Cranfield UNIVERSITY

School of Engineering

Msc Thesis

Parallelization of lattice Boltzmann method using CUDA platform

Tamás István Józsa

Supervisors: Dr. Irene Moulitsas
Dr. László Könözsy

September 2014.

Abstract

In the current thesis two existing two-dimensional, laminar lattice Boltzmann solvers were combined, parallelized and extended. First, the original C++ codes were rewritten using the C language. Second, the features of nVidia's CUDA software development kit were investigated using a simple benchmark. Third, the main steps of the solver were parallelized with CUDA using one GPU. Fourth, the performance gain was examined and analysed on two different platforms and the effects of double precision execution were investigated. Fifth, the solver was stabilized with a first order opening boundary condition (since the second order formulation often led to divergence). Sixth, the implementation was verified through the channel flow test case (Poiseuille flow) and a grid convergence study was carried out. Finally, several simulations were performed and the results were validated with the available data.

The parallel code proved efficient in every investigated case. The performance gain depended on the platform: on the first platform (a simple personal computer) the speed-up was between three and six, whilst on the second platform (a GPU cluster) it was between eight and sixteen. The double precision execution of the code needed 30% more time, and the double precision speed-up was less. The solver proved more scalable with single precision. The BGKW and the MRT models were able to capture the flow in the channel and showed good convergence properties. The models produced similar results in the other test cases which were in good agreement with the reference data. All in all, the prepared code can be a good basis for further development and optimization.

Acknowledgements

In general, people do not say thank you for something which seems natural. First of all that is why I would like to thank my whole family, especially my parents, Gabriella Kovácsik and István Józsa, for their care. They have supported and encouraged me totally naturally and honestly, ever since I can remember. I hope they know that their love is an infinite source of energy for me. My grandparents worked all their lives for our family. I would like to express my gratitude to them too: thank you Julianna Sulcz, Éva Kenyó, József Kovácsik and István Józsa. My sister Kata Józsa also helped me a lot. Her happy personality showed me the sunny side of the life even in the shadow of work.

I am really grateful for the assistance of my Supervisors, Irene Moulitsas and László Könözsy. After our consultations I could leave their office in a more optimistic and cheerful mood. I think both of them added important bricks to the house of my professional knowledge. I would not be here without the help of my former supervisor, György Paál. He planted the seeds within me, which became the basis of my scientific interest.

The first part of the thesis was done with Máté Szőke. I am glad that the common work forged our friendship, and that I could start to conquer the United Kingdom with him. Thanks to his patience, I am able to use Ubuntu without daily re-installations. Thanks to Jimmy-John Hoste, my first Belgian friend, for the high quality Belgian beers and the late night conversations. Both of them had an important relaxing effect. Since friends are our chosen family, I do not want to forget to mention Gábor Budai, Bianka Farkas, Márton Fogarassy, Csaba Kasza, Dávid Molnár and Károly Puhr. The nostalgic memories about our childhood and early adulthood bring a smile to my face every time. I hope that distance will not break the strong bond between us. I would like to thank Robert Rutherford for his time. His English instructions improved considerably the quality of the thesis. I think the environment determines us essentially. So last but not least thanks to everybody in my environment.

Contents

List of Figures

List of Tables

List of symbols

Latin letters

Symbol	Unit	Description
C	[-]	Collision operator
$\vec{c}$	[m/s]	Microscopical velocity
c_s	[-]	lattice speed of sound
$\vec{e}_i$	[-]	Discrete lattice velocity vector (i^{th})
$\vec{F}$	[N]	Force vector
F	[Hz]	Frequency
f	[-]	Distribution function
f^{eq}	[-]	Equilibrium part of the distribution function
f^{ne}	[-]	Non-equilibrium part of the distribution function
$\vec{g}$	[m/s^2]	Gravitational acceleration vector
H	[m]	Channel height
k_B	[m^{2}kg/s^2/K]	Boltzmann constant
Kn	[-]	Knudsen number
L	[m]	Characteristic length
m	[kg]	Mass
Ma	[-]	Mach number
N	[-]	Number of particles
n	[-]	Number of lattices
O	[-]	Numerical error
$\vec{p}$	[kg m/s]	Impulse vector
p	[Pa]	Pressure
p_a	[-]	Order of accuracy
Re	[-]	Reynolds number
St	[-]	Strouhal number
T	[K]	Absolute temperature
t	[s]	Time
$\vec{u}$	[m/s]	Macroscopical velocity vector
u	[m/s]	x-directional component of the macroscopical velocity
u_{lattice}	[-]	Lattice velocity
V	[m^3]	Volume
v	[m/s]	y-directional component of the macroscopical velocity
w_i	[-]	Weighting factor of the i^{th} direction
$\vec{x}$	[-]	Particle position vector
x	[-]	Horizontal axis of the Cartesian coordinate system
y	[-]	Vertical axis of the Cartesian coordinate system

Greek letters

Symbol	Unit	Description
δ	[m]	Grid spacing
λ	[m]	Mean free path
μ	[Pa s]	Dynamical viscosity
ν	[m^2/s]	Kinematical viscosity
$\nu_{lattice}$	[-]	Lattice viscosity
ω	[-]	Collision frequency
ρ	[kg/m^3]	Density (microscopical or macroscopical)
τ_r	[s]	Relaxation time

Abbreviations

2D	Two-Dimensional
3D	Three-Dimensional
API	Application Programming Interface
Arg.	Argument
BFS	Backward Facing Step
BGKW	Bhatnagar–Gross–Krook–Welander
CAD	Computer Aided Design
CAE	Computer Aided Engineering
CFD	Computational Fluid Dynamics
CPU	Central Processing Unit
CT	Computed Tomography
CUDA	Compute Unified Device Architecture
D2Q9	The Applied Lattice Speed Model
DDR3	Double Data Rate Type Three
DNA	Deoxyribonucleic Acid
DP	Double Precision
DRAM	Dynamic Random Access Memory
FLOP	Floating-Point Operation
GCI	Grid Convergence Index
GPGPU	General-Purpose Computing on Graphics Processing Unit
GPU	Graphics processing unit
HPC	High Performance Computing
L1, L2, L3	Cache Levels
LBM	Lattice Boltzmann Method
M1, M2	First and Second Machine
MPI	Message Passing Interface
MRI	Magnetic Resonance Imaging
MRT	Multiple Relaxation Time
NS	Navier-Stokes
RAM	Random Access Memory
SIMPLE	Semi-Implicit Method for Pressure Linked Equations
SDK	Software Development Kit
SLI	Scalable Link Interface
SoC	System on Chip
SP	Single Precision
TRT	Two Relaxation Times
UPC	Unified Parallel C programming Language

1 Introduction

High performance computing (HPC) plays an important role in the field of scientific computations, such as computational fluid dynamics (CFD). The rapid improvement in the performance of computers, combined with the availability of ever more powerful computational facilities has been the main reason for the development of efficient parallel computational methods. The basic concept of parallelization follows the philosophy of the ancient Roman Empire, known as "Divide et impera" (divide and conquer). In practice it means that instead of using one processing unit, the tasks are distributed between several processing units. The theoretical speed up is equal to the number of processing units.

It is not a secret that the a common purpose of the HPC society is to create an exascale supercomputer. The implementation has numerous stumbling block both on the hardware and the software side. The technologies used for classical supercomputers are too energy-, and money-consuming. Consequently new ideas and technologies are required. Probably the future hardware structures will demand novel programming techniques also. Maybe the rapidly developing graphical cards will be able to pick up the gauntlet.

The structure of graphics processing units (GPUs) offers a massively parallelizable computational resource which was realized in the beginning of the 2000s. The software development kit (SDK) of nVIDIA called CUDA makes the general-purpose computing on graphics processing unit (GPGPU) possible [1]. The platform was applied to modern technical problems, like medical image processing, humanoid robot programming and engineering simulations. Nowadays several research institutions have their own GPU cluster including graphical cards designed for scientific computations. Although the GPU is an excellent way to improve performance, the complex memory architecture of the graphical cards makes the programming a real challenge.

The goal of CFD is to analyze different fluid flows by solving the governing equations with numerical methods. As the name implies, CFD is a field that relies heavily on the usage of computational resources. Therefore it is only natural that the evolution in computing can advance the frontiers of CFD studies and the related numerical techniques. The classical approach of fluid simulations is to treat fluids as continuums and describe the motion of fluid packages based on the Navier-Stokes equation. Modelling of the particle level of the materials is another way to represent the fluids but this approach is computationally very intensive, produces unnecessary data from a practical point of view, and makes large-scale modelling impossible. The third option is the so called mesoscopic modelling. The Boltzmann equation gives a statistical representation of non-equlibrium thermodynamical systems and the discretized Boltzmann equation led to the family of lattice Boltzmann methods [2, 3] which proved a reliable tool to model fluid

flows in practice. Nowadays the lattice Boltzmann methods can be used to solve complex fluid mechanical problems such as multiphase turbulent flows [4] etc.

1.1 Objectives of the thesis

The aim of this study is to parallelize the critical parts of the existing two-dimensional in-house lattice Boltzmann code using the CUDA SDK. The two codes [5, 6] were written in C++. In order to achieve better portability and speed, the best properties of the codes will be combined and implemented in C. The computationally costly sections of the new code will be identified using detailed profiling of the program. The effects of double precision execution will be examined. On the physical side a new boundary condition will be avoid numerical instabilities. After parallelization of the critical parts, validation and verification will be performed.

2 Literature survey

2.1 Short overview of fluid mechanics

Continuous technological development based on fluid flow modelling plays an important role in our everyday life too. However a layman may not see how fluid dynamics influences his or her life, even though many branches of technical evolution were highly related to the experimental and computational techniques of fluid dynamics in the last century. Some examples without attempting to be comprehensive are:

- pursuit of efficiency in the energy sector: gas [7], steam [8], and wind turbine design [9];
- aerospace development including helicopters [10] and aeroplanes [11];
- developments connected to the car industry, e. g. shape optimization of cars to reduce drag [12], internal combustion engine improvements [13] etc;
- pipe network analyses of potable [14] and waste water;
- biomechanical research, e. g. simulation of blood flow in arteries [15], lungs [16];
- cooling system engineering, from reactors [17] to computer servers [18];
- design of reliable inkjet printers [19];
- etc.

The list has no end.

One can categorize the techniques of fluid dynamics based on compressibility which is an obvious property of the investigated fluid. One can distinguish compressible and incompressible flows [20] but the situation is not so simple, because every fluid is compressible in an extreme environment. Anyway, a general assumption is that e. g. water can be modelled as an incompressible liquid. In the case of air it is possible to choose a compressible or incompressible approach too: if we are below the compressible limit (typically Mach number smaller than 0.2) then the incompressible description is allowed. Since the Mach number is related to the velocity of the fluid, one can separate high speed and low speed flows too. It is worth estimating the importance of compressible effects because capturing of wave phenomena in compressible materials is quite difficult and sometimes it is unnecessary.

Based on this categorization of the fluid medium, the question arises: is it possible to describe the dynamic behaviour of fluids? As far as we know the Navier-Stokes equations (NS

eqs.) are capable of modelling this behaviour. The compressible or incompressible assumption clearly determines the form and the nature of the Navier-Stokes equations. Since the topic of the current study is limited to incompressible flows, the remaining parts are restricted to this group. The NS eqs. in this case can be written as:

$$\nabla \vec{u} = 0, \tag{2.1}$$

$$\frac{\partial \vec{u}}{\partial t} + \nabla \cdot (\vec{u} \otimes \vec{u}) = -\frac{1}{\rho}\nabla p + \frac{\mu}{\rho}\nabla^2 \vec{u} + \vec{g}. \tag{2.2}$$

In the continuity equation (2.1) ∇ is the vector differential operator, known as the Nabla or Del operator and $\vec{u}$ denotes the velocity vector. In Equation (2.2) t stands for the time, ρ for the density, and μ for the dynamic viscosity of the fluid. $\vec{g}$ symbolizes outer force fields, e. g. gravity. While the former equation expresses the conservation of mass, the latter ones symbolize the conservation of momentum. The first step of the derivation of the above expressions is based on the continuum structure of fluids; thus it is assumed that the fluid is built up from small continuous "packages". It is well known that our world is constituted from particles so that this description method seems like some kind of paradox. Actually the truth is, provided our "packages" include a high enough number of particles, the modelling of fluids following the continuum approach works well. In practice the dimensionless Knudsen ($Kn = \lambda/L$) number is used to quantify what is a high enough number of particles. While λ stands for the mean free path of a particle in the fluid, L is a physical length scale (e. g. the diameter of a hole). If the Knudsen number is close to one the continuum approach loses its validity (it works better in cases of lower Kn). We have arrived to another point where we can categorize the technical problems: the continuum description technique can be treated as macroscale modelling, while one can distinguish microscale and mesoscale modelling too. In [21] Gad-el-Hak has presented a detailed categorisation of the CFD methods based on the Knudsen number.

In microscale modelling we use the particle behaviour of the material as a way to describe our system. Even in small scales it proved impossible to take into account every particle (just think of one mole of water containing $6 \cdot 10^{23}$ molecules). The governing equations of this level are Hamilton's equations [2]. It is more feasible to use statistical modelling, and investigate the molecules not individually but in collectively. The Boltzmann equation as the basic equation of statistical mechanics is the known "mathematical tool to investigate analytically or numerically the properties of fluids far from equilibrium" [3]. This field of fluid dynamics has individual techniques, e. g. Monte Carlo method. Although hydrodynamical modelling on a microscale basis is not impossible it is a computationally intensive and quite time consuming family of the simulation methods even nowadays. The lattice Boltzmann method is generally referred as a

mesoscale modelling technique because in this case the detailed description of the micro-world is ignored and only particle motions in discrete directions are allowed. Although the basic idea sounds a little bit strange the reported numerical technique proved reliable in several fields.

2.2 Applications of the lattice Boltzmann method

The application of the LBM is relatively extensive. One of the most typical application fields is multiphase flows [3, 22]. The applicability of the method was presented through the basic benchmark of the Rayligh-Taylor instability by He et al. [23]. Later on He et al. [24] gave a detailed description of the thermodynamical background of multiphase modelling with LBM. Earlier the method worked only with a relatively high error in cases of large multiphase density ratio (the limit was around 100) which was presented by Kim et al. [25] who also extended the equilibrium density function formulation. In this way the method became suitable for multiphase flows with a density ratio around $1,000$. Another approach was used for bubbly flows (density ratio again around $1,000$) and the results were validated too [26]. LBM also was combined with Lagrangian particle tracking to model flow in filters [27]. Reactive flows were also captured [28, 29], e. g. flame propagation through the Rayligh-Taylor instability was simulated [30]. Pan et al [31] used the method to investigate the flow through porous media. Later on Pan et al. [32] examined also multiphase flows through porous materials. (Spatially resolving the flow through porous materials is out of the Knudsen limit of the Navier-Stokes equation.)

Another field of application is biomedical flows. The simple meshing technique makes the LBM popular for these flows when it is necessary to mesh irregular, complex geometries (e. g. blood vessels) [33]. The easy porous media implementation helps considerably in modelling the effect of stents [34, 35]. Furthermore Ouared, R. and Chopard, B. simulated the clotting process in aneurysms with a non-Newtonian description of the blood [36]. Závodszky and Paál validated the in vitro flow simulations of intracranial aneurysms with laser optical measurement techniques [15]. Fluid–structure interaction simulations [37, 38], and several other advanced modelling techniques form further research and development areas.

From an industrial point of view the LBM is highly relevant to the chemical industry because of its reactive and multiphase flow treatment. LBM is applied in the automotive industry too (for commercial vehicle design low speed flow simulations are generally satisfactory.) Fares performed the Ahmed body simulation as a basic test case [39]. In this work the LBM was combined with the k-ε turbulence model. Turbulence modelling is a challenging part of CFD and it is an active research topic to implement feasible turbulence models combined with the LBM [4, 22].

One of the biggest advantages of the LBM is its good scalability. LBM codes are highly parallelizable since the collision as one of the main steps of the loops is totally independent from the neighbouring cells. CPU and GPU parallelizations were also carried out. On the CPU side the Message Passing Interface (MPI) was used [40], while on the GPU side the CUDA developer platform of nVidia was found popular both in two-dimension (2D) [41] and three-dimension (3D) [42]. The solvers were also implemented for multiple GPUs [43] (up to two-billion nodes). Tölke [41] and Habich [42] et al. reported approximately one order of magnitude gain in performance compared to multi-core CPUs. In the following subsection a bridge between physical modelling and programming will be presented.

2.3 From differential equations to high performance computing

The desire to describe our environment has concerned humanity since time immemorial. The methods used to investigate the world around us were developed in parallel with human evolution. For reasons beyond curiosity it has become important to be able to forecast the behaviour of different systems. Just think of the monumental ancient buildings, for example the aqueduct system of the ancient Romans. The structural design itself is a basic static and strength of material problem from an engineering point of view, while the water distribution is related to fluid mechanics.

We approach the engineering problems typically through physics. The projection of a real world problem to a physical problem is called modelling. Modelling neglects several phenomena compared to the real world problem. (In the case of the aqueduct design for example three-dimensional unsteady behaviour of fluids can be neglected, and a one-dimensional steady state approach can be applied.) Since the language of physics is mathematics, we need to "translate" the physical problem to equations. As the physical problems have become more and more difficult, the describing mathematics also has become more complex. Whilst algebraic equations is often satisfactory to model simple physical systems, the description of modern physical problems are often related to differential equations (NS eqs.) or integro-differential equations (Boltzmann equation).

It is known that an analytical solution is mostly not available for the governing differential equations especially in complex cases. To overcome this barrier several numerical techniques were worked out to discretize differential equations [44]. Discretization means that we give up describing the system with functions and we try to use finite points to recover the behaviour of the system. The discretiaztion results in time steps in time and grid points (lattice points) in space. The aim of numerical schemes is to represent the derivatives of the original differential equations based on the discretized points. After all this, we have a discretized, algebraic

equation system which can be solved using computers [45].

In the current thesis the engineering problem is to describe the behaviour of the fluids. The governing equation is the lattice Boltzmann equation. The Chapman-Enskog theory [46, 3] makes a clear connection between the lattice Boltzmann equation and the incompressible Navier-Stokes equations (in the form proposed by Chorin [47], known as the artificial compressibility approach). Consequently the physical model describes the incompressible, laminar, Newtonian fluid flow. The discretized equations are presented in Chapter 3.1.2. The 2D solvers were implemented earlier [6, 5].

One may think that after the implementation there is nothing else to do but anybody, who has performed CFD simulations, knows, that is not the end of the story. CFD solvers are quite resource intensive and it takes often weeks or months to obtain a converged solution. Even in 2D, the computers sometimes take days to solve the equations in the discretized domain. The lattice density (number of cells) influences the accuracy significantly but higher spatial resolution leads to increased computational costs. Serial implementations are not able to provide results quickly, and the time is an important factor especially in industry related projects. This is the reason why we need HPC and special implementation techniques to speed up the solvers.

2.4 Past, present, and future of HPC

Seymour Roger Cray is often refered as the "Father of supercomputing" [48]. He was a famous electrical engineer who designed the fastest computers for years. Cray Research, funded by Cray, is a market leader company in the field of supercomputers even nowadays [49]. The revolution started in Manchester in the early 1960s, when the first Atlas machine (designed by Cray) began its operation [50]. It was four times faster than the rival IBM 7094. Atlas owed its success partially to the applied parallel structure.

The author would like to underline one more computer from the early stage of supercomputing, namely the CDC 6600. This computer worked in the CERN laboratory in Switzerland, and it was also designed by Cray. The machine was ten times faster than other computers in the market. The key concept was to use silicon transistors in place of germanium transistors. Furthermore Cray worked out an efficient refrigeration technique to overcome the cooling problems. As the reader will see soon, cooling is a serious dilemma in the case of supercomputers.

In the early stage of the development the designers increased mainly the clock rate of the processors to obtain better performance. The clock rate of CPUs almost reached the theoretical limit in the 1990s. The research centres then changed their strategy: while the earlier supercomputers were built only with a few processors, the later ones included several collaborating processors [51]. We can see the same tendency in the field of electronics in general. While

parallel computing was the privilege of the researchers at the beginning, later on several manufacturers started to produce CPUs with multiple cores. The beast ran away. On the one hand, the fastest supercomputers include hundreds of thousands of cores [52]. On the other hand, parallel computing was totally integrated into our everyday life too. Multi-core CPUs work in our desktops, notebooks, and mobile phones...

Figure 1 represents the main properties of CPUs in the last 40 years. The dashed lines indicate the expected trend based on Moore's law. Moore's law "states that processor speeds, or overall processing power for computers will double every two years" [53]. The law also means that the number of transistors is doubled every second year. The curve of transistor number follows this law quite accurately. It is also clearly visible how the progressively increasing trend of the clock rate (frequency) slowed down, then stagnated between 2005 and 2010. The introduction of dual-core processors is also indicated (before 2005).

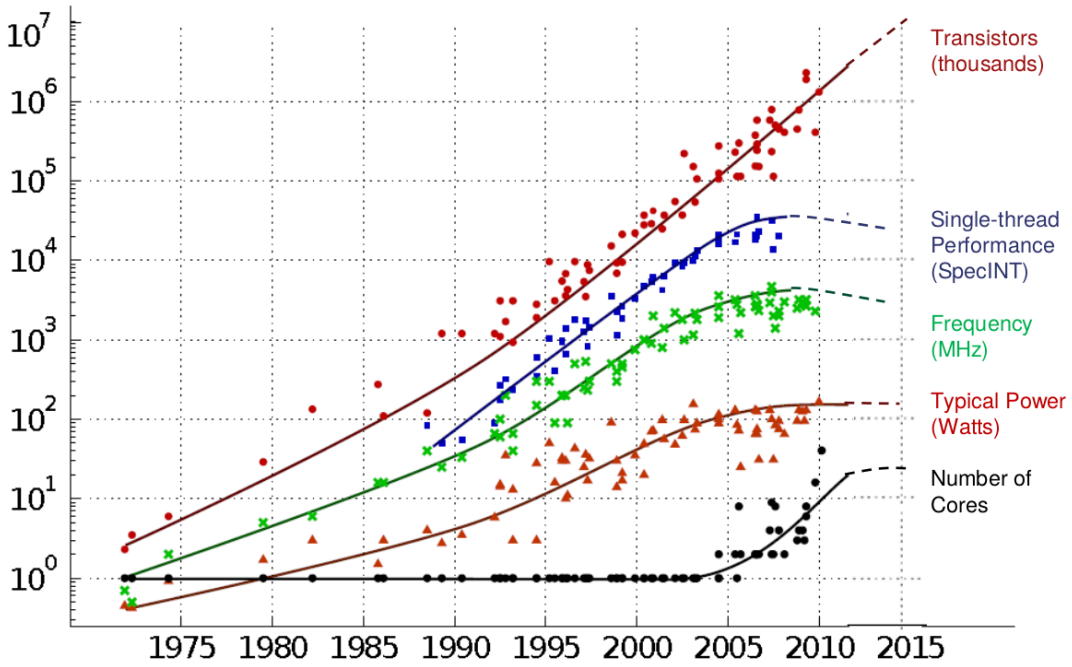


Figure 1: Properties of CPUs over time [54]. (Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten.)

Although the expansion of electronics seems like a triumph of technology, a coin has two sides; the performance of the best supercomputer is above thirty petaFLOPs/second but the power consumption of the CPUs multiplied (Figure 1). (FLOP stands for floating point operation. The performance of computers is typically measured in FLOPs/second.) Several supercomputers from the top 500 rank list [52] need megawatts to work and most of the used power is converted to heat. Almost the half of the energy consumption cover the cooling since overheating has been a critical problem since the first supercomputers.

The first exaFLOP supercomputer "is scheduled to begin working in 2020" [55]. The power requirement of this computing monster would be 30 to 40 megawatts. "One megawatt per year in the United States is about a million dollars. So just to turn it on and power it will cost you between $30 million and $40 million" said Jack Dongarra [56]. Scientists are looking forward to the next challenge but we can see that development in this direction is not sustainable. The energy demand and cooling are the bottlenecks of high performance computing. This is where mobile device processors and GPUs come in.

The Mont-Blanc project target was to design an efficient supercomputer architecture using the technological achievements of mobile devices [57]. For this purpose two supercomputer prototypes were built from system-on-chip (SoC) units. A SoC architecture includes every necessary element of a computer in an integrated circuit. The CPUs have 2-8 cores with a clock rate up to 2.5 GHz. GPUs are built in only for visualization purpose currently.

The ARM company specialized in producing CPUs and GPUs for mobile devices [58]. SoC units (for example Samsung Exynos [59] or nVidia Tegra series [60]) are typically built from ARM Cortex CPUs, and ARM or nVidia GPUs [61]. Although these systems may reach competitive performance in the field of HPC one day, they struggle with some limitations nowadays [62, 63]. Anyway, it seems a promising direction, since the heat production of optimized SoC processors may be 15-30 times less than the currently used HPC processors. (By the way, Göddeke at al. has already published about the implementation of the LBM on the prototype of the Mont-Blanc project [64].)

The author would like to mention another promising HPC project from the world of microelectronics, namely Parallella. "The 66-core version of the Parallella computer delivers over 90 GFLOPs on a board the size of a credit card while consuming only 5 watts under typical work loads. For certain applications, this would provide more raw performance than a high end server costing thousands of dollars and consuming 400W." [65]. The small board is programmable with C/C++ and it has a shared memory architecture.

So far only the hardware of supercomputers has been discussed and analysed but HPC requires special software support too. The most typical programming languages of parallel computing are C, C++, and Fortran. One needs "packages", like MPI (Message Passing Interface) [66], OpenMP [67], or, OpenCL [68] appeared to add the commands which make the execution distributed. MPI was introduced in the early 1990s, OpenMP in the late 1990s, while OpenCL in the late 2000s. Although MPI is the oldest of the listed languages, it is also the most widely used. MPI programming takes high effort and often results in complex, nearly unreadable codes. If we keep in mind the exa-scale supercomputer target, there are other languages that may be considered. Both Co-Array Fortran [69], and UPC [70] (Unified Parallel C) offer a

more user-friendly environment (compared to MPI) and the well implemented compilers facilitate their use. Unfortunately only a small handful of research groups applies these in practice, so that MPI remains probably the dominant approach on the CPU side for the next few years.

Some researchers predict a shiny future for GPUs in HPC. On the GPU side, no doubt OpenGL [71] is the most popular language (since 1992) but it is related mainly to graphics programming. The author found that CUDA provides a more general purpose programming method, but it can be used only for nVidia cards. The reader can find further information about GPU programming in the following sections. The author will try to highlight why GPU implementation is suitable for the LBM and how it can be so efficient.

2.5 GPGPU - supercomputers at home

2.5.1 Historical background of the graphics processing unit

Cicero's famous expression, "Historia est Magistra Vitae" (history is life's teacher) could be the motto of the current chapter in which the author will provide a short walk-through of GPU evolution. Although the historical review of a scientific field is not essential to understand the ideas and to be able to apply the knowledge, it can be instructive to look into the past.

Once upon a time, in the late 1970s, the first display controllers functioned mainly as converters and broadcasters between the processor and the displaying device. "The incoming data stream was converted into serial bitmapped video output" [72]. These display controllers gave only lines of pixels. Compared to the first generation controllers the "Pixie" chip of the Radio Corporation of America (RCA) meant a significant step forward: the chip was able to produce a video outlet with a resolution of 64×32. In the early 1980s Motorola's and Commodore's video display generators were found to be suitable for the first home computers.

In the 1980s Intel targeted the computer graphics industry and made the iSBX 275 chipset from $82,720$ graphics display controllers. The new chipset was suitable for displaying eight different colors with a resolution of 256×256 [72, 73]. Intel's product speeded up simple graphical operations, like line and curve drawings, and it was used inter alia for flight simulators, and for military and space simulations. (As usual in technological advancement, the military industry was a basic driver of the GPU development). ATI Technologies Incorporation was introduced its graphical unit onto the market in the middle of the 1980s. While ATI's Color Emulation Card had a memory 16 kB in the middle of the 1980s, thanks to the rapid development the memory of the cards reached 1 MB at the end of the decade. In the following years the resolution reached 800×600 pixels [72] and the graphical units "freed up the CPU for video processing (such as drawing and colouring filled polygons)" [74]. The GPU has became an individual part of the

computers.

As time went on, the entertainment industry slowly conquered informatics. Resource intensive games for PCs and different consoles were introduced to the market in the early 1990s. At this time CPU-assisted visualization has become typical, and graphics accelerators were used to speed up the processes. In 1992 the Silicon Graphics Incorporation created an application programming interface (API) called OpenGL. The goal of the new API was clearly to ensure a standardized programming environment for GPUs, to hide the differences between the platforms, and to avoid the complicated direct programming of the hardware. The basis of the new API was the C programming language, and it was designed to be platform independent [75]. It is a fact that OpenGL is widely used nowadays for display purposes (in different computer aided design (CAD), computer aided engineering (CAE) softwares and games) indicates well the success of the API. It is also a typical requirement for new GPUs to be compatible with OpenGL.

Naturally Microsoft tried to answer the challenge of the Silicon Graphics Incorporation, and the Direct3D API was created for the Windows operating system. Although Direct3D was a failure, it formed the basis of DirectX which has become an essential part of the Windows operating systems. From the middle of the 1990s Glide was also a competitive rival of OpenGL, especially in the fields of computer games. Glide was the API of the 3dfx Interactive company (founded in 1994) and the success of the API was mainly caused by the incredible popularity of the 3dfx accelerator devices, the Voodoo series.

The Voodoo cards were intended to provide an enjoyable three-dimensional gaming experience. Since the basic graphical hardware could handle only two-dimensional operations with a high enough speed, producing 3D accelerators was found to be a logical development direction by the 3dfx company. The Voodoo cards were responsible only for 3D operations (for example z-buffering), while the original graphical units worked only on the simple 2D operations. (Depth buffering, or z-buffering is a popular algorithm for 3D rendering; the algorithm decides how to overlap objects and determines which one is closer to the observer.) The Voodoo products were coupled in series after the original graphical units, and the displaying device was joined directly to the Voodoo cards.

There was another notable result of 3dfx's developments, namely scan line interleave (SLI) technology. The technology allowed sharing the the load of the graphical unit between two (or more) units. To this end the GPUs were connected with a simple ribbon cable (the same technology is used actually for the modern GPU clusters). Despite its sudden prosperity, in the early 2000s 3dfx lost its nearly hegemonic role in the market; consequently the Glide API lost its favourable position too [72].

Before the rise of 3dfx, the giant, nVidia was already born in 1993. Although the first nVidia products were not unquestionably successful, the company introduced the GeForce 256 GPU into the market in 1990. The GeForce 256 card had a high enough capacity to perform the necessary calculations alone without an accelerator card (it had 23 million transistors, 32 MB of 128-bit DRAM (dynamic random-access memory), and DirectX support) [76]. Only ATI's Radeon 7500 card had similar capacity [74]. The new trend led to the breakdown of 3dfx in some years and it started to rule the market. While in the early 1990s the CPU was mainly responsible for graphics and it could send more data to the GPU than the GPU could handle, approximately ten years later, at the beginning of the 2000s, the GPUs were able to accept the sent data and process it in parallel.

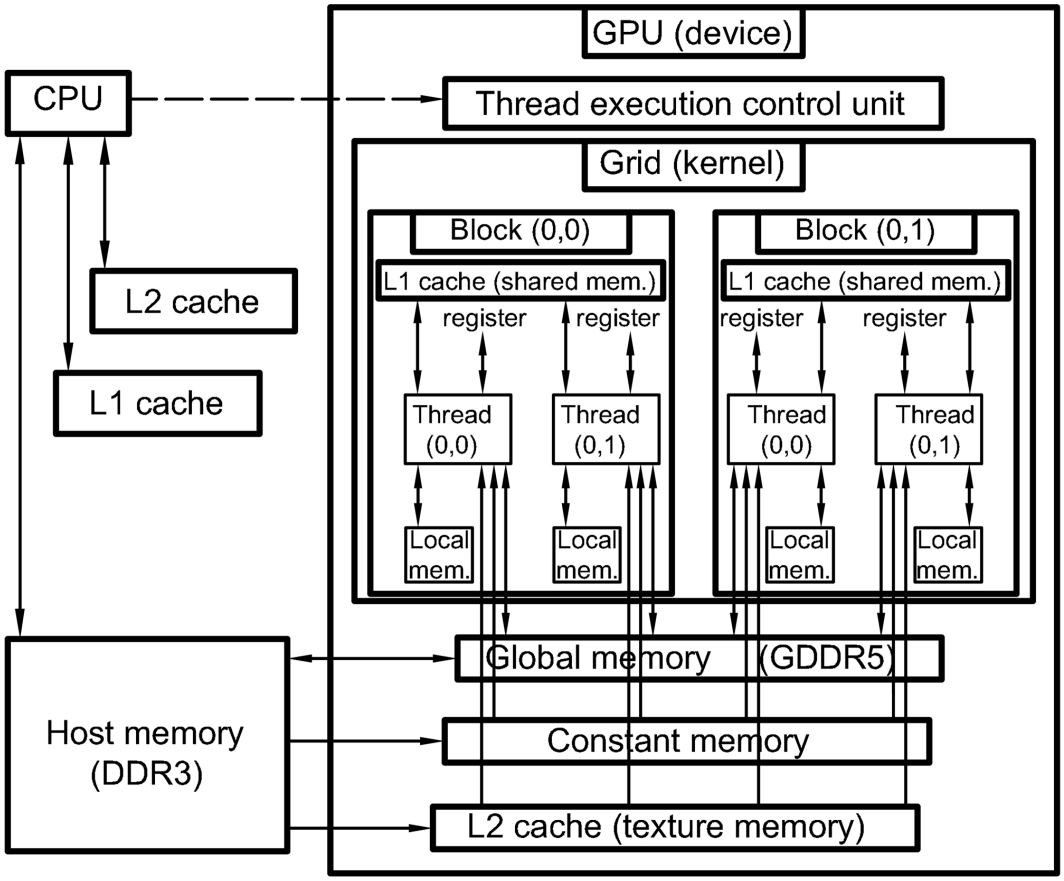


Figure 2: Schematic drawing of the Fermi architecture

The first cards from the new wave did not have very user-friendly programmability properties. (The operations were programmed in the hardware itself and the user could only send the data to the GPU. It was relayed fast to programmable cards but the language was low level, assembly-like shading language [74].) The nVidia GeForce FX and the ATI Radeon 9700 as "the first fully programmable graphics cards hit the market" in 2002 [74]. The floating point

operation support was an important step forward to GPGPU in 2003. In the next few years a large amount of effort was taken by the companies to improve the programmability of cards, which helped the GPUs to break into the field of scientific computing too.

In the early 2000s "the rate of GPU hardware technology was accelerating at a rate much faster than Moore's law" [77]. This represents well the incredibly fast advancement of GPUs. The financial collapse of the 3dfx company led to the bi-polarization of the GPU's market. The two racers, ATI and nVidia defeated the rivals year after year (for example nVidia annexed 3dfx, the biggest competitor of the '90s, in 2002, and several companies of the '90s (Terayon, ArtX, etc.) had a similar fate). Since the beginning of the 90s the customers have seen a race between the two giants.

While the cards were built with more and more memory and higher computational capacity the companies fought for more user friendly programmability to utilize the GPU's massively parallel nature for non-graphics application too [74]. In 2007 nVidia released the CUDA language, a CPU-like programming language only for nVidia cards. (Of course ATI tried to answer the challenge with its own language, Stream, and also DirectX offers similar features.) CUDA is based on the C language and includes several additional functions. As the reader will see later on, the code parts, executed on the GPU, can be called as functions. (GPU functions are typically called kernel functions). The language partially revolutionized HPC: nearly every modern supercomputer has a GPU cluster too. The role of GPU clusters is clearly to speed up the suitable part of the resource intensive programs.

2.5.2 The structure of nVidia GPUs

To understand the limitations and the key for successful parallelization using GPU, one has to know the structure of the applied card. The Fermi microarchitecture was the basis of the new nVidia supercomputing architectures and it had the most important elements of a modern, massively parallel GPU. Fermi architecture was followed by the Tesla and Kepler architectures. Since one of the applied GPUs had Fermi architecture the current chapter will present only the structure of these devices.

Figure 2 shows a schematic drawing of a CPU - Fermi GPU coupling [60]. On close inspection, we can see that the memory structure of the host and the device is quite different: the GPU has more memory "layer". Although nVidia aims to provide a CPU-like programming platform with CUDA, it is not possible to bridge the physical differences and connections between the CPU and the GPU. Consequently the memory management is a key part of GPGPU which determines the theoretical speed up limits too.

In general computer codes are written first in serial. The variables are declared then allo-

cated in the host memory. After the initialization (setting variable values) the user can define a sequence of commands which is executed by the processor (CPU). The L1 and L2 cache was made to speed up computations: if an operation is in progress at a memory location the neighbouring values from the host memory are copied to the cache. Probably the next operation will happen with one of the neighbouring elements, thus it can be read from the cache. The trick is that the maximum bandwidth of the double data rate 3 (DDR3) host memory is approximately 25 GB/s [78] while the reading bandwidth of the L1/L2 cache is around 300/200 GB/s. Although the cache bandwidth is higher by one order of magnitude compared to the host memory, the storage capacity is limited: users can have easily 20 or more GB DDR3 host memory (random access memory - RAM) in a single PC, but the size of the cache memory is only around 10 MB. In the case of the CPU programming mainly the operating system and the applied compiler are responsible for the cache memory handling.

nVidia GPUs are built up from multiprocessors which include some groups of cores. The used GPUs had 32 cores in every multiprocessor (cores are sometimes referred to as stream processors). "Each core can execute a sequential thread, but the cores execute in what nVidia calls SIMT (Single Instruction, Multiple Thread) fashion; all cores in the same group execute the same instruction at the same time" [79]. The multiprocessors need two cycles to execute single precision (SP) operations within the warps. In general only one-third of the cores of the multiprocessors are able to perform double precision (DP) operations. This means that the peak speed of double precision execution is only one-third of the single precision execution. While double precision operations occur on one-third of the cores the other "free" cores are capable to performing single precision operations.

It is shown in Figure 3 how the performance of GPUs and CPUs increased in the case of double and single precision. The progressive development of GPUs is clear. GPU performance is almost ten times higher nowadays than the CPU performance. The graph emphasizes the much lower GPU performance for double precision operations. As we can see the difference between single and double precision peak performance is less significant if we use CPUs. The author notes that double precision was available earlier only in the more expensive GPUs which are especially designed for scientific computations [60]. Based on the literature, it seems that the limited speed of double precision operations is one of the main disadvantages of GPU computing.

As we saw, in the case of a GPU device the computing elements are the threads (besides the host processor). The GPUs are called massively parallel computational tools since theoretically every thread can carry out operations at the same time. In practice 32 threads form a warp and threads within a warp work together. The commands for threads come from the CPU and the

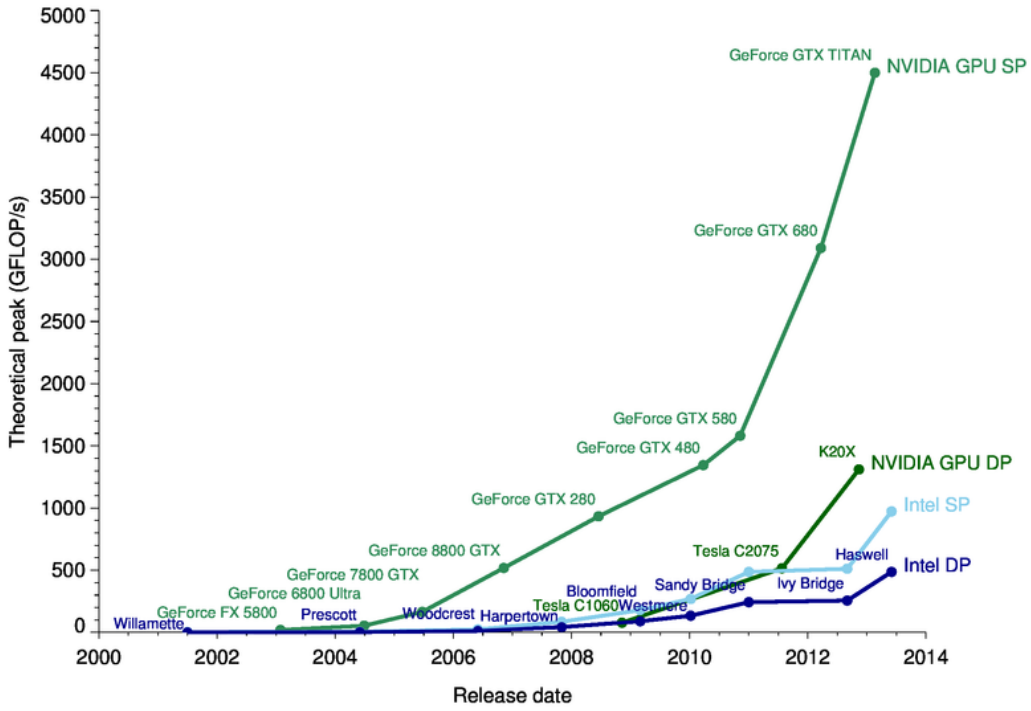


Figure 3: Theoretical peak performance of Intel CPUs and nVidia GPUs over time [80]. (SP stands for single precision, DP stands for double precision.)

thread execution control unit is responsible to mediate the tasks to the threads. As the arrows show in Figure 2, every thread has readable and writeable register memory and local memory. Register memory is the fastest memory but it is slow. Local memory is slower and can store more data which cannot fit to the register memory [81].

The group of threads are organized to blocks. The blocks have their individual L1 cache, generally referred as shared memory. The shared memory is available for every thread within a block and the bandwidth is in the order of 1.5 TB/s. The size of shared memory is around 50 MB, and the "lifetime" of shared variables is only the actual kernel function. The group of blocks form a grid, which is the working part of the device. In the case of multiple GPUs one has more grids, and the communication between them also has to be managed [43, 82].

Examining of Figure 2 leads to the recognition that the threads of the device do not have direct access to the host memory where the data is stored initially. (It is unable to read/write data from/to files directly to/from the device memory consequently these operations require to store the variables in the host memory.) One can see that there are three ways from the host memory to the threads: the global, the constant, and the texture memories of the device. From these memory types only the global memory is readable and writeable too. These facts clearly show that global memory has a key role in the communication between the host and the device.

The unquestionable bottleneck of our process is the global memory because from the host the writing bandwidth is only ≈ 10 GB/s. Once our data is in the global memory the threads can reach it with a bandwidth ≈ 150 GB/s, which sound a lot, but on the device this is the lowest bandwidth. The "lifetime" of the global memory is the whole computation, and the size is around some GB. While the host memory typically can be extended the device global memory is fixed. This is the reason why one needs multiple GPUs for huge computations (for example 3D lattice Boltzmann solver [43]).

As the arrows indicate the constant and texture memories can only be read by our device. The threads have a higher bandwidth to both of them compared with the global memory. The constant memory can be used to store our constant parameters during our calculations. The texture memory operates like a constant cache, and can prepare the neighbouring data in 2D for the threads. The "lifetime" of the texture and the constant memory is the whole computation. The storage capacity of the texture memory is some hundreds of MB while the constant memory size is ≈ 50 MB.

The programmers using CUDA have the freedom to decide how the memory is distributed. It is possible to draw up two rules of thumb as a consequence of the listed properties of the nVidia GPUs:

1. The data transfer between the host and the device should be minimized. Once data is on the device it should be kept there for the computations as long as possible.

2. Memories with higher bandwidth have to be used to obtain the maximum speed up.

Based on this short description of the GPU architecture we can see that there are some important properties of the hardware. For example the maximum number of threads per block (block size), the maximum number of blocks per grid (grid size), the size of the mentioned memories, etc. The author used two different machines for the computations (M1 and M2). The computers were basically different (different processors, memories, graphical cards...) The most important features and properties of the computers and the hardware can be found in Appendix A. (One can list the host properties typing the command `cat /proc/cpuinfo` in Ubuntu terminal. To find the listed information about the GPU one has to run the deviceQuery executable file. The code of deviceQuery should be installed together with other CUDA samples which are ready to run after compilation.)

The author found the block size and the grid size parameters misleading. To prevent misunderstanding it is important to clarify the meaning of these parameters. The maximum number of threads per block is $1,024$ for the GPUs. It can be distributed in 2D for example as 1024×1, or 32×32. The grid size is totally similar: the blocks per grid limit is $65,535$. The grid and the

block size determine the maximum number of threads per GPU: $65535 \cdot 1024 = 67,107,840$. This is the available thread number, so that the maximum number of the lattices is the same in our numerical grid. (In the current study only one GPU was used to perform the calculations.)

2.5.3 Application of CUDA in practice

The goal of the current chapter is to examine CUDA in practice. In a former thesis [5] Scopus [83] was used to map the field of applications. The same strategy was followed. The Scopus search engine is able to list the publications related to the given keywords. The first keyword was "CUDA" then the results were filtered based on further keywords for disciplines. The results are summarized in Table 1. The sum of the portions is higher than 100% because of overlapping (for example fluid mechanics and solid mechanics are the subsets of physics).

Keywords	**Results**	**Portion [%]**
CUDA	4920	100
Computer science	3356	68.2
Image processing	1662	33.8
Physics	1378	28.0
Mathematics	1292	26.3
Biology	669	13.6
Electronics	572	11.6
Chemistry	325	6.6
Material science	321	6.5
Molecular dynamics	317	6.4
Fluid mechanics (CFD)	217 (219)	4.5
Astronomy	159	3.2
Solid mechanics	69	1.4

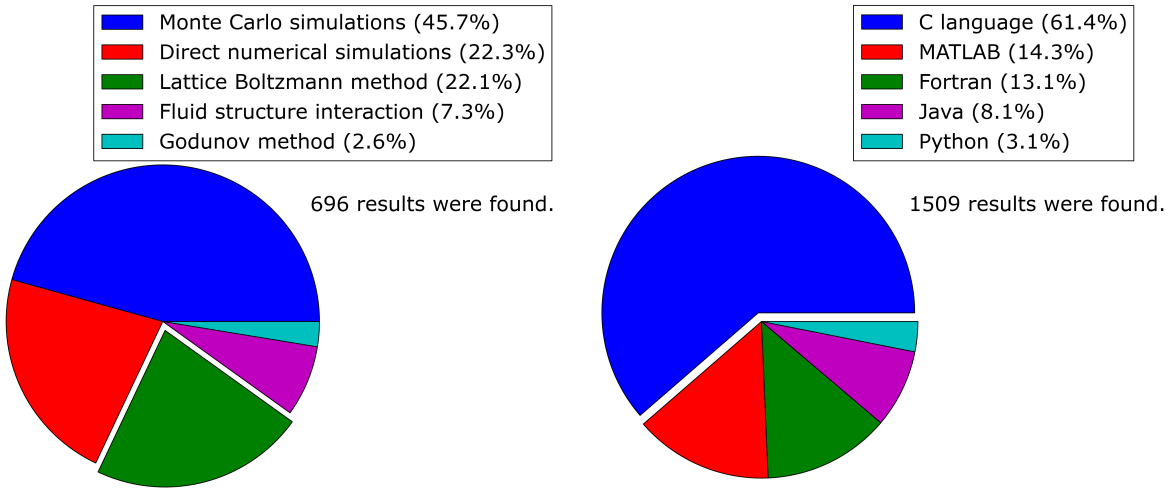
Table 1: Results of Scopus research engine for the listed keywords [83]

Computer science Since GPGPU and CUDA are relatively new tools in the scientists' hand, there are lots of articles investigating the basic behaviour of the graphical cards through benchmarks. These papers help the reader to learn efficient CUDA programming. They are related often to programming strategies [84], or practical memory treatment investigation [85] etc.

Image processing One can find within these results typical application examples of GPGPU. Medical images are built up from voxels. We can imagine these images as a 3D grid with pixel data at the grid points (responsible for colour). In cases of magnetic resonance imaging (MRI) and computed tomography (CT) images the pixel values are connected the measured contrast value, and the grid is taken from the spatial resolution of the machine. The image reconstruction was implemented and accelerated with CUDA [86, 87].

Physics These results involve a whole branch of applications including electronics, material science, molecular dynamics, CFD, astronomy, solid mechanics, etc. Some example without attempting to be comprehensive: Tanno et al. [88] described an efficient CUDA implementation of the "classical" artificial compressibility method. Mielikainen et al [89] accelerated the micro-physical part of an existing weather forecasting software. Joldes and Miller applied CUDA successfully to a real-time, nonlinear finite element solver [90], and a nonlinear material model was also modelled by Taylor et al. [91].

Mathematics As Gauss said: "The Queen of the Sciences". We cannot imagine any natural science without Her. Some papers in this category show the details of a CUDA implementation of different mathematical algorithms. The most cited paper of the category is about the Smith-Waterman algorithm [92]. The algorithm looks for local alignments in different sequences and it is used to search for similarities "in protein and DNA databases" [92].



(a) The distribution of the applied languages in CUDA programming of GPUs

(b) The distribution of different simulations in the field of computational fluid dynamics

Figure 4: Results of Scopus search engine [83]

As the current thesis aims to parallelize an existing CFD code, the author investigated the application of CUDA in the field CFD separately. Figure 4(a) indicates the results of the Scopus search engine. We can conclude that the new line of CFD, namely modelling based on the particle approach, is quite a popular area. It seems logical because the GPU is able to make computations in parallel on the individual particles. Probably this is the reason for the popularity of Monte-Carlo [93, 94] and lattice Boltzmann simulations. Shinn et al. parallelized the semi-implicit method for pressure linked equations (SIMPLE algorithm) and direct numerical

simulations were performed on GPU [95]. The LBM was also tested with direct numerical simulation (DNS) [96], whilst DNS results were compared to large eddy simulation (LES) [97] and coupled to particle modelling too [98].

CUDA is available not only in the C/C++ language. MATLAB, Fortran (CUDA Fortran), Python (PyCUDA), Java (JCUDA) also support CUDA. The results of the Scopus search engine for different languages is shown in Figure 4(b). As we can see, C is clearly the dominant one. MATLAB is in second place, while Fortran is third. CUDA Fortran was worked out by the Portland Group [79] because a lot of research institutions use this "ancient" language (since 1957) for development purposes. If we are talking about scientific computing, it has the speed of C but the efficient implementation itself takes less time in general. CUDA Fortran is special compared to the other listed languages: the kernel itself is not written in C. Other languages need a kernel function written in C (they save time for the user through the declaration).

As explained in this chapter, graphical cards have become popular HPC devices. GPGPU and CUDA proved its legitimacy in different fields of applications. Although the GPGPU approach has some basic hardware related limitations, it will probably play an important role in the future of supercomputing. In addition, anybody can use their own nVidia device for computing thanks to CUDA.

3 Methodology

3.1 Theoretical background of the lattice Boltzmann method

Since the goal of the current study is the parallelization of an existing lattice Boltzmann code, a short description of the lattice Boltzmann method is given below.

3.1.1 From kinetic theory to the macroscales

To understand the concept of the lattice Boltzmann method (LBM) one has to start at the level of particles. A typical starting point could be a box with volume V including N moving particle. Every particle has its individual velocity ($\vec{c}$) and mass m (it is important to distinguish the macroscale velocity $\vec{u}$ from the microscale velocities $\vec{c}$). The impulse of a particle can be written as $\vec{p} = m\vec{c}$. Let the position of the i^{th} particle be denoted by $\vec{x}_i$. For simplicity the particles have spherical shape. The motion of the particles can be predicted based on Newton's second axiom, which can be written now in the following form:

$$\frac{\mathrm{d}\vec{x}_i}{\mathrm{d}t} = \frac{\vec{p}_i}{m}, \tag{3.1}$$

$$\frac{\mathrm{d}\vec{p}_i}{\mathrm{d}t} = \vec{F}_i, \tag{3.2}$$

where $\vec{F}_i$ stands for the force acting on the i^{th} particle [3].

As previously mentioned, solving the system covered by Eq. (3.1) and (3.2) results in an unnecessarily detailed and expensive description. To make the conversion and start to look at the system from a statistical point of view, a distribution function $f(\vec{x},\vec{p},t)$, (sometimes referred as probability density) is introduced; f represents the probability of "finding a molecule around position $\vec{x}$ at time t with momentum $\vec{p}$ ". The Boltzmann equation was derived to follow the history of f and it can be written as

$$\left[\partial_t + \frac{\vec{p}}{m}\cdot\partial_{\vec{x}} + \vec{F}\cdot\partial_{\vec{p}}\right] f(\vec{x},\vec{p},t) = C_{1,2}. \tag{3.3}$$

The left hand side of the Boltzmann equation expresses the change of the density function in time caused by the streaming of the particles, while the right hand side is responsible for the changes caused by the interaction of the particles (collisions). $C_{1,2}$ is the two-body collision operator which can be formulated from the two-body distribution function $f_{1,2}$ (the probability of finding two molecules in state 1 and 2 at the same time). The problem of this approach is that one need the three-body distribution function $f_{1,2,3}$ for the two-body distribution function

and so on. The closure of the problem is the so called Stosszahlansatz: $f_{1,2} = f_1 \cdot f_2$. This assumption represents molecular chaos, so there is "no correlation between molecules entering a collision" [3]. Consequently the Boltzmann equation can be solved, and it is possible to follow the evolution of the distribution function.

The question may arise: why is it good to solve the Boltzmann equation, and how is it related to the macroscopic world? It is known that the Chapmen-Enskog procedure forms a clear bridge between the microscopic and the macroscopic description so the Navier-Stokes equations (describing large-scale fluid motions) can be derived from the Boltzmann equation. (The derivation is beyond the the scope of the current work.) The first connection can be found through the temperature T. The relation between the temperature and the mean-square of the microscale velocity $< c^2 >$ can be formed as

$$\int_0^\infty c^2 f(c) \mathrm{d}c =< c^2 >= \frac{T k_B}{m}. \tag{3.4}$$

In the above equation the role of the distribution function is also visible (k_B is the Boltzmann constant). In the case of LBM the temperature cannot be calculated so simply,because (3.4) is just a basic example to show how one can make a connection between the microscopic and macroscopic variables. However the usual variables like density and momentum per unit volume ($\rho\vec{u}$) can be obtained easily from the distribution function (Eq. (3.5) and (3.6)).

$$\rho = m \int f \mathrm{d}\vec{c} \tag{3.5}$$

$$\rho\vec{u} = m \int f \vec{c} \mathrm{d}\vec{c} \tag{3.6}$$

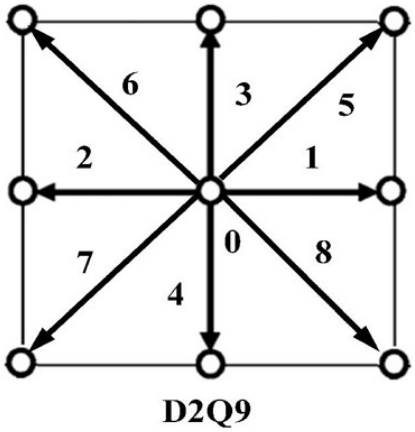
The pressure can be obtained from the lattice speed of sound $c_s = 1/\sqrt{3}$ as $p = c_s^2 \rho$.

It is important to note that the distribution function can be split into an equilibrium and a non-equilibrium part $f = f^{eq} + f^{ne}$. After a characteristic time, called relaxation time τ_r the system of moving particles reaches the equilibrium state when $f^{ne} = 0$. In the equilibrium state the distribution function shapes the Maxwell-Boltzmann distribution. Later on the distribution function will be used for the simulation of the collision operator, and the relaxation time will be a basic parameter of the scheme.

3.1.2 Lattice Boltzmann method for the discretized Boltzmann equation

The first attempt to discretize the Boltzmann equation was the lattice gas cellular automata in the late '80s. The basic idea was to resolve the space by a lattice and to model the streaming only along discrete directions. The method worked only based on logical operations so that it could

be implemented efficiently. Although it was capable of solving fluid flow problems, it proved a dead end, because of some basic limitations like statistical noise, exponential complexity of the collision operator etc. The lattice Boltzmann method rose from the ashes of the lattice gas cellular automata. In [99] McNamara and Zanetti first proposed first use of the lattice Boltzmann method to describe fluid flow. It relatively quickly became a robust algorithm for incompressible fluid flow problems but there ain't no such thing as a free lunch: the complexity increased and the new method needed more than logical operations.



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

$$e_1 = (1,0) \qquad e_2 = (-1,0)$$
$$e_3 = (0,1) \qquad e_4 = (0,-1)$$
$$e_5 = (1,1) \qquad e_6 = (-1,1)$$
$$e_7 = (-1,-1) \qquad e_8 = (1,-1)$$
$$e_0 = (0,0)$$

Figure 5: Lattice for the discretization [6]

To discretize spatially the Boltzmann equation, the LBM requires only a Cartesian grid (lattice). After the spatial discretization the streaming is allowed only at discrete points (The vertices and the midpoints of the sides determine the points; symbolically the particles can move only in finite directions). In this way the type of the lattice is related to the speed model which is clearly formed from the shape of the lattice and the number of the allowed directions. LBM is often categorized based on the applied speed model as DaQb, where a is the number of spatial dimensions and b is the number of the allowed directions. Figure 5 represents the D2Q9 speed model which was used for domain discretization in the earlier codes [6, 5]. In Figure 5 $\vec{e}_i = e_{0...8}$ denotes the discrete lattice velocity vectors and w_i is the weighting factor corresponding to the different directions (i here stands for the numbering of the directions). Because of the simple Cartesian grid representation, the geometry could not be resolved accurately: the error from the spatial resolution is $\delta/2$ where δ is the spacing of a uniform grid. In the thesis of Teschner ICEM CFD was used for meshing butthe advantages of the fast meshing algorithms were lost [5], and the LBM became less flexible. For exact and curved boundary treatment Mei et al. worked out stable meshing algorithms [100, 101].

After the spatial discretiation the velocity discrete Boltzmann equations can be written as

$$\frac{\partial f_i(\vec{x},t)}{\partial t} + \vec{e}_i \nabla f_i(\vec{x},t) = -\omega(f_i(\vec{x},t) - f_i^{eq}(\vec{x},t)). \tag{3.7}$$

In the above expression f_i is the distribution function related to the discretized directions. One can see how the collision operator is simplified, and simply connected to the directional distribution functions. $\omega = 1/\tau_r$ denotes collision frequency which can be written as a function of the relaxation time. This is the single-relaxation time scheme. In earlier studies two-, and multi-relaxation time schemes were also implemented. For details check [6, 5]. The Maxwell-Boltzmann distribution function leads to an equilibrium function

$$f_i^{eq} = w_i \rho \left(1 + \frac{\vec{e}_i \vec{u}}{c_s^2} + \frac{(\vec{e}_i \vec{u})^2}{2c_s^4} - \frac{\vec{u}^2}{2c_s^2}\right). \tag{3.8}$$

After all this, a time step can be performed in two steps:

1. Collision→interaction inside a lattice between the directions

$$\tilde{f}_i(\vec{x},t+\Delta t) = f_i(\vec{x},t) - \omega\left[f_i(\vec{x},t) - f_i^{eq}(\vec{x},t)\right]; \tag{3.9}$$

2. Streaming→the distribution functions "travel" to neighbouring lattices

$$f_i(\vec{x}+\vec{e}_i\Delta t, t+\Delta t) = \tilde{f}_i(\vec{x},t+\Delta t). \tag{3.10}$$

This results in a second order accurate numerical scheme both in space and time for incompressible flows.

3.1.3 Boundary conditions for the LBM

The simplest boundary condition is the periodic boundary condition: a quantity leaving the domain on the right side is described as an inlet on the left side and vice versa. It can be implemented easily both at the micro-, and macro-scale.

The so called bounce back boundary condition can be used for no slip wall treatment in LBM [2, 3]. Since the macroscale variables are computed from the distribution functions, the distribution functions have to be well-behaved near the wall so that they give zero macroscopic velocity. During the boundary refreshment the molecules virtually "bounce back" from the wall and change direction. The procedure includes the interchange of the opposite distribution functions. If the 0^{th} direction of a lattice (middle point of the lattice from Figure 5) is on the

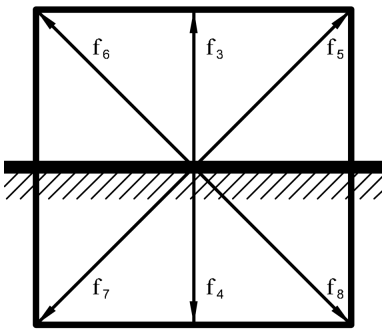


Figure 6: Bounce back on the wall

wall then the situation looks like in Figure 6 and the boundary condition can be written as

$$\begin{aligned} f_3 &= f_4; \\ f_5 &= f_7; \\ f_6 &= f_8. \end{aligned} \tag{3.11}$$

It is logical not to modify the distribution functions parallel to the wall (one should not forget that small particles have a non-zero velocity arbitrarily close to the wall).

The method, reported by Zou and He [102], is quite popular for treatment of the inlet and outlet boundary conditions (BCs). In Figure 7 i stands for the vertical lattice number, and the inlet BC is located at $i = 1$ while the outlet BC is placed at $i = n$. The j^{th} directional distribution function of the n^{th} lattice can be denoted as f_j^n. In Figure 7 the distribution functions corresponding to the dashed lines are unknown. Based on the sketched notations and Figure 7 the formulae for the inlet boundary condition look like

$$\begin{aligned} f_1 &= f_2 + \tfrac{2}{3}u; \\ f_5 &= f_7 + \tfrac{1}{2}(f_4 - f_3) + \tfrac{1}{2}\rho v + \tfrac{1}{6}\rho u; \\ f_8 &= f_6 + \tfrac{1}{2}(f_3 - f_4) - \tfrac{1}{2}\rho v + \tfrac{1}{6}\rho u. \end{aligned} \tag{3.12}$$

In the above expressions u and v are the two components of the $\vec{u}$ macroscopic velocity vector. A first order BC can be formulated easily at the outlet: the unknown values of the lattice $i = n$ have to set the values of the lattice $i = n - 1$. A second order formulation is described in the literature [2].

The question may arise what we can prescribe at the corners. The boundary conditions determine the results thus it is important to be consistent. A lack of the exact corner treatment description was reported [5] earlier. However it was experienced that the wall-wall prescription did not cause distortions, non-physical noises in the flow field [6]. The question is critical

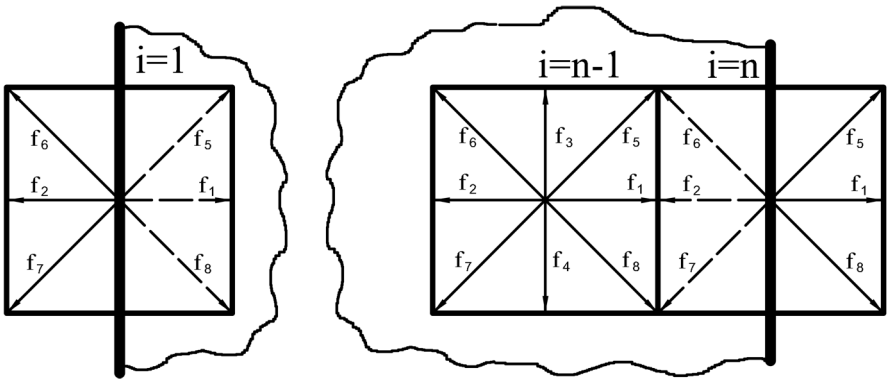


Figure 7: Inlet (left) and outlet (right) boundary treatment

when for example inlet and wall boundaries form the corner. The author followed the wall-wall approach.

As the reader has become familiar with the lattice Boltzmann method now we can have a closer look at the existing codes.

3.2 The in-house lattice Boltzmann codes

Both of the given codes were written in C++ [6, 5]. The codes are referred to from now on as Code1 [6] and Code2 [5]. The 2D geometry has to be created in Pointwise. A structured meshing algorithm, written in C++, corresponds to the solver creating a suitable lattice for the LBM. The author found the meshing algorithm parallelizable but this is beyond the scope of the current thesis. The generated grid is equidistant and has a rectangular shape. It often leads to uncertain geometrical resolution. The solver and the mesher are totally separate codes, but it would be possible to couple them and make a code that includes remeshing (often necessary for fluid-structure interaction). This approach has the advantage that fluid-solid cells are distinguished by simple 0/1 logical decisions. This made the LBM suitable for modelling clotting in blood flow [103]. Although the automatic mesher has several limitations yet, further development is possible. The optional automatic structured mesh generation is one of the biggest advantages of the lattice Boltzmann method. (Mesh generation takes sometimes months, especially if we simulate the flow in complex geometries.)

Regarding the implemented LBM in Code1, the code is not user friendly. The parameters are hidden inside the code because the developer did not use an *.ini file. First, the code reads the two output files of the meshing algorithm. Second , an initialization is performed. The user can initialize a parabolic velocity profile or a uniform one. These are the implemented boundary conditions too. The initialization is followed by the main loop which refreshes the

distribution functions in the cells of the fluid domain. Three different collision models were implemented. A time dependent boundary condition was not implemented. The code has some further limitations because of the generated mesh. The inlet and the outlet surfaces have to be parallel to the x or y axis. Curved wall boundary treatment was worked out based on a polygonal representation of the curve. A second order boundary condition, or parabolic velocity profile is available at the outlet boundary.

Code2 [5] was written to be able to use a mesh generated in ICEM CFD. To this end a CGNS reader is the first step of the code. Although in this way the user loses the comfortable automatic mesh generation, this approach offers a more accurate geometrical treatment and more freedom to choose the domain shape. From a physical point of view Code2 has the same features. All in all Code2 is more user-friendly: a menu is given in the terminal and the user can choose the settings and define the boundary conditions. Unfortunately the menu needs direct communication with the user which makes it difficult to run the code in a cluster environment. The code is highly object-oriented which has to be mentioned as a disadvantage since C is not object-oriented. Both of the existing codes were validated with basic measurement data and the Poiseuille flow (analytical solution).

3.3 CUDA - programming of nVidia's graphical cards

The goal of this subsection is to demonstrate the basic features of CUDA and GPU programming. The applied commands of the SDK will be described and analysed. One has to install the CUDA toolkit to be able to program the GPU. Furthermore, installation of the up-to-date driver of the nVidia graphical card is essential.

As previously mentioned the user can instruct the GPU through kernel calls. The kernel function is similar to a simple C function: the user has to list the variables in the function argument to be able to use them within the function. The difference is that we do not have the variables originally on the GPU so it is necessary to copy them from the host to the device. After the calculations are finished on the GPU we need to copy back the data from the device to the host. As we saw in Chapter 2.5.2, the GPU has a special structure including threads and blocks. The control of these units forms the tricky part of CUDA programming. Since it is part of the basic philosophy of the CUDA SDK to make a user friendly GPGPU platform, one can expect relatively simple, additional functions which make it possible to control the device.

The main steps of a simple CUDA code are:

1; Loading preprocessors: this step is part of every C program. For example loading the standard input/output header (`#include <stdio.h>`) is necessary if we want to use for

example the `printf()` function to write something to the terminal. One has to load the CUDA tools with the command `#include <cuda.h>`. In this part of the code programmers often define some constant with the `define` command. Similarly, this is the suitable part in which we can allocate constant memory locations of the GPU. The corresponding command is `__constant__ <arg1> <arg2>[<arg3>]`. (Here "arg." stands for argument). "Arg1" is the type of the variable (for example float or integer) while arg2 is the name of the variable. The third argument is a positive integer and it determines the allocated memory size (number of memory places). It is important to note that the author did not find a way to allocate the constant memory dynamically.

2; Defining the kernel function: as mentioned earlier, the kernel function is a function executed on the GPU. A usual kernel function definition looks like `__global__ void FUNCTION(int *dev_in, int *dev_out)`. FUNCTION is the name of the function and the listed variables between brackets are the input and output pointers on the GPU. (Of course there can be more than one input and output.) Afterwards we have to define a suitable indexing. On the GPU every thread has its individual index within a block, and a block itself also has an index (Figure 2). The vector indexing starts from zero and the index increases by one after every element. Since the GPU indexing is not identical with this, it is necessary to "project" the GPU indexing to the vector indexing. There are some inbuilt variables in CUDA to this end. The variables `blockIdx.x`, `blockIdx.y` and `blockIdx.z` store the x, y and z-directional indices of the current block, while `threadIdx.x`, `threadIdx.y` and `threadIdx.z` store the directional indices of the threads. The directional block and grid dimensions are stored in the variables `blockDim.x`, `blockDim.y`, `blockDim.z` and `gridDim.x`, `gridDim.y`, `gridDim.z` respectively. As shown in Figure 2, there is a (1,0) thread in every block; consequently we need both the thread indices and the block indices to identify one single thread uniquely. It is possible to form a suitable index (`ind`) from the listed inbuilt variables. Finally, the line `dev_out[ind]=1;` means that every element of the `dev_out` vector is equal to one. This command was executed in parallel.

3; The main function: the main function is part of every C code. First of all it includes the declaration and the allocation of the variables. Although the GPU variables can be declared as normal C variables (`int *dev_in;`), the allocation is different, since we allocate these variables in the GPU memory. (Variables allocated within the kernel functions are "alive" only while the kernel function is running.) We use `cudaMalloc` to allocate on GPU. The structure of the command is `cudaMalloc((void**)&<arg1>,<arg2>)`,

where the first argument stands for the name of the variable, the second for the data size of the vector. The data size in C can be calculated with the `sizeof()` command. The declaration and allocation is followed typically by some host command (for example initialization). Afterwards, we have to copy the data to the GPU and determine the block and the grid size before the kernel function call. The command `cudaMemcpy` is responsible for data copying between the host memory and the GPU global memory. The line `cudaMemcpy(<arg1>,<arg2>,<arg3>,<arg4>);` copies data from the second argument to the first argument with the data size of the third argument. (The first and second arguments are variables and the third is a positive integer). The fourth argument is responsible for showing the direction of the data copying, namely `cudaMemcpyHostToDevice` or `cudaMemcpy DeviceToHost`. One has to use a different command, `cudaMemcpyToSymbol`, to copy data to the constant memory. It would be pointless to copy data from the constant memory to the host so the fourth argument is missing in this case compared to `cudaMemcpy`.

When we have the data on the GPU, it is time to make the kernel function call in the following form: `FUNCTION<<<GS,BS>>>(dev_in,dev_out);` (This kernel call corresponds to the above kernel function definition.) The two new variables `GS` and `BS` are the grid size and the block size. These are 3D vectors, and they can be created with the command `dim3 BS(100, 10, 1)`. The latter instruction sets the grid size to be 1000: 100 threads in the x-direction, and 10 in the y-direction. Thus the value of `blockDim.x` is 100. After a successful kernel call the programmer has to copy the data back to the host, where further calculations can be done. It is important to note that every kernel function call is a synchronisation point. Originally the kernel functions are executed concurrently to the host executions but `cudaMemcpy` breaks the asynchronous behaviour. At the end it is important to free the allocated memory on the GPU (and on the host too). This can be done by `cudaFree(<arg>)`, where the argument is the name of a variable.

4 Results and discussion

4.1 Analyses of the sample code

The author prepared a simple benchmark to test the basic features of CUDA. The program creates a matrix, adds the four neighbouring cells and multiplies the sum with a constant. (This step is typical for example in 2D, steady state, heat conduction problems.) The parallel code is attached and can be found in Appendix B. The blue words are original C commands, the green ones are CUDA commands, and comments are red.

The *first line* of the code includes the command for the compilation and execution (Linux terminal). It was mentioned earlier that the first step is to include the necessary header files of the preprocessors (between *line 2 and 6*). *Lines 8–9* define the width and the height of the matrix. Between *line 12 and 17* one can see definitions related to the GPU. These are the dimensions of the grid and the blocks, and they are used in the main function. In addition, a parameter (`c_d`) is declared and allocated in the constant memory of the device (*line 20*). The grid dimensions are 32 threads in the x-direction (`tx`) and 32 threads in the y-direction (`ty`) which means 1024 threads within every block. Similarly, 7 blocks are in the x-direction (`bx`), 5 in the y-direction (`by`), and 2 in the z-direction (`bz`). All in all it means 70 blocks, so the total number of threads is 71680.

The following part of the code is the kernel function (between *line 25 and 50*). A suitable indexing can be seen based on the inbuilt CUDA variables (*line 35*). The same indexing was used during the parallelization. It makes it possible to use 2D blocks and 3D grid.

The first `if` statement (*line 39*) restricts execution to the elements of the vector. It is necessary because the number of threads is often greater than the number of the elements of the vector. In the current example we have 71680 working threads but the matrix has only `W`×`H`= 62500 elements. The difference is 9180. The non-working threads within a warp execute the same operations as their working partners.

The second `if` statement (*line 43*) limits the calculations to the inner nodes (no calculations on the boundaries). Since we have a matrix in vector form the boundary indexing is not trivial. The C language processes the matrices line by line. Consequently the boundaries are excluded by the following logical conditions

- `ind>W-1` prevents calculations on the northern boundary;
- `(ind%W)!=0` prevents calculations on the western boundary;
- `ind<(W*(H-1))` prevents calculations on the southern boundary;

- `(ind+1)%W!=0` prevents calculations on the eastern boundary.

After the two `if` statements, *line 45* includes the parallel execution of summation and multiplication. Thanks to the vector representation of the indexing of the four neighbouring cells it is also tricky:

- `[ind-W]` is the northern neighbour;
- `[ind-1]` is the western neighbour;
- `[ind+W]` is the southern neighbour;
- `[ind+1]` is the eastern neighbour.

The author notes that `c_d` is used from the constant memory as a pointer. The reader may feel the analysis of the kernel function is unnecessary. The author describes the details because similar indexing was used during the parallelization of the lattice Boltzmann code and an appropriate indexing is a key for CUDA parallelization.

The main function is placed after the kernel function (*lines 52-192*). The declaration can be seen between *lines 56 and 80*. It is important to note that every GPU variable has its twin on the host and both of them have to be pointers. In this part of the declaration one may find some new CUDA command such as `cudaEvent_t`, and `cudaEventCreate` which are used for time measurement. (Classical C time measurement is not suitable for measuring GPU execution time.) The CUDA time measurement needs two more functions `cudaEventRecord` and `cudaEventElapsedTime`. As usual the declaration is followed by the allocation (*lines 81-98*). Two-dimensional and one-dimensional arrays are allocated on the host, and only one-dimensional arrays on the GPU.

One of the declared matrices is initialized (*line 123*). To this end two `for` loops are used. As we can see the initialization time was measured based on the clock rate of the CPU (*lines 101 and 111-112*). The matrix was initialized with the indices of the elements. The indexing starts from zero from the upper left corner of the matrix, and it increases by one.

Lines 114 and 115 include the grid and block size definition of the GPU. *Lines 119–126* contain a matrix-vector conversion. This is necessary because the GPU stores the data in vector form but the data was declared in 2D matrix form on the host. The author wrote the code like this because the reformulated serial lattice Boltzmann C code is based on 2D matrix formulation (similarly to the original C++ code). The conversion step had a great role during the parallelization. After the conversion it is time to copy the data to the GPU (*lines 130-135*) and make the kernel function call (*line 139*). Before the kernel function call three different time

measurements are started (*lines 117, 127, 136*). Based on the measured data the times of kernel execution, data copying, and matrix-vector conversion were evaluated.

After the kernel finished the computations, the data can be copied back to the host (*line 146*). Another conversion is necessary to return to matrix form (*lines 152–158*). The measured data are written to the terminal (*lines 164–168*). (Time is classically measured in seconds, but it is reported in milliseconds by CUDA. The written data are in milliseconds.) After saving the data (*lines 171-181*) it is a good idea to free the memory of the host and the device arrays (although the compiler will probably perform this task). The allocated host and device memories are freed in *lines 182-191*. Table 2 summarizes the time measurements for the serial and parallel codes. On the GPU several simulations were performed with different settings. In every case the blocks contained 1024 threads. The further abbreviations in Table 2 are for the following grid-block approaches:

GPU I. one-dimensional block and one-dimensional grid. The grid size varies as a function of the vector size (dynamic grid size). Grid size $\approx$ (number of elements / 1024) + 1.

GPU II. one-dimensional block and three-dimensional grid. dynamic grid size with constant y- and z-dimensions.

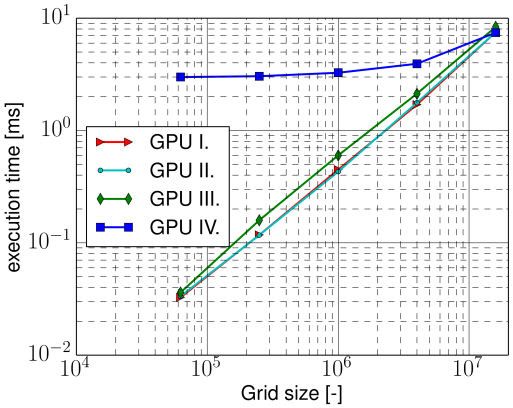
GPU III. two-dimensional block and three-dimensional grid. Block size: 32×32 threads; dynamic grid size with constant y- and z-dimensions.

GPU IV. one-dimensional block and one-dimensional grid. Non-dynamic grid size (The grid size was fitted to the maximum number of elements. The maximum number of elements was 8000×2000. Consequently the grid contained $8000 \times 2000/1024 = 15,625$ blocks.)

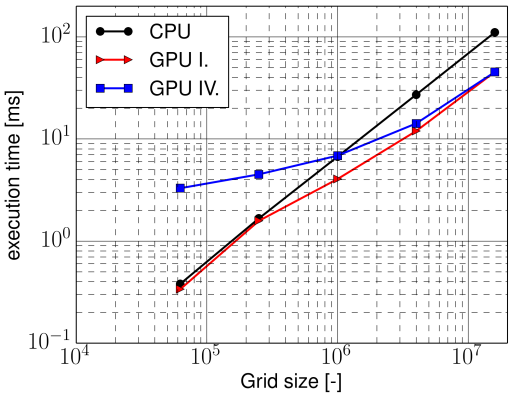
Table 2 shows in addition the time spent copying data and carrying out matrix-vector and vector-matrix conversions.

–	Mesh 1	Mesh2	Mesh3	Mesh4	Mesh5
Measured part	500×125	1000×250	2000×500	4000×1000	8000×2000
CPU main task [ms]	0.38	1.66	6.71	27.19	110.94
GPU I. main task [ms]	0.03	0.12	0.45	1.70	7.52
GPU II. main task [ms]	0.03	0.12	0.43	1.77	7.51
GPU III. main task [ms]	0.04	0.16	0.59	2.13	8.37
GPU IV. main task [ms]	2.99	3.04	3.27	3.93	7.44
GPU data copying [ms]	0.30	1.46	3.61	10.27	37.87
GPU transformation [ms]	0.37	1.61	6.09	23.23	94.61

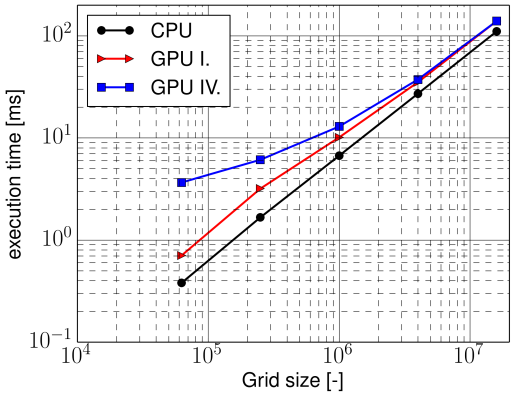
Table 2: Time measurement data based on the sample codes



(a) Comparison of the different thread and block size treatment: only the main task execution



(b) Comparison of CPU and GPU execution time: data copying included



(c) Comparison of CPU and GPU execution time: data copying and matrix-vector transformation included

Figure 8: Time measurement data of the sample codes

Figure 8(a) displays the results of the different grid-block size treatments. We see that the "GPU IV." approach proved inefficient. This seems logical, since in this case an unnecessarily high grid was used. Hence the non-working threads prevent the efficient operation of the working threads. We can draw two more conclusions from Figure 8(a): i) "GPU I." and "GPU II." show very similar characteristics so there is no significant difference between one-dimensional and two-dimensional grid spacings; ii) there is a noticeable difference between the "GPU I." and "GPU III." Curves which means that one-dimensional and two-dimensional block spacings are not equivalent. The one-dimensional block spacing (GPU I.) proved approximately 10% faster in every case. Thus if the data array is one-dimensional it is suggested that the one-dimensional block structure is kept but the dimension of the grid is not so important.

The CPU and GPU execution times are compared in Figure 8(b). Although the GPU finished

the main task more than ten times faster (with dynamic grid sizing) compared to the CPU (Table 2) the data copying takes a lot of time. Figure 8(b) shows that the GPU is only slightly faster than the CPU if we take into account the data copying and the advantages of the GPU can be used only if we work with a large enough datasets. The graphs clearly highlight that the GPU is not efficient for simple tasks, and the data copying is dangerously expensive.

Figure 8(c) was prepared to show the "price" of matrix-vector transformation. It has become clear that during the code reformulation this kind of transformation should be avoided. The GPU based solver was in every case slower because of the conversions. During the development the conversions were used because in this way the author could modify a part of the code without influencing other parts.

4.2 Reformulation of the code

The goal of this section is to present the reformulation procedure, which was performed with Máte Szőke [104]. Since C is a lower level language than C++, we expected a higher speed from the reformulated code. The other reason for the reformulation was portability because C++ codes are less portable if varying platforms are considered. The basis of the new serial code was the two existing 2D lattice Boltzmann solvers [6, 5] (see Chapter 3.2). Finally we chose Code1 as the core of the new code.

The main difference between C and C++ is that C++ is an object-oriented language while C is not. In the original Code1 an object was used which included every property related to the individual cells (for example x and y coordinates, u and v velocities, density, etc.) We tried to replace the objects with a structure called `Cells`. In this way we could write `Cells[j][i].U` which substitutes the x-directional velocity of the cell determined by the `i` and `j` indices. This structure included 21 different properties.

The flowchart of the new code is shown in Figure 9. The parallel code (presented later on) has the same structure. The "main.c" function first calls the "ReadIniData" function to read the basic settings from the "SetUpData.ini" file. Then the "Iteration" function is called which manages the computations.

The "Iteration" function first call two other function to read the generated mesh files ("ReadNodes" and "ReadBCconn"). Second, the "CompDataNode" and the "CompDataConn" function compute the basic parameters of the mesh (for example the grid spacing `Delta`). These functions are in the "CellFunctions.c" file.

Third, variables are declared and allocated. Fourth, the lattice quantities are initialized by the "D2Q9Vars" function (if the MRT model is used then additional constants are set with "MRTInitializer"). Fifth, "CellIni" initializes the individual cells based on the *.ini file (for

example, a parabolic profile or uniform velocity distribution is set here). "CellFunctions.c" includes these functions. The "WriteResults" function saves the data ("FilesWriting.c" file). Afterwards a `while` loop starts the iteration process itself.

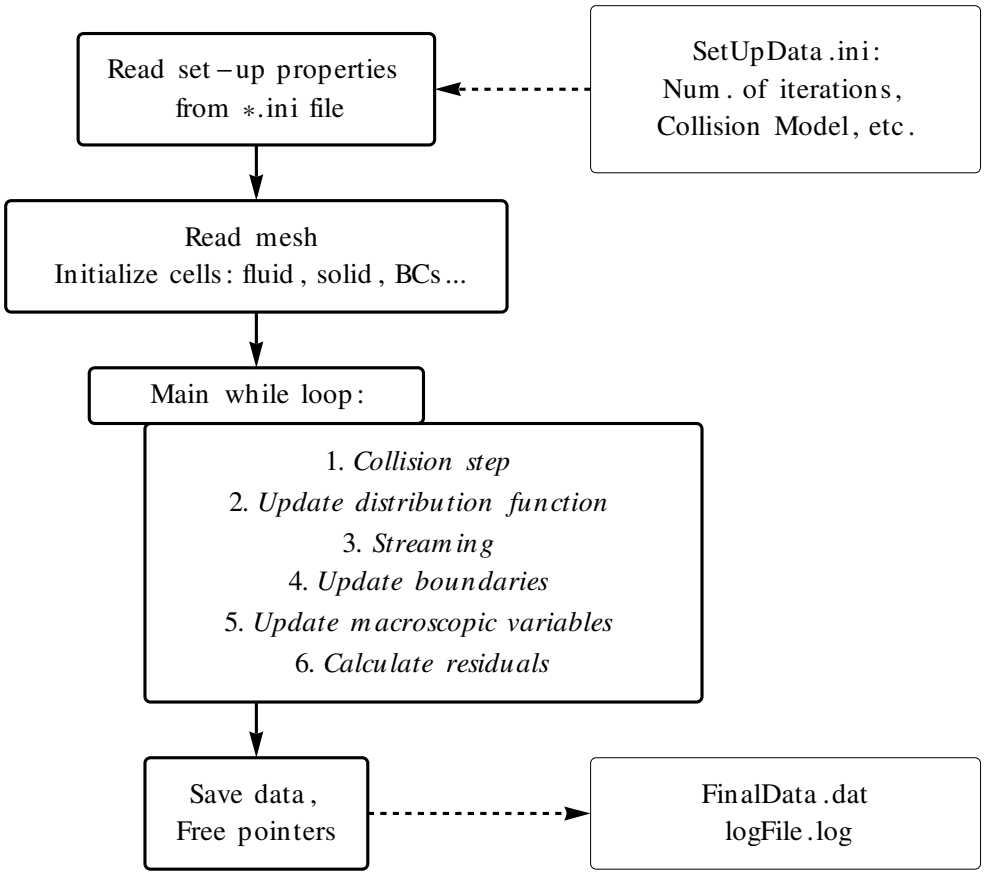


Figure 9: Flowchart of the rewritten codes [104]

The lattice Boltzmann method is built up from the following steps:

1. collision based on the collision model ("BGKW", "TRT", or "MRT" function);
2. update distribution function ("UpdateF");
3. streaming;
4. boundary treatment ("InletBC", "CurvedWallBoundaries", "WallBC", "OutletBoundaries" function from the "BoundaryConditions.c file");
5. update macroscopic variables ("UpdateMacroscopic").

The residuals are computed at the end of every time step ("ComputeResiduals" from the ComputeResiduals.c file). The data is saved based on the set frequency (*.ini file).

The following lines present the streaming step of the original C++ and the rewritten C code. The first `for` loops sweep through every element of the domain. The first `if` gate decides whether the current lattice is part of the fluid domain. The third `for` loop is responsible for the nine discrete directions whilst the second `if` gate allows streaming only from the right locations (for example no streaming from the corners, etc.) The `get` command in C++ "call" named variables from objects (the object was the `Cells` object. The same was done with the `Cells` structure in C as shown below.

- The original C++ code

```
1 for (i=0; i<n;i++){
2   for (j=0; j<m;j++){
3     if (Cells[j][i].getFluid()==true){
4       for(k=0;k<9;k++){
5         if (Cells[j][i].getStreamLattice()[k]==true){
6           Cells[j][i].setF(Cells[j-INI.getcy()[k]][i-INI.getcx()[k]].
    getMETAF()[k],k);
7 }  }  }  }  }
```

- The rewritten C code

```
1 for (i=0; i<*m; i++){
2   for (j=0; j<*n; j++){
3     if (Cells[j][i].Fluid == 1){
4       for(k=0; k<9; k++){
5         if ( (Cells[j][i].StreamLattice[k]) == 1){
6           Cells[j][i].F[k] = Cells [j-cx[k]] [i-cy[k]].METAF[k];
7 }  }  }  }  }
```

4.3 Main steps forward to a fully parallel code

It was essential to identify the most time-consuming part of the serial code before the parallelization. Profiling was carried out based on time measurement data. Since CUDA GPU execution is way faster with single precision, a serial code was prepared with single precision. The author chose the flow around the cylinder test case to investigate the execution time of the different parts. The details of the settings will be presented later on (Chapter 4.6.4). For the time measurement $1,000$ iterations were run in every case. The spatial resolution of the mesh was $602 \times 123 = 74,046$ lattices. The development was carried out mainly on machine M1. The time measurement data presented in this section also correspond to this machine.

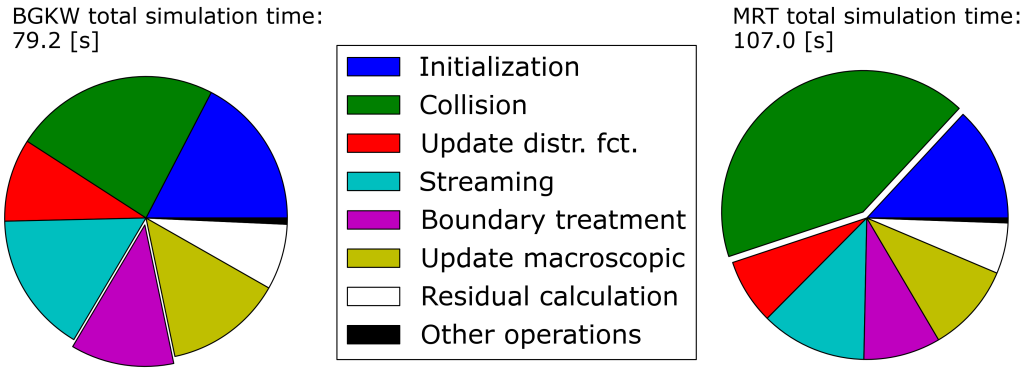


Figure 10: Results of the serial code profiling (the abbreviation distr. fct. stands for distribution function)

Figure 10 presents the results of the profiling (BGKW and MRT models). It is clear that the initialization was relatively expensive (approximately 15% in the case of 1,000 iterations). Fortunately the streaming (most expensive operation) was related only to the individual cells; consequently it was relatively easy to parallelize this step. As the total simulation times indicate, the MRT model is more resource-intensive. The only difference between the computations of the different models is in the collision step. The streaming and the boundary treatment are critical parts from a coding point of view because these parts are related to the neighbouring cells. The boundary treatment is quite expensive; however, only a few cells are on the boundaries typically. A global search is performed in the code to find the boundaries which is not very efficient. This is the reason for the high cost of the boundary treatment. The other operations include for example saving the data, declaration and allocation, etc. The author observed again that a rule of thumb of GPU programming is to keep the data on the device as long as possible. Based on this rule the author decided to try to execute every step on the device. The best solution would be if data copying were performed only when data saving happens (writing data to a file is possible only on the host). As we will see this is not an easy task.

The CUDA parallelization was started with the vector-matrix conversion presented earlier in Chapter 4.1. The steps of the lattice Boltzmann method were parallelized one by one. After every successfully rewritten part the data were copied back to the host, vector-matrix conversion was carried out and the author checked the results. Naturally these codes were very inefficient (because of the continuous data copying and matrix-vector/vector-matrix conversions) but every main operation was executed on the GPU except the residual computation. The author notes that several functions were split into two or three parts to perform the necessary synchronization before the next operation. The streaming had the form

```
1 if (ind<((*width_d)*(*height_d))){
```

```
2   if (Cells_d[ind].Fluid==1){
3     for(k=0; k<9; k++){
4       if ((Cells_d[ind].StreamLattice[k]) == 1){
5         Cells_d[ind].F[k] = Cells_d[ind+c_d[k]].METAF[k];
6 }   }   }   }
```

Since the number of threads in general is higher than the number of cells, the first `if` closes out the unnecessary threads from the calculations. The following `if` gates and `for` loop have the same roles as earlier. It is clear that the streaming happens here in nine separate steps, because of the `for` loop. Furthermore the indices of the neighbours have to be worked out based on the vector formulation. The necessary shifting constants for the nine directions were included in the `c_d` vector. The `c_d` vector does not change during the calculations so the author put this variable into the constant memory. (Similarly every constant variable is stored in the constant memory.) This formulation allowed parallel computations in every cell at the same time (theoretically) but the streaming in the nine directions was sequential. In this way the GPU limited the maximum number of cells based on the maximum number of threads, calculated as $(\text{maximum block size}) \times (\text{maximum grid size}) = 1024 \times 65535 = 67,107,840$. (The author did not check the limitation coming from the memory size of the GPU.)

It is worth mentioning here the reason why residuals are computed on the host. The residual calculation typically requires summation of the elements, or searching for the maximum of the elements of the solution vector. One may think that these algorithms are typical serial codes but efficient algorithms were implemented in parallel. These algorithms include some tricky GPU computations and several articles have been written about the most efficient technique [105, 106]. The summation is often referred to as "prefix sum" or "scan' and a speed-up around five can be expected from CUDA implementation of the efficient version. The author focused mainly on the other parts of the code but a well parallelized scan algorithm would be important to use the power of the GPU. Without this, one needs to copy the data back to the host at the end of every iteration which is a strong barrier. Implementation of a simple scan algorithm on the GPU was tried but the execution was 50 times slower compared to the code executed only on the CPU. The simple summation based on a `for` loop is very inefficient since in this way the user cannot use the parallel structure of the GPU. In addition the communication between the blocks is high.

To avoid matrix-vector conversions within the main `while` loop the residual computation and the data saving were rewritten in vector form. In this way there was only one conversion after the initialization. After this step the author obtained speed-up first time since the beginning of the parallelization. The main `while` loop proved faster by $\approx 120\%$ (the execution time was less than the half of the original). The total speed-up of the solver was only $\approx 80\%$.

To get rid of the matrix-vector conversion the author reformulated the initialization. The former and the latter formulations were as follows:

The former formulation

```
1 Cells[A][B].ID = i;
```

The new formulation

```
1 (Cells_h+ind)->ID = i;
```

This part can be found in the "CellIni" function, and it simply gives an identification number `i` to the elements. As shown above, the 2D indexing (`[A][B]`) was changed and a simple 1D index (`ind` in terms of `A` and `B`) was used. The newer formulation had another advantage. Besides writing `Cells[ind].ID = i`, the author changed the formulation. In this way, the new code shifted only the memory locations to assign values to the variables. It should be essentially faster since the command itself is "closer" to the hardware structure. (Originally the compiler "rewrites" `Cells[ind].ID = i;` to the form `(Cells_h+ind)->ID = i;` at every step which is not for free.) Although the author expected some performance gain, the initialization slowed down. It was 70% slower.

It is important to note again that the rewritten C code used the matrix formulation everywhere, similarly to the C++ code. Until the last changes the data from the mesh files were read to a matrix. A performance reduction was experienced because the elements of a vector were made equal to the elements of a matrix. Besides reading the grid files to a matrix, the code was modified again to read the data to simply vectors. Finally the code did not include any matrix-vector/vector-matrix conversion. The serial initialization was almost 20% faster thanks to the vector representation and the memory shifting.

The next step was to execute the initialization in parallel. Initialization is not part of the main loop, and the execution time converges to the main loop execution time in the case of a large enough number of iterations. However, the author found that the initialization needs a lot of time, especially if the grid has more than one million cells, in which case initialization may take an hour or more. The operations in this part are related only to the individual cells so the author expected an efficient kernel function. After the initialization was executed in parallel, the data had to be copied to the GPU at the beginning of the code. Afterwards data had to be copied back to the host only to make residual computations and data saving. The initialization proved 17 times faster with the sample code.

The author found the state of the solver suitable for a more detailed investigation. The effect of the thread number was examined. Several simulations were carried out with different block sizes. The results are shown in Figure 11. The graph shows the normalized execution time of the different parts as a function of the threads/block number. (Lower values in the graph

mean faster execution.) The results were contrary to expectation. (The same simulations were performed with machine M2 but the curves in the graph had similar characteristics.) Based on the released nVidia occupancy calculator the optimal number of threads per block is 256. As we can see in Figure 11, the efficiency of the code decreased as the block size increased, and the code was the slowest at 256 (main loop execution). The graph shows that the most efficient execution was when the block size was 16. This limits the number of cells drastically with the maximum number of cells for efficient execution being $= 16 \times 65,535 = 1,048,560$. Although this code was 4 times faster when 16 threads/blocks were prescribed, we could not allow this restriction. The key to the mystery was in the inefficient use of the GPU.

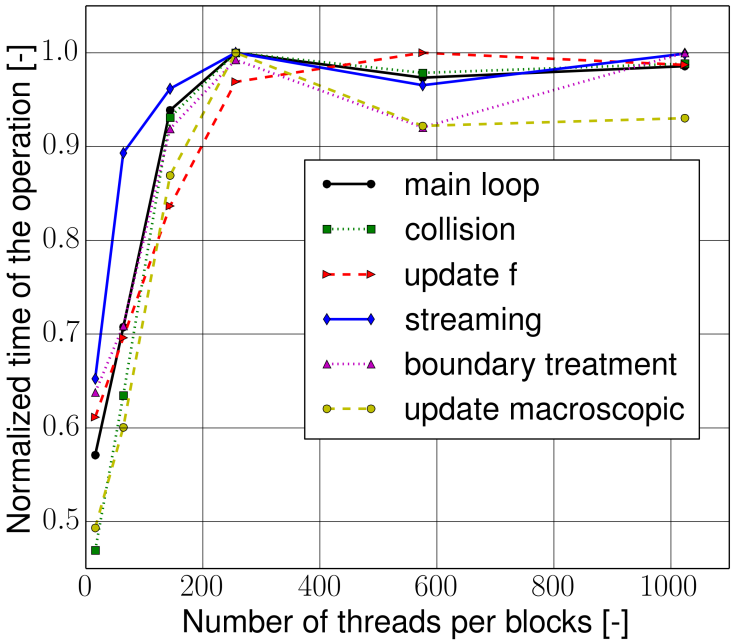


Figure 11: Effect of the block size I. (not the final version of the code)

As mentioned above, the kernels executed e. g. the streaming in nine steps within a `for` loop. This is a waste of resources. The execution could be done totally in parallel but it requires a completely new structure where the nine layers of the distribution functions are in one long vector. This structural change had a clear disadvantage: whilst the earlier version could work (inefficiently) with $67,107,840$ lattices, the new version will be able to handle only $67,107,840/9 \rightarrow 7,456,426$ cells.

4.4 The parallel code

Before the data structure was changed, it was decided to split the `Cells` structure into smaller structures. The idea was that for example the x and y coordinates of the lattices are

unchanged during the calculations; consequently it is enough to copy these variables only once to the GPU. Following this idea the `Cells` struct was split into four parts: the constant 1D quantities (e. g. x coordinate of the cells), the variable 1D quantities (e. g. u as the x-directional velocity), the constant 9D quantities (e. g. the boundary condition identifier of the nine directions), the variable 9D quantities (e. g. f_i as the ith distribution function). Thanks to the splitting the quantity of the copied data decreased by $\approx 85\%$ (only the variable 1D quantities have to be copied for the residual computations and the data saving).

Because of the simple vectorial representation of the nine directions, new indexing had to be derived. The elements of the distribution function vector went from 0 to $9 \times$ (number of cells) $- 1$. The other indices had to be "projected" from this index. As shown earlier, the 1D grid and block are more efficient in the case of a vector formulation (4.1). The index of the 9D vectors based on the inbuilt CUDA variables had the following form:

```
int bidx=blockIdx.x;
int tidx=threadIdx.x;
int ind = tidx + bidx*blockDim.x;
```

Using this variable one can express the index of the simple 1D quantities as

```
int ind_s = ind - ((*width_d)*(*height_d))*((int)(ind/((*width_d)*(*
    height_d))));
```

where the `width_d` and `height_d` pointers point at the locations, which store the vertical and horizontal size of the 2D rectangular matrix, representing the fluid zone. This index was used for the simple 1D quantities. Finally one can project the identification number of the discrete directions (from zero to eight) as

```
int ind_c = (int)(ind/((*width_d)*(*height_d)));
```

Based on these indices it was possible, for example, to execute the streaming in every direction in parallel. The final form of streaming on the GPU was

```
if (ind<(9*(*width_d)*(*height_d))){
  if (Cells_const_d[ind_s].Fluid==1){
    if ((Cells_const_9d_d[ind].StreamLattice) == 1){
      Cells_var_9d_d[ind].F = Cells_var_9d_d[ind+c_d[ind_c]].METAF;
}  }  }
```

Thanks to the these changes the code proved ≈ 9 times faster.

It was again time to investigate the code. The effect of thread number was investigated again (Figure 12). This time the author saw a more natural trend. Basically two factors are responsible for the characteristic of the curves, namely the communication between the blocks and the speed of the individual threads. As presented in Chapter 2.5.2 if we increase the block

size, then after a certain size we reach a point where the execution cannot be done in parallel but in more steps. If we try to keep the number of threads per blocks low, then we clearly increase the inter-block communication. This communication is slow since it happens through the global memory. As we see in Figure 12, the minimum of the main loop execution time corresponded to 256 threads per block. For further simulations the author kept the solver at the optimum. (This led again to the limitation of the cells, but as we see in this case the solver performance is not influenced significantly by the thread number.)

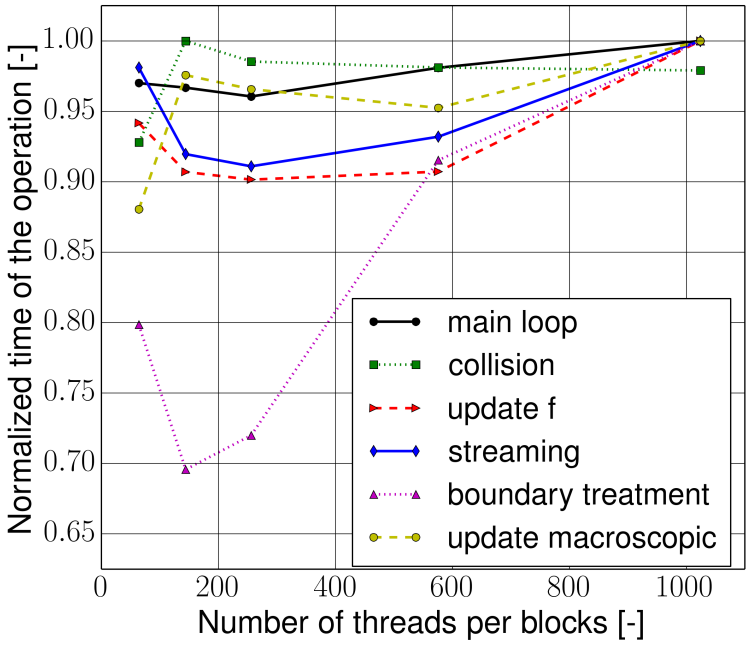


Figure 12: Effect of the block size II. (the final version of the code)

The profiling of the new code was carried out. The pie charts in Figure 13 show the results of the profiling for the BGKW and the MRT models. (The TRT model was more expensive than the BGKW model but less expensive than the MRT model.) As shown, the distribution of the computational load changed considerably (compared to the serial code shown in Figure 10). The boundary treatment, the residual computation, and the other operations have become more significant; the solver spent the most of the time with these steps. It is logical that the residual calculation is expensive compared to the other parts since it was not parallelized. The other operations include the data copying between the host and the device. It could be reduced if the residual calculations were executed on the GPU.

In the case of the MRT model the execution time of the parts show fundamentally different distribution. The collision is clearly the dominant and most time consuming part. The collision step in the MRT model includes summations, which requires a lot of communication between the blocks. This could be reduced with the use of the shared memory. As shown in Figure 13,

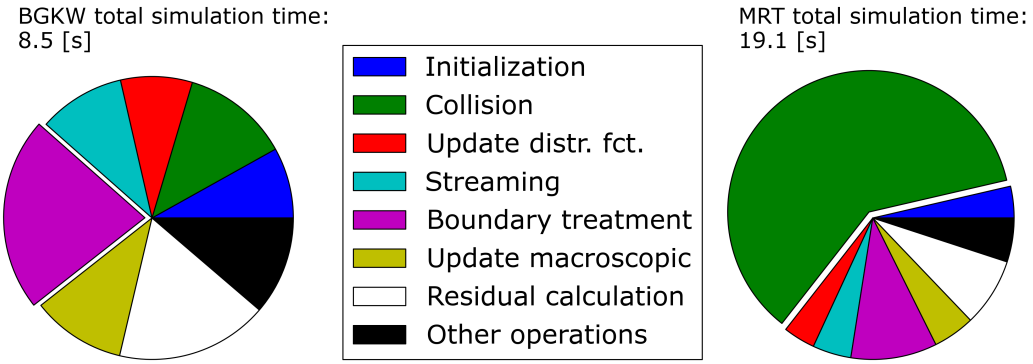


Figure 13: Results of the parallel code profiling - BGKW model, fine mesh (2402×483) (the abbreviation distr. fct. stands for distribution function)

every other part of the code needs less than half of the execution time. The performance of the solver was best with the last modifications.

The author prepared both single and double precision versions of the parallel and serial codes. (The `float` variables within the main structure(s) were changed to `double`.) To perform simulations with double precision the compiler needs an additional flag. The flag is related to the CUDA capability version number (check Appendix A). One can get this value from the deviceQuery sample code. The compilation command was modified from the simple `nvcc` to `nvcc -arch sm_21` (M1) and `nvcc -arch sm_20` (M2). The simple compilation based on the `nvcc` command did not have any limitation on the machines (the author could use the maximum size and block size without any problem). When the architecture was defined (`-arch`) the solver did not work with $1,024$ threads/block. The reason was probably the register limitation but the author could not find a direct answer for this question. The double precision solver (compiled with `-arch`) worked properly with 512 threads/block in machine M2.

	Machine	t_{CPU} (SP) [s]	t_{CPU} (DP) [s]	t_{GPU} (SP) [s]	t_{GPU} (DP) [s]
Streaming	M1	1085.4	—	175.6	—
	M2	1440.9	1508.7	98.3	130.6
Collision	M1	245.4	—	26.4	—
	M2	259.8	288.6	8.2	11.5
Main loop	M1	288.1	—	28.6	—
	M2	388.4	394.4	9.8	16.9

Table 3: Execution time on the investigated machines (SP stands for single precision; DP stands for double precision)

Some time measurement data are listed in Table 3. The data correspond to a mesh with spatial resolution 2402×483 and the BGKW model. As we see, the double precision execution of the main while loop was only slightly more expensive on the CPU ($\approx 5\%$ difference). The

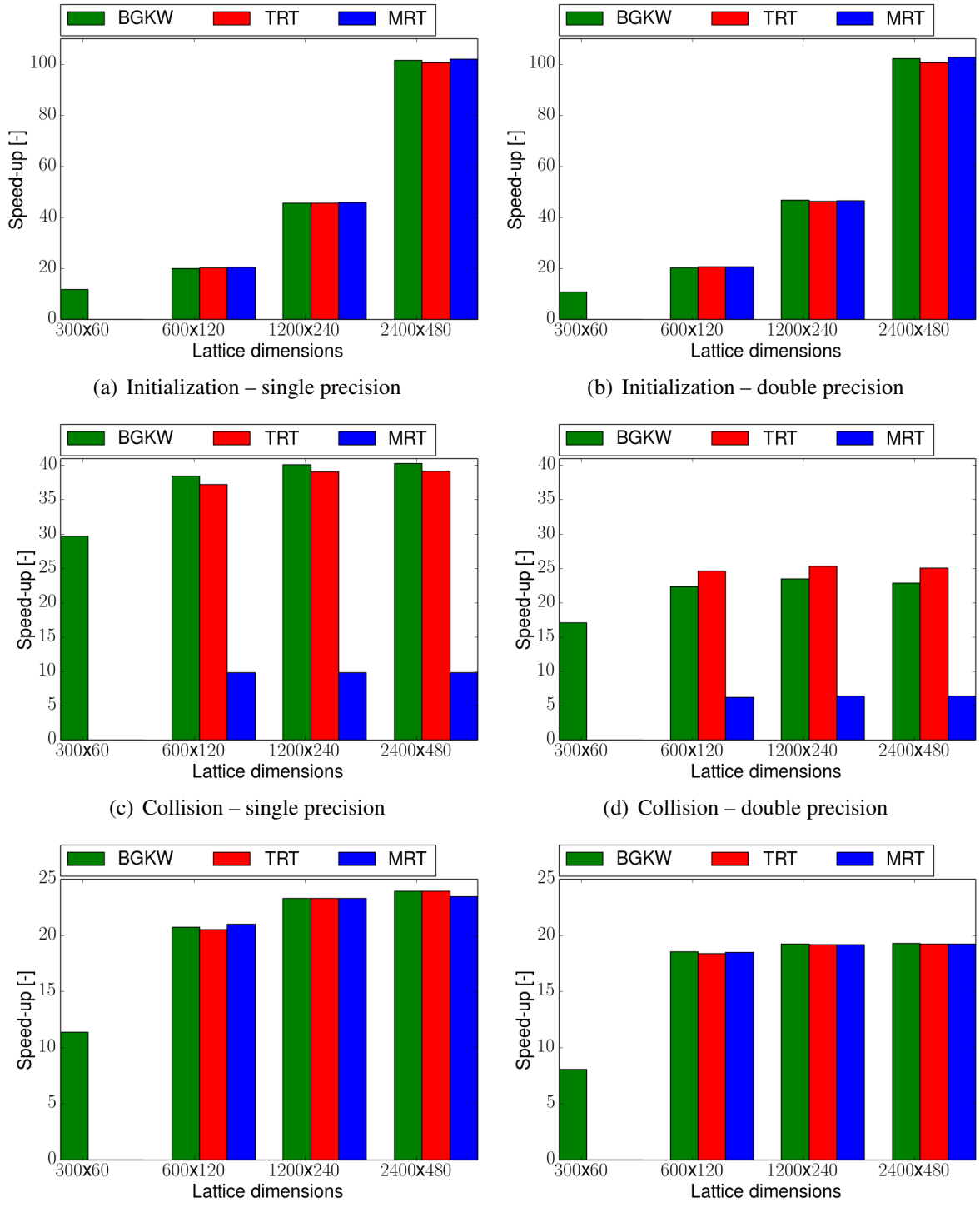
streaming needed more time by $\approx 5\%$ but the collision was slower by $\approx 10\%$. The collision step includes multiplications and divisions, whilst the streaming is basically copying data from one place to the other. Probably this is the reason for the differences. We can see that the CPU of machine M1 was $\approx 35\%$ faster. Which is clearly because of the 50% higher clock rate of M1.

As expected the double precision operation proved more expensive on the GPU. (It needed 30% more time). The cost difference is more obvious again in the case of the collision. Probably the data copying is also responsible for the more significant difference between simple and double precision. The GPU of machine M2 was almost three times faster than the device of M1.

In the following pages detailed analysis of the speed-up on machine M2 is displayed. The data copying is included only when the main while loop is analysed. Similar graphs for M1 are in Appendix C. (In the speed-up graphs for the M1 machine one can see a declining part. Probably the register memory of the machine was not big enough to store the structures.) The initialization proved incredibly efficient on the GPU (Figures 14(a) and 14(b), speed-up around 100). There was no difference between the single and the double precision speed-up of the initialization. Naturally the different collision models did not influence this step since the same initialization was performed. We can see the good scalability of the initialization.

The collision is a more interesting part of the code. It includes long equations to recover the particle interaction based on the relaxation time. As we saw in Figure 13 this was the most resource-intensive part in the case of the MRT solver. Figures 14(c) and 14(d) highlight the performance gap between single and double precision; however the solver proved efficient in both case. The speed-up of the MRT model was almost constant. It was less than 10 when single precision was used, and around 6 when double precision simulations were run. The scalability was poor compared to the initialization but the speed-up was still high, around 40 and 25 for single and double precision respectively.

The update f part makes the elements of two vectors equivalent. The speed-ups are shown in Figures 14(e) and 14(f). The streaming is similar because it also manages only data transfer from one location to the other. The single precision speed-up was 24 for the update f section, and 33 for the streaming part. Although the single precision graphs (Figure 14(e) and 15(a)) show a relatively good scalability, the double precision graphs (Figure 14(f) and 15(b)) display stagnation. The update macroscopic section had similar speed-up properties.



(a) Initialization – single precision

(b) Initialization – double precision

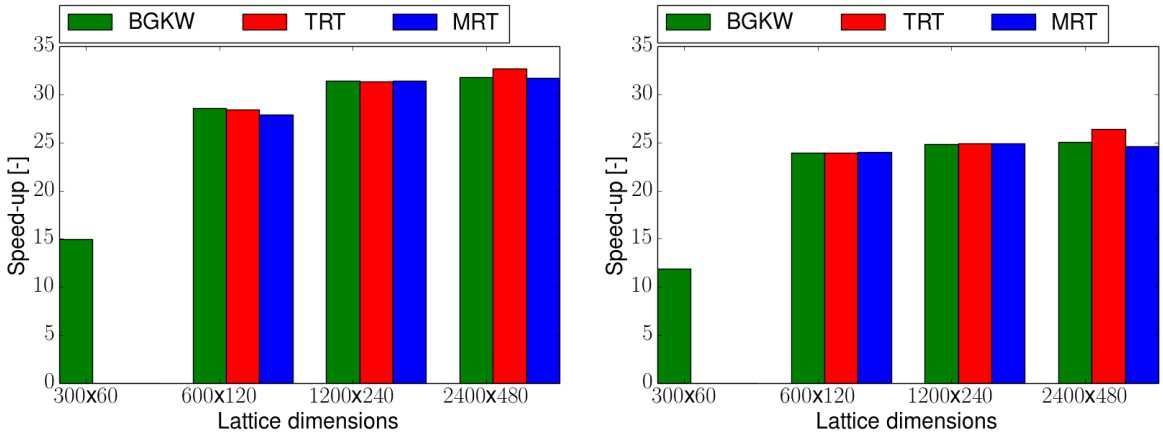
(c) Collision – single precision

(d) Collision – double precision

(e) Update f – single precision

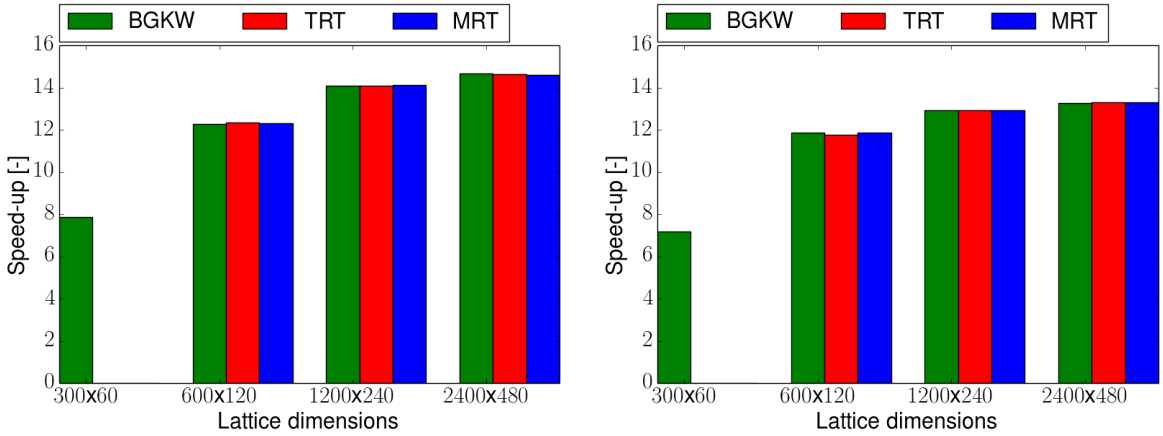
(f) Update f – double precision

Figure 14: Speed up of the different sections on machine M2 I.



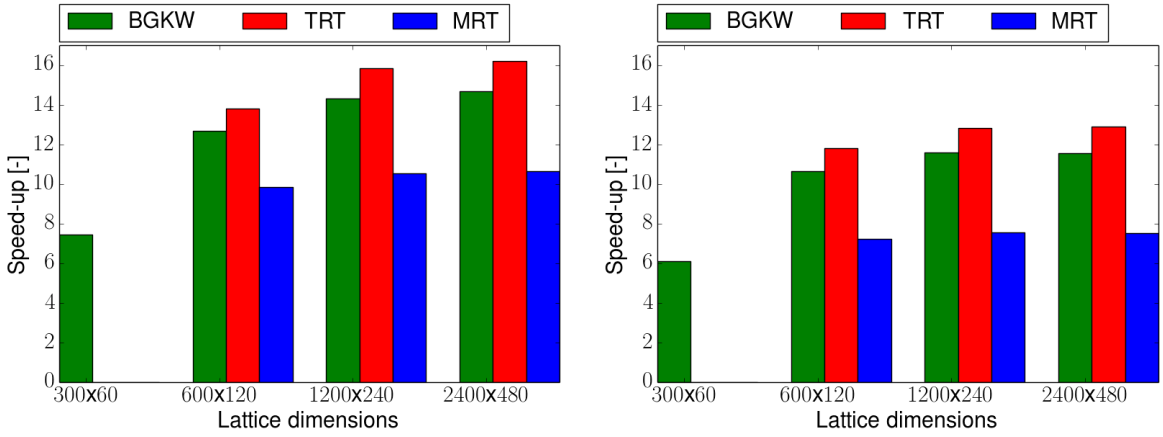
(a) Streaming – single precision

(b) Streaming – double precision



(c) Boundary treatment – single precision

(d) Boundary treatment – double precision



(e) Main loop – single precision

(f) Main loop – double precision

Figure 15: Speed up of the different sections on machine M2 II.

The boundary treatment is a critical part of the CFD code from a physical point of view, and as we see in Figures 15(c) and 15(d), it was critical from the speed-up point of view too. Both the single precision and the double precision diagrams show good scalability, but the speed-up peak was only around 15 and 13 for single (Figure 15(c)) and (Figure 15(d)) double precision respectively.

The last graphs display the speed up of the main loop which is the main target of parallelization. We should not forget that the main loop includes the residual computation which was not parallelized. Thanks to the vector formulation, that part has higher performance (2–4 times faster), but it is the actual bottleneck of the code. The speed-up is clearly visible both on the single precision (Figure 15(e)) and the double precision graphs (Figure 15(f)). A relatively good scalability was obtained and the solver speed-up is much higher when the number of elements is around $1,000,000$. The maximum speed-up of the MRT model (10) is quite poor compared to the BGKW (14) and the TRT (16) models. The speed up is obviously less in the cases of double precision simulations: 11 for the BGKW, 14 for the TRT, and 7 for the MRT model. All in all the parallelization could be considered successful. A significant speed-up was obtained in every part of the code. Similar speed-up was achieved based on CPU parallelization with more than 20 cores (grid size 2400×480); however, the parallel CPU code does not have such rigid memory limitations and the MRT model has scalability properties similar to the BGKW and the TRT models [104].

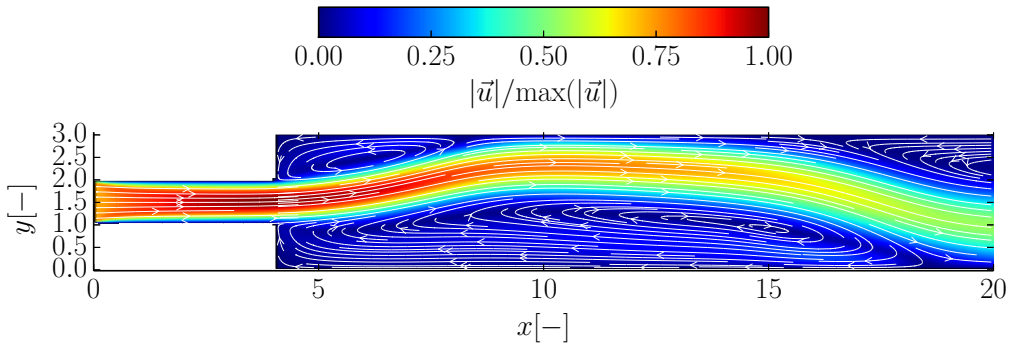


Figure 16: Captured backflow at the outlet with the first order opening formulations ($Re = 100$, BGKW model)

4.4.1 The first order opening outlet

As the reader will see later on, the original second order formulation of the opening outlet boundary condition proved unstable. The BGKW model especially was sensitive to this ap-

proach. The same numerical instabilities were reported earlier too [6, 5] and it was suggested that a first order formulation should be used instead [6]. The author implemented the simple first order boundary treatment at the outlet. The solver proved more stable with this formulation and several simulations could be carried out without divergence. Figure 16 shows a flow field which the second order formulation could not capture because of the backflow at the outlet. The results were obtained at Reynolds number $Re = 140$ (further details of the setting in Chapter 4.6.3).

4.5 Verification

The proof of the pudding is in the eating. Since there is no guarantee that one did not make some bugs during the parallelization, verification is essential. Roache described the verification as a process to check whether we are solving the equations correctly [107]. A grid convergence study is a widely used technique to this end. In addition a grid convergence study provides some useful informations about the accuracy of the applied scheme.

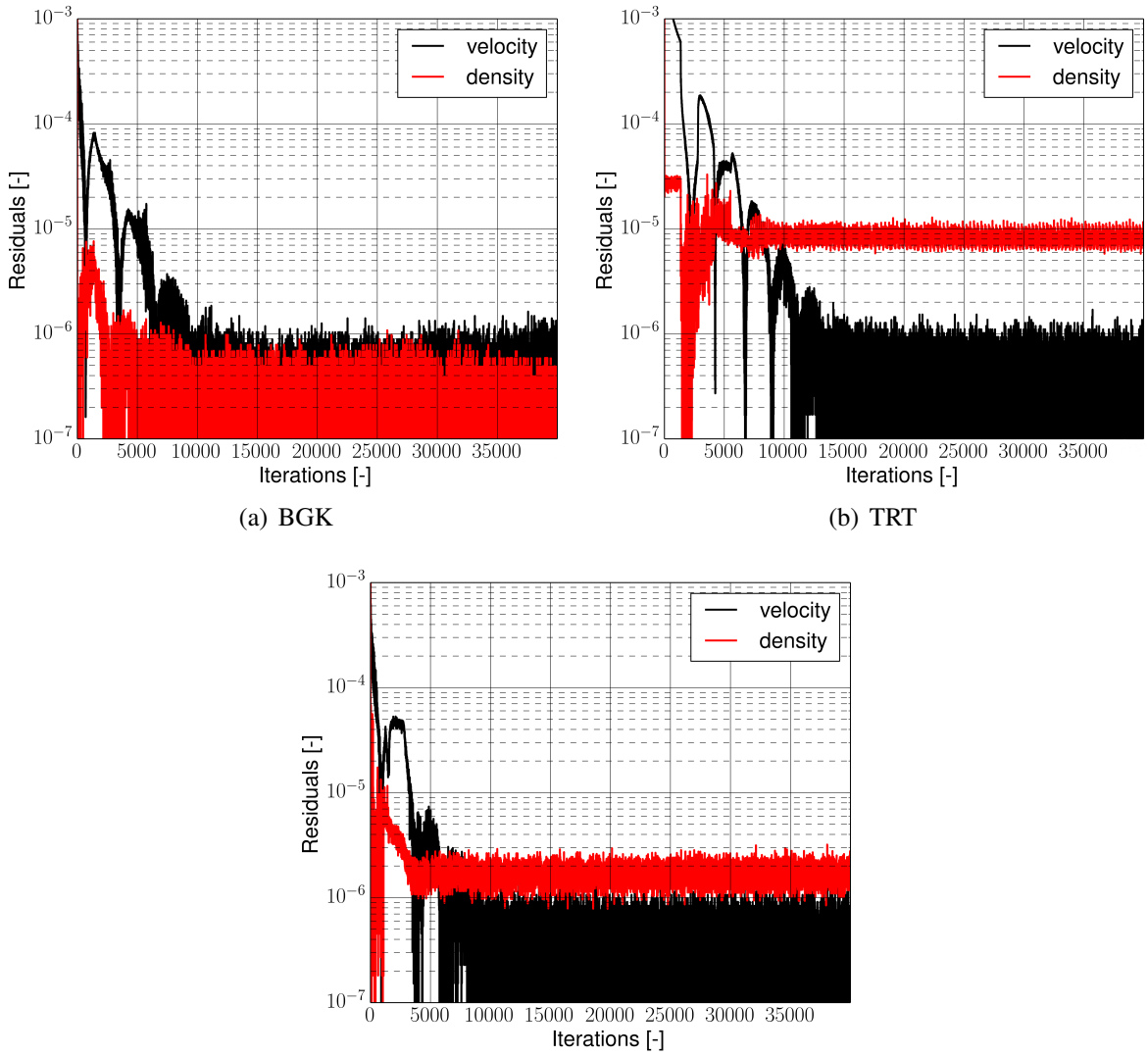
	mesh1	mesh2	mesh3
$n_x \times n_y$	402×23	802×43	1602×83
u_{lattice}	0.05	0.025	0.0125
Normalized grid spacing	4	2	1

Table 4: Channel flow simulation parameters

Since an analytical solution can be derived for the 2D channel flow, this test case is suitable for verification. Three different grids were created and all of them were investigated with the implemented collision models. Table 4 shows the parameters of the applied meshes (n_x and n_y stand for the x- and y-directional number of lattices. The geometry was given from the earlier code development [6].) First, simulations with Reynolds number $Re = 100$ were carried out. Since the LBM is totally dimensionless it is necessary to redefine the Reynolds number based on lattice quantities: $Re = \frac{nu_{\text{lattice}}}{\nu_{\text{lattice}}}$. In this expression n is the number of lattices along the characteristic length, u_{lattice} is the prescribed, dimensionless lattice velocity, and $\nu_{\text{lattice}} = 0.01$ is the dimensionless lattice viscosity. The characteristic length in the case of channel flow is the channel height $H = 5$ m. The Cartesian coordinates were made dimensionless by the channel height. The channel length was 100 m.

At the inlet a uniform velocity profile was prescribed (based on the lattice velocity) and at the outlet the first order opening formulation was used. Constant velocity was initialized in the domain (same as at the inlet). From a convergence point of view, $40,000$ iterations were found satisfactory with single precision. It is worth noting that CFD solvers typically need

more iterations if the mesh is finer. In the LBM one has to decrease the lattice velocity to keep the Reynolds number constant if the spatial resolution increases. In this way the solver needs almost the same number of iterations, nearly independently from the spatial resolution (if the Reynolds number is kept constant).



(a) BGK

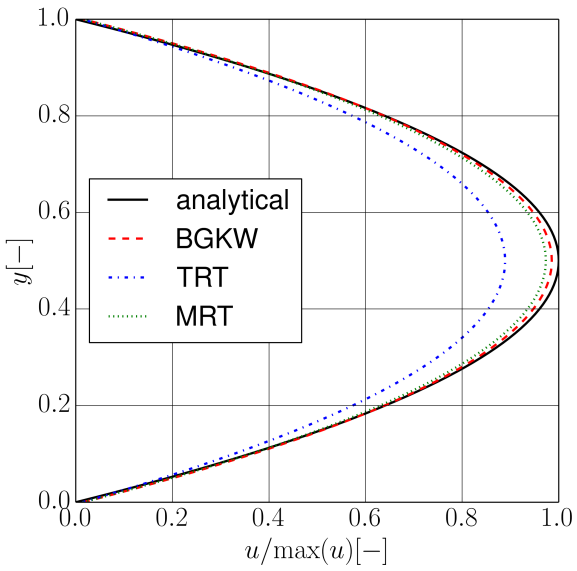
(b) TRT

(c) MRT

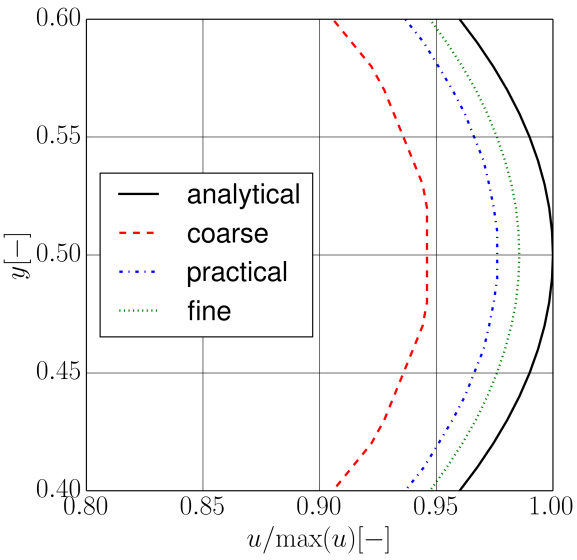
Figure 17: Residuals during the channel flow simulations (practical mesh)

The presented results correspond to the parallel code. Post processing was carried out in Python. Before we check the results, usually we examine the residuals. Figure 17 displays the residuals for the three collision model. As shown, there are two or three oscillations at the beginning of the residuals with a constant period time. The propagation of the distribution function causes waves travelling by the lattice speed of sound. These waves typically are reflected from the boundaries and they need time to vanish. Probably these waves are responsible for the

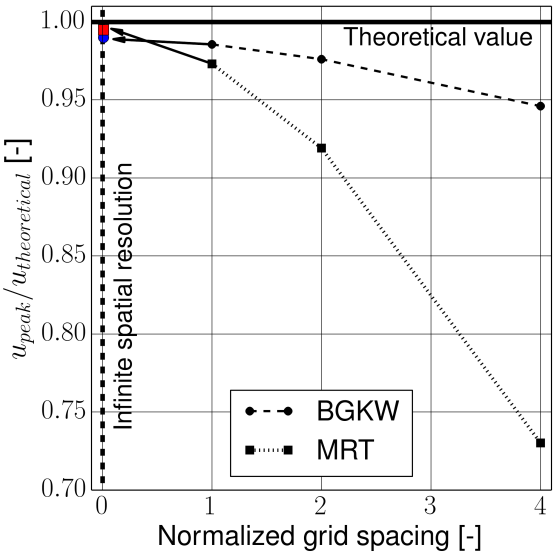
fluctuations. After 15,000 iterations the residuals were unchanged in every case. The convergence level of the single precision simulations was between 10^{-6} and $5 \cdot 10^{-6}$. In the case of double precision a convergence level $\approx 10^{-8}$ was reported [104].



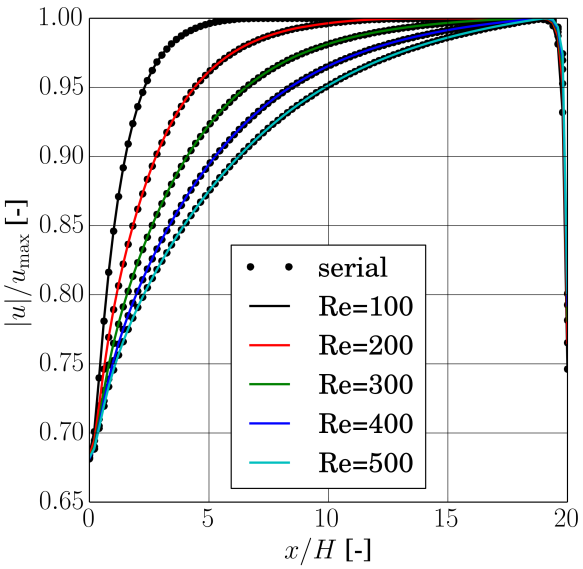
(a) Developed profiles at $x = 90$ m ($n_x \times n_y = 1602 \times 83$)



(b) Convergence of the velocity profile



(c) Richardson extrapolation of the peak velocities



(d) Velocity development ($n_x \times n_y = 802 \times 43$)

Figure 18: Results of the channel flow simulations

Although the residual curves in Figure 17(b) have similar characteristic than in the other graphs, we can note a higher convergence limit, especially for the density. Since the density should be constant at the end of the steady state lattice Boltzmann simulations, the higher convergence level is a bad omen. As we will see, the TRT model was not able to produce a reliable

solution for the channel flow.

Figure 18(a) shows the computed velocity profiles. We can see that every collision model gave qualitatively good results but the TRT model proved quantitatively poor. Figure 18(b) displays the convergence of the BGKW collision model. This model gave a relatively accurate solution with the coarse mesh too. Similar figures for the other models can be found in Appendix D. The TRT model was quite inaccurate with the fine mesh too. Whilst the MRT model also proved inaccurate with the coarse mesh, it showed promising convergence behaviour. The convergence of the MRT and the BGKW models are also presented in Figure 18(c). The normalized grid spacing (also shown in Table 4) was calculated as the ratio of the number of elements in the finest mesh and in the actual mesh. (Zero normalized grid spacing is equivalent to an infinitely fine grid, which theoretically gives back the analytical value.) One can see in Figure 18(c) the convergence of the peak velocities. The values tend clearly to the analytical solution.

	Dimensionless velocity peak						
	mesh1	mesh2	mesh3	extrap.	$GCI_{1,2}$	$GCI_{2,3}$	p_a
BGKW	0.9459	0.9761	0.9856	0.9899	1.77	0.55	1.67
TRT	0.5126	0.7309	0.8889	–	–	–	–
MRT	0.7302	0.9191	0.9732	**0.9948**	10.29	2.78	1.81

Table 5: Grid convergence study of the channel flow (extrap. stands for the extrapolated value)

The author performed Richardson extrapolation based on the peak values. (The details of the extrapolation technique are available on the web page of NASA [108]. The arrows in Figure 18(c) indicate the extrapolation and the colourful marks show the extrapolated values. Although the MRT was less accurate in the case of the coarse mesh, the extrapolated value from the model was closer to the analytical solution. Table 5 summarizes the results of the Richardson extrapolation and also contains the grid convergence indices (GCI). (The author found the TRT model inappropriate for channel flow simulations with the current spatial resolutions and settings.) Based on the GCI values we can conclude that MRT has a slightly higher spatial resolution requirement compared to BGKW. Both the MRT and the TRT model gave an order of accuracy (p_a) close to the right value (2).

The time, needed for one iteration, was calculated as the ratio of the total execution time and the number of iterations. The parallel code proved approximately ten, nine, and six times faster for the BGKW, the TRT and the MRT models. These speed-ups correspond to the simulations with the finest mesh.

After the grid convergence study some simulations were run with different Reynolds numbers. The development of the peak velocity at the centreline of the channel is displayed in

Figure 18(d). The black dots indicate the results of the serial code, while the lines show the results of the parallel one. The serial and the parallel solvers provided the same, physical results.

4.6 Validation

Verification is mostly related to mathematics and the implementation itself but we are interested in the physical description. During the validation procedure we should ask the question: "are we solving the right equations?" [107]. To check the connection between simulation results and reality we need measurement data. In the current chapter the author used four test cases to investigate the validity of the solver. Figure 19(a) and 19(b) present the mesh used for the lid-driven cavity and the von Kármán vortex street simulations (further pictures about the structured grids can be found in Appendix E). In these figures the grey cells indicate the fluid domain. Table 6 includes the most important parameters of the examined domains.

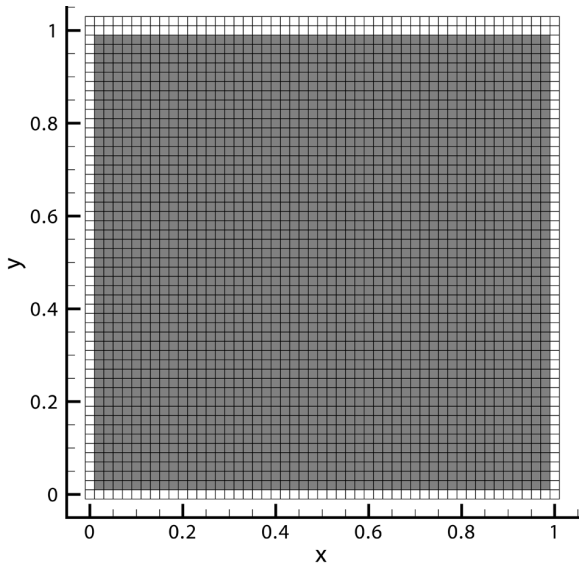
Geometry	Size ($x \times y$)	Spatial resolution ($n_x \times n_y$)
Cavity	1×1	202×203
Step	20×2	402×43
Expansion	30×3	802×83
Cylinder	5×1	602×123

Table 6: Parameters of the investigated geometries [6]

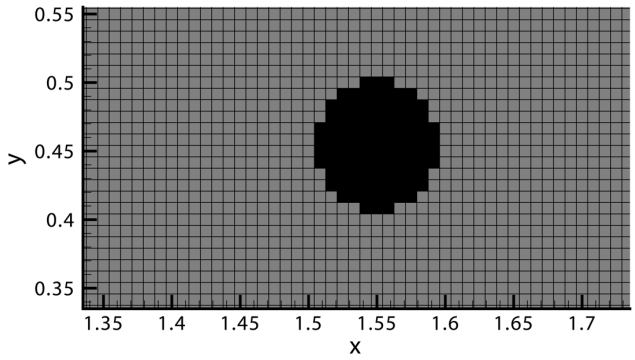
4.6.1 Lid-driven cavity

The lid-driven cavity is a widely used validation test case in CFD. While Prasad and Koseff [109] carried out measurements, Ghia et al. [110] performed numerical simulations to study the flow field and provide data for validation. (Ghia et al. used the vorticity-stream function approach to overcome the pressure-velocity coupling problem.) The LBM simulations were compared to the results of Ghia et al. [110].

In the case of the lid-driven cavity the characteristic length is the height or the length of the cavity. (It is natural that the characteristic length depends on the aspect ratio of the cavity. The smaller size determines the length.) The lattice viscosity was $\nu_{\text{lattice}} = 0.01$ and the number of cells along the characteristic edge was $n_x = 200$. Based on these parameters one can calculate the necessary lattice velocity for the three applied Reynolds numbers. The investigated cases are listed in Table 7. The author ran $200,000$ iterations to reach converged, steady state solutions. The Cartesian coordinates were made dimensionless by the length of the cavity. In Table 7 t_{CPU} and t_{GPU} stand for the necessary time for one iteration on the CPU and the GPU. The speed-ups



(a) Mesh for the cavity flow (52×53)



(b) Mesh around the cylinder (602×123)

Figure 19: Structure of the applied grids (the grey cells indicate the fluid zone)

of the BGKW, TRT and MRT models are ≈ 9, ≈ 8, and ≈ 5.5 for the used mesh. The results correspond to machine M1.

The flow fields visualized by streamlines and the velocity contours are displayed on the following page. Drawings of the reference flow fields (simulation results of Ghia et al. [110]) for the three Reynolds number are attached (Appendix F). Figure 20(a) corresponds to $Re = 100$ and it shows the flow field obtained by the BGKW model. We can clearly recognize the swirling flow and a small vortex in the lower right corner. In addition there should be a small corner vortex in the lower left corner but it seems that the models could not capture it with the current spatial resolution. Maybe the same spatial resolution would be satisfactory with double precision. The centre of the big vortex is not at the middle of the domain which is in good agreement with the reference data.

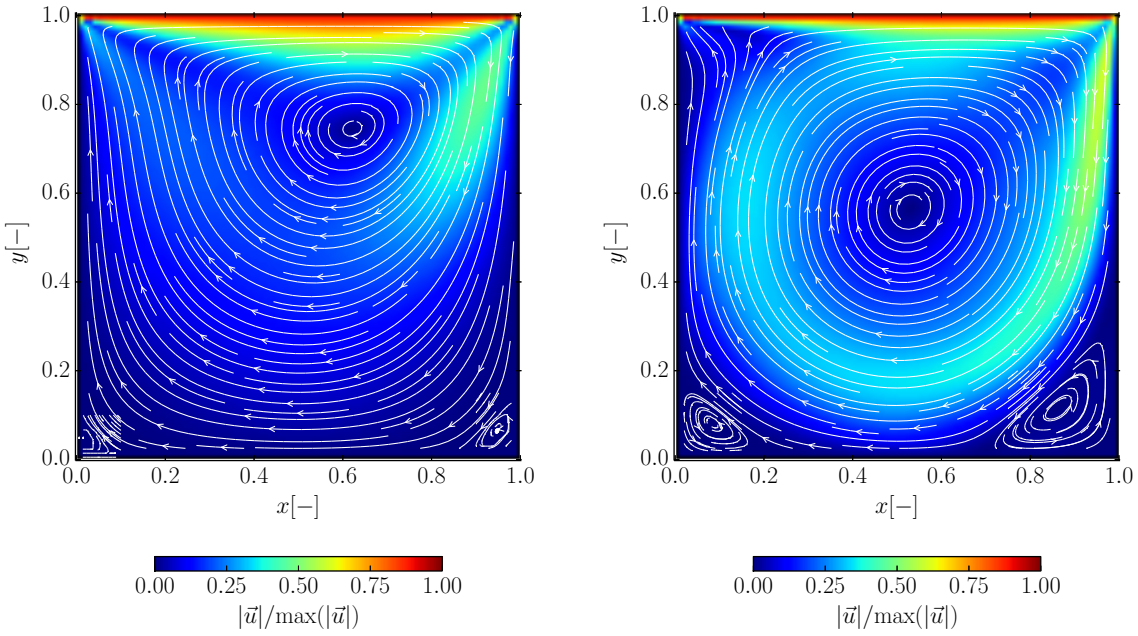
As Figure 20(b) displays, the corner vortices enlarged as the Reynolds number increased up to $1,000$. The right corner vortex was significantly bigger than the right one. The centre of the big vortex shifted to the middle of the domain (compared to $Re = 100$). The flow field shows a good qualitative agreement with the reference data. In the case of $Re = 3,200$ we see a similar flow field (Figure 20(c)) but one can identify a third vortex close to the upper left corner. Although the size of the right corner vortex was almost unchanged compared to $Re = 1,000$, the left corner vortex grew considerably. The flow field is again in great concordance with the reference pictures. The results of the TRT and the MRT models are attached (Appendix G and Appendix H).

Model	Reynolds	u_{lattice} (inlet)	ν_{lattice}	Notes	t_{CPU}	t_{GPU}
BGKW	100	0.005	0.01	OK	36	4
	1000	0.05	0.01	OK		
	3200	0.0866	0.0054125	inaccurate		
TRT	100	0.005	0.01	OK	42	5
	1000	0.05	0.01	OK		
	3200	0.0866	0.0054125	diverged		
MRT	100	0.005	0.01	OK	55	10
	1000	0.05	0.01	OK		
	3200	0.0866	0.0054125	inaccurate		

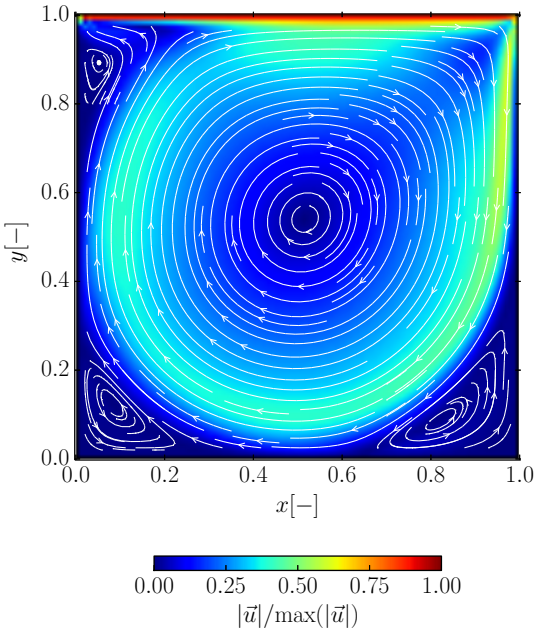
Table 7: Lid-driven cavity simulations

After the qualitative flow field analysis it is instructive to have a closer look at the velocity profiles. The x-directional velocity profile was investigated along the $x = 0.5$ sample line. Similarly the y-directional velocity profile was investigated along the $y = 0.5$ sample line. As displayed in Figures 21(a) and 21(b) the collision models produced very similar profiles. For $Re = 100$ and $Re = 1,000$, the results were very close to the simulation of Ghia et al. [110]. Whilst the profiles are smooth in Figure 21(a), the reader can see higher gradients near the wall in Figure 21(b) which seems logical based on our physical sense. Although we are not able to distinguish the profiles of the three collision models from Figures 21(a) and 21(b), there was a slight discrepancy between the computed values, which is presented in Figure 21(c). This graph highlights that the MRT model proved slightly more accurate than the other models at $Re = 1,000$ and the TRT model was the less accurate. (Linear interpolation was used to achieve the results at the location of the reference simulations.)

Figure 21(d) visualizes the profiles corresponding to the highest investigated Reynolds number, namely $Re = 3,200$. The profiles computed by the BGKW and the MRT models were quite inaccurate near the left and right walls and the velocity magnitude was overestimated. The simulations had to be stabilized with decreased lattice velocity and viscosity to run $Re = 3,200$



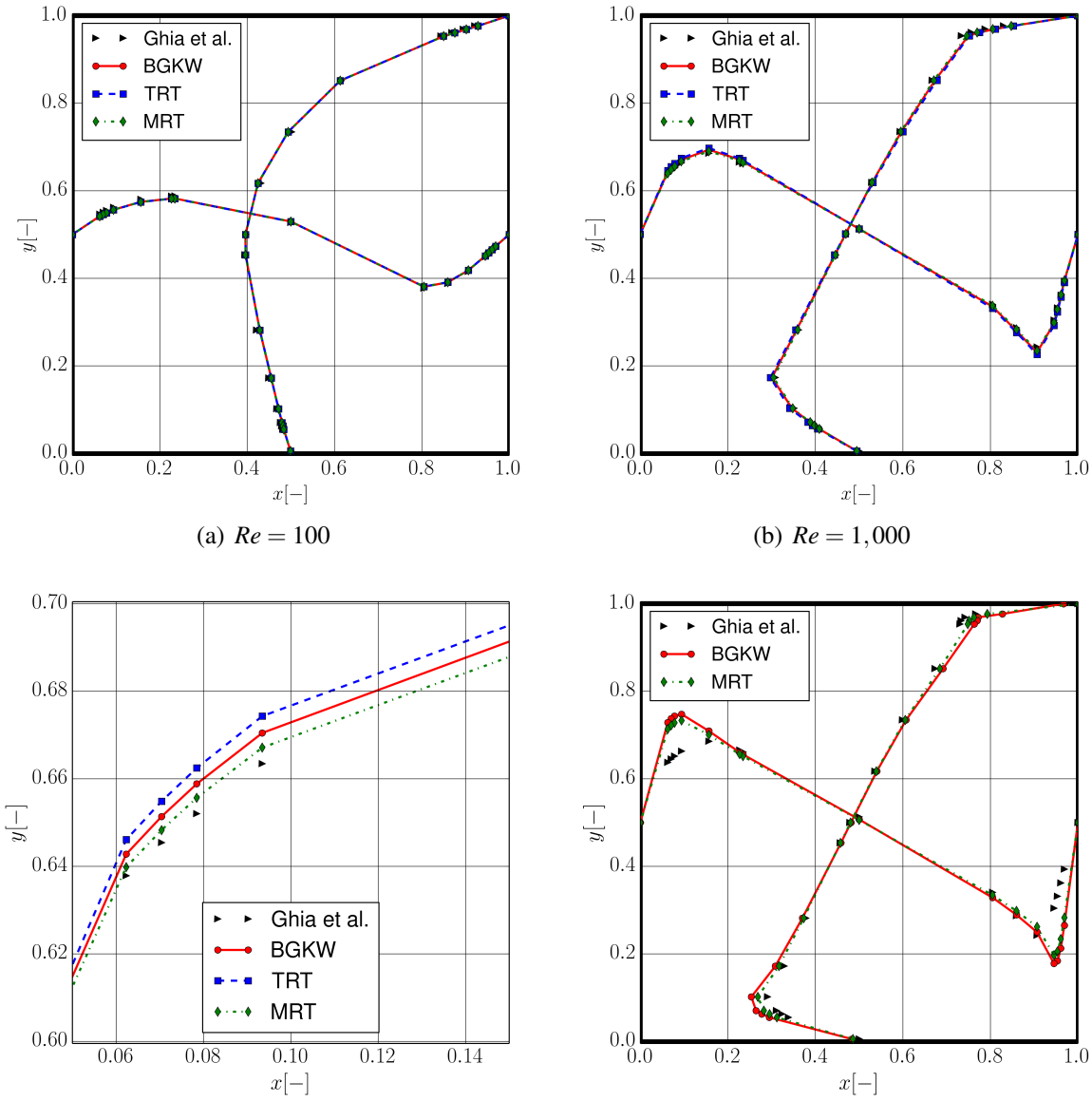
(a) $Re = 100$ (b) $Re = 1,000$



(c) $Re = 3,200$

Figure 20: Results of the lid-driven cavity simulations (BGKW)

(see Table 7). A typical criterion on the LBM can be formed based on the Mach number ($Ma = u_{\text{lattice}}/c_s$), which is calculated from the lattice speed of sound $c_s = 1/\sqrt{3}$. The method is more accurate in the case of low Mach numbers (thus $Ma << 1$) because the error (O) increases



(a) $Re = 100$ (b) $Re = 1{,}000$

(c) $Re = 1{,}000$ (d) $Re = 3{,}200$

Figure 21: Velocity profiles in the lid-driven cavity compared to the simulations of Ghia et al [110]

with the Mach number as $O(Ma^2)$. A rule of thumb is to set $Ma = \sqrt{3}u_{\text{lattice}} < 0.15$ [111]. This condition was satisfied in every case but the TRT model diverged with the settings. The author could not find a suitable settings for every collision model. The profiles obtained by the parallel and the serial solvers were compared and the graphs presenting the comparison can be found in Appendix I. The parallel and the serial results agreed within line thickness.

4.6.2 Backward facing step

The backward facing step is another typical validation test case in CFD. Whilst in the last chapter the results were compared to simulation data, the author will compare the simulations against the measurements of Armaly et al. [112] in the current section.

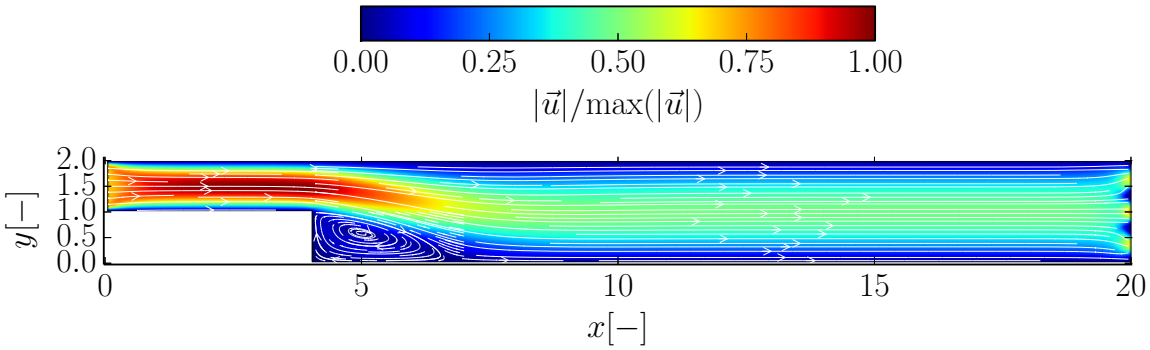
The characteristic length in the backward facing step is the height of the inlet section. The Cartesian coordinates were made dimensionless by the inlet section height. There were 20 cells along the inlet edge. The investigated Reynolds number was $Re = 100$. As usual, the lattice viscosity was set to be $\nu_{\text{lattice}} = 0.01$. A uniform velocity of 0.025 was prescribed at the inlet. For the Reynolds number calculation the maximum inlet lattice velocity ($u_{\text{lattice}} \approx 0.025$) has to be substituted. Furthermore Armaly et al [112] used the hydraulic diameter for the Reynolds number calculations so the number of lattices in the Reynolds expression was $n = 40$. The author found $60,000$ iterations enough to reach the steady state.

Model	RE
BGK	-8.7%
TRT	-18.7%
MRT	-10.9%

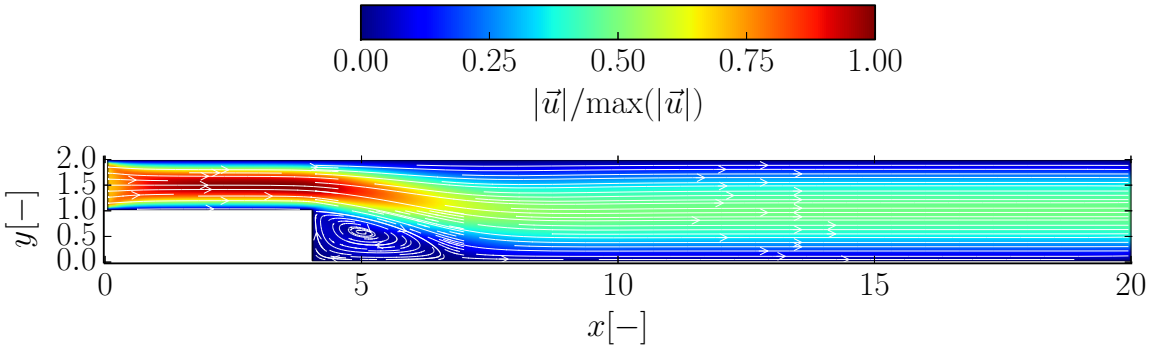
Table 8: Relative error (RE) of the reattachment length in the BFS compared to the measurements of Armaly et al. [112]

These simulations proved suitable for testing the implemented first order opening boundary condition at the outlet. The velocity fields, computed by the BGKW collision model, are displayed in Figures 22(a) and 22(b). Whilst Figure 22(a) shows the flow field obtained by the second order opening formulation, Figure 22(b) represents the results based on the first order formulations. We can see that the second order formulations resulted in non-physical distortion of the flow field near the outlet (Figure 22(a)). Compared to the second order formulation the first order formulation was more stable and it did not cause any disturbance. The author noted that the second order formulation proved unreliable only in the case of the BGKW collision model and it led to divergence when more iterations were performed. The same effect was also realized in a former thesis [6]. For the other models it is better to keep the second order formulation because the first order outlet decreases the accuracy of the solver.

The flow field itself was in good agreement with the measurement results. After the step a vortex was formed at the bottom of the domain. Naturally a reattachment followed the vortex. The same flow features are reported in the literature [112]. The three collision models produced similar flow fields including the same flow features. (The velocity fields and the streamlines obtained by the other models are attached as Appendix J.) The reattachment distance after the



(a) BGKW model with second order outlet boundary condition formulation

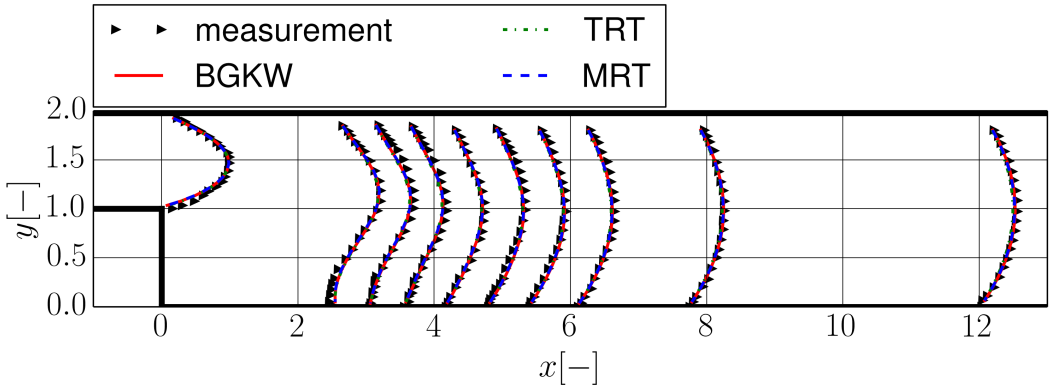


(b) BGKW model with first order outlet boundary condition formulation

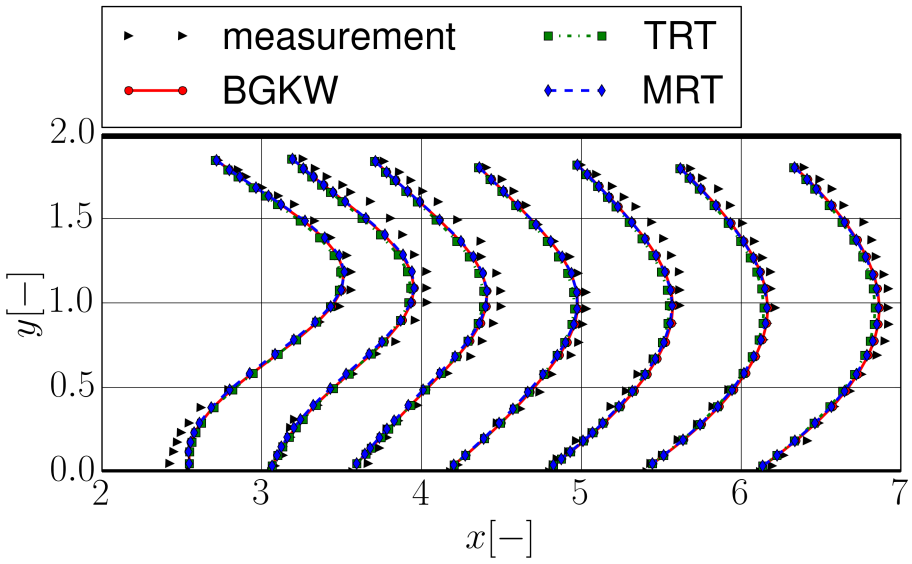
Figure 22: Results of the backward facing step simulations

step is suitable for quantitative comparison. The relative error (RE) of the reattachment is listed in Table 8. The error of the TRT model was almost twice the error of the other models and the BGKW model gave the most accurate reattachment length. The negative relative error values mean that the lattice Boltzmann simulations underestimated the distance. (The reattachment length was determined based on the prepared streamlines. Although the represented data is informative, a more accurate solution would be to compute the wall shear stress at the walls. Reattachment occurs where the wall shear stress is equal to zero.)

The measurement results are available at several locations. The locations, furthermore the simulated and measured x-directional velocity profiles at these locations are shown in Figure 23(a). A relatively good qualitative agreement is visible in this graph. Figure 23(b) shows the same profiles in the region of interest. The profiles were typically underestimated by the models. Typically the results of the BGKW or the MRT models were closer to the measurement data. The TRT model proved the least accurate again. All in all the models provided very similar profiles. The profiles obtained by the parallel and the serial solvers were compared again



(a) Velocity profiles after the backward facing step



(b) A closer look at the profiles

Figure 23: Velocity profiles after a backward facing step compared to the measurements of Armaly et al. [112]

and the graphs presenting the comparison can be found in the Appendix K. The parallel and the serial results agreed within line thickness.

4.6.3 Sudden expansion

There is another validation test case for incompressible flows in the investigated flow regime, namely the sudden expansion. Fearn et al. studied the stability of the flow after a sudden expansion [113]. The data are highly suitable for validating solvers both qualitatively and quantitatively.

The characteristic length in the case of sudden expansion is half the height of the inlet section. There were 26 cells along the inlet edge. The investigated Reynolds number was

Collision model	Reynolds	u_{inlet}	Second order opening	First order opening
BGKW	25	≈ 0.0192	divergence after 90,000 iterations	200,000 iteration
	80	≈ 0.0410	divergence after 80,000 iterations	300,000 iteration
TRT	25	≈ 0.0192	200,000 iterations	200,000 iteration
	80	≈ 0.0410	500,000 iteration	500,000 iteration
MRT	25	≈ 0.0192	200,000 iterations	200,000 iteration
	80	≈ 0.0410	500,000 iterations	500,000 iteration

Table 9: Sudden expansion simulation settings and results

$Re = 100$. As usual, the lattice viscosity was set to be $\nu_{\text{lattice}} = 0.01$. A uniform velocity profile was prescribed at the inlet with a lattice velocity u_{inlet}. For the Reynolds number calculation the lattice velocity $u_{\text{lattice}} = 1.5u_{\text{inlet}}$ and half the channel height ($n = 13$) have to be substituted. Table 9 summarizes the prescribed velocity values for the different simulations. The author found $60,000$ iteration enough to reach the steady state. The Cartesian coordinates were made dimensionless by the inlet section height.

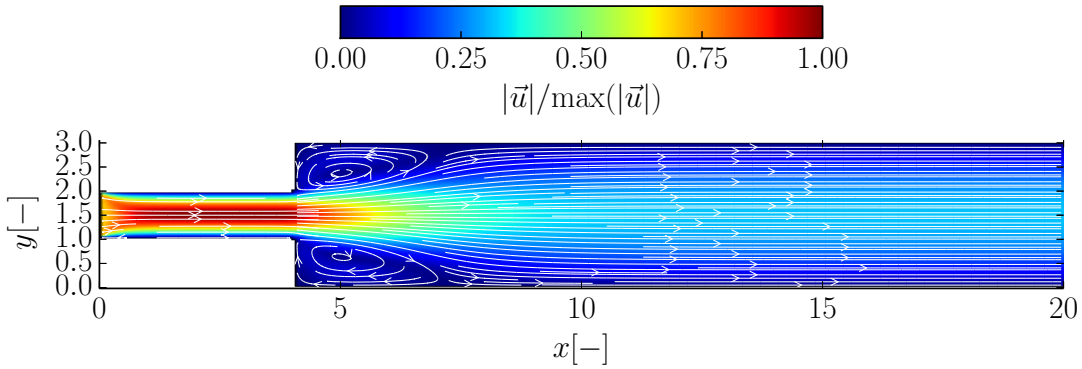
The visualized flow fields at the two investigated Reynolds numbers are displayed in the following page. (The collision models produced similar results so only the BGKW model is presented here.) Under these conditions the "Navier-Stokes equation loses stability at a critical Reynolds number $Re = 40.45 \pm 0.17\%$". Figure 24(a) shows the flow field at $Re = 25$ which is clearly in the stable region whilst the flow field at $Re = 80$ is visible in Figure 24(b). The LBM gave a symmetric flow field in the stable region as expected (Figure 24(b)). The formed vortices had the same size, so the same reattachment length could be measured at the top and the bottom of the domain after the expansion. After the step the streamlines had a typical spreading, hand-held fan-like pattern, similar to a jet.

Reynolds	25	80	
location	–	upper	lower
BGK	-18.5%	-7.5%	-8.4%
TRT	-15.9%	-7.0%	-10.2%
MRT	-14.6%	-8.8%	-7.3%

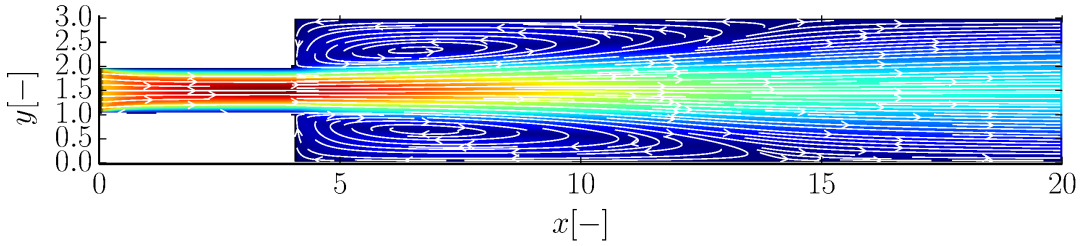
Table 10: Relative error of the reattachment length in the sudden expansion compared to the measurments of Fearn et al. [113]

Since the LBM is essentially transient, we can follow the stability losing process of the jet in the case of $Re = 80$, which is clearly in the unstable regime. Figures 24(b) 24(c) and 24(d) show the flow field at three different time instants. The flow field after $250,000$ iterations (Figure 24(b)) at $Re = 80$ is qualitatively only slightly different compared to $Re = 25$. We can see the vortices both at the top and the bottom, and a mainly symmetric flow field. If we look

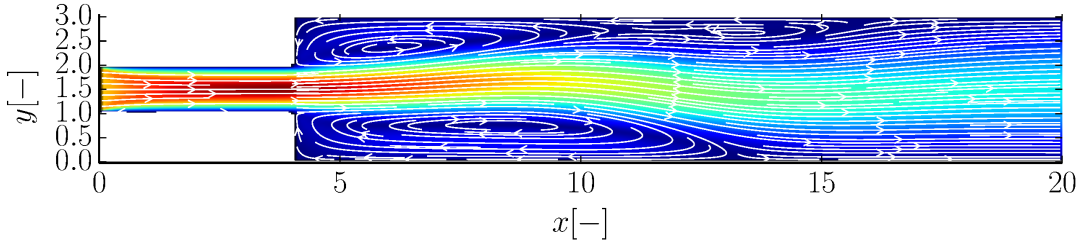
at the details, we may recognize that the vortex core at the bottom has already been shifted a little bit closer to the outlet. (The vortex itself seems somewhat longer at the bottom.) Later on (Figure 24(c)) one can recognize clearly the bent state of the jet. The elongated vortices are unstable and they brake easily to smaller vortices. One may identify a second, smaller vortex at



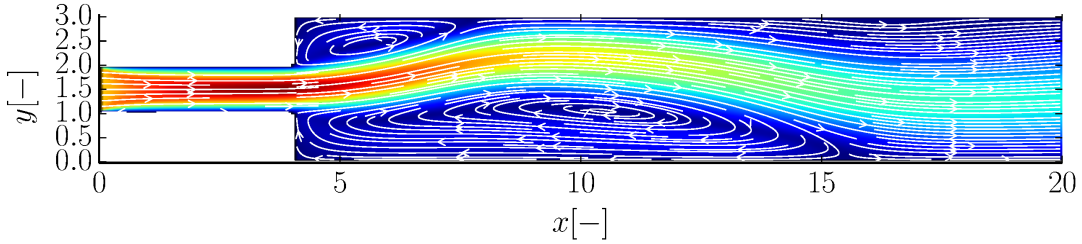
(a) $Re = 25$



(b) $Re = 80$ (after $250,000$ iterations)



(c) $Re = 80$ (after $300,000$ iterations)

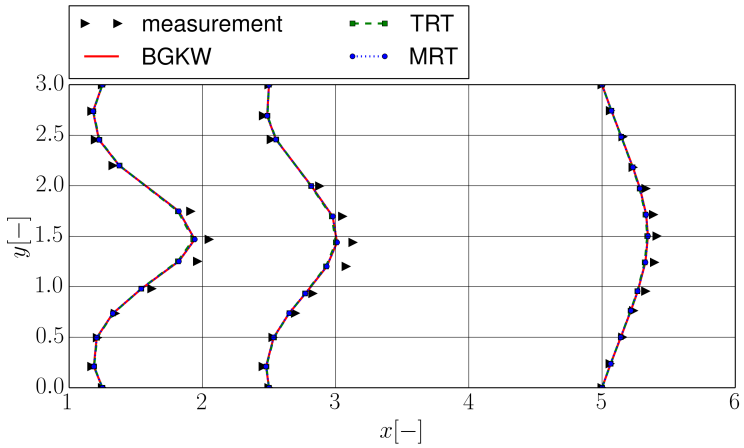


(d) $Re = 80$ (after $500,000$ iterations)

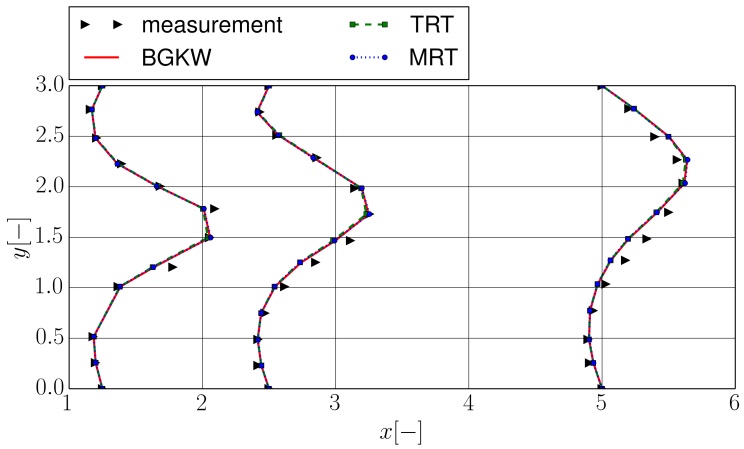
Figure 24: Results of the sudden expansion simulations (BGKW)

the top (between $x = 10$ and $x = 15$). The upper vortices were formed from the elongated upper vortex after braking.

The transition happened after $250,000$ iterations and it took several steps to reach the steady state. After $500,000$ iterations the solver produced the bent and reattached jet as displayed in Figure 24(d). Quantitatively the whole process occurred as expected. The flow fields are in good agreement with the measurement [113] at the examined Reynolds numbers. The flow fields computed by the TRT and the MRT models are attached in Appendix L and Appendix M respectively.



(a) Velocity profiles after the sudden expansion ($Re = 25$)



(b) Velocity profiles after the sudden expansion ($Re = 80$)

Figure 25: Velocity profiles after a sudden expansion compared to the measurements of Fearn et al. [113]

The velocity profiles were compared to the available measurement data. The models computed quite similar profiles, and the TRT proved the least accurate again. Figure 25(a) shows the symmetric profiles corresponding to $Re = 25$ whilst Figure 25(b) displays the asymmetric

ones at $Re = 80$. The velocity profiles were again typically underestimated by the LBM but qualitatively they predicted the velocity distribution well. The back-flows are visible in both figures.

The relative error of the reattachment lengths are collected in Table 10. This error is quite high in the case of $Re = 25$ but it is important to note that small quantities often result in a high relative error. The error at $Re = 80$ was satisfactory, and it was significantly lower (almost a half) than the errors at $Re = 25$.

One might consider $500,000$ iterations too much but it is usual if we are talking about LBM. The parallel solver needed only around half an hour to resolve the transition with the BGKW model but it took 2.5 hours to produce the same results with the MRT model. Data was saved after every $50,000$ iterations. It meant ≈ 3.6 ms/itaration with the BGKW model and ≈ 18 ms/iteration with the MRT model. The necessary times for one iteration with the serial solver are ≈ 49 ms and ≈ 86 ms respectively.

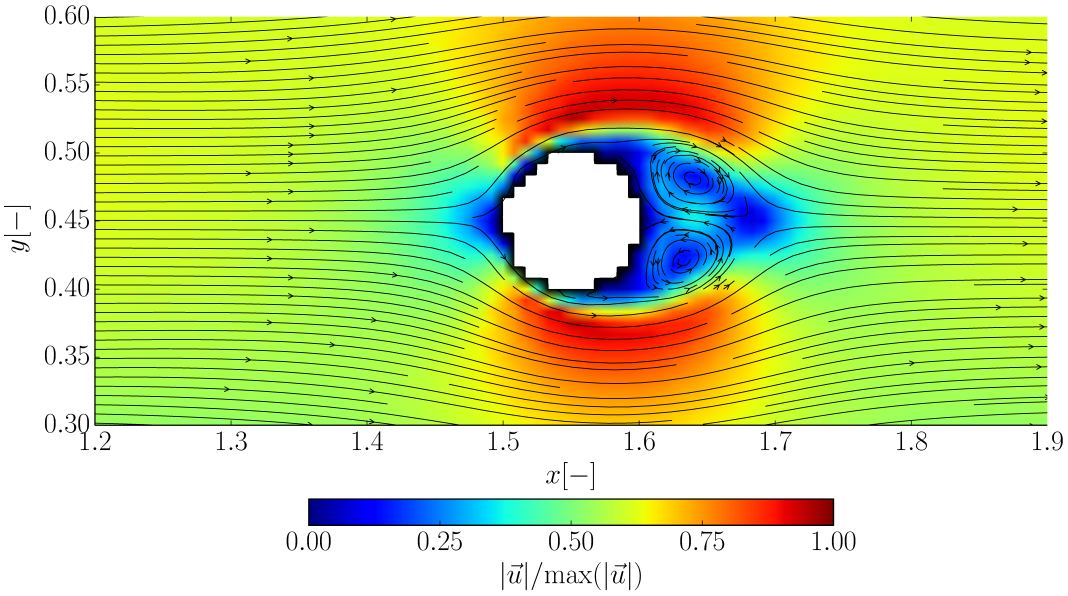
4.6.4 Flow around a cylinder

So far the validation has included mainly steady state analysis (although the sudden expansion analysis touched the time dependent transition). The flow around a cylinder is a simple test case which results in a very interesting flow field. Although the boundary conditions are stationary the flow field becomes periodic at certain conditions. Consequently the test case is suitable for examining the temporal behaviour of the solver at least qualitatively.

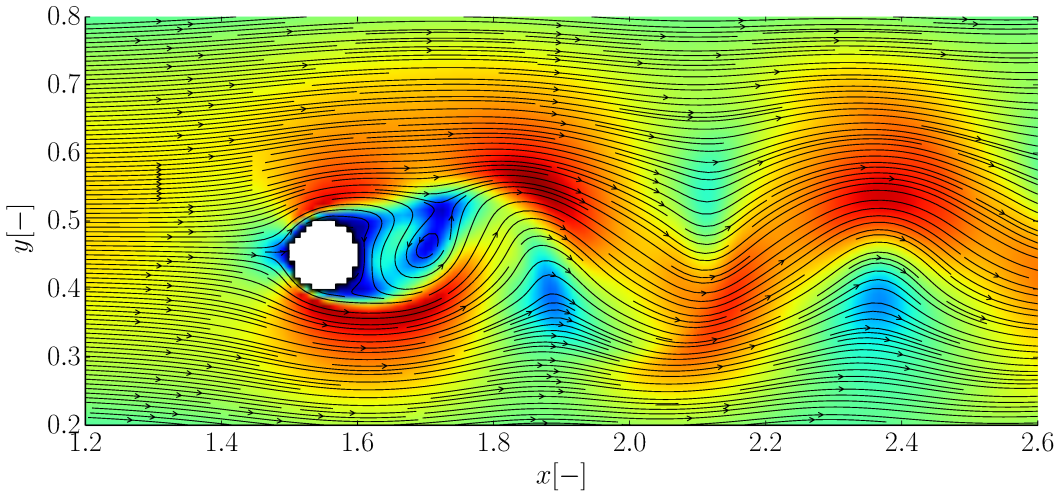
In this chapter the flow around a cylinder will be examined at $Re = 100$. The lower stability limit is around 50 in this case. Usually the characteristic length is chosen as the diameter of the cylinder. The number of lattices along the diameter was $n = 12$ (shown in Figure 19(b)). The lattice viscosity was $\nu_{\text{lattice}} = 0.01$ so the lattice velocity was $u_{\text{lattice}} \approx 0.083$. At the inlet a parabolic profile was prescribed, and the same profile was initialized along the channel. At the outlet the first order opening boundary condition was prescribed. The simulation with the BGKW model was run until $10,000$ iterations.

Figure 26 presents two pictures at different time instants. Whilst the velocity field is symmetric in Figure 26(a), the streamlines show an asymmetric flow in Figure 26(b). The two vortices behind the cylinder in the separation zone broke the stability between 400 and 500 iterations. Then one of the vortices grew and a periodic vortex shedding started. The series of vortices is named after Theodore von Kármán (von Kármán vortex street). The phenomenon was investigated both experimentally [114] and numerically [115] several times. The presented results are qualitatively in good agreement with the expectations. An advanced analysis of the process could be done based on the dimensionless Strouhal number. The Strouhal number is

given by $St = FL/u$, where F is the frequency of the shedding, L is the characteristic length and u is the velocity peak of the developed profile. Viggen provided a detailed description of the rescaling the dimensionless LBM results [116].



(a) Flow field after 400 iterations (BGKW)



(b) Flow field after 7,000 iterations (BGKW)

Figure 26: Formation of the von Kármán vortex street after the cylinder

5 Conclusion and outlook

Two existing 2D laminar lattice Boltzmann solvers were reformulated successfully from C++ to the C language. The advantages of the two codes were combined. In addition the new serial code has some new features which makes it more user-friendly: i) *.ini file was added which serves as an easy way to define the basic settings; ii) at the end of the simulations the solver creates a *.log file to store the most important data of the simulations, including time measurement results of the main steps; iii) the user can choose between two different output file format (Tecplot and Paraview).

A detailed literature survey was carried out to research and present theoretical basis of the lattice Boltzmann method and how it has been applied. Furthermore the historical background of high performance computing was briefly analysed both on the CPU and the GPU side. The author summarized the most important features of the applied physical approach and the parallelization technique with particular emphasizes on the advantages and disadvantages.

After a short validation of the serial code, the reformulated code was parallelized with a novel approach, namely general-purpose computing on graphics processing units. To this end nVidia's CUDA software development kit was used. To identify the most time consuming part of the code, a detailed profiling was performed with classical, invasive measurement techniques. The development was demonstrated through sample code sections in detail.

Finally six of the seven basic operations were executed on the GPU. The code was analysed through profiling and the author highlighted that the computational costs of the different collision models are significantly different. The structure of the GPU makes some basic serial algorithms, like "scan" difficult. (Summation of every element of a vector is called "scan" and it is necessary e. g. for the residual computation.) These parts of the code were found quite inefficient but the problem could be solved by further optimization using the shared memory. A description of an efficient "scan" algorithm was found in the literature.

Thanks to the parallel structure of the lattice Boltzmann method and the massively parallel structure of the GPU, the parallel code was considerably faster. Two machines (with different CPUs and GPUs) were tested and the new code proved more efficient in every case. The performance peak of the GPU on the first machine was approximately nine times higher compared to the serial execution. This machine was a simple personal computer. The second machine was the Fermi GPU cluster of Cranfield university. The GPU of the cluster was almost three times faster in general, and a speed-up maximum between fourteen and sixteen was reported for the less expensive collision models (BGKW and TRT). (The speed-up of the expensive collision model (MRT) was between four and ten for the different grids.) The double precision execu-

tion increased the simulation time by 30% when the GPU was used, but it was only 5% higher with the CPU. All in all the parallelization can be considered efficient and the GPU proved an efficient high performance tool compared to multiple CPUs. A GPU, designed for scientific computing might be equivalent to a workstation. The solver had good scalability. Further optimization would be required to obtain higher performance. One could overcome the relatively strong memory limitation of one single graphical card with multiple GPU implementation.

The solver was verified with the Poisseuile flow test case. A grid convergence study was performed to check the different models. The BGKW and the MRT models captured the parabolic profile quite accurately and showed good convergence properties. The TRT model probably needs much higher spatial resolution compared to the other models. With the investigated resolution the TRT model failed to resolve the flow field.

On the physical side a new boundary condition was implemented. The second order formulation of the opening outlet boundary condition proved unstable. Especially for higher Reynolds number the BGKW model gave a distorted, non-physical flow field at the outlet, which often led to divergence. As an alternative to the second order formulation a simple first order opening outlet was introduced and tested. The new boundary condition proved more stable and reliable during the validation.

The solver was validated through basic test cases both qualitatively and quantitatively. The lid-driven cavity, the backward facing step, the sudden expansion, and the flow around a cylinder were investigated as basic test cases for 2D solvers. The results were compared to the available data. On the one hand, the models showed very similar characteristics. On the other hand they were able to capture the steady and unsteady flow features including stability loss and the von Kármán vortex street. The BGK and the MRT models were more accurate than the TRT.

The current study suggests several way to continue the research. One direction of future development could be extension of the code to include additional physics. Turbulence modelling would be essential for practical use. As mentioned in the literature survey, the lattice Boltzmann method is suitable for multiphase and reactive flows. The implementation of a fluid-structure interaction solver based on the 2D lattice Boltzmann solver would be a real challenge. Since flows are essentially 3D, a next step could be to extend the meshing and the solver to 3D cases.

There are several further tasks on the programming-coding side too. The optimization of the code could lead to further acceleration. The goal should be a fully parallel code. The shared memory is currently an untapped feature of the GPU which could release more performance gain in. With multiple GPUs one could get rid of the embarrassing memory (mesh size) barriers. The solver and the meshing itself is limited to simple geometries so the solver should be generalized.

References

[1] J. Sanders and E. Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[2] A. A. Mohamad. *Lattice Boltzmann method*. Springer, 2011.

[3] S. Succi. *The lattice Boltzmann equation: for fluid dynamics and beyond*. Oxford university press, 2001.

[4] S. Chen and G. D. Doolen. Lattice Boltzmann method for fluid flows. *Annual Review of Fluid Mechanics*, 30(1):329–364, 1998.

[5] T.-R. Teschner. Development of a two dimensional fluid solver based on the lattice Boltzmann method. Master's thesis, Cranfield University, 2013.

[6] G. Abbruzzese. Development of a 2D lattice Boltzmann code. Master's thesis, Cranfield University, 2013.

[7] W.-W. Kim, S. Menon, and H. C. Mongia. Large-eddy simulation of a gas turbine combustor flow. *Combustion Science and Technology*, 143(1-6):25–62, 1999.

[8] M. A. Trigg, G. R. Tubby, and A. G. Sheard. Automatic genetic optimization approach to two-dimensional blade profile design for steam turbines. *Journal of Turbomachinery*, 121(1):11–17, 1999.

[9] M. O. L. Hansen, J. N. Sørensen, S. Voutsinas, N. Sørensen, and H. A. Madsen. State of the art in wind turbine aerodynamics and aeroelasticity. *Progress in Aerospace Sciences*, 42(4):285–330, 2006.

[10] A. Datta, J. Sitaraman, I. Chopra, and J. D. Baeder. CFD/CSD prediction of rotor vibratory loads in high-speed flight. *Journal of Aircraft*, 43(6):1698–1709, 2006.

[11] F. T. Johnson, E. N. Tinoco, and N. J. Yu. Thirty years of development and application of CFD at Boeing Commercial Airplanes, Seattle. *Computers & Fluids*, 34(10):1115–1151, 2005.

[12] F. Muyl, L. Dumas, and V. Herbert. Hybrid method for aerodynamic shape optimization in automotive industry. *Computers & Fluids*, 33(5):849–858, 2004.

[13] Y. Ra and R. D. Reitz. A reduced chemical kinetic model for IC engine combustion simulations with primary reference fuels. *Combustion and Flame*, 155(4):713–738, 2008.

[14] J. G. Bene and I. Selek. Water network operational optimization: Utilizing symmetries in combinatorial problems by dynamic programming. *Civil Engineering*, 56(1):51–61, 2012.

[15] G. Závodszky and Gy. Paál. Validation of a lattice Boltzmann method implementation for a 3D transient fluid flow in an intracranial aneurysm geometry. *International Journal of Heat and Fluid Flow*, 44:276–283, 2013.

[16] N. Nowak, P. P. Kakade, and A. V. Annapragada. Computational fluid dynamics simulation of airflow and aerosol deposition in human lungs. *Annals of Biomedical Engineering*, 31(4):374–390, 2003.

[17] M. Naitoh, S. Uchida, S. Koshizuka, H. Ninokata, N. Hiranuma, K. Dosaki, K. Nishida, M. Akiyama, and H. Saitoh. Evaluation methods for corrosion damage of components in cooling systems of nuclear power plants by coupling analysis of corrosion and flow dynamics (I) Major targets and development strategies of the evaluation methods. *Journal of Nuclear Science and Technology*, 45(11):1116–1128, 2008.

[18] J. Choi, Y. Kim, A. Sivasubramaniam, J. Srebric, Q. Wang, and J. Lee. A CFD-based tool for studying temperature in rack-mounted servers. *Computers*, 57(8):1129–1142, 2008.

[19] H. Wijshoff. Free surface flow and acousto-elastic interaction in piezo inkjet. In *Proc. NSTI Nanotechnology Conf. and Trade Show*, volume 2, pages 215–218, 2004.

[20] J. D. Anderson et al. *Computational fluid dynamics*, volume 206. Springer, 1995.

[21] M. Gad-el Hak. The fluid mechanics of microdevicesthe Freeman scholar lecture. *Journal of Fluids Engineering*, 121(1):5–33, 1999.

[22] C. K. Aidun and J. R. Clausen. Lattice-Boltzmann method for complex flows. *Annual Review of Fluid Mechanics*, 42:439–472, 2010.

[23] X. He, S. Chen, and R. Zhang. A lattice Boltzmann scheme for incompressible multiphase flow and its application in simulation of rayleigh–taylor instability. *Journal of Computational Physics*, 152(2):642–663, 1999.

[24] X. He and G. D. Doolen. Thermodynamic foundations of kinetic theory and lattice Boltzmann models for multiphase flows. *Journal of Statistical Physics*, 107(1-2):309–328, 2002.

[25] S. H. Kim and H. Pitsch. On the lattice Boltzmann method for multiphase flows. *Center for Turbulence Research, Annual Research Briefs*, 2009.

[26] H. W. Zheng, C. Shu, and Y. T. Chew. A lattice Boltzmann model for multiphase flows with large density ratio. *Journal of Computational Physics*, 218(1):353–371, 2006.

[27] O. Filippova and D. Hänel. Lattice-Boltzmann simulation of gas-particle flow in filters. *Computers & Fluids*, 26(7):697–712, 1997.

[28] D. L. Marchisio and R. O. Fox. *Multiphase reacting flows: modelling and simulation*, volume 492. Springer, 2007.

[29] S. P. Dawson, S. Chen, and G. D. Doolen. Lattice Boltzmann computations for reaction-diffusion equations. *The Journal of Chemical Physics*, 98(2):1514–1523, 1993.

[30] L. Biferale, F. Mantovani, M. Sbragaglia, A. Scagliarini, F. Toschi, and R. Tripiccione. Reactive Rayleigh-Taylor systems: Front propagation and non-stationarity. *EPL (Europhysics Letters)*, 94(5):54004, 2011.

[31] C. Pan, L.-S. Luo, and C. T. Miller. An evaluation of lattice Boltzmann schemes for porous medium flow simulation. *Computers & Fluids*, 35(8):898–909, 2006.

[32] C. Pan, M. Hilpert, and C. T. Miller. Lattice-Boltzmann simulation of two-phase flow in porous media. *Water Resources Research*, 40(1), 2004.

[33] A. M. M. Artoli. *Mesoscopic computational haemodynamics*. Ponsen en Looijen, 2003.

[34] M. Hirabayashi, Makoto O., D. A. Rüfenacht, and B. Chopard. A lattice boltzmann study of blood flow in stented aneurism. *Future Generation Computer Systems*, 20(6):925–934, 2004.

[35] Y. H. Kim, X. Xu, and J. S. Lee. The effect of stent porosity and strut shape on saccular aneurysm and its numerical analysis with lattice Boltzmann method. *Annals of Biomedical Engineering*, 38(7):2274–2292, 2010.

[36] R. Ouared and B. Chopard. Lattice Boltzmann simulations of blood flow: non-Newtonian rheology and clotting processes. *Journal of Statistical Physics*, 121(1-2):209–221, 2005.

[37] M. Keijo. *Implementation techniques for the lattice Boltzmann method*. University of Jyvaskyla, 2010.

[38] M. Garcia, J. Gutierrez, and N. Rueda. Fluid–structure coupling using lattice-Boltzmann and fixed-grid FEM. *Finite Elements in Analysis and Design*, 47(8):906–912, 2011.

[39] E. Fares. Unsteady flow simulation of the Ahmed reference body using a lattice boltzmann approach. *Computers & Fluids*, 35(8):940–950, 2006.

[40] J.-C. Desplat, I. Pagonabarraga, and P. Bladon. LUDWIG: A parallel Lattice-Boltzmann code for complex fluids. *Computer Physics Communications*, 134(3):273–290, 2001.

[41] J. Tölke. Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA. *Computing and Visualization in Science*, 13(1):29–39, 2010.

[42] J. Habich, T. Zeiser, G. Hager, and G. Wellein. Performance analysis and optimization strategies for a D3Q19 lattice Boltzmann kernel on nVIDIA GPUs using CUDA. *Advances in Engineering Software*, 42 (5):266–272, 2011.

[43] W. Xian and A. Takayuki. Multi-GPU performance of incompressible flow computation by lattice Boltzmann method on GPU cluster. *Parallel Computing*, 37(9):521–535, 2011.

[44] K. A. Hoffmann and Steve. T. Chiang. *Computational fluid dynamics, Vol. 1*. Wichita, KS: Engineering Education System, 2000.

[45] W. H Press. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[46] L.-S. Luo. *Lattice-gas automata and lattice Boltzmann equation for two-dimensional hydrodynamics*. PhD thesis, Georgia Institute of Technology, 1993.

[47] A. J. Chorin. A numerical method for solving incompressible viscous flow problems. *Journal of Computational Physics*, 2(1):12–26, 1967.

[48] Computer History Museum. URL `www.computerhistory.org`. Accessed 20.08.2014.

[49] Cray. URL `www.cray.com`. Accessed 20.08.2014.

[50] S. H. Lavington. *A history of Manchester computers (2nd edition)*. NCC Publications Manchester, 1998.

[51] M. D. Hill, N. P. Jouppi, and G. Sohi. *Readings in computer architecture*. Gulf Professional Publishing, 2000.

[52] Top 500 supercomputer. URL `www.top500.org/`. Accessed 20.08.2014.

[53] Moore's law. URL `www.mooreslaw.org/`. Accessed 20.08.2014.

[54] S. Naffziger. Technology impacts from the new wave of architectures for media-rich workloads. In *VLSI Technology (VLSIT), 2011 Symposium on*, pages 6–10. IEEE, 2011.

[55] Riken. URL `www.riken.jp/en/pr/topics/2013/20131226_1/`. Accessed 20.08.2014.

[56] Spectrum. URL `http://spectrum.ieee.org`. Accessed 20.08.2014.

[57] Mont-Blanc Project. URL `www.montblanc-project.eu/`. Accessed 20.08.2014.

[58] ARM. URL `www.arm.com/`. Accessed 20.08.2014.

[59] Samsung Exynos. URL `www.samsung.com/exynos`. Accessed 20.08.2014.

[60] nVidia. URL `www.nvidia.co.uk`. Accessed 20.08.2014.

[61] N. Rajovic, A. Rico, J. Vipond, I. Gelado, N. Puzovic, and A. Ramirez. Experiences with mobile processors for energy efficient HPC. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 464–468. EDA Consortium, 2013.

[62] N. Rajovic, P. M. Carpenter, I. Gelado, N. Puzovic, A. Ramirez, and M. Valero. Supercomputing with commodity CPUs: are mobile SoCs ready for HPC? In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, page 40. ACM, 2013.

[63] N. Rajovic, L. Vilanova, C. Villavieja, N. Puzovic, and A. Ramirez. The low power architecture approach towards exascale computing. *Journal of Computational Science*, 4(6):439–443, 2013.

[64] D. Göddeke, D. Komatitsch, M. Geveler, D. Ribbrock, N. Rajovic, N. Puzovic, and A. Ramirez. Energy efficiency vs. performance of the numerical solution of PDEs: an application study on a low-power ARM-based cluster. *Journal of Computational Physics*, 237:132–150, 2013.

[65] Parallella. URL `http://www.parallella.org`. Accessed 20.08.2014.

[66] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[67] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering*, 5(1):46–55, 1998.

[68] R. Tsuchiyama, T. Nakamura, T. Iizuka, A. Asahara, S. Miki, and S. Tagawa. The OpenCL Programming Book. *Fixstars Corporation*, 63, 2010.

[69] R. W. Numrich and J. Reid. Co-Array Fortran for parallel programming. In *ACM Sigplan Fortran Forum*, volume 17, pages 1–31. ACM, 1998.

[70] T. El-Ghazawi and L. Smith. UPC: unified parallel C. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 27. ACM, 2006.

[71] M. Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL programming guide: the official guide to learning OpenGL, version 1.2*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[72] Techspot. URL `www.techspot.com/article/650-history-of-the-gpu/`. Accessed 20.08.2014.

[73] M. Swaine. New chip from Intel gives high-quality displays. 1983.

[74] C. McClanahan. History and evolution of GPU architecture. *A Paper Survey*, 2010.

[75] T. S. Crow. *Evolution of the graphical processing unit*. PhD thesis, Citeseer, 2004.

[76] MaximumPC. URL `www.maximumpc.com/print/6338`. Accessed 20.08.2014.

[77] S. Venkatasubramanian. Understanding the graphics pipeline. *Lecture 2*, 2005.

[78] Intel. URL `www.intel.co.uk`. Accessed 20.08.2014.

[79] Portland Group. URL `www.pgroup.com/lit/articles/insider/v2n1a5.htm`. Accessed 20.08.2014.

[80] Michael Galloy's webpage. URL `http://michaelgalloy.com/2013/06/11/cpu-vs-gpu-performance.html`. Accessed 20.08.2014.

[81] Cornell University. URL `www.cac.cornell.edu/VW/gpu/memory_arch.aspx`. Accessed 20.08.2014.

[82] Y. Okitsu, F. Ino, and K. Hagihara. High-performance cone beam reconstruction using CUDA compatible GPUs. *Parallel Computing*, 36(2):129–141, 2010.

[83] Scopus. URL `www.scopus.com`. Accessed 06.08.2014.

[84] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. 6(2):40–53, 2008.

[85] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium*, pages 163–174. IEEE, 2009.

[86] H. Scherl, B. Keck, M. Kowarschik, and J. Hornegger. Fast GPU-based CT reconstruction using the common unified device architecture (CUDA). In *Nuclear Science Symposium Conference Record, 2007. NSS'07. IEEE*, volume 6, pages 4464–4466. IEEE, 2007.

[87] S. S. Stone, J. P. Haldar, S. C. Tsao, W.-M. Hwu, B. P. Sutton, Z.-P. Liang, et al. Accelerating advanced MRI reconstructions on GPUs. *Journal of Parallel and Distributed Computing*, 68(10):1307–1318, 2008.

[88] I. Tanno, K. Morinishi, N. Satofuka, and Y. Watanabe. Calculation by artificial compressibility method and virtual flux method on GPU. *Computers & Fluids*, 45(1):162–167, 2011.

[89] J. Mielikainen, B. Huang, H. A. Huang, and M. D. Goldberg. Improved GPU/CUDA based parallel weather and research forecast (WRF) single moment 5-class (WSM5) cloud microphysics. *Selected Topics in Applied Earth Observations and Remote Sensing*, 5(4):1256–1265, 2012.

[90] G. R. Joldes, A. Wittek, and K. Miller. Real-time nonlinear finite element computations on GPU–Application to neurosurgical simulation. *Computer Methods in Applied Mechanics and Engineering*, 199 (49):3305–3314, 2010.

[91] Z. A. Taylor, O. Comas, M. Cheng, J. Passenger, D. J. Hawkes, D. Atkinson, and S. Ourselin. On modelling of anisotropic viscoelasticity for soft tissue simulation: Numerical solution and GPU execution. *Medical Image Analysis*, 13(2):234–244, 2009.

[92] S. A. Manavski and G. Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics*, 9(Suppl 2):S10, 2008.

[93] J. Liang. Parallel direct simulation Monte Carlo using graphics processing unit with CUDA. In *Parallel Computational Fluid Dynamics*, pages 354–362. Springer, 2014.

[94] M. J. Goldsworthy. A GPU–CUDA based direct simulation Monte Carlo algorithm for real gas flows. *Computers & Fluids*, 94:58–68, 2014.

[95] A. F. Shinn, S. P. Vanka, and W. W. Hwu. Direct numerical simulation of turbulent flow in a square duct using a graphics processing unit (GPU). In *40th AIAA Fluid Dynamics Conference*, 2010.

[96] S. S. Chikatamarla, C. E. Frouzakis, I. V. Karlin, A. G. Tomboulides, and K. B. Boulouchos. Lattice Boltzmann method for direct numerical simulation of turbulent flows. *Journal of Fluid Mechanics*, 656: 298–308, 2010.

[97] H. Yu, S. S. Girimaji, and L.-S. Luo. DNS and LES of decaying isotropic turbulence with and without frame rotation using lattice Boltzmann method. *Journal of Computational Physics*, 209(2):599–616, 2005.

[98] Q. Xiong, B. Li, J. Xu, X. Wang, L. Wang, and W. Ge. Efficient 3D DNS of gas–solid flows on Fermi GPGPU. *Computers & Fluids*, 70:86–94, 2012.

[99] G. R. McNamara and G. Zanetti. Use of the Boltzmann equation to simulate lattice-gas automata. *Physical Review Letters*, 61(20):2332, 1988.

[100] R. Mei, L.-S. Luo, and W. Shyy. An accurate curved boundary treatment in the lattice boltzmann method. *Journal of Computational Physics*, 155(2):307–330, 1999.

[101] R. Mei, W. Shyy, D. Yu, and L.-S. Luo. Lattice Boltzmann method for 3-D flows with curved boundary. *Journal of Computational Physics*, 161(2):680–699, 2000.

[102] Q. Zou and X. He. On pressure and velocity boundary conditions for the lattice Boltzmann BGK model. *Physics of Fluids*, 9(6):1591–1598, 1997.

[103] J. Bernsdorf, S. E. Harrison, S. M. Smith, and P. V. Lawford. Concurrent numerical simulation of flow and blood clotting using the lattice Boltzmann technique. *International journal of Bioinformatics Research and Applications*, 2(4):371–380, 2006.

[104] M. Szőke. Efficient implementation of a 2d lattice boltzmann solver using modern parallelisation techniques. Master's thesis, Cranfield University, 2014.

[105] M. Harris, Shubhabrata S., and J. D. Owens. Parallel prefix sum (scan) with CUDA. *GPU gems*, 3(39): 851–876, 2007.

[106] S. Sengupta, M. Harris, and M. Garland. Efficient parallel scan algorithms for GPUs. *NVIDIA, Santa Clara, CA, Technical Report NVR-2008-003*, (1):1–17, 2008.

[107] P. J. Roache. Quantification of uncertainty in computational fluid dynamics. *Annual Review of Fluid Mechanics*, 29(1):123–160, 1997.

[108] NASA. URL `http://www.grc.nasa.gov/WWW/wind/valid/tutorial/spatconv.html`. Accessed 20.08.2014.

[109] A. K. Prasad and J. R. Koseff. Reynolds number and end-wall effects on a lid-driven cavity flow. *Physics of Fluids*, 1(2):208–218, 1989.

[110] U. Ghia, K. N. Ghia, and C. T. Shin. High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method. *Journal of Computational Physics*, 48(3):387–411, 1982.

[111] X. He and L.-S. Luo. Lattice Boltzmann model for the incompressible Navier–Stokes equation. *Journal of Statistical Physics*, 88(3-4):927–944, 1997.

[112] B. F. Armaly, F. Durst, J. C. F. Pereira, and B. Schönung. Experimental and theoretical investigation of backward-facing step flow. *Journal of Fluid Mechanics*, 127:473–496, 1983.

[113] R. M. Fearn, T. Mullin, and K. A. Cliffe. Nonlinear flow phenomena in a symmetric sudden expansion. *Journal of Fluid Mechanics*, 211:595–608, 1990.

[114] C. Y. Wen and C. Y. Lin. Two-dimensional vortex shedding of a circular cylinder. *Physics of Fluids*, 13(3): 557–560, 2001.

[115] M. Schäfer, S. Turek, F. Durst, E. Krause, and R. Rannacher. *Benchmark computations of laminar flow around a cylinder*. Springer, 1996.

[116] E. M. Viggen. The lattice Boltzmann method in acoustics. In *Proc. 33rd Scandinavian Symp. on Physical Acoustics, Geilo, Norway, 7–10 February 2010*, 2010.

Appendix A: The properties of the computers

properties	Machine 1 (M1)	Machine 2 (M2)
Operation system	Ubuntu 14.04	Ubuntu 12.04
Processor type (CPU)	Intel i7-3820	Intel Xeon E5620
Processor clock rate	3.6 GHz	2.4 GHz
Processor cores (threads)	4 (8)	4 (8)
Processor cache size	10 MB	12 MB
Host memory type	4 x Kingston blu DDR3	unknown
Host memory size	16 GB	24 GB
Device type (GPU)	nVidia GeForce GTX 550 Ti	nVidia Tesla C2050
CUDA capability version number	2.1	2.0
Device clock rate	1940 MHz	1147 MHz
Device multiprocessors	6	14
Device CUDA cores	192	448
Maximum block size (x,y,z)	$(1024, 1024, 64)$	$(1024, 1024, 64)$
Maximum grid size (x,y,z)	$(65535, 65535, 65535)$	$(65535, 65535, 65535)$
Global memory type	DDR5	DDR5
Global memory size	1 GB	3 GB
Shared memory size (per block)	49 MB	49 MB
Constant memory size	65 MB	65 MB
Texture memory size	390 MB	786 MB
Warp size	32	32
Double precision	YES	YES

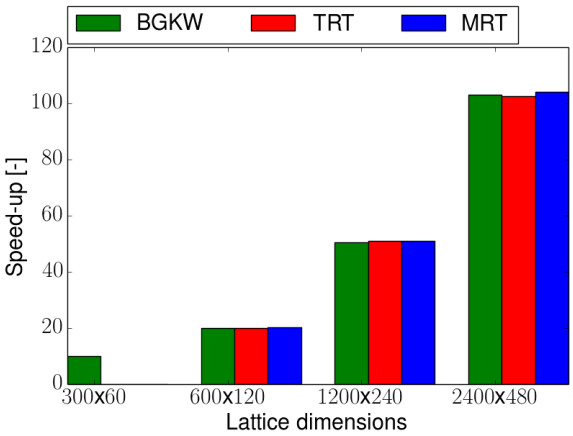
Appendix B: The sample CUDA code

```
// nvcc sample_cuda_code.cu -o parallel && ./parallel
#include <stdio.h> // writing & reading data
#include <stdlib.h> // dynamic memory allocation
#include <time.h> // time measurement
#include <cuda.h> // CUDA functions


#define W 500 // define width of the matrix
#define H 125 // define height of the matrix

// GPU grid and block size
#define tx 32 // number of threads in the x direction
#define ty 32 // number of threads in the y direction
#define bx 7 // number of blocks in the x direction
#define by 5 // number of blocks in the y direction
#define bz 2 // number of blocks in the z direction


// declare and allocate parameter in the GPU's constant memory
__constant__ float c_d[1];


///// GPU KERNEL FUNCTION /////
__global__ void sg( float *a, float *b)
{

  int bidx=blockIdx.x; // index of the actual block (x direction)
  int bidy=blockIdx.y; // index of the actual block (y direction)
  int bidz=blockIdx.z; // index of the actual block (z direction)
  int tidx=threadIdx.x; // index of the actual thread (x direction)
  int tidy=threadIdx.y; // index of the actual in the (y direction)

  // index along the vector
  int ind = tidx + bidx*blockDim.x + tidy*(blockDim.x*gridDim.x)
          + bidy*(blockDim.x*blockDim.y*gridDim.x)
          + bidz*(blockDim.x*blockDim.y*gridDim.x*gridDim.y);

  if (ind<W*H) //execution only within the vector
  {
    // if statement --> no computations on the ''boundaries''

    if (ind>(W-1) && ind<(W*(H-1)) && (ind % W)!= 0 && ((ind+1) % W)!= 0)
    {
      // sum the neigbouring cells and multiply by a constant
      b[ind]=(*c_d)*(a[ind+1]+a[ind-1]+a[ind-W]+a[ind+W]);
    }
  }
}


///// THE MAIN FUNCTION /////
int main()
{

  // declare variables
  int i, j; // declare for loop variables

  int width=W, height=H; // declare width and height of the matrix

  float **matrix_o, **matrix_n; // declare host matrices

  float *vec_o, * vec_n; // declare host vectors
  float *dev_vec_o, *dev_vec_n; // declare device vectors

  float *c_h; // declare host variable for GPU's constant memory

  FILE * fp1; // declare file pointer to output file

  // C time measurement variables
  clock_t tInstant1, tInstant2;

  // cuda time measurement variables
  cudaEvent_t start1, start2, start3, stop1, stop2, stop3;
```

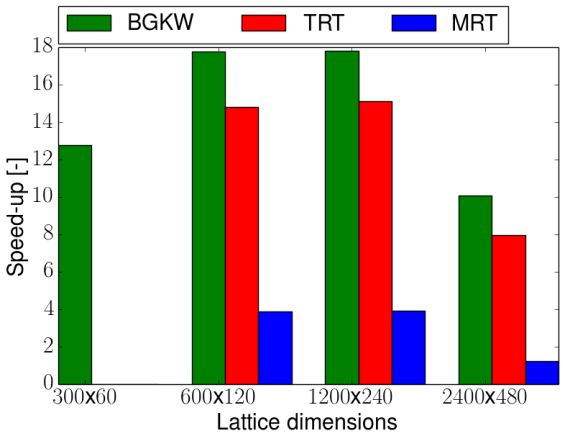
```
74  float cudatime;
75  cudaEventCreate(&start1);cudaEventCreate(&start2);
76  cudaEventCreate(&start3);
77  cudaEventCreate(&stop1);cudaEventCreate(&stop2);cudaEventCreate(&stop3);
78
79  float tinit = 0.0, tGPU_ex=0.0, tGPU_cop=0.0, tGPU=0.0;  // elapsed time
     of CPU and GPU executions
80
81  // allocate host matrices
82  matrix_o = (float **)calloc(height,sizeof(float*));
83  for (i = 0; i < height; i++)
84    matrix_o[i] = (float *)calloc(width,sizeof(float));
85
86  matrix_n = (float **)calloc(height,sizeof(float*));
87  for (i = 0; i < height; i++)
88    matrix_n[i] = (float *)calloc(width,sizeof(float));
89
90  // allocate host vectors
91  c_h = (float *)calloc(1,sizeof(float*));
92  vec_o = (float *)calloc(height*width,sizeof(float*));
93  vec_n = (float *)calloc(height*width,sizeof(float*));
94
95  // allocate device vectors
96  cudaMalloc((void**)&dev_vec_o, height * width * sizeof(float));
97  cudaMalloc((void**)&dev_vec_n, height * width * sizeof(float));
98
99  *c_h=0.1; // define the value of the constant
100
101 tInstant1 = clock(); // start classical time measurement
102
103 for (i = 0; i < height; i++)
104 {
105   for (j = 0; j < width; j++)
106   {
107     matrix_o[i][j]=i*width+j;  // initializations
108   }
109 }
110
111 tInstant2 = clock(); // stop classical time measurement
112 tinit = (float)(tInstant2-tInstant1) / CLOCKS_PER_SEC;
113
114 dim3 tpb(tx, ty, 1); // define number of threads/blocks
115 dim3 bpg(bx, by, bz); // define number of blocks/grid
116
117 cudaEventRecord(start1, 0); // start cuda time measurement
118
119 for (i = 0; i < height; i++)
120 {
121   for (j = 0; j < width; j++)
122   {
123     vec_o[i*width+j]=matrix_o[i][j]; // convert matrix to vector
124   }
125 }
126
127 cudaEventRecord(start2, 0);
128
129 // copy to constant memory
130 cudaMemcpyToSymbol(c_d, c_h, sizeof(int));
131
132 // copy vectors to the device
133 cudaMemcpy(dev_vec_o, vec_o, height*width*sizeof(float),
     cudaMemcpyHostToDevice);
134 cudaMemcpy(dev_vec_n, vec_n, height*width*sizeof(float),
     cudaMemcpyHostToDevice);
135
136 cudaEventRecord(start3, 0);
137
138 // kernel function call
139 sg<<<bpg,tpb>>>(dev_vec_o, dev_vec_n);
140
141 cudaEventRecord(stop3, 0); cudaEventSynchronize(stop3);
142 cudaEventElapsedTime(&cudatime, start3, stop3);
143 tGPU_ex = cudatime;
144
145 // copy results back to host
```

```
146   cudaMemcpy(vec_n, dev_vec_n, height*width*sizeof(float),
        cudaMemcpyDeviceToHost);
147
148   cudaEventRecord(stop2, 0); cudaEventSynchronize(stop2);
149   cudaEventElapsedTime(&cudatime, start2, stop2);
150   tGPU_cop = cudatime;
151
152   for (i = 0; i < height; i++)
153   {
154     for (j = 0; j < width; j++)
155     {
156       matrix_n[i][j]=vec_n[i*width+j]; // convert vector to matrix
157     }
158   }
159
160   cudaEventRecord(stop1, 0); cudaEventSynchronize(stop1); // stop cuda time
        measurement
161   cudaEventElapsedTime(&cudatime, start1, stop1);
162   tGPU = cudatime;
163
164   printf("Initialization needed %f ms\n", tinit*1000);
165   printf("Main step needed %f ms\n", tGPU_ex);
166   printf("Data copying needed %f ms\n", tGPU_cop-tGPU_ex);
167   printf("Conversion needed %f ms\n", tGPU-tGPU_cop);
168
169   fp1=fopen("test_GPU.dat", "w"); // open file
170
171   // write results to the file
172   for(i=0;i<height;i++)
173   {
174     for(j=0;j<width;j++)
175     {
176       fprintf(fp1, "%f ", matrix_n[i][j]);
177     }
178     fprintf(fp1, "\n");
179   }
180   fclose(fp1); // close file
181
182   free(matrix_o);
183   free(matrix_n);
184   free(vec_o);
185   free(vec_n);
186   free(c_h);
187
188   cudaFree(dev_vec_o);
189   cudaFree(dev_vec_n);
190   cudaFree(c_d);
191
192   return 0;
193 }
```

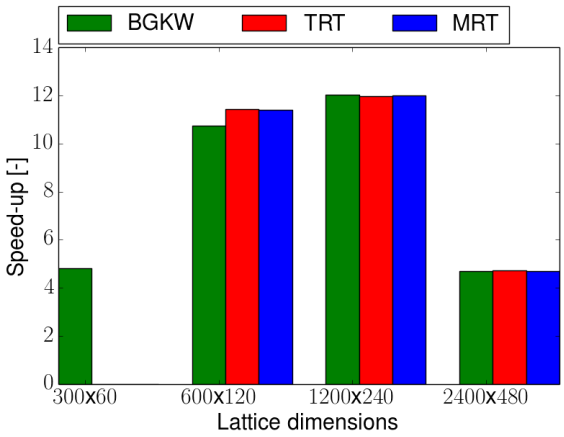
Appendix C: Speed-up analysis of machine M1



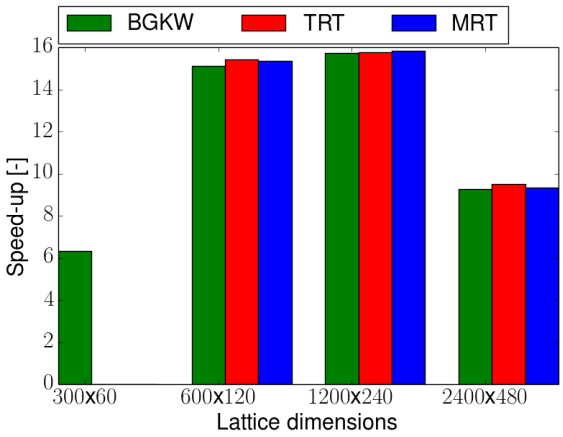
(a) Initialization – single precision



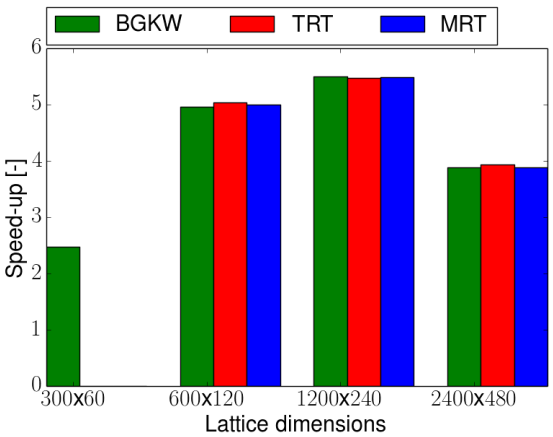
(b) Collision – single precision



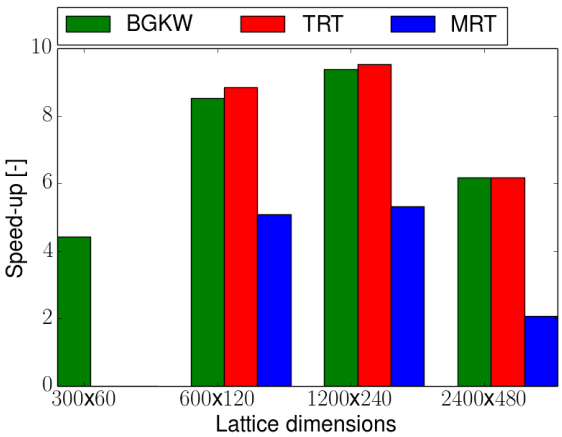
(c) Update f – single precision



(d) Streaming – single precision



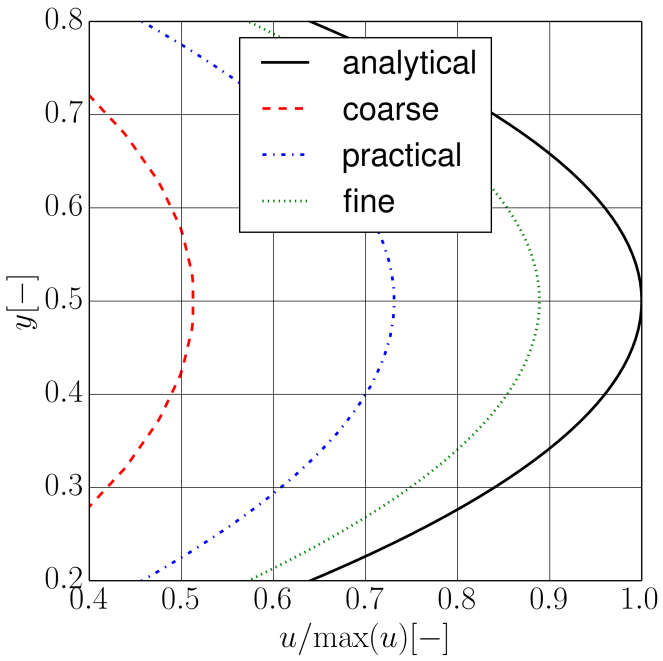
(e) Boundary treatment – single precision



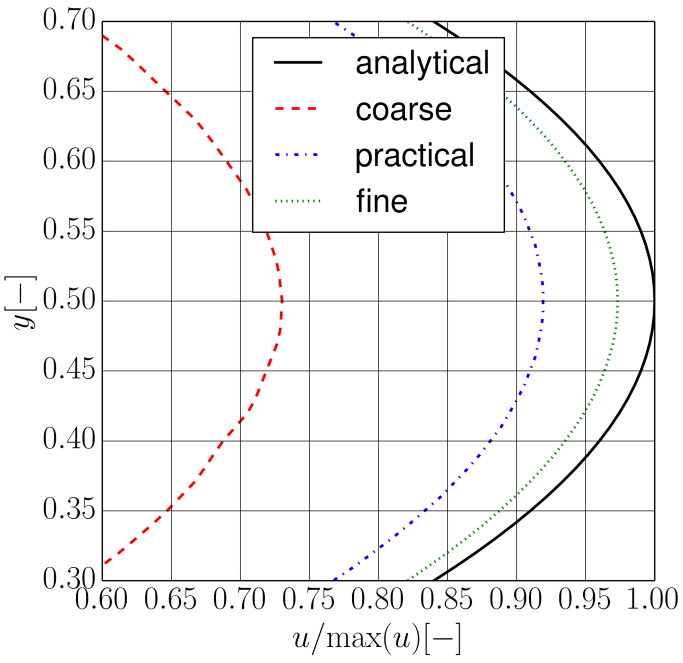
(f) Main loop – single precision

Speed up of the different sections on machine M1

Appendix D: Convergence of the TRT and the MRT models (channel flow)

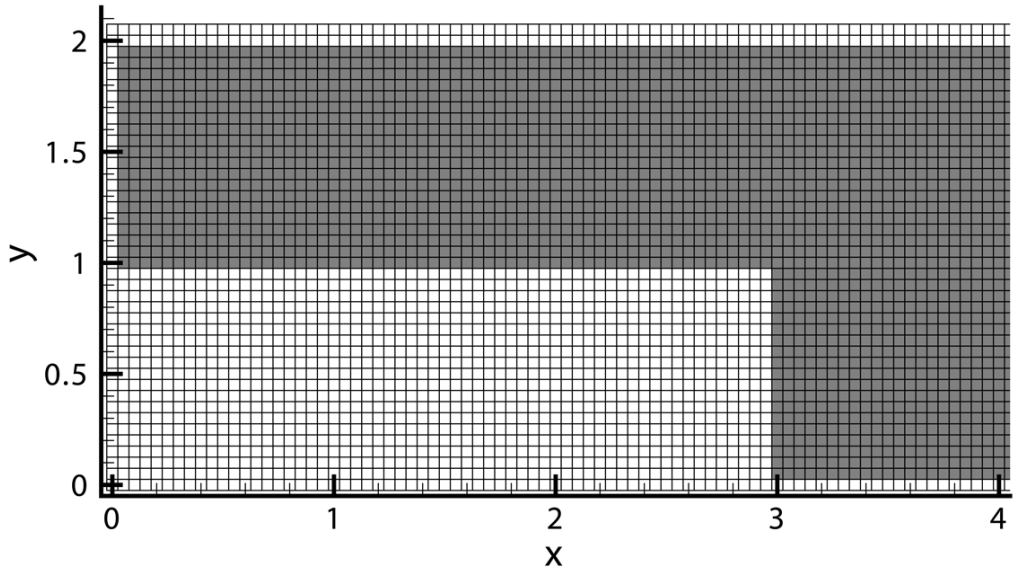


(a) TRT

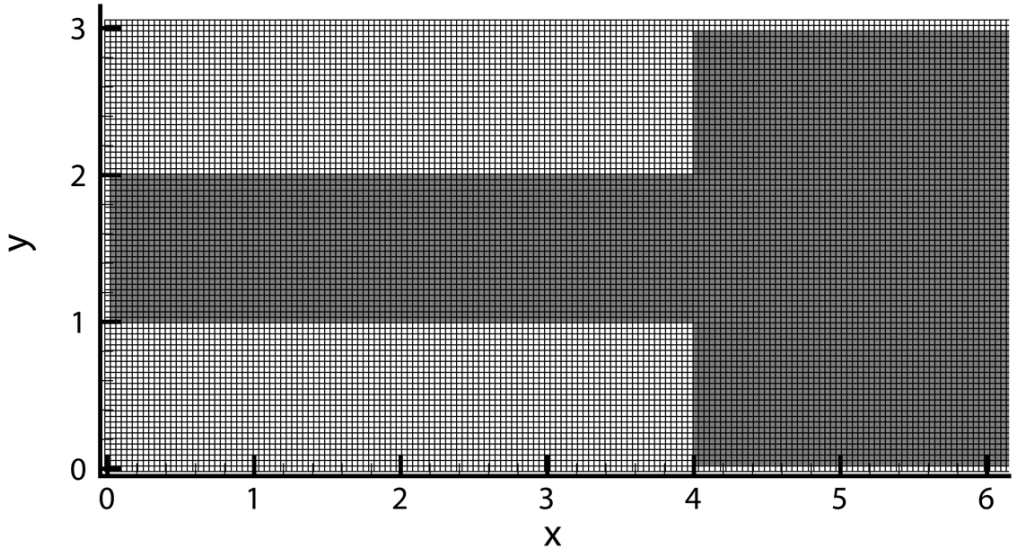


(b) MRT

Appendix E: Further details of the applied grids



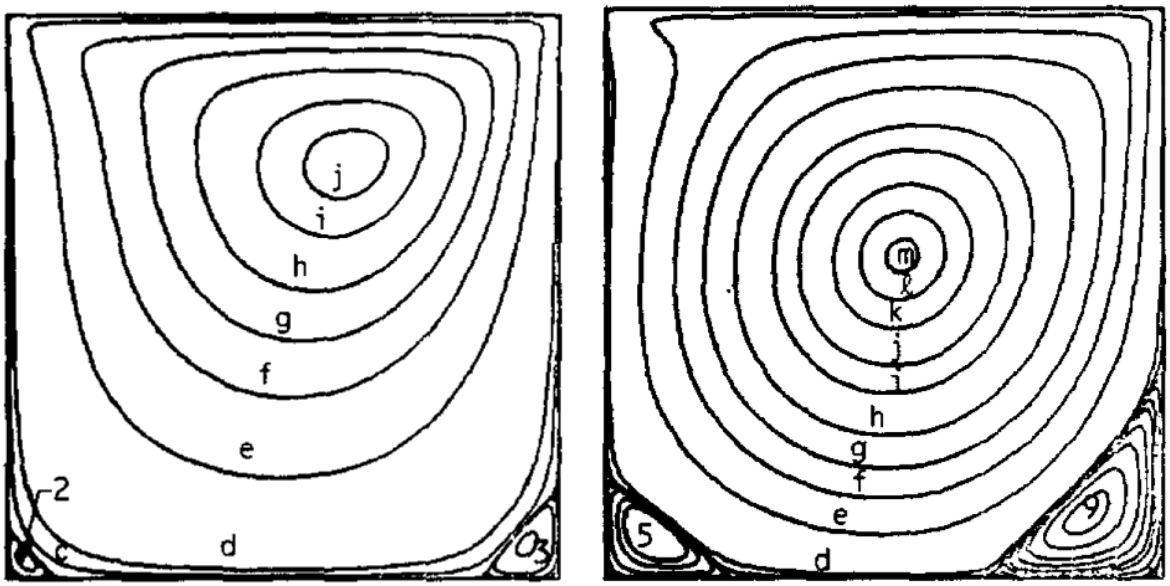
(a) Mesh of the backward facing step around the inlet (402×43)



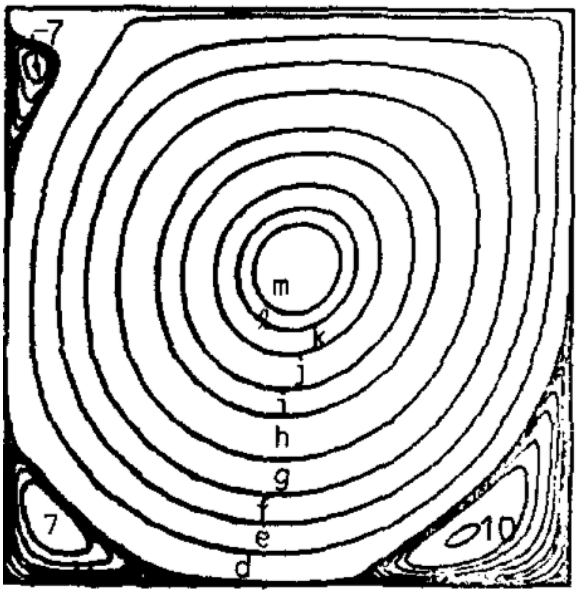
(b) Mesh of the sudden expansion around the inlet (802×83)

Fluid and solid regions in the used domains

Appendix F: Reference flow field for the lid-driven cavity (simulations of Ghia et al.)

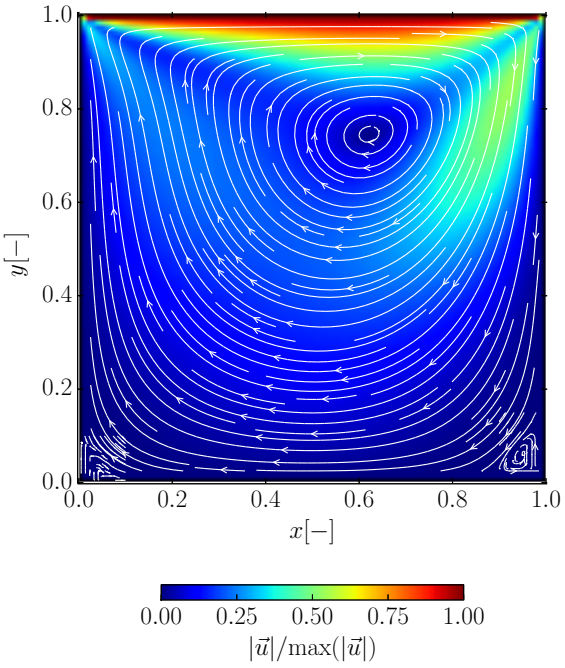


(a) $Re = 100$ (b) $Re = 1,000$

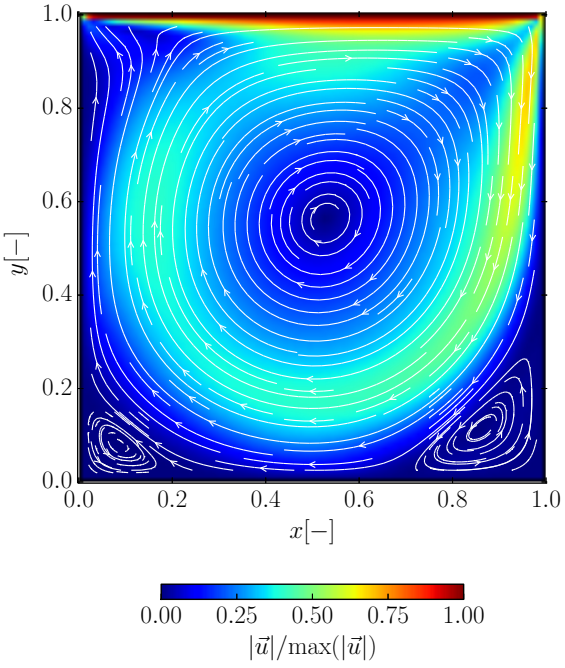


(c) $Re = 3,200$

Appendix G: Lid-driven cavity flow field (TRT)



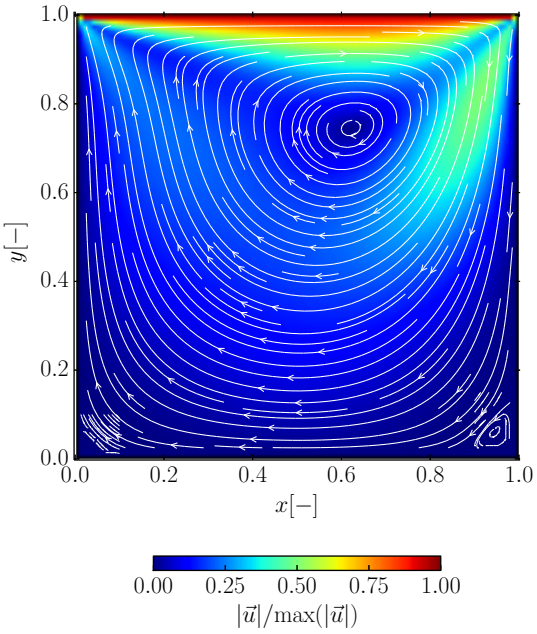
(a) $Re = 100$



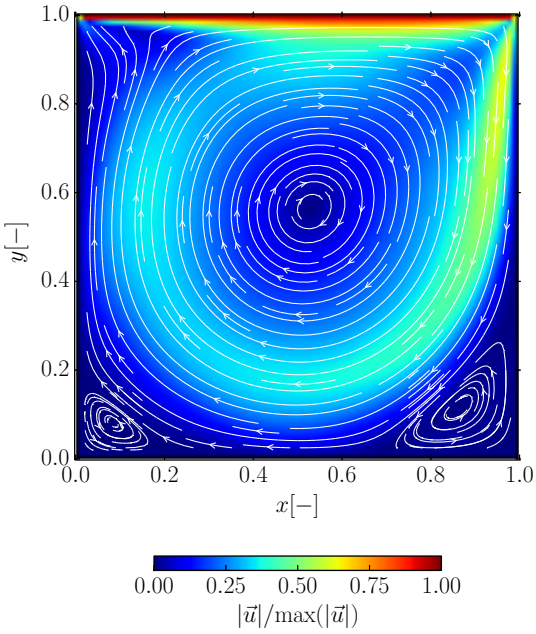
(b) $Re = 1,000$

Results of the TRT collision model

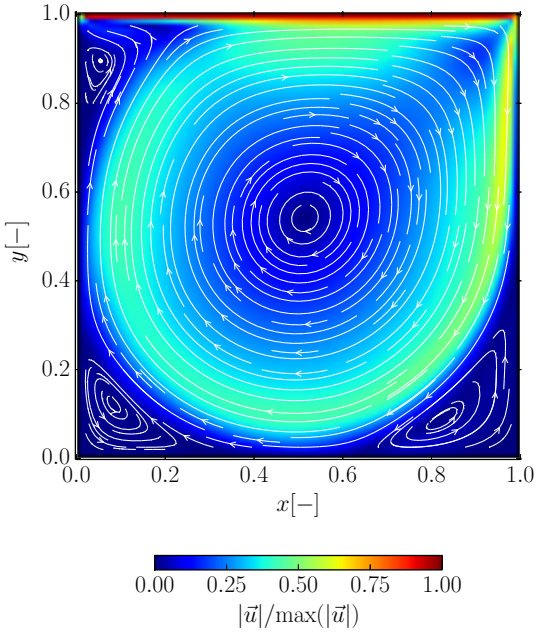
Appendix H: Lid-driven cavity flow field (MRT)



(a) $Re = 100$



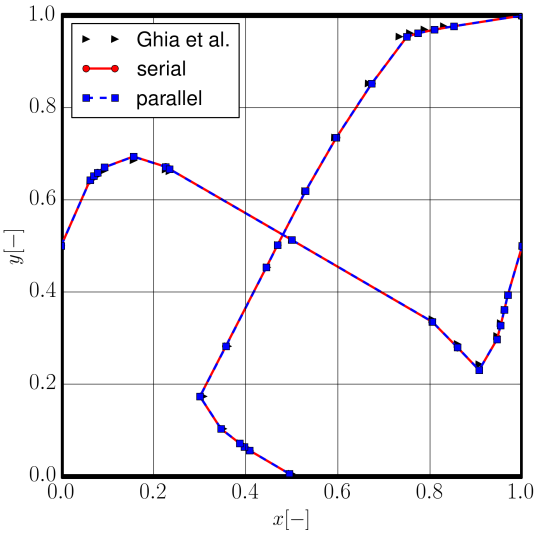
(b) $Re = 1,000$



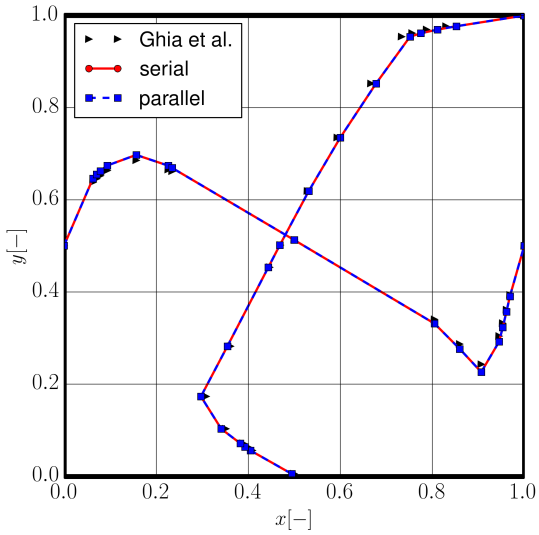
(c) $Re = 3,200$

Results of the MRT collision model

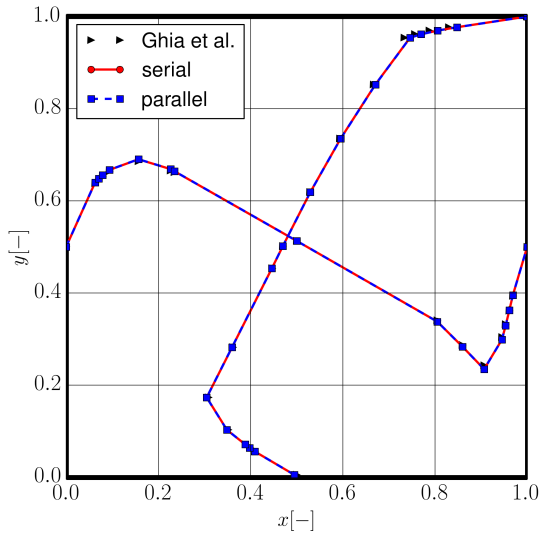
Appendix I: Comparison of the serial and the parallel code (lid-driven cavity)



(a) BGK



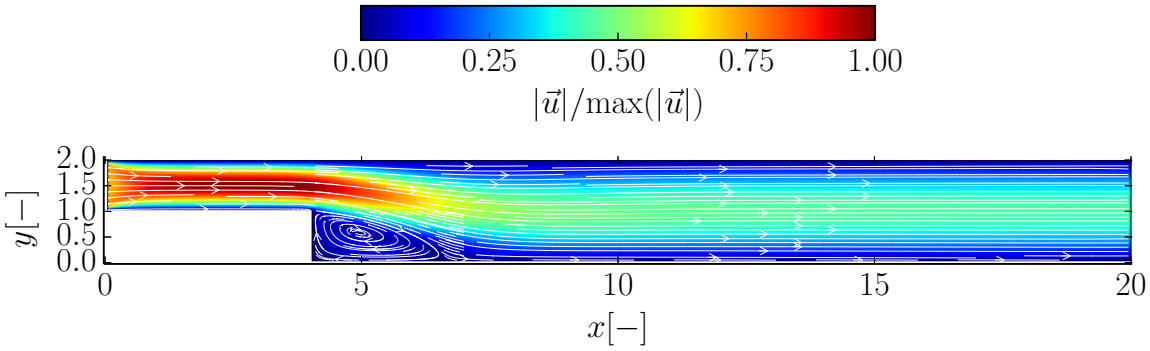
(b) TRT



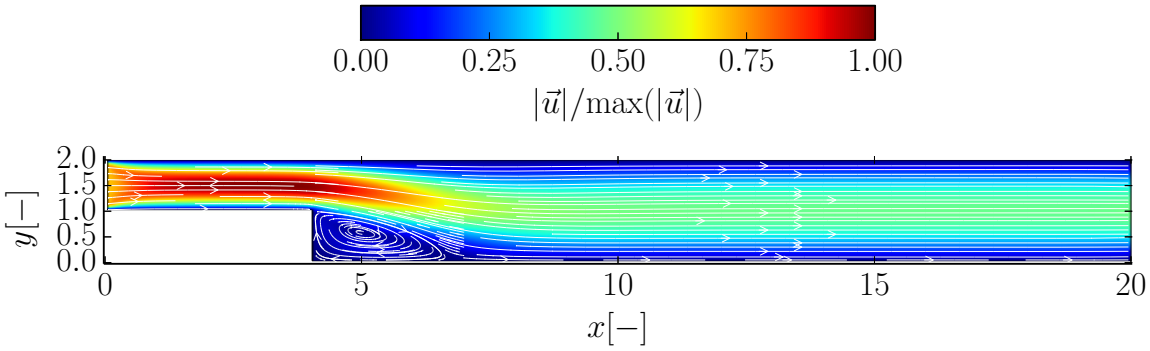
(c) MRT

Profiles compared to the simulations of Ghia et al [110] ($Re = 1,000$)

Appendix J: Backward facing step flow fields



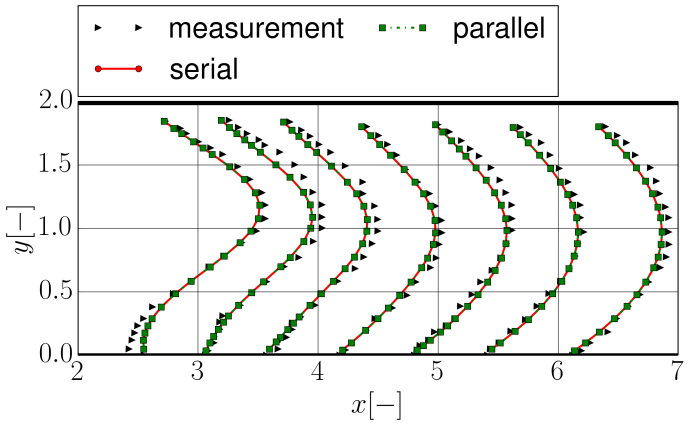
(a) TRT model



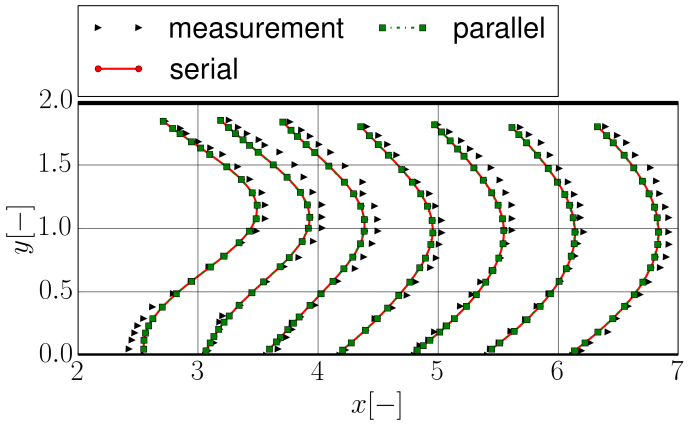
(b) MRT model

Streamlines and velocity contours as the results of the backward facing step simulations

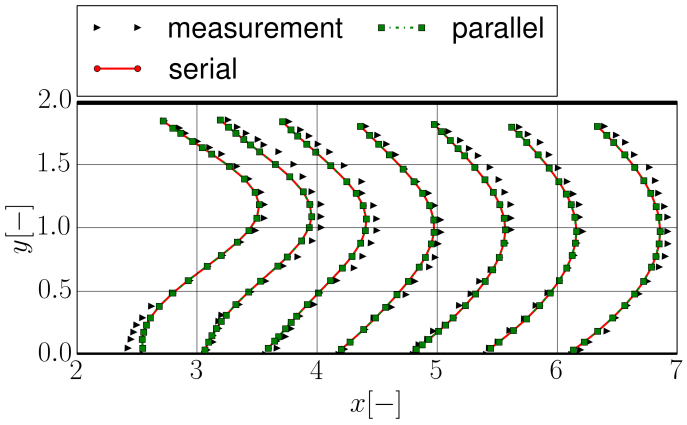
Appendix K: Comparison of the serial and the parallel code (backward facing step)



(a) BGK



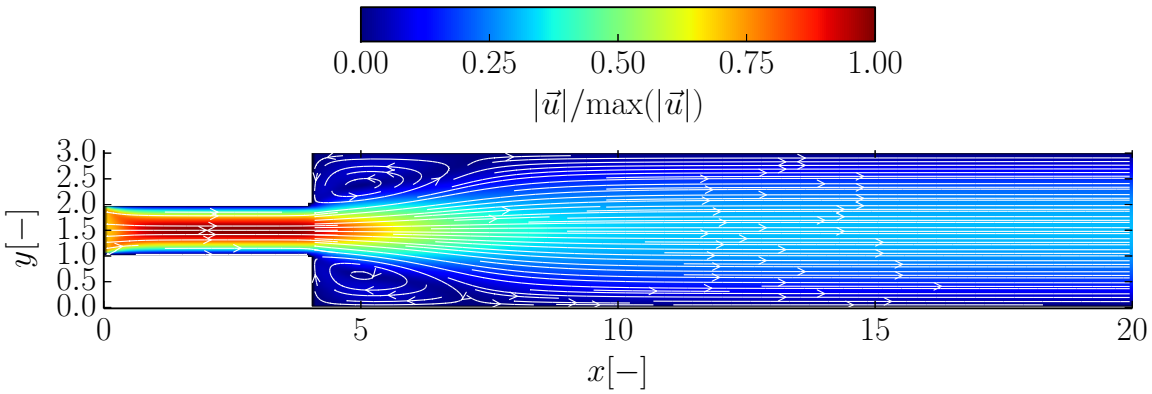
(b) TRT



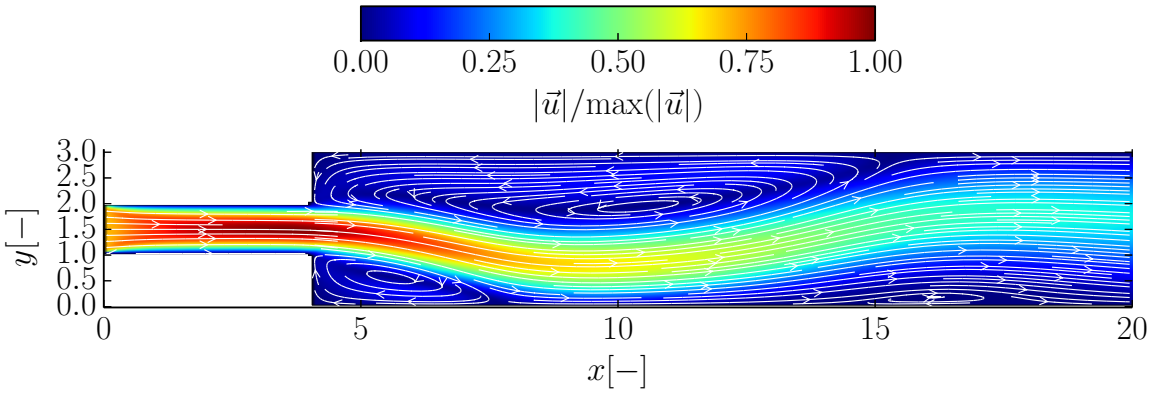
(c) MRT

Velocity profiles compared to the measurements of Armaly et al. [112]

Appendix L: Sudden expansion flow field (TRT)



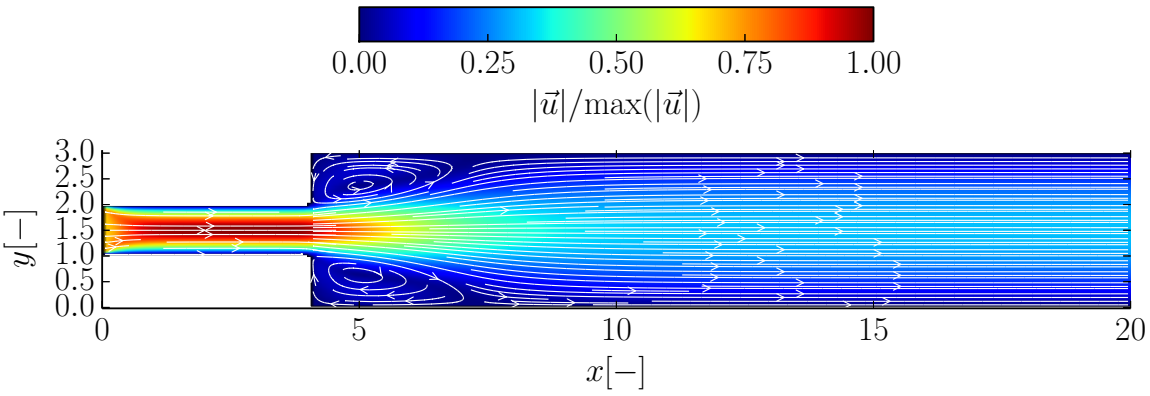
(a) $Re = 25$



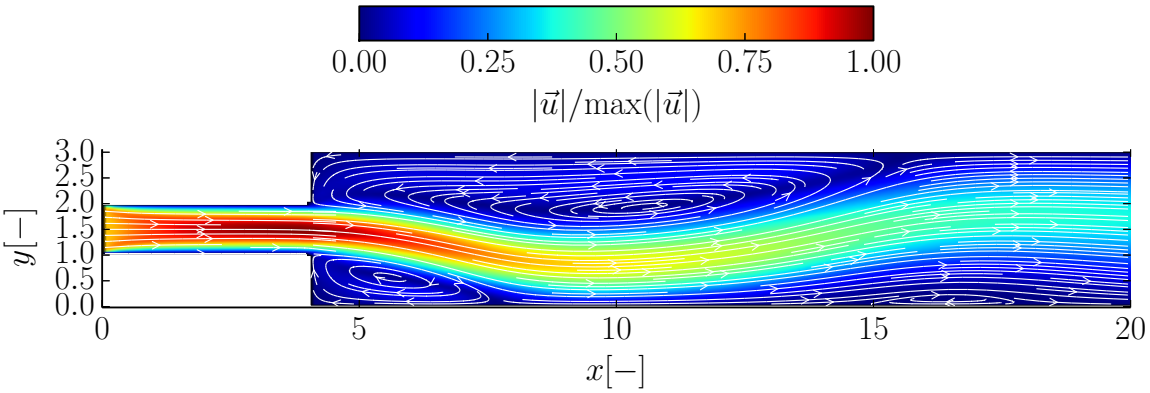
(b) $Re = 80$

Results of the TRT collision model

Appendix M: Sudden expansion flow field (MRT)



(a) $Re = 25$



(b) $Re = 80$

Results of the MRT collision model