, W. R., O'Leary, P., Whiting, E. T. and Su, S. (2011). Immersive ParaView: A community-based, immersive, universal scientific visualization application, *2011 IEEE Virtual Reality Conference*, IEEE, Singapore, Singapore, pp. 239–240.
URL: *http://ieeexplore.ieee.org/document/5759487/*

Spadafora, M., Chahuneau, V., Martelaro, N., Sirkin, D. and Ju, W. (2016). Designing the Behavior of Interactive Objects, *Proceedings of the TEI '16: Tenth International Conference on Tangible, Embedded, and Embodied Interaction - TEI '16*, ACM Press,

Eindhoven, Netherlands, pp. 70–77.
URL: *http://dl.acm.org/citation.cfm?doid=2839462.2839502*

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

Su, S., Perry, V., Bravo, L., Kase, S., Roy, H., Cox, K. and R. Dasari, V. (2020). Virtual and Augmented Reality Applications to Support Data Analysis and Assessment of Science and Engineering, *Computing in Science & Engineering* **22**(3): 27–39.
URL: *https://ieeexplore.ieee.org/document/8978600/*

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*

Succi, S. (2018). *The lattice boltzmann equation: For complex states of flowing matter*. Pages: 1-762.

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

Thürey, N., Rüde, U. and Körner, C. (n.d.). Interactive Free Surface Fluids with the Lattice Boltzmann Method, *Technical Report* p. 9.

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/*

Victor, B. (2006). Magic ink: Information software and the graphical interface.
URL: *http://worrydream.com/MagicInk/*

Victor, B. (2018). Inventing on principle.
URL: *https://www.youtube.com/watch?v=8QiPFmIMxFc*

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.

Yang, X.-S. (2017). *Engineering mathematics with examples and applications*, Academic Press, an imprint of Elsevier, London ; San Diego, CA.

Yuan, H. Z., Wang, Y. and Shu, C. (2017). An adaptive mesh refinement-multiphase lattice Boltzmann flux solver for simulation of complex binary fluid flows, *Physics of Fluids* **29**(12): 123604.
URL: *http://aip.scitation.org/doi/10.1063/1.5007232*

Zhang, J. (2011). Lattice Boltzmann method for microfluidics: models and applications, *Microfluidics and Nanofluidics* **10**(1): 1–28.
URL: *http://link.springer.com/10.1007/s10404-010-0624-1*

Appendices

Appendix A System Manual

Appendix B User Manual

Appendix A

System Manual

Hardware Requirements

Virtual reality software in this work support VR headsets from Oculus and HTC out-of-the-box. How to install Oculus Rift and HTC Vive is descibed in section 8.1.5 and 8.1.5. To actually run VR content on your PC, it needs to meet at least minimum recommended hardware specifications from Table 8 – 1.

Component	Recommended Specs	Minimum Specs
Processor	Intel i5-4590 / AMD Ryzen 5 1500X or greater	Intel i3-6100 / AMD Ryzen 3 1200, FX4350 or greater
Graphics Card	NVIDIA GTX 1060 / AMD Radeon RX 480 or greater	NVIDIA GTX 1050 Ti / AMD Radeon RX 470 or greater
Alternative Graphics Card	NVIDIA GTX 970 / AMD Radeon R9 290 or greater	NVIDIA GTX 960 4GB / AMD Radeon R9 290 or greater
Memory	8GB+ RAM	8GB+ RAM
Operating System	Windows 10	Windows 10
USB Ports	3x USB 3.0 ports, plus 1x USB 2.0 port	1x USB 3.0 port, plus 2x USB 2.0 ports
Video Output	Compatible HDMI 1.3 video output	Compatible HDMI 1.3 video output

Table 8 – 1: Hardware specifications for VR support.

Windows 10 is the minimum supported operating system because Microsoft stopped supporting the Windows 7 and 8.1, beginning from 14th of January 2020.

Oculus Rift installation

Guide for installing Oculus Rift headset support on a Windows PC combined with hardware configuration manual is provided through interactive Oculus software, that can downloaded from https://www.oculus.com/rift/setup/.

Oculus Rift comes in a set including headset, two cameras for tracking that can stand on your desk, hand controllers called Touch controllers, USB and HDMI cables. Oculus Rift powers itself through USB port

HTC Vive installation

Detailed installation manual for HTC Vive virtual reality headset can be found at https://www.htc.com/managed-assets/shared/desktop/vive/vive_pre_user_guide.pdf or by searching the Support page at https://www.vive.com/us/support/.

HTC Vive comes in a set including headset, base stations called Lighthouses for creating the laser field for precision tracking, hand controllers, USB and power cables.

Appendix B

User Manual

Installation

Setting Up Virtual Reality Headset

Running The Application

Virtual Reality User Interface