CRANFIELD UNIVERSITY

MACIEJ KUBAT

DEVELOPMENT OF PHYSICS OR HPC OPTIMISATION OF A PARELLEL 2D LATTICE BOLTZMANN SOLVER USING GPUS CUDA

SCHOOL OF AEROSPACE, TRANSPORT AND MANUFACTURING
Computational & Software Techniques In Engineering

MSc Thesis
Academic Year: 2015 - 2016

Supervisor: Irene Moulitsas
August 2016

CRANFIELD UNIVERSITY

SCHOOL OF AEROSPACE, TRANSPORT AND MANUFACTURING
Computational and Software Techniques in Engineering

MASTER OF SCIENCE

Academic Year 2015 - 2016

MACIEJ KUBAT

DEVELOPMENT OF PHYSICS OR HPC OPTIMISATION OF A PARELLEL 2D LATTICE BOLTZMANN SOLVER USING GPUS CUDA

Supervisor: Irene Moulitsas
August 2016

This thesis is submitted in partial fulfilment of the requirements for the degree of Master of Science
(NB. This section can be removed if the award of the degree is based solely on examination of the thesis)

ABSTRACT

Main goal of this thesis was to develop existing two-dimensional LBM CUDA solver into 3D version. First, structure of the memory was rearranged in order to store and simulate 3D problem cases. Second, all main steps and functions of the solver were adapted to handle three-dimensional simulations. Third, developed code were verified and validated though a 3D lid driven cavity flow. Fourth, the memory was rebuilt again, in order to simulate bigger problem cases. Fifth, the code was optimised to decrease runtimes of the simulations.

Second task of the thesis was to prepare software to generate input data for the 3D LBM solver. To achieve that, there has been used existing mesh generator. First, the code was modified in order to handle three-dimensional geometries. Secondly, the program was optimised to improve performance. Finally, improved code is over 100x faster than the first version, adapted directly from 2D mesh generator.

Optimisation of both applications were required to simulate real problems with appropriate accuracy in reasonable time. Each change of once validated code has been validated again, using the output files comparator – another feature of the LBM solver.

ACKNOWLEDGEMENTS

Working on this thesis was a really challenging task. Now I am sure, it would not be possible without help of a few, special people. First, I would like to thank my supervisor, Dr Irene Moulitsas for her valuable support. All the time during my work on the thesis I could rely on her guidance and precious advices.

I would also thank my friend, Alfonso Aguilar. We have been developing together the first version of the solver. Without his CFD knowledge, it would never be possible to finish this working 3D version of the LBM solver. He also was a great motivator, without his involvement in this project, I probably would not finish this thesis.

Finally, I would thank my family, my parents Katarzyna and Marek, and my brother Filip. They have always supported me and believed in me.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF EQUATIONS

LIST OF SYMBOLS

Symbol	Unit	Description
λ	[m]	Mean free path
ν	[m^2/s]	Kinematical viscosity
$\nu_{lattice}$	[-]	Lattice viscosity
ρ	[kg/m^3]	Density (microscopic or macroscopic)
μ	[Pa s]	Dynamical viscosity
τ	[s]	Relaxation time
ω	[-]	Collision frequency
f^{eq}	[-]	Equilibrium part of the distribution function
f^{ne}	[-]	Non-equilibrium part of the distribution function
k_B	[m^{2}kg/s^2/K]	Boltzmann constant
N	[-]	Number of particles
p	[Pa]	Pressure
Re	[-]	Reynolds number
t	[s]	Time
u	[m/s]	x-directional velocity
v	[m/s]	y-directional velocity
w	[m/s]	z-directional velocity
x	[-]	First axis of the Cartesian coordinate system
y	[-]	Second axis of the Cartesian coordinate system
z	[-]	Third axis of the Cartesian coordinate system

LIST OF ABBREVIATIONS

2D	Two-Dimensional
3D	Three-Dimensional
BC	Boundary condition
BGKW	Bhatnagar-Gross-Krook-Welander collision model
CAD	Computer Aided Design
CAE	Computer Aided engineering
CFD	Computational Fluid Dynamics
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
D2Q9	The Two-Dimensional Applied Lattice Speed Model
D3Q19	The Three-Dimensional Applied Lattice Speed Model
Double	8 Byte Floating-point Data Type
DP	Double Precision
Float	4 Byte Floating-point Data Type
FLOP	Floating-Point Operation
GPU	Graphics Processing Unit
HPC	High Performance Computing
Int	4 Byte Integer Data Type
M1	GRID - Cranfield HPC Machine
M2	FERMI – Cranfield HPC Machine
M3	Third Machine – Private Laptop
MRT	Multiple Relaxation Time collision model
MLUPS	Mega Lattice node Updates Per Second
NS	Navier-Stokes (equation)
LBM	Lattice Boltzmann Method
SM	CUDA Streaming Multiprocessor
RAM	Random Access Memory
VRAM	Video RAM
UPC	Unified Parallel C programming Language

1 Introduction

Computational fluid dynamics (CFD) is a good alternative to wind tunnels. When we want to prototype for example an aircraft's wing, firstly we should simulate its behaviour using CFD solver. It is faster and cheaper than building real prototype for wind tunnel. It was just one from many examples of usage of the CFD. It shows how important and valuable is the CFD in real life, not only for scientific purposes.

Subject of my thesis – lattice Boltzmann method (LBM) is relatively new class of CFD methods. LBM combines advantages of two classical approaches in CFD – macroscopic and microscopic (section 2.2). Besides the advantages related purely to the simulated physics, LBM is a well parallelisable algorithm. Nodes in LBM solver require very small amount of data from neighbours, therefore we can partition the domain of the problem among many threads and the communication cost between them will be really low.

Talking about parallelisable capabilities of the LBM, it is necessary to introduce very important term – high performance computing (HPC). Nowadays, CFD simulations are so complex and time consuming, therefore there is a need to reduce runtimes of those simulations. The simplest way to achieve it, is parallelisation. Parallel codes can be run on almost every computer, it is really difficult today to buy PC with a single core CPU. However, to obtain really high performance of the parallel code, what is necessary in scientific or industrial world, we need HPC machines. They are so powerful machines, each of them has hundreds of thousands CPUs [8].

Until 2007, each HPC was accelerated only by CPUs [20]. However, after that time we could notice single HPCs, accelerated by GPU. Now that trend is rising, using GPUs in computing are getting more and more popular. How is it possible to use GPU for computing? They are not only supposed to display graphics? Until 2007 it had been a truth. Thereafter, NVIDIA has released CUDA – parallel computing platform, designed for NVIDIA GPUs. It was a milestone for HPCs. CUDA platform is very innovative, it has a complex memory model, but because of that, CUDA also offers great capabilities.

1.1 Objectives of the thesis

Main task of this thesis was to adapt existing 2D LBM CUDA solver [12,[13] into three-dimensional version. This 3D LBM solver will be developed by two students: author of this thesis and Alfonso Aguilar [23]. Author will be responsible for all software aspects of this project, whereas Aguilar will be responsible for all physical aspects of the solver: models, validation, etc.

Besides developing 3D LBM solver, there is also required another software, for preparing input data for the simulations. It will be adapted from existing 2D mesh generator [14] to handle three-dimensional meshes. Similarly as the LBM solver, lattice generator will be developed by two students. Distribution of the duties will be the same, author of this thesis will be responsible for software issues of this task (algorithms, optimisation, etc.), whereas Aguilar will be responsible for the physical aspects of meshes and lattices.

2 Literature review

In this chapter I will present a short overview of what has already been done by others in fields of my thesis. I will write something about the physics, which is necessary to understand the whole thesis. Then I will introduce very modern and fast growing topic of high performance computing.

2.1 General overview of computational fluid dynamics

The topic of my thesis – lattice Boltzmann method is one of the Computational fluid dynamics (CFD) methods, so firstly what we should know before I will describe the examined method, is what exactly the CFD is.

To talk about the CFD, we should know the definition: “CFD is the science of determining a solution to fluid flow through space and time.” [1]. CFD uses computers to simulate dynamic of fluids, by using various numerical methods, sometimes few of them are combined together, to solve complex simulations.

Computational fluid dynamics is a very huge, and fast growing part of science, used not only by scientists in universities, but also by industry, in many different sectors. Below we can find a few examples of usage of the CFD:

- simulations of methane partial oxidation [2];
- designing jet engine internal cooling system [3];
- designing microchannel networks for liquid-cooling of electronic devices [4];
- simulations of blood flow [5], etc.

As we can see above, CFD is used very widely, in different sectors of science and industry. Definitely few of our own items were designed with assistance of CFD simulations (e.g. our cars).

2.2 Two main approaches to CFD

In computational fluid dynamics, we can find two main approaches in simulations: continuum (macroscopic) and discrete (microscopic) [6]. We should acquaint with those approaches, to understand later the lattice Boltzmann method.

2.2.1 Macroscopic scale

In this approach, we are considering the fluid as a whole. We can describe the fluid by using traditional properties of fluids: velocity, pressure, temperature, etc. All the macroscopic scale methods are based on conservation of energy, mass and momentum rules. Those rules can be combined into system of algebraic equations, which can be solved iteratively, but we have to be sure about convergence of those methods [6].

2.2.2 Microscopic scale

Microscopic scale approach is based on treating the medium (fluid) as small particles (atoms, molecules). In those methods, we have to use Newton's second law and compute location and velocity of every particle of the fluid. As we know, each fluid is built from huge number of particles, for example 10^3 cm of air at normal condition contains about 3×10^{22} particles [6]. Therefore, we can use those methods only for simulations for small amount of fluids, to ensure that the simulation will finish in reasonable time.

2.3 Lattice Boltzmann method

In section 2.2 we were introduced to the two main approaches of CFD. We know the advantages and disadvantages of those methods, so now we can introduce lattice Boltzmann method, which is something between macroscopic scale and microscopic scale methods – we can classify the lattice Boltzmann method as a Mesoscopic scale [6].

Lattice Boltzmann method (LBM) is similar to microscopic scale methods, but it does not consider every particle separately. This method creates groups of particles (for example 9 particles are considered as one node in 2D case (Figure 2.1) or 19 particles in 3D version (Figure 2.2)). These collections of molecules are described by a distribution function which is the most important part of the LBM [6].

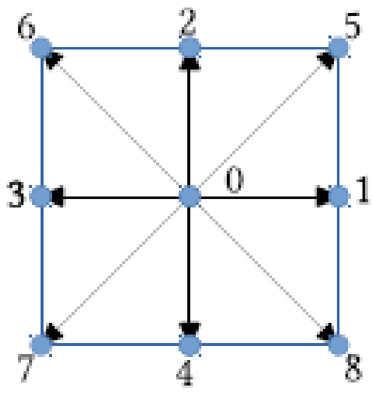


Figure 2.1: 2D LBM molecule model (D2Q9)

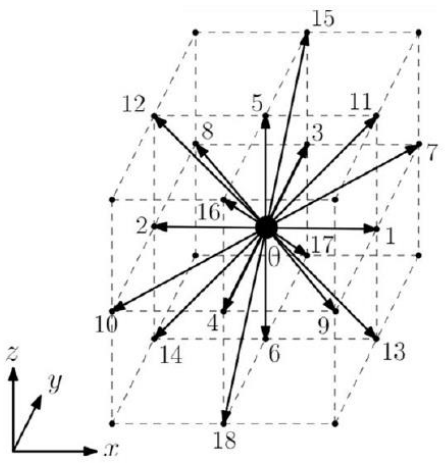


Figure 2.2: 3D LBM molecule model (D3Q19)

LBM takes the best features from both microscopic and macroscopic scale methods. The most important feature of LBM (for our purpose – parallelization) is that LBM requires very small exchange of data between nodes during the simulation. Therefore, LBM offers great opportunities to parallelise.

2.4 High performance computing

High performance computing (HPC) is a fast growing field of informatics. Need of computing performance is still growing, many sectors of industry, science are simulating more and more complex scenarios. Without powerful machines, those simulations could take weeks, or even months. HPC were introduced in the 1960s by Seymour Cray [7]. He established Cray Research company, which is still functioning. Moreover, Cray is one of the bests HPC concerns (if not the best). Currently, one of the Cray's HPC machines is the second of the most powerful HPCs on the world (Figure 2.4) [8]. That HPC is called Titan (Figure 2.3), and its peak computing capability is 27,112.5 TFLOPs/s. To have a comparison to that score, computing performance of my CPU in my laptop is 16.94 GFLOPs/s. We can see that the difference is incredible.

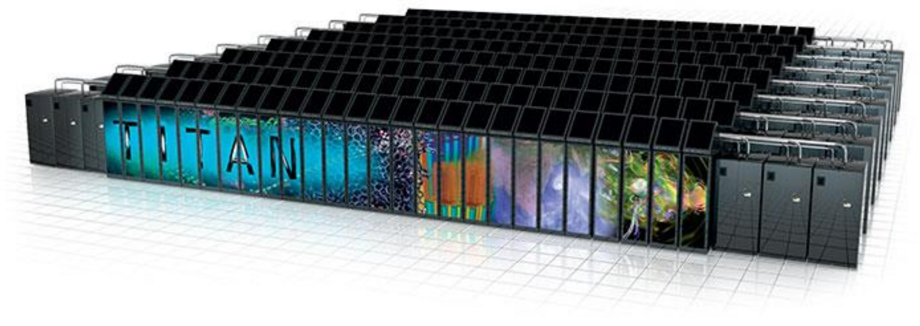
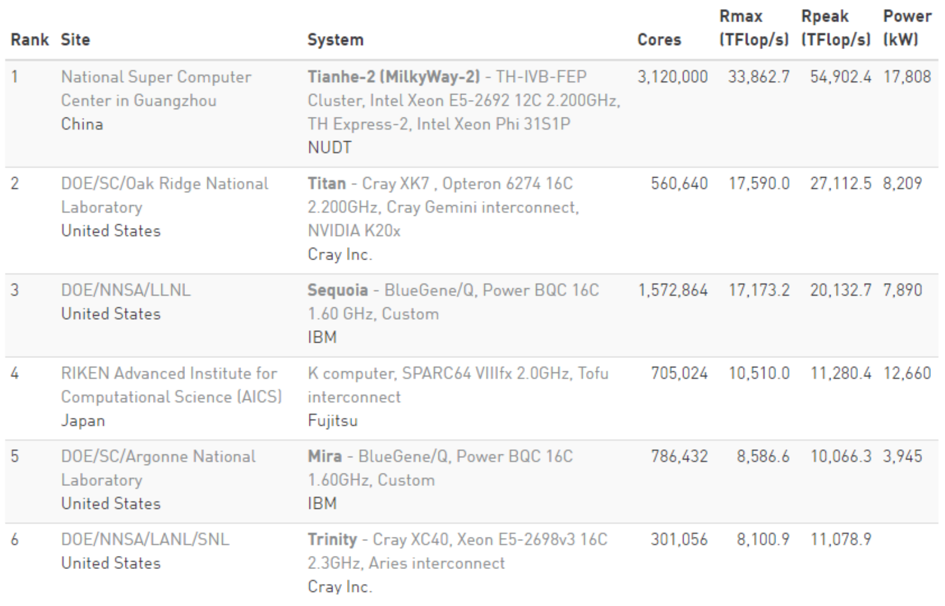


Figure 2.3: Cray Titan [9]



Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
2	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
3	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
5	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,586.6	10,066.3	3,945
6	DOE/NNSA/LANL/SNL United States	**Trinity** - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Aries interconnect Cray Inc.	301,056	8,100.9	11,078.9	

Figure 2.4: Top HPC machines in November 2015 [8]

HPC clusters are very expensive machines, so they are used mainly by big industry machines, or scientific research centres. Therefore, our developed parallel code should be reliable, to not waste expensive HPC resources unnecessarily.

In the Figure 2.5 we can see, how fast the HPC evolved in the past few years. The plot presents bandwidth of networks used in HPC clusters. It is just a comparison of HPC hardware, it does not mean, that parallel codes run on those machines have the same speedup as hardware. It is very difficult to obtain maximum, theoretical speedup in parallel applications, because we have to struggle with many other bottlenecks.

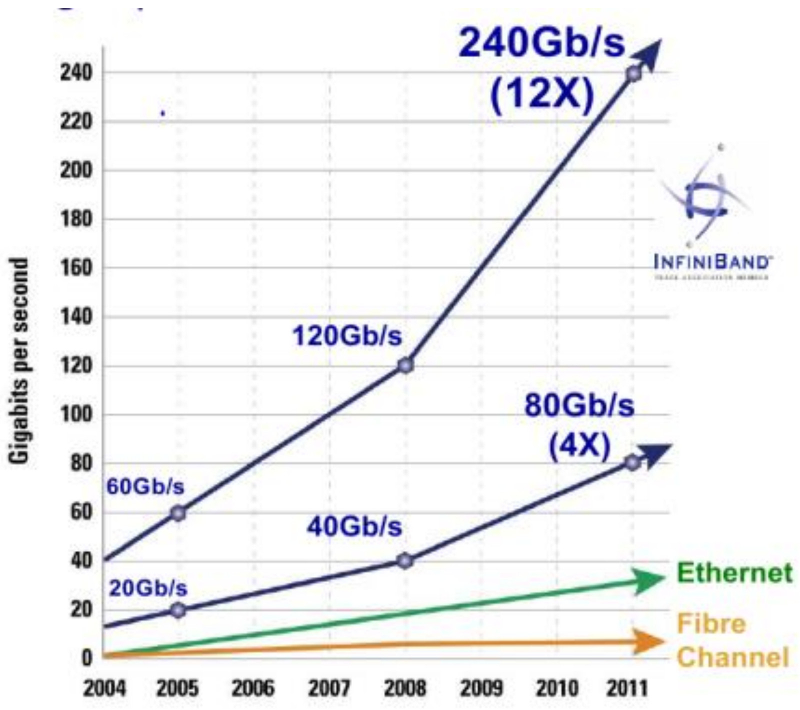


Figure 2.5: Comparison of HPC networks [10]

2.4.1 Main approaches to HPC parallelism

To talk about parallel codes, firstly we should introduce two main parallel models, which are widely used in nowadays HPC.

2.4.1.1 Shared memory model

First of the two main approaches is shared memory model. In this case, all threads (the smallest sequence of programmed instructions) are using the same, shared memory (Figure 2.6). That is why we also call this model as thread-based parallel programming or data-sharing scheme [11]. The most popular programming platform for this model is OpenMP (Open Multi-Processing). I would recommend this API for programmers who wants to start their adventure with parallelisation, because it is the easiest way to transform serial code into parallel. Using OpenMP, we can parallelise every single loop of serial code just by adding a few lines of code. There is no need for any advanced knowledge about HPC, to use OpenMP. We could also classify CUDA as shared memory model, but that architecture is much more complicated, I will describe CUDA in details in further part of the thesis (section 2.7).

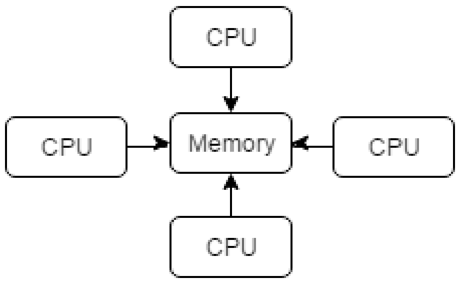


Figure 2.6: Shared memory model

2.4.1.2 Distributed memory model

Second approach to parallelise is distributed memory model. In this model, each process has its own memory(Figure 2.7) [11]. Another term for parallelisation scheme is message-passing or process-based parallel programming. To exchange data between processes, we have to establish connection between them (that is why we called it message passing model), which is also time consuming. Main advantage of this approach is preventing race conditions. Race conditions occur when we want to access the same memory in the same time by different threads. This is also the main disadvantage of shared memory model. Open MPI (Message Passing Interface) [10] is a library which helps us to implement parallel message-passing codes. It combines resources from a few other projects (FT-MPI, LA-MPI, LAM/MPI, PACX-MPI).

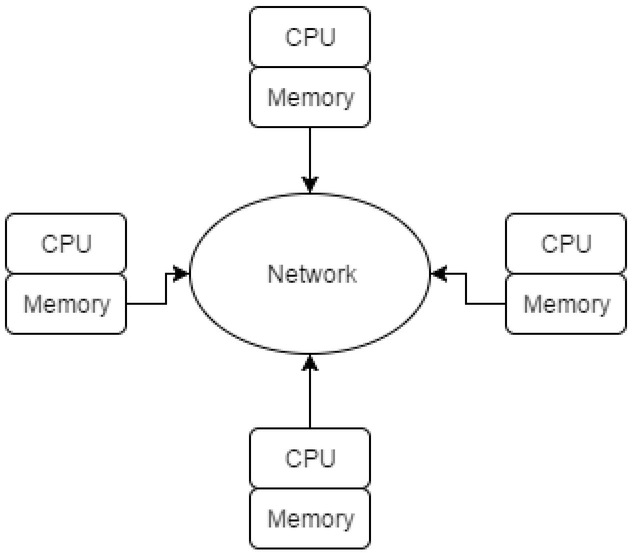


Figure 2.7: Distributed memory model

2.4.2 Motivation for HPC

There are three main issues why do we use HPC instead of our own single processor personal computers (Figure 2.8)[10]:

- multiple simulations at the same time;
- faster results by using parallel solvers on multiple HPC processors;
- bigger memory allows to simulate bigger problem cases;
- increased accuracy of CFD simulations by using models with higher resolution (bigger memory).

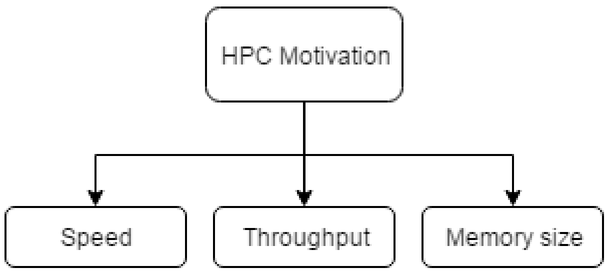


Figure 2.8: Motivation for HPC

2.5 Existing implementations of LBM

Lattice Boltzmann method is a quite popular CFD method, so we can find a few implementations of LBM, even the parallel versions. In my thesis, I will base on two parallel implementations of LBM, developed independently by two Cranfield University's students: Tamás István Józsa and Ádám Koleszár [12[13]. These students worked on parallel implementations of LBM with using CUDA technology. I will explain what is the CUDA in further part of this chapter.

Tamás István Józsa was the student, who implemented the first version of LBM solver on CUDA platform. This implementation was based on two different versions of C++ implementation of LBM solver [14, [15]. Those implementations was the 2D laminar LBM solvers. Józsa took the best parts of those two implementations, he transformed it into C code, and then, he parallelised the most time-consuming parts of the code, by using the CUDA technology. Józsa solver was about 15 times faster than the serial version of the same solver, so we can say, his result was quite impressive.

Ádám Koleszár was the student who developed the Józsa's version of LBM solver. This solver has already used the CUDA technology, but Koleszár tried to increase performance of that solver. He has done his work properly, his version of the LBM CUDA solver was 2 times faster than the first version of LBM CUDA solver (developed by Józsa). Finally, Adam's solver was 30 times faster than the serial version of the same LBM solver, so for example if we have a simulation which runs 30 minutes on single thread (serial version of the solver), Adam's LBM solver will run just a minute on the GPU.

There is also one interesting parallel implementation of the LBM solver, developed with using different technology than CUDA. This solver was implemented by Máté Tibor Szőke [16]. He used Unified Parallel C (UPC) which is a Partitioned Global Address Space (PGAS) based language. This approach for parallelisation is quite different than CUDA. UPC is a high level programming language [17], what means, that many of necessary mechanisms used in parallel application (synchronisation between threads, protection from deadlocks, etc.) are provided by the programming language. Using CUDA, we have to care about all of those details. This difference is both an advantage and a disadvantage in the same time. The advantage is that we have an influence on almost everything, every synchronisation, so we can optimise the code in a better way, but it will be harder for programmer.

All of those LBM solvers described above were supposed to simulate only 2D problems. However, LBM solver implemented on GPU by Alistair Revell [18] is designed also for 3D cases, not only 2D. Performance of his solver is really high (812 MLUPs (Million node updates per second) on NVIDIA K20c). He tried to maximise usage of GPU registers, which are the fastest type of GPU memory. Therefore LBM solver developed by Revell is the fastest GPU LBM solver which I have found.

All the solvers described above, were verified and validated, to prove that they compute the simulations properly, with small, acceptable errors. These solvers support 3 different collision models: BGKW, TRT and MRT. All of those collision models are scalable differently, for example the most scalable is the TRT. The best speed-up was obtained with using this collision model. The worst scalable was the MRT model.

2.6 Lattice generation for LBM solver

To simulate any physical problem case in LBM solver, we need appropriate input data. All LBM solvers developed by Cranfield Students (section 2.5) had been using input data prepared by software written by Gennaro Abbruzzese [14]. His mesh generator uses STAR-CD files, which can be exported from popular meshing software – Pointwise, which is widely used by CFD engineers. His software uses two input files: ".vrt" and ".bnd". In the first file, there are stored all geometrical nodes of the mesh with their real coordinates. ".bnd" file contain information about boundaries of the mesh: indices of the nodes of the cell with BC (Boundary condition), ID of BC and type of BC (e.g. wall, inlet, outlet, etc.).

STAR-CD only contains information about real coordinates of the nodes and boundaries, however LBM solver requires information which directions of LBM molecule cross the boundary (Figure 2.9). The main part of the mesh generator is "FluidTracking". This part of the code iterates through all nodes of the lattice and if it encounters boundary, it will write appropriate information to the output file: index of the node, direction which crosses boundary, real coordinates of the intersection, type of the boundary.

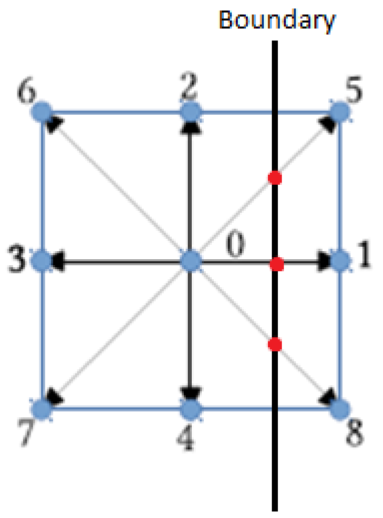


Figure 2.9: Boundary placed on directions 1, 5, 8 of the LBM molecule

Gennaro's software prepares off-grid lattice, which means that boundaries are places between nodes (Figure 2.10). There is also second approach – On-grid lattice (Figure 2.11). Each of those approaches requires different physics models for the LBM solver.

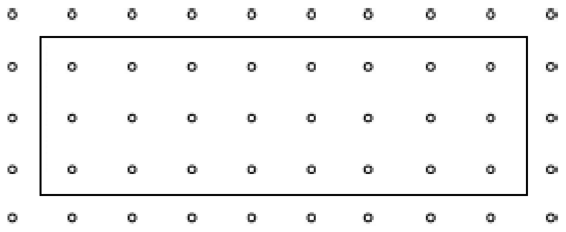


Figure 2.10: Off-grid lattice

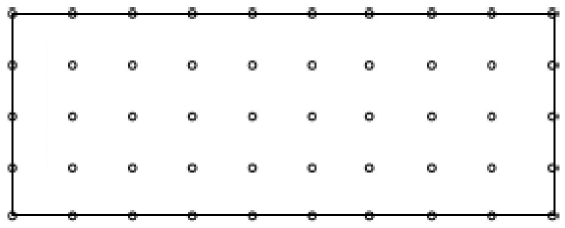


Figure 2.11: On-grid lattice

2.7 GPGPU

What is this mysterious abbreviation? GPGPU means general-purpose computing on graphics processing units [11]. Originally, graphic cards were intended only for rendering (preparing graphics). However, in the 2000s, GPUs started to being used also for general-purpose computing. Vectors and matrices are native "data formats" for GPUs, so calculations on those formats shall be fast, maybe faster than by using CPUs. Finally, the first program which runs faster on GPUs than CPUs was LU factorization. It happened in 2005. Now, GPUs are quickly getting more and more popular in HPC world (Figure 2.12).

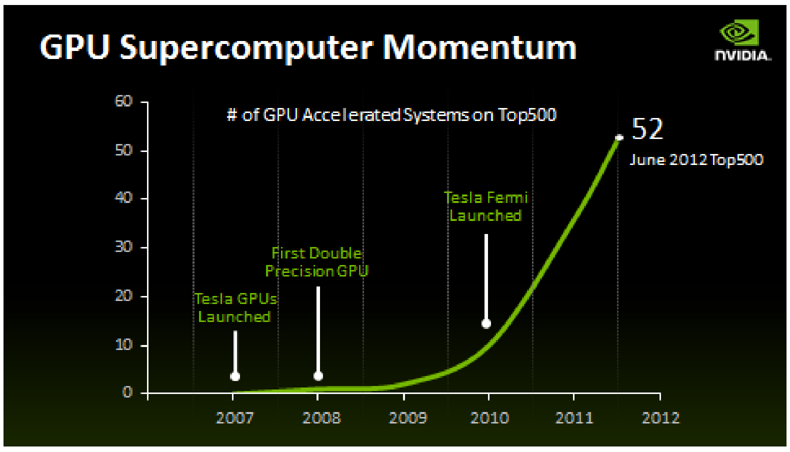


Figure 2.12: HPC accelerated by GPU [20]

The biggest difference between GPU and CPU computing is the architecture of them. Normally, popular CPUs used in desktops and laptops have up to 8 threads, so we can run up to 8 parallel threads. However in our GPUs, there are hundreds of thousands threads. We can imagine how many tasks we can do at the same time, using GPU.

The most popular GPGPU platform is CUDA, developed by NVIDIA Corporation. First version of CUDA was released in the middle of 2007 [19]. CUDA code is very similar to C language, there are couple of extensions to the C language, provided by NVIDIA. They are necessary for using CUDA. Figure 2.13 presents how the GPUs evolved in the past few years. Until 2004 graphics cards had the same performance as CPU, however in 2014 GPUs are much more powerful than CPUs (almost 10 times faster in single precision floating-point operations).

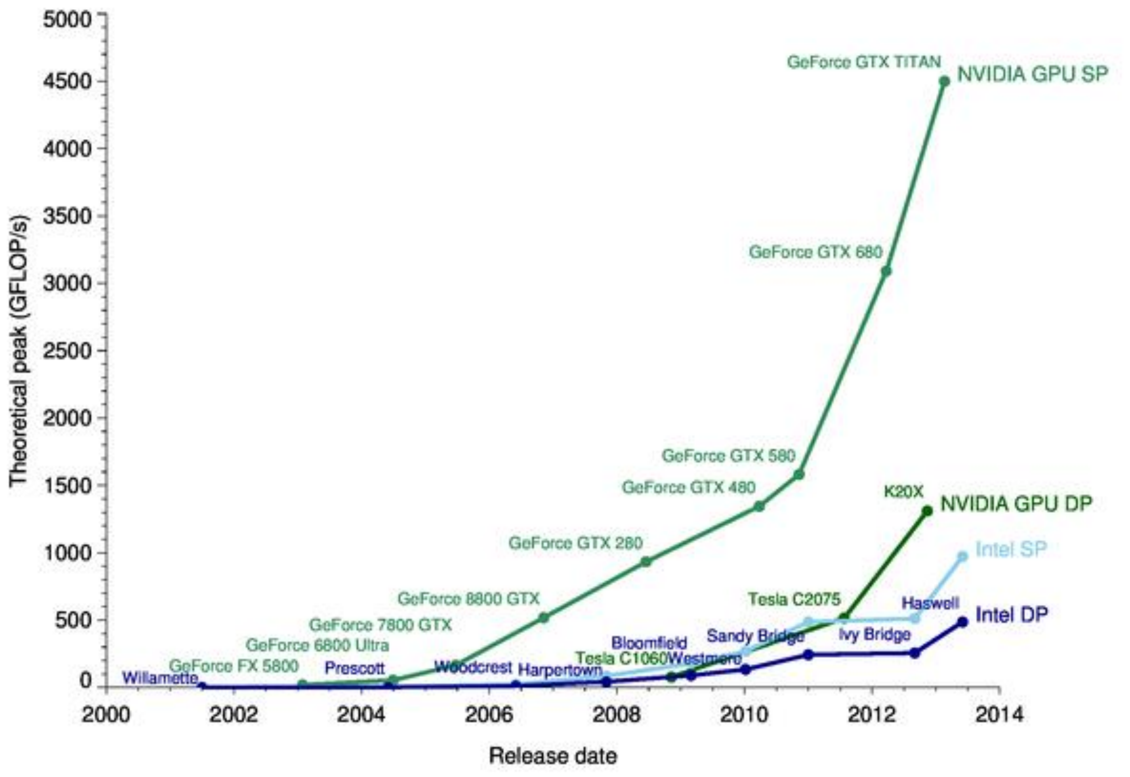


Figure 2.13: GPU and CPU performance comparison [21]

CUDA is a really low level programming, we can control almost everything: synchronisation mechanisms, memory management, etc.[22]. This feature of CUDA lets us develop really fast piece of code. However, it can be also a disadvantage: CUDA will not protect us from develop really bad and slow code, we have to care about everything. One small oversight can slow-down our code 100 times. This situation will not happen in high level parallel languages, but those platforms will never be as fast as low level platforms.

3 Methodology

3.1 Three-dimensional space – directions

Introducing three-dimensional space, it is necessary to assume consistent nomenclature for all six directions in the 3D space - Figure 3.1. It helps a lot, while it is certain, that all readers has no doubts about directions, when any of 3D geometry is described by the author. I have encountered many different nomenclatures for directions in 3D space, therefore it is necessary to clarify it.

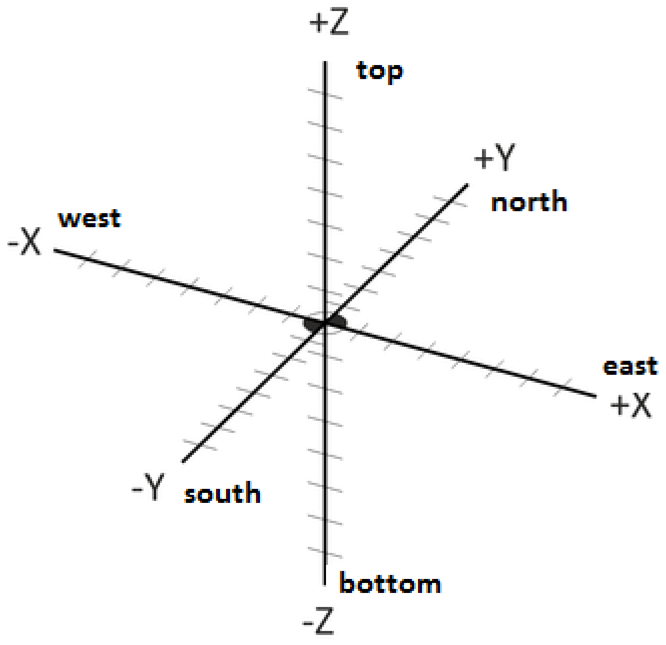


Figure 3.1: Nomenclature for directions in 3D space

3.2 Lattice Boltzmann Method

LBM is a CFD method with a mesoscopic approach (section 2.3). To describe physical system, we will use the Boltzmann equation[6], which is an advection equation:

$$\frac{\partial f}{\partial t} + c\nabla f = \Omega \tag{3.1}$$

Where f is a distribution function, used to static description of the physical system.

To calculate macroscopic values (density, velocity), we will use following equations:

$$\rho(r,t) = \int m f(r,c,t)\, \mathrm{d}c \tag{3.2}$$

$$\rho(r,t)u(r,t) = \int m c f(r,c,t)\, \mathrm{d}c \tag{3.3}$$

Where m is the molecular mass.

Two above equations are following the rule of conservation of mass and momentum. There is also one more equation, used to calculate internal energy, but in the LBM solver this macroscopic value will not be calculated, therefore the author did not show this equation.

3.2.1 Collision models

If particles would travel through the physical domain without any collision, we could use eq (3.1) with $\Omega = 0$. However, in real world this situation is impossible, therefore we need to introduce collisions into our model. In this LBM solver, developed by the author of this thesis and Aguilar [23], there will be used BGKW collision model. Using this collision model, we will replace the Ω from eq. (3.1) with:

$$\Omega = \omega(f^{\text{eq}} - f) = \frac{1}{\tau}(f^{\text{eq}} - f) \tag{3.4}$$

Where $\omega = 1/\tau$.

The coefficient ω is called the collision frequency, whereas τ is called relaxation time. Term f^{eq} is used for local equilibrium distribution function, which is Maxwell-Boltzmann distribution function. The BGKW collision model is not the only one, there are also MRT (multiple relaxation time), and a few others. Using eq. (3.4), the Boltzmann equation – eq. (3.1) can be written now as:

$$\frac{\partial f}{\partial t} + c \cdot \nabla f = \frac{1}{\tau}(f^{\text{eq}} - f) \tag{3.5}$$

3.2.2 Discretisation of the Boltzmann equation

Equation above is still in continuous form, therefore we need to discretize it. Discretisation is necessary in order to implement the solver – write the computer code which will work properly. In LBM we discretise the eq. (3.4). This discretisation will be valid along specific directions, linkages (Figure 2.1, Figure 2.2). The discretized eq. (3.5) can be written along a specified direction as [6]:

$$\frac{\partial f_i}{\partial t} + c_i \nabla f_i = \frac{1}{\tau}(f_i^{\text{eq}} - f_i) \tag{3.6}$$

Implementing 3D LBM solver, there will be used D3Q19 lattice arrangement (Figure 2.2). Each of the directions has its own weighting factor, used e.g. to calculate macroscopic values:

- Weight of $f_0 = 12/36$
- Weight of $f_1\ to\ f_6 = 2/36$
- Weight of $f_7\ to\ f_{18} = 1/36$

3.2.3 Streaming

While the collision is the first sub step in each LBM main loop, streaming is the second step. Streaming is necessary to pass information (distribution function) between neighbour nodes. Figure 3.2 explains how the streaming process works for D2Q9 lattice arrangement, for D3Q19 used in developed 3D LBM solver, streaming is analogous.

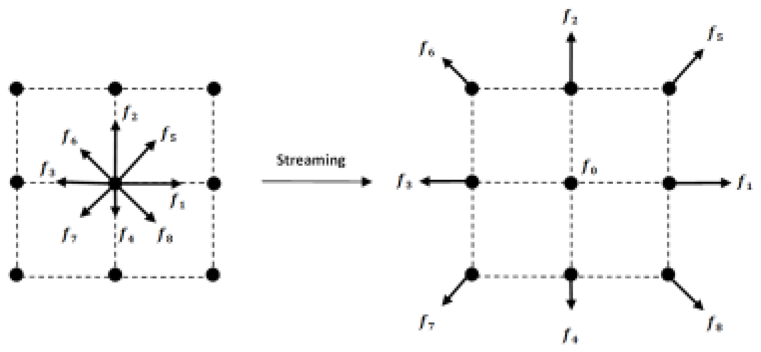


Figure 3.2: Explanation of the streaming process [26]

3.2.4 Boundary conditions for the LBM

Next sub step in LBM solver, after streaming, is BC handling. There will be implemented five different types of boundary conditions.

- Inlet – simulates an input flow of the fluid into the physical domain [25].
- Outlet – simulates an output flow of the fluid, from the physical domain to outside
- Wall – simulates a solid wall. The Fluid particle is bounced back when it encounters the wall.
- Periodic – Particles which leaves the domain e.g. on the west side, will enter the domain on the opposite side, in this case - east.
- Symmetric – unknown directions of the particles on this BC will be copied from known values as a mirror image.

All of those BCs above are described in details by Aguilar [23], who was responsible for implementation of the physical models.

3.3 3D Lattice generation

To use LBM solver, we need appropriate input data. Lattice Boltzmann method requires lattices as input data. Lattice is a structured mesh, with the same distances between nodes in each direction [24]. Therefore, for 3D problem cases, we need following condition:

$$deltaX = deltaY = deltaZ \tag{3.7}$$

where deltaX is the distance between lattice nodes in x direction, similarly for y and z.

New lattice generator, intended for 3D meshes, is based on existing software, designed for 2D meshes [14]. 3D LatticeGen has been developed in cooperation with Aguilar [23]. Aguilar has developed conceptual design of three-dimensional lattices. Therefore, author of this thesis was able to implement appropriate algorithms, which are necessary to generate correct lattices.

Introducing third dimension makes the algorithms much more complicated. In 2D case, boundary conditions are placed on lines – connectors between nodes. Therefore algorithms to find intersections of the molecule directions and BC are quite simple. However in 3D case, BCs are placed on faces – planes spread among 4 nodes. Thus the algorithms to detect intersections will be much more complicated.

Besides the main difference between old and new mesh generator - third dimension, there is also one more change. Current software prepares on-grid lattices, whereas Abbruzzese's mesh generator was intended for off-grid lattices (section 2.6). The difference is caused by using different physical models for inlet and walls. These differences are described in details by Aguilar, who was responsible for implementing equations of the models [23].

3.3.1 Used meshes

There has been prepared three different geometries of the meshes, which will be used by 3D LBM solver, for verification, validation and performance tests.

The first type of the prepared geometry is a channel - Figure 3.3. The figure shows the mesh, prepared in Pointwise (popular meshing software), by Aguilar [23]. The orange cell (on the west) represents inlet of the channel, whereas the yellow face is an outlet. The other four sides of this cuboid are walls. Therefore, using this geometry, we will simulate a flow from the west side to the east side of the channel.

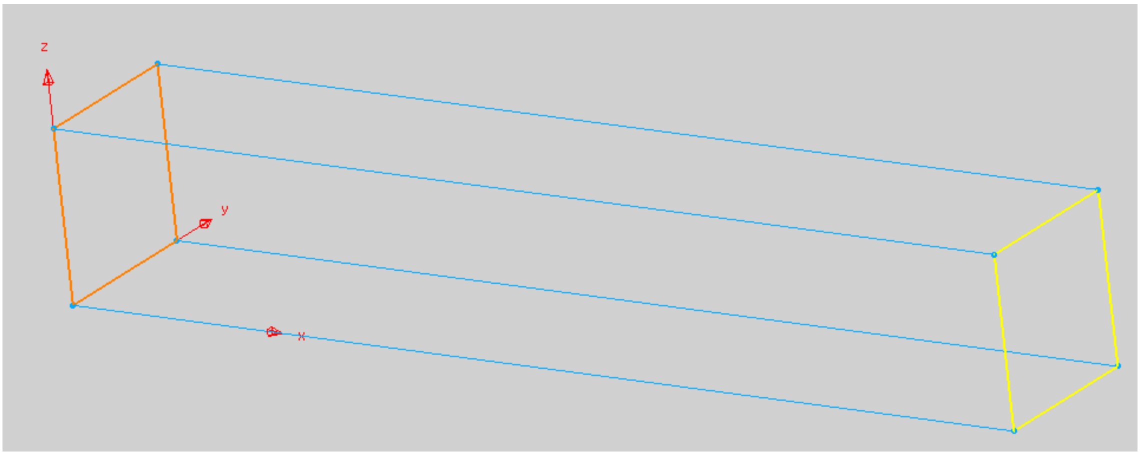


Figure 3.3: Channel

Second used mesh is a cavity – Figure 3.4, also prepared by Aguilar [23]. This geometry has 5 walls and inlet on the top (yellow face). The geometry showed below will be used to simulate popular CFD problem case – lid driven cavity.

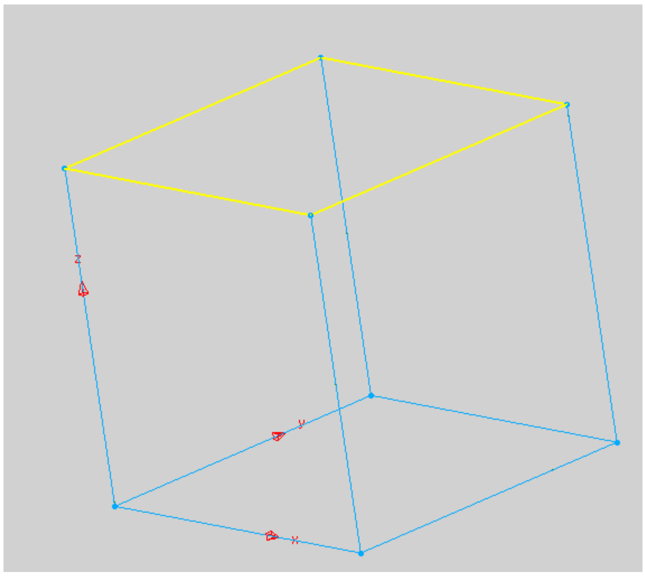


Figure 3.4: Cavity

The last used mesh is a channel with a square cylinder inside – Figure 3.5. The channel is the same as presented in the Figure 3.3 – inlet on the west side, outlet on the east side. All other faces are walls, all sides of the cylinder as well.

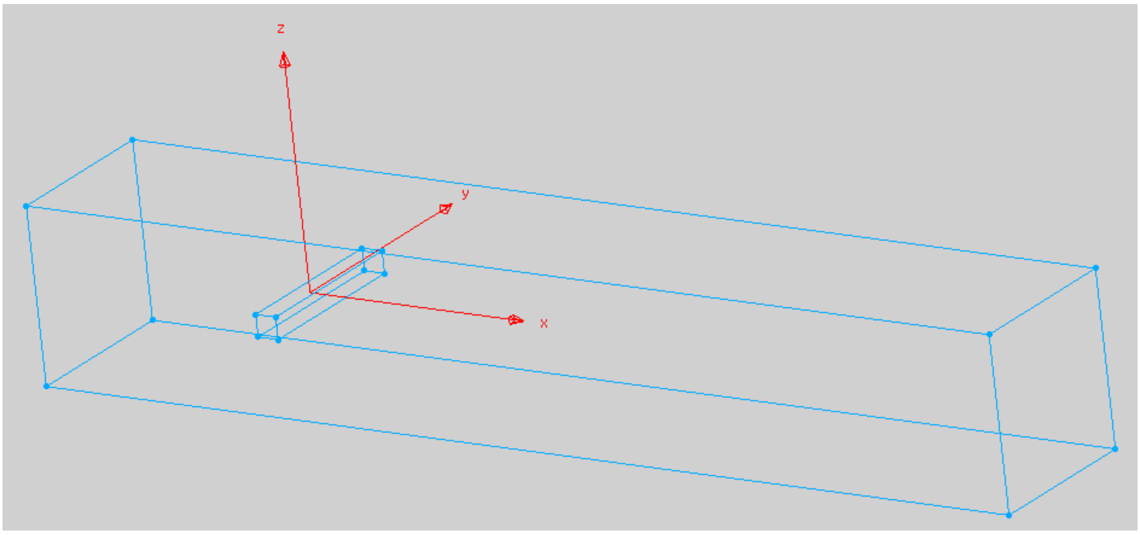


Figure 3.5: Channel with a square cylinder inside

3.4 CUDA programming

CUDA is a relatively new parallel programming platform, released by NVIDIA. It offers great opportunities for programmer, to develop parallel code in many different ways. This is caused by very advanced and complicated memory model (Figure 3.6).

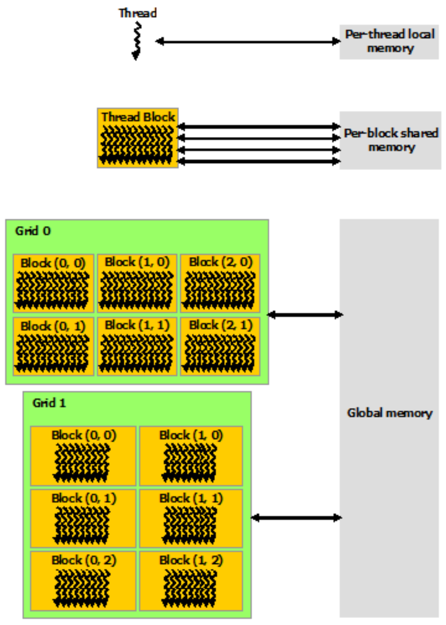


Figure 3.6: CUDA memory hierarchy [27]

In the Figure 3.6 we can notice three different types of memory:

- Local memory (registers) – the fastest type of memory. Lifetime of this memory is equal to life time of the thread, therefore we cannot pass this type of variables between different GPU kernels. Registers are also only accessible by the specified thread, which creates the local variable. Because of that, registers cannot be used to pass values between different threads. The main advantage of registers is their speed, latency of registers are only about 24 cycles of the SM.
- Shared memory – This memory can be accessed by all threads within one block. Lifetime of shared memory is the same as lifetime of the block, so it also cannot be passed between different kernels. This type of memory is slower than registers, but it is still faster than global memory (around 100x faster, in case when there is no bank conflicts between the threads[28]). Bank conflict means that multiple threads are trying to request the same memory bank. In this case, memory access is serialised. Shared memory is widely used in many popular parallel algorithms e.g. sum of vector elements, etc.
- Global memory – The biggest and slowest type of the CUDA memory. Lifetime of this memory is unlimited, global memory variable will last until we will not deallocate it. Therefore, this memory can be used by all threads, can be passed between kernels. It is also used to pass variables between CPU and GPU, because host memory (CPU) cannot be accessed by GPU, and vice versa. Global memory is the slowest memory, but we can reduce access time, if memory is properly coalesced, then, multiple threads can access different memory banks in one transition.

In the shared memory description I have used “block” term. To explain what it means, we need to be introduced to another hierarchy in CUDA programming – threads arrangement (Figure 3.7). As we can see, each thread is located in block and each block is located in grid. This classification is mainly used to design memory model for CUDA application.

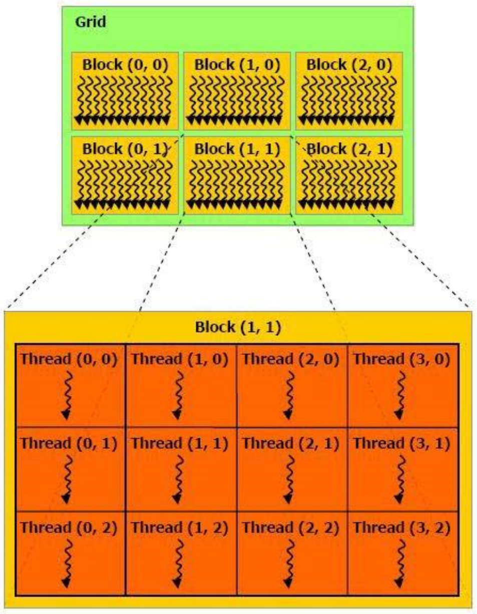
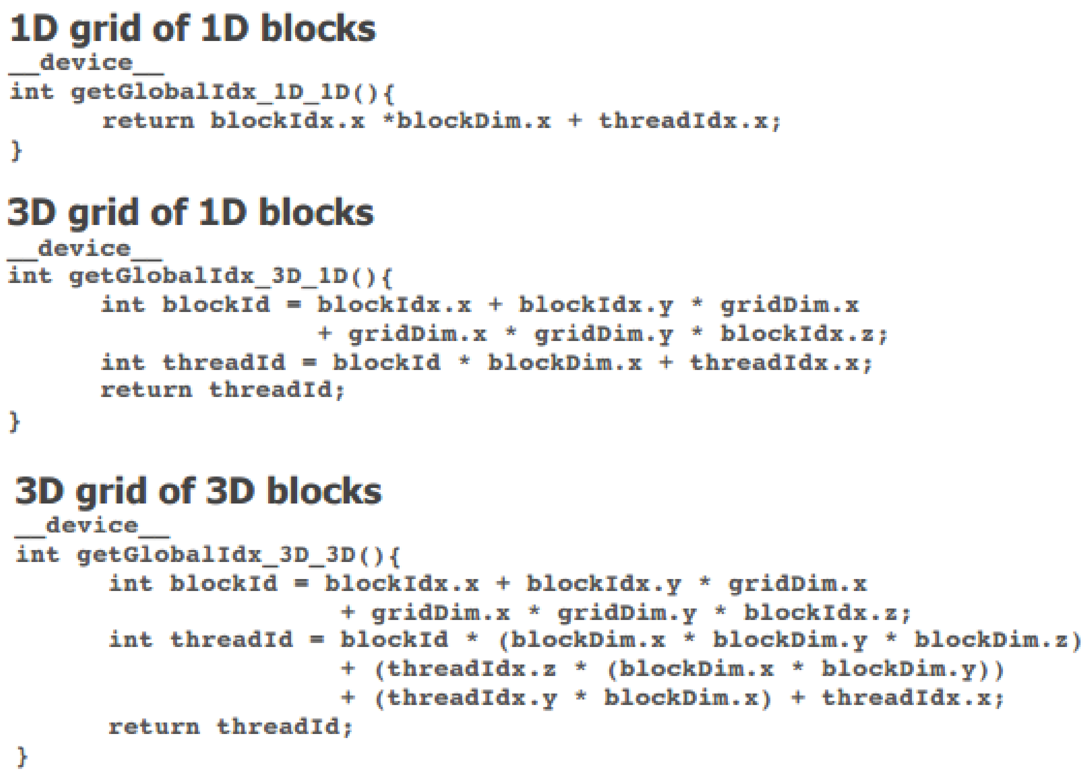


Figure 3.7: CUDA threads arrangement [29]

Cuda functions which are supposed to be run on GPU are called kernels. To define kernel function, we have to add following prefix to the definition “*__global__*”. It means , that the function is supposed to be run on GPU. Every “*__global__*” function cannot return anything, so type of those functions has to be “*void*”. When we want to call kernel function, we have to use following form: *“function<<<GS,BS>>>(a,b)”.* As we can see, there are strange “<<<>>>” signs, which are not used in any other programming language. They are used to specify grid size (GS) and block size (BS), which will be used to run this GPU kernel. Therefore, the total number of threads which will execute this kernel is GS*BS. We also have to remember that GS and BS can be 1,2 or 3-dimensional. Another important thing worth to notice is that we can allocate even a few millions of threads, but it does not mean that all the threads will run concurrently. It just means, that GS*BS threads will be used at all, not all in the same time.

Another very useful and important part of the CUDA programing is thread indexing. Each thread and block has its own index, which is used to calculate global thread index [30]. This index is used e.g. to access different variables from arrays allocated in global memory by different threads. Both blocks and grid can be indexed in many different ways, they can be one-dimensional, two-dimensional and three-dimensional. Figure 3.8 presents few examples of calculating global thread index.



1D grid of 1D blocks

```
__device__
int getGlobalIdx_1D_1D(){
      return blockIdx.x *blockDim.x + threadIdx.x;
}
```

3D grid of 1D blocks

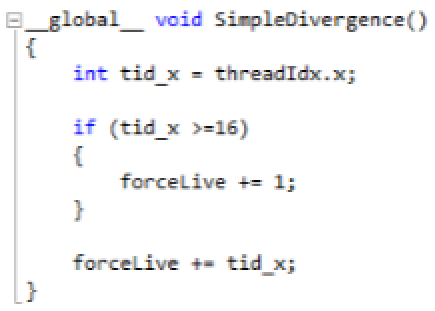
```
__device__
int getGlobalIdx_3D_1D(){
      int blockId = blockIdx.x + blockIdx.y * gridDim.x
                  + gridDim.x * gridDim.y * blockIdx.z;
      int threadId = blockId * blockDim.x + threadIdx.x;
      return threadId;
}
```

3D grid of 3D blocks

```
__device__
int getGlobalIdx_3D_3D(){
      int blockId = blockIdx.x + blockIdx.y * gridDim.x
                   + gridDim.x * gridDim.y * blockIdx.z;
      int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z)
                   + (threadIdx.z * (blockDim.x * blockDim.y))
                   + (threadIdx.y * blockDim.x) + threadIdx.x;
      return threadId;
}
```

Figure 3.8: Examples of global thread index calculations

There is also one more necessary term used in CUDA programming – warp. Warp is a group of 32 threads which are allocated automatically by CUDA mechanisms on SM. Each Streaming multiprocessor has 32 processing cores [31]. Therefore each processing core processes instructions of single thread of the same warp. However, each processing core within the same SM can execute only the same instruction. That is why we should avoid “warp divergence” [32], which occurs, when threads within the same warp are executing different instructions. Figure 3.9 shows an example code with the warp divergence. The warp size is 32, but 16 threads will execute some instruction, and another 16 threads within the warp will be inactive, but they cannot be assigned to do any other task, so this code just waste 16 threads, which could be used to calculate something else. This example showed us, if we want to do a couple of different tasks in the same kernel, we should split the work among full warps. Therefore, to optimise the example in the Figure 3.9 if statement should be changed to: tid_x<32.



```
__global__ void SimpleDivergence()
{
    int tid_x = threadIdx.x;

    if (tid_x >=16)
    {
        forceLive += 1;
    }

    forceLive += tid_x;
}
```

Figure 3.9: Example of the warp divergence

4 Results and discussion

4.1 Properties of used machines

To run the developed programs (3D LBM CUDA solver and LatticeGen) we need machines equipped with an NVIDIA graphics card. There have been used 3 different machines, to compare the results obtained on each of them. Properties of those machines can be found in Table 4.1.

	M1 - GRID	**M2 - FERMI**	**M3 – private laptop**
CPU	Intel E5-2660	Intel Xeon E5620	
RAM	64GB	24GB	
GPU	Tesla K40m	Tesla C2050	
CUDA capability	3.5	2.0	3.5
CUDA Cores	2880	448	
VRAM	12228 MB	3071 MB	

Table 4.1: Properties of used machines

4.2 Developing 2D lattice generator into 3D version

Lattice generator is a necessary software to generate appropriate input files for LBM solver simulations. 3D version of this program is based on 2D mesh generator, developed by Cranfield Msc Student in 2013 – Abbruzzese [14]

First step to adapt this software to handle 3D meshes, was to thoroughly analyse the existing 2D software. Therefore, there have been prepared flowchart presenting the algorithm to generate lattice files for 2D lbm solver (Figure 4.1)

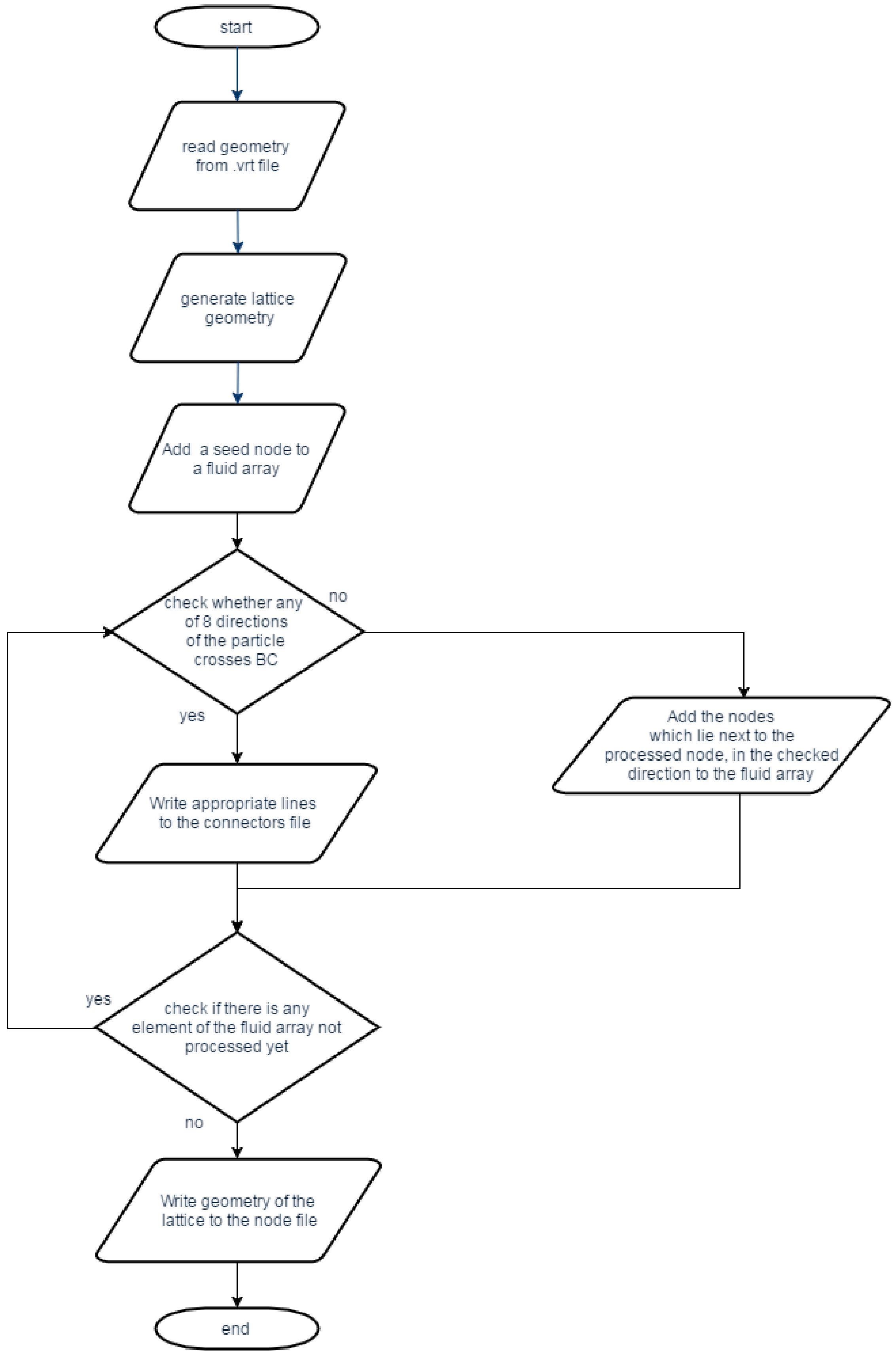

start
read geometry from .vrt file
generate lattice geometry
Add a seed node to a fluid array
check whether any of 8 directions of the particle crosses BC
no
yes
Add the nodes which lie next to the processed node, in the checked direction to the fluid array
Write appropriate lines to the connectors file
yes
check if there is any element of the fluid array not processed yet
no
Write geometry of the lattice to the node file
end

Figure 4.1: Flowchart of 2D mesh generator

Figure 4.1 presents very generally the idea of the algorithm to generate 2D lattice files. There are a lot of necessary details, not included in this flowchart, but overall approach to generate lattices has been presented in this figure.

Next step to develop 3D lattice generator, was analysing two input files used in 2D mesh generator, exported by Pointwise meshing software. These files has ".vrt" and ".bnd" extension. I have discovered, those files are STAR-CD files, which is popular CAE package, developed by CD-adapco company [33]. Those input files are generated by Pointwise software, however we are specifying the target solver while we prepare mesh. Therefore those files exported from Pointwise are designed for STAR-CD solver.

First file which is used by mesh generator is a ".vrt" file. This file contains 4 columns [34] - Table 4.2.

Vertex ID	Real coordinates		
	X	Y	Z

Table 4.2: ".vrt" file format

".vrt" file is used by mesh generator to read the geometry of the problem. Algorithm reads coordinates of the boundaries, and then it will use them to generate appropriate lattice. In 2D case, 4th column is always equal to zero, all the nodes are placed on the same Z direction coordinate. However, in 3D case, we will use also the 4th column.

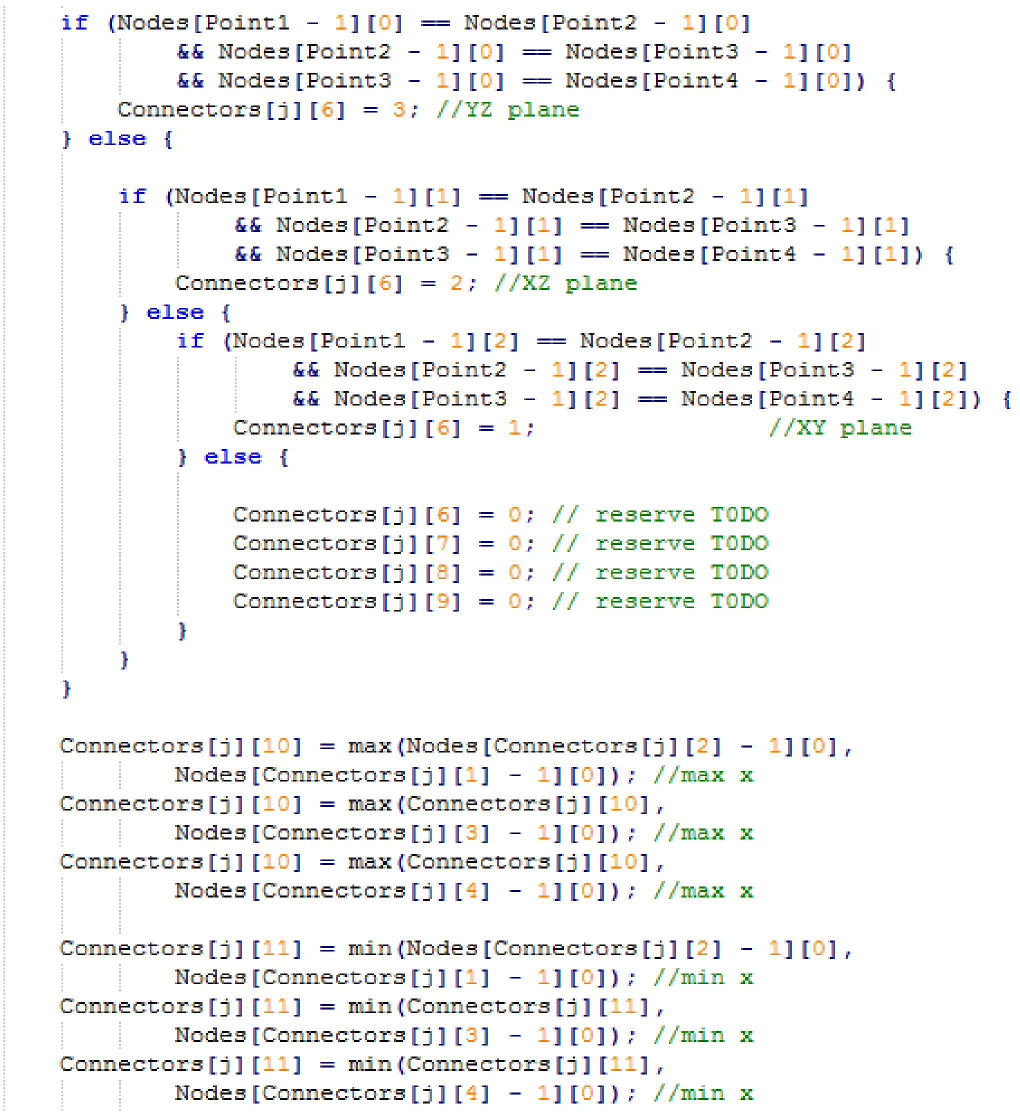
Second file used by mesh generator is a ".bnd" file - Table 4.3. This file consists of 8 columns and gives us information about boundaries of the mesh[34].

Boundary number	vertex				region number	patch number	region type
	#1	#2	#3	#4			

Table 4.3: ".bnd" file format

In 2D case, boundaries are spread between 2 vertexes, therefore 4th and 5th column is unused. 6th column is used to identify the surface of the boundary, it will be used in LBM solver to calculate drag/lift forces. Whereas the last column is the column with the type of boundary (e.g. wall, inlet, outlet, etc.). ".bnd" file is used in mesh generator to determine types and positions of the BC, which will be used then in "FluidTracking" function, to find the intersections of directions of the particles with BCs.

Analysing 2D version of mesh generator, it seems to be quite easy to adapt it into 3D version. At first glance, the only one chances necessary to do are transforming all 2D arrays into 3D (e.g. arrays which stores nodes in 2D/3D space), increase number of directions from 8 to 18 and read 4 vertexes for each boundary face instead of 2 vortexes (2D case). When I was implementing those changes, I realised, the biggest problem in transforming 2D into 3D version will not be programming issues, but maths. In 2D case we just have to find intersections of 2 lines (BCs and particle directions). However in 3D case we need to track down intersections of the line (directions) with a plane (boundary condition).Since the mesh is three-dimensional, BC has to be described by the plane, not only by the line, as in the 2D case. Therefore, intersection algorithms are much more complicated. In the final version, the 3D lattice generator is handling only BC faces which are parallel to XY, XZ or YZ planes. However, the code is prepared for implementation all BC faces handling. I have prepared relevant reserves in the arrays (Figure 4.2), structure of "intersections" class is also ready to implement sloping BCs without major changes in the structure of the code. Those unused cells in the "Connectors" array can be used e.g. to store normal vector of the sloping BC face. Normal vector would be helpful to find intersections of a line and a plane, which cannot be found easily (e.g. case when the direction of the particle is normal to BC face is very easy to manage).



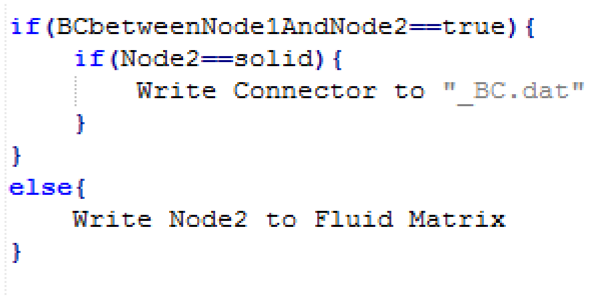
```
if (Nodes[Point1 - 1][0] == Nodes[Point2 - 1][0]
        && Nodes[Point2 - 1][0] == Nodes[Point3 - 1][0]
        && Nodes[Point3 - 1][0] == Nodes[Point4 - 1][0]) {
    Connectors[j][6] = 3; //YZ plane
} else {

    if (Nodes[Point1 - 1][1] == Nodes[Point2 - 1][1]
            && Nodes[Point2 - 1][1] == Nodes[Point3 - 1][1]
            && Nodes[Point3 - 1][1] == Nodes[Point4 - 1][1]) {
        Connectors[j][6] = 2; //XZ plane
    } else {
        if (Nodes[Point1 - 1][2] == Nodes[Point2 - 1][2]
                && Nodes[Point2 - 1][2] == Nodes[Point3 - 1][2]
                && Nodes[Point3 - 1][2] == Nodes[Point4 - 1][2]) {
            Connectors[j][6] = 1;                   //XY plane
        } else {

            Connectors[j][6] = 0; // reserve TODO
            Connectors[j][7] = 0; // reserve TODO
            Connectors[j][8] = 0; // reserve TODO
            Connectors[j][9] = 0; // reserve TODO
        }
    }
}

Connectors[j][10] = max(Nodes[Connectors[j][2] - 1][0],
        Nodes[Connectors[j][1] - 1][0]); //max x
Connectors[j][10] = max(Connectors[j][10],
        Nodes[Connectors[j][3] - 1][0]); //max x
Connectors[j][10] = max(Connectors[j][10],
        Nodes[Connectors[j][4] - 1][0]); //max x

Connectors[j][11] = min(Nodes[Connectors[j][2] - 1][0],
        Nodes[Connectors[j][1] - 1][0]); //min x
Connectors[j][11] = min(Connectors[j][11],
        Nodes[Connectors[j][3] - 1][0]); //min x
Connectors[j][11] = min(Connectors[j][11],
        Nodes[Connectors[j][4] - 1][0]); //min x
```

Figure 4.2: part of the LatticeGen code, prepared for development

At the beginning of work on the LatticeGen, I have assumed that the 3D LBM solver will use off-grid lattices (Figure 2.10), as the 2D version. However, during Aguilar's work on physics models of the solver [23], we realised, that we will need on-grid lattices (Figure 2.11) for the solver. Theoretically, it was easy to adapt the code to produce on-grid lattices, however, to develop general code, capable to process different geometries, it was necessary to implement quite complex algorithms.

On-grid lattice means, that the lattice node is placed exactly on the BC. Basic algorithm of intersection class checks if there is any BC between 2 nodes. We should notice, that the first node taken to intersection function, is the fluid node, previously classified (in the previous iteration of the "FluidTracking") as fluid. Second node is unknown at this moment, we will classify it as fluid or solid, it depends on that, if any BC is placed between those two nodes (Figure 4.1).

To adapt the algorithm to handle on-grid lattices, we should skip the situations when the second node taken to intersection function is placed on BC, because we want to classify the nodes which lie on BC as fluids. Therefore, in next iteration, we will check all directions from the node, which is placed exactly on the BC. Using the first algorithm for "FluidTracking", we would classify all directions of this node as connectors to "_BC.dat" file, because the algorithm will find the boundary in the position of the first node. Therefore, it was necessary to add another condition - Figure 4.3. Now, the second node has to be solid, to classify specific direction as connector for the LBM solver.



```
if(BCbetweenNode1AndNode2==true){
    if(Node2==solid){
        Write Connector to "_BC.dat"
    }
}
else{
    Write Node2 to Fluid Matrix
}
```

Figure 4.3: Pseudocode of the "FluidTracking"

The code after introducing previously described changes was able to produce correct output files ("_BC.dat" and "_Node.dat") only for simply geometries. In case when the mesh was more complicated than the cavity or the channel, generated output contained bugs, e.g. for the channel with a cylinder inside. It was caused by the fact that "FluidTracking" algorithm checks in the same iteration whether the neighbour node is solid or fluid, and then the code wrote appropriate information to connectors file. However, when the geometry was more complex, the neighbour node which was still solid (so connector will be written), can change his state to fluid, e.g. when the algorithm encounter this node from different direction. Therefore, I have implemented two options of LatticeGen: simple geometry and complex geometry. The first option is the same as I just described, but the second option firstly just recognizes fluid nodes, and adds them to the "FluidMatrix". Then, second time there is executed the main loop of "FluidTracking", but now it will only write connectors to the output file. The advantage of the "complex geometry" option of the software is that we already know fluid/solid states of all nodes, so the condition described in previous paragraph will work correctly for every mesh. Disadvantage of this solution is the runtime, it will be slower than the normal option of the program, because there will be executed twice the same, main loop.

4.3 Developing 2D LBM solver code into 3D version

The main task of this thesis was to adapt the existing 2D LBM CUDA solver [12,[13] into 3D version. It was realised with another student - Alfonso Aguilar [23], who was responsible for the physics of the solver, while I was responsible for the software aspects.

First step to develop the existing code is always thorough analysis of the code. To achieve it, there were helpful both the codes and the theses about those implementations. I have prepared simple flowchart of the solver, to clarify all the steps of this software – Figure 4.4.This existing 2D code was quite well commented, there was also prepared Doxygen documentation, therefore it was possible to familiarize with the code quickly and without any bigger problems.

Next step in developing the 3D version was to prepare new arrays, capable to store 3D molecule models (Figure 2.2). These arrays were designed e.g. to store distribution function, or information about allowed directions for streaming. We had to also add a few another variables/arrays, to store values for the third dimension, e.g. velocity along the z axis, etc.

All physical models, e.g. 3D boundary conditions, etc. were prepared by Aguilar [23], therefore I will not describe the implementation of the physics in this thesis.

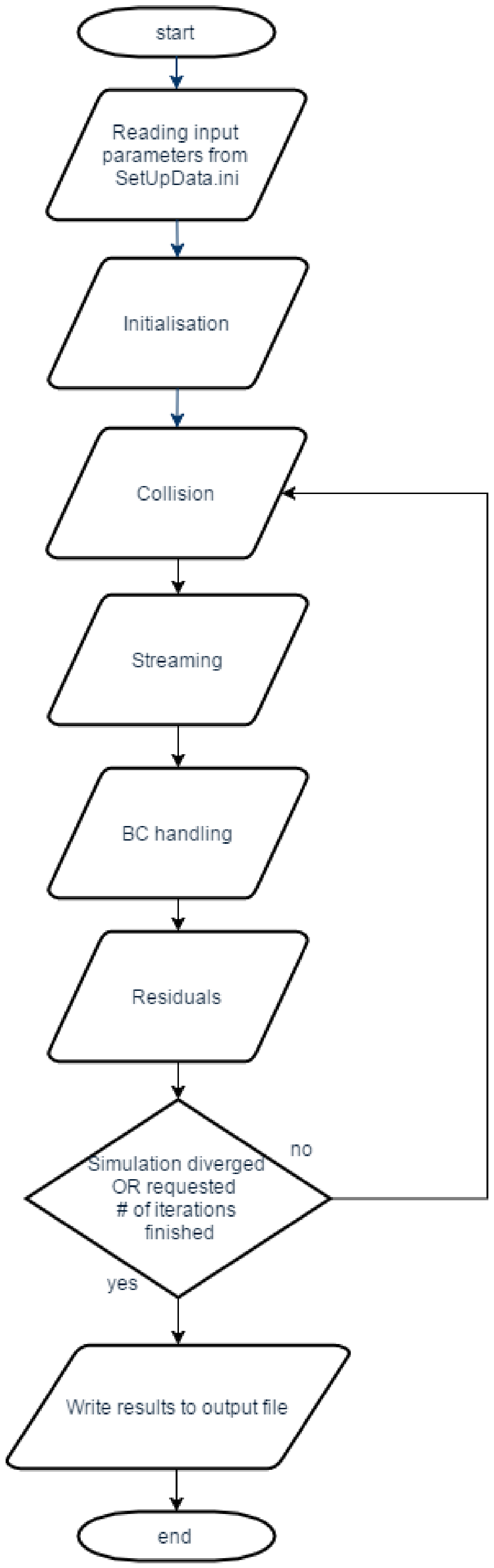
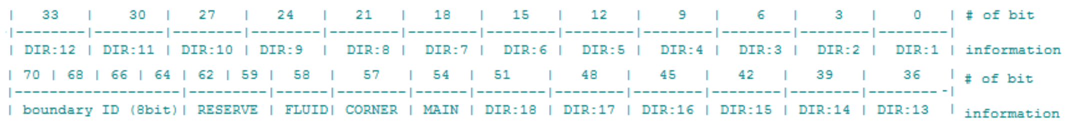

start
Reading input parameters from SetUpData.ini
Initialisation
Collision
Streaming
BC handling
Residuals
Simulation diverged OR requested # of iterations finished
no
yes
Write results to output file
end

Figure 4.4: General flowchart of the existing 2D CUDA LBM solver

Koleszár [13] during his work on the first version of the 2D LBM CUDA solver, has implemented bitmask used for storing information about the boundary conditions of the node. This idea saves a lot of memory, because in the 2D version he just needed one int variable (8 bytes) per node, to store all necessary information of the BCs for one node. I appreciated this idea of storing BCs, therefore I have decided to adapt this bitmask into 3D version. However, 3D version uses D3Q19 molecule, what means, we have 19 directions instead of 8 (2D case). Second issue which I had to challenge, was new types of BCs – periodic and symmetric. Now we have 5 types of BCs, therefore, 2 bits will not be enough anymore to store the information about type of the BCs for the particular direction, as it was implemented in 2D version, where they have used only 3 types of BCs (wall, inlet, outlet). Those two issues (new directions and new types of BCs) caused that previously used datatype of the bitmask – integer, was not enough anymore to store the BCs for 3D implementation. Figure 4.5 presents new version of the bitmask, designed to handle 3D version of the LBM solver. As we can see, we need 70 bits, to store all necessary information, while the “int” type provides only 64 bits. Therefore, I had to change the datatype of this array to “unsigned long long”, which can store 16 bytes – 128 bits, what is enough for the new BCmask.



```
|   33    |    30   |   27    |   24    |   21    |   18    |   15    |   12    |    9    |    6    |    3    |    0    | # of bit
|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|
| DIR:12  | DIR:11  | DIR:10  | DIR:9   |  DIR:8  |  DIR:7  |  DIR:6  |  DIR:5  |  DIR:4  |  DIR:3  |  DIR:2  |  DIR:1  | information
| 70 | 68 | 66 | 64 | 62 | 59 |   58   |    57   |   54   |   51    |   48    |   45    |   42    |   39    |   36    | # of bit
|-------------------|----------|-------|---------|-------|---------|---------|---------|---------|---------|---------|
| boundary ID (8bit)| RESERVE  | FLUID| CORNER  | MAIN  | DIR:18  | DIR:17  | DIR:16  | DIR:15  | DIR:14  | DIR:13  | information
```

Figure 4.5: BCmask - 3D version

Implementing 3D version of the LBM solver, leads to significantly increasing problem sizes, what also means increasing output files, containing the results from the simulations. Most simulations of our solver will be executed on HPCs, so the results will have to be downloaded on our own computers, for further post processing. Therefore the output file sizes are the issue which should be considered during the development process. 2D version of the solver has implemented two types of the output files: ".csv" and ".dat". The first one can be used e.g. by Excel, whereas the second one is designed to be read by Tecplot software, which is professional commercial plotting software. Because of the fact, that the Tecplot is not a free software, I decided to implement new type of output file format, which could be read by some open-source plotting program, e.g. ParaView. I have chosen ".vti" format, which is a native format for ParaView [38]. In opposition to the ".dat" Tecplot format, ".vti" does not contain real coordinates of each node, this format allows us to represent the domain as structured mesh, what means that we just need to specify problem size and spacing. It requires only 3 lines in the file, instead of writing all three coordinates of each node for the ".dat" format. In the result of the implementation of the new format file, we obtained quite impressive memory saving - Figure 4.6. Smaller file size does not only mean smaller storage usage, but it also means that we can download the results from the HPC faster and the post processing can be performed faster.

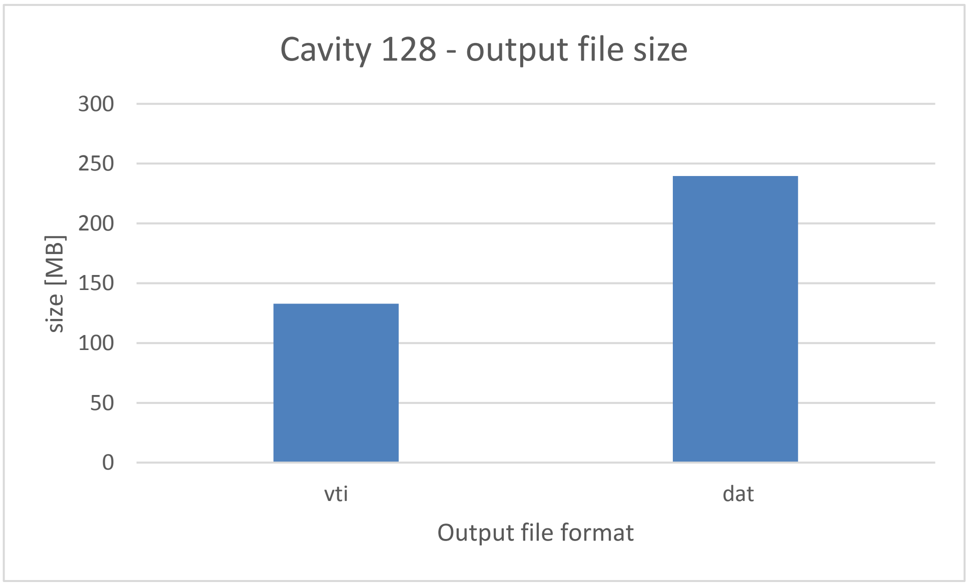


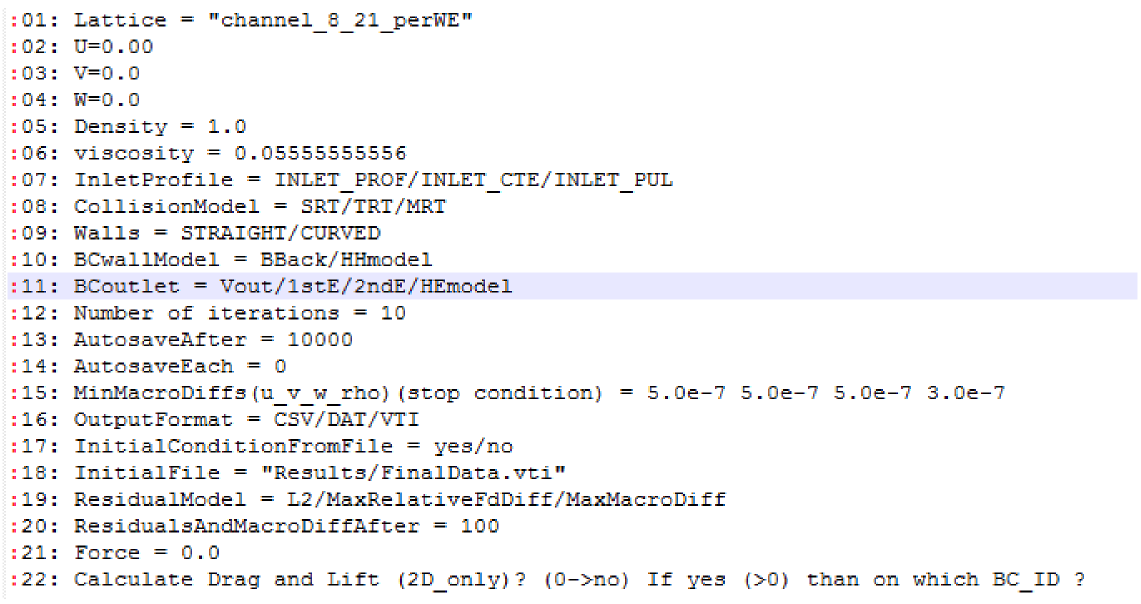
Figure 4.6: Output file sizes

During the development process of the 3D LBM solver, me and Aguilar [23] have many ideas, how to improve the usability of the solver. First way to make the solver more usable, was a few modifications about the residuals. As I will explain further (section 4.6 and 4.7), residuals are not necessary to produce correct results, but they are significantly slowing down the program. Therefore, I have implemented possibility to choose how often we want to calculate the residuals. For example, when we want to simulate 10 million iterations, it is meaningless to calculate residuals after each iteration. It would be totally sufficient to calculate residuals e.g. after each 10000 iterations, we would be still able to monitor the state of the simulation (converged/diverged), but our simulation would be much faster.

Another new feature of the solver, comparing to the 2D version, is stop condition. We can automatically terminate the simulation, when it is already converged and there is no sense to still calculate new iterations, because they will not produce anything new. This feature is based on comparing residuals with the specified stop condition, if the residuals are smaller than the stop condition, the main loop will be terminated, and the results will be written into the file.

3D problem cases can be really huge, therefore the simulations can last e.g. a couple of days. Therefore, it would be useful, to be able to “restore” the simulation when the results are not satisfying us, and we would like to simulate more iterations, but starting from the beginning would be really time consuming. Therefore I implemented a possibility to load output files from the simulation as the initial conditions of the physical problem – simulated problem will be initiated by the loaded macro values – u, v, w, rho. This feature lets us to restore the simulation from the result file. It is not perfectly restoring, because we are initialising only macroscopic values, not the distribution functions, but in collision step, the major influence on the new value of the distribution function comes from previous macroscopic values of the node, therefore our idea for restoring the simulation is quite accurate and it requires only the output file from the previous simulation.

Figure 4.7 presents the actual form of the SetUpData.ini file, which is used to initialise the input parameters. In this file we can see parametrisation of the new features of the solver, described above.

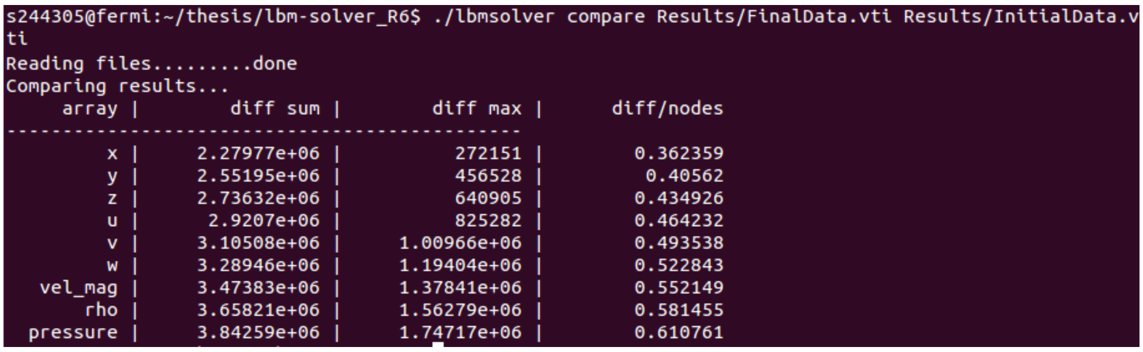


```
Lattice = "channel_8_21_perWE"
U=0.00
V=0.0
W=0.0
Density = 1.0
viscosity = 0.05555555556
InletProfile = INLET_PROF/INLET_CTE/INLET_PUL
CollisionModel = SRT/TRT/MRT
Walls = STRAIGHT/CURVED
BCwallModel = BBack/HHmodel
BCoutlet = Vout/1stE/2ndE/HEmodel
Number of iterations = 10
AutosaveAfter = 10000
AutosaveEach = 0
MinMacroDiffs(u_v_w_rho)(stop condition) = 5.0e-7 5.0e-7 5.0e-7 3.0e-7
OutputFormat = CSV/DAT/VTI
InitialConditionFromFile = yes/no
InitialFile = "Results/FinalData.vti"
ResidualModel = L2/MaxRelativeFdDiff/MaxMacroDiff
ResidualsAndMacroDiffAfter = 100
Force = 0.0
Calculate Drag and Lift (2D_only)? (0->no) If yes (>0) than on which BC_ID ?
```

Figure 4.7: File with an input parameters (SetUpData.ini)

4.4 Verification and validation

Developed 3D version of the CUDA LBM solver has been verified and validated by Aguilar [23], who was responsible for the physics of the solver. Verification of the code - Appendix B, shows that the models have been implemented correctly. Whereas validation - Appendix C proved, that the models were correct, therefore our LBM solver is able to simulate and produce correct results. The solver has been validated with two different sets of reference data for lid-driven cavity simulations [35,[36]. There has been also showed results of the simulation for lid drive cavity - Appendix D. Once validated results by Aguilar has been used as reference results, helpful to validate the code after optimisation changes. To do that, there has been implemented appropriate functionality of the solver – output file comparator (Figure 4.8). This is a high speed comparator of the output files, written in CUDA as well as the solver. The slowest part of this feature of the solver is reading files into memory. After that, comparison is really fast, because of used high optimised CUDA kernels, used for comparing values from 2 different files.



```
s244305@fermi:~/thesis/lbm-solver_R6$ ./lbmsolver compare Results/FinalData.vti Results/InitialData.v
ti
Reading files.........done
Comparing results...
     array |        diff sum |        diff max |      diff/nodes
--------------------------------------------
         x |     2.27977e+06 |          272151 |        0.362359
         y |     2.55195e+06 |          456528 |         0.40562
         z |     2.73632e+06 |          640905 |        0.434926
         u |      2.9207e+06 |          825282 |        0.464232
         v |     3.10508e+06 |     1.00966e+06 |        0.493538
         w |     3.28946e+06 |     1.19404e+06 |        0.522843
   vel_mag |     3.47383e+06 |     1.37841e+06 |        0.552149
       rho |     3.65821e+06 |     1.56279e+06 |        0.581455
  pressure |     3.84259e+06 |     1.74717e+06 |        0.610761
```

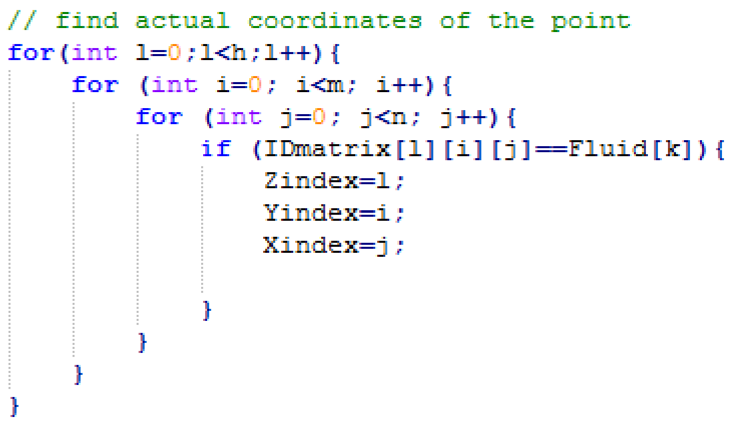
Figure 4.8: Example output from the file comparator

4.5 LatticeGen optimisation

First working version of LatticeGen, directly adapted from 2D version [14] was too slow to produce necessary lattices in a reasonable time. Therefore, it was necessary to optimise the code. 2D lattices can be relatively small, to simulate proper physical problems. However, to simulate real problems in three-dimensional space, inputted lattices have to be quite huge, to provide convergence of the solution and other necessary conditions.

To find the bottlenecks of the LatticeGen, firstly I have measured runtimes of each part of the code. Then I have realised, that over 90% time of the overall runtime was consumed by “FluidTracking” function. After that, I have measured each sub step of this algorithm. Finally, I have found two very slowing down parts of the code - unnecessarily nested loops:

First of the main two bottlenecks was following part of the code:



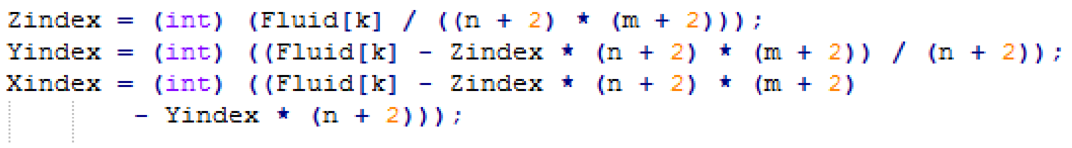
```
// find actual coordinates of the point
for(int l=0;l<h;l++){
    for (int i=0; i<m; i++){
        for (int j=0; j<n; j++){
            if (IDmatrix[l][i][j]==Fluid[k]){
                Zindex=l;
                Yindex=i;
                Xindex=j;

            }
        }
    }
}
```

Figure 4.9: The first bottleneck of the LatticeGen

As we can see in the Figure 4.9, information about fluid nodes is stored in 1D array, which is flattened in order to simplify another parts of the algorithm. Code showed in the Figure 4.9 is supposed to find appropriate element in 3D array which is equivalent to the element in the 1D Fluid array. Flattening is realised by following calculation:

$$Index_{1D} = Index_x + Index_y * size_x + Index_z * size_x * size_y \qquad \textbf{(4.1)}$$

Therefore, it is possible to do opposite operation analytically, not iteratively, as presented in the Figure 4.3. Thus the presented code has been rebuilt in order to improve the performance of the code:



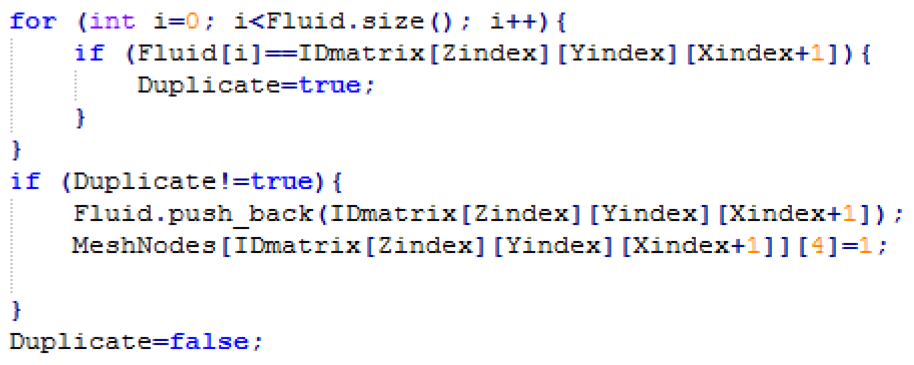
```
Zindex = (int) (Fluid[k] / ((n + 2) * (m + 2)));
Yindex = (int) ((Fluid[k] - Zindex * (n + 2) * (m + 2)) / (n + 2));
Xindex = (int) ((Fluid[k] - Zindex * (n + 2) * (m + 2)
        - Yindex * (n + 2)));
```

Figure 4.10: Optimised first bottleneck of the LatticeGen

Code presented in the Figure 4.9 noticeable reduced the runtimes of the software (exact comparisons of the runtimes will be showed further). Those unnecessary loops in the first version produces $n * m * h * FluidSize$ iterations, which were completely useless, as we can see.

Rewriting the first found bottleneck gave us huge speedup, however to prepare lattices appropriate for high Reynold number simulations, the performance was still not sufficient. Therefore, the optimisation of the code was continued.

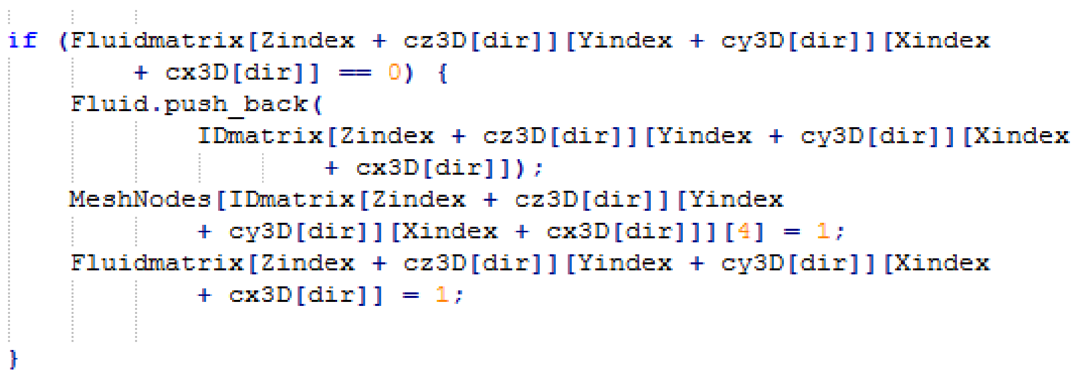
Second part of the code, which causes a significant increase of the runtimes was following code:



```
for (int i=0; i<Fluid.size(); i++){
    if (Fluid[i]==IDmatrix[Zindex][Yindex][Xindex+1]){
        Duplicate=true;
    }
}
if (Duplicate!=true){
    Fluid.push_back(IDmatrix[Zindex][Yindex][Xindex+1]);
    MeshNodes[IDmatrix[Zindex][Yindex][Xindex+1]][4]=1;

}
Duplicate=false;
```

Figure 4.11: The second bottleneck of the LatticeGen

Figure 4.11 presents algorithm for avoiding duplication of the Fluid nodes. This functionality is necessary, because we could encounter the same node from different directions, what could lead to classify multiple times the same node as the fluid. Algorithm presented above was directly adapted from Abbruzzese’s 2D mesh generator [14] to my 3D implementation. However, as I mentioned before, 2D cases were much smaller and less time consuming, therefore this code was acceptable in that version. However to handle 3D meshes, it was necessary to rebuild (in order to optimise) that code:



```
if (Fluidmatrix[Zindex + cz3D[dir]][Yindex + cy3D[dir]][Xindex
        + cx3D[dir]] == 0) {
    Fluid.push_back(
            IDmatrix[Zindex + cz3D[dir]][Yindex + cy3D[dir]][Xindex
                    + cx3D[dir]]);
    MeshNodes[IDmatrix[Zindex + cz3D[dir]][Yindex
            + cy3D[dir]][Xindex + cx3D[dir]]][4] = 1;
    Fluidmatrix[Zindex + cz3D[dir]][Yindex + cy3D[dir]][Xindex
            + cx3D[dir]] = 1;

}
```

Figure 4.12: Optimised second bottleneck of the LatticeGen

Figure 4.12 presents my solution to optimise that code. I have removed the loop for iterating through whole fluid array to check whether the analysed node is already placed in the Fluid array. Instead of iterating through the 1D array, I have prepared 3D array containing information about type of the node (fluid=1, solid=0). Implementing 3D matrix allows us to directly address specified cell of this matrix, which will contain information about state of the node. Elements of the 3D matrix are ordered in the same way as the nodes in the lattice, therefore we exactly know the indices of the “Fluidmatrix” array which we want to check. Old, “Fluid” array was not ordered, elements were written into this array in order as the nodes were encounter in “Fluidtracking” algorithm. Therefore it was impossible to simply address the specified node. My implemented solution needs a little bit more memory, but the speed-up of this solution is huge, so it was worth to sacrifice that small amount of memory.

Another issue which affects performance of the LatticeGen was meshes generated by Pointwise. In the beginning, when I was not familiarized with the Pointwise, I was generating unnecessarily complicated meshes, for example to generate lattice for lid-driven cavity, we just need mesh with a 8 nodes, on each corner of the cube. This mesh will be sufficient to give all necessary information about BC and the geometry of the cube. However, in the beginning my cavity mesh had 5 nodes in each dimension, what gives total number of nodes equal to 125. It leads to have unnecessary lines in “.bnd” file, which will be iterated many times during the “FluidTracking” algorithm. “.bnd” file contains information about the BC for each cell (face spread among 4 vertexes), therefore, if we have redundant nodes in our mesh, we will also have unnecessary cells and lines in ".bnd" file, which could be replaced by one big cell (case when cavity has just 8 nodes). Figure 4.13 – shows the influence of the mesh complexity on the runtime of the LatticeGen. This comparison was performed on the final version of the software, using M2 machine.

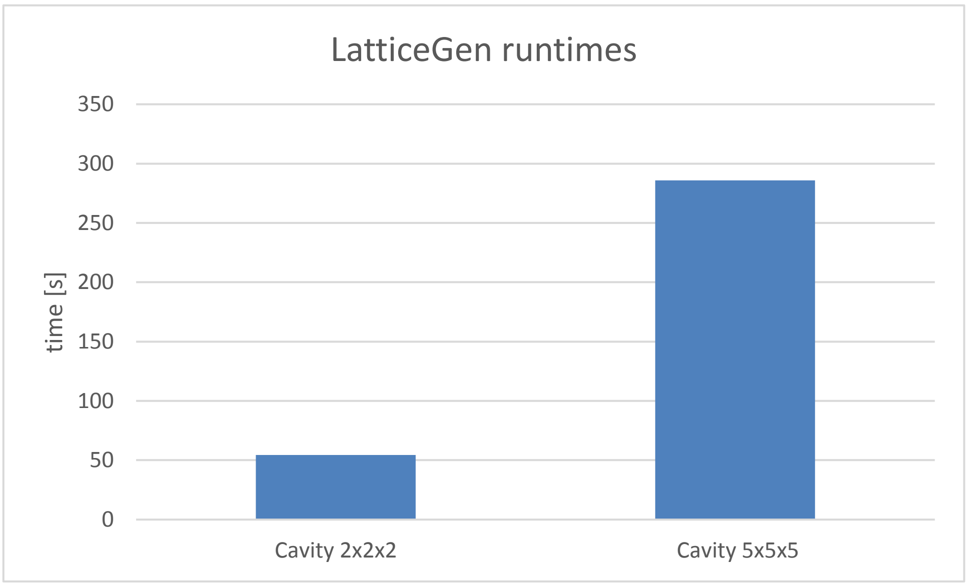


Figure 4.13: Influence of the mesh complexity on the runtime

Figure 4.14 and Figure 4.15 present the results of the optimisation, obtained on M3 machine. We can notice that the speedup is colossal. Finally, the LatticeGen lets us to generate huge lattices in a reasonable time - Figure 4.16.

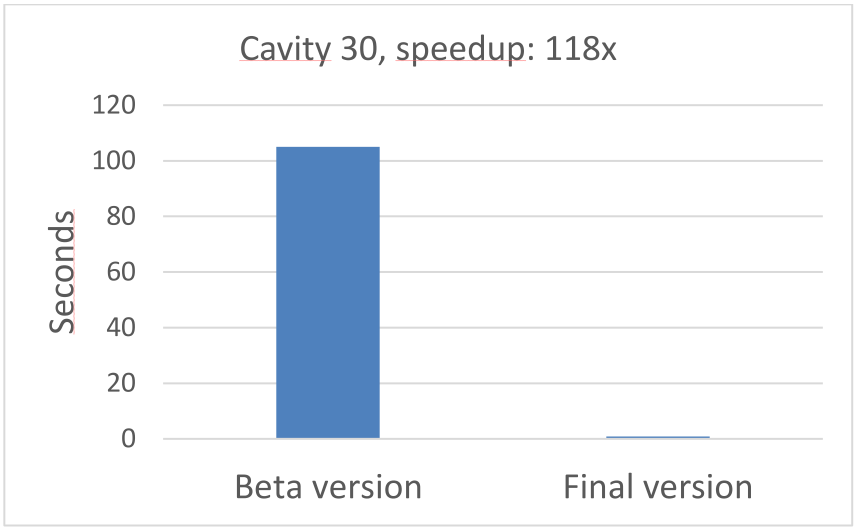


Figure 4.14: LatticeGen runtimes for Cavity 30

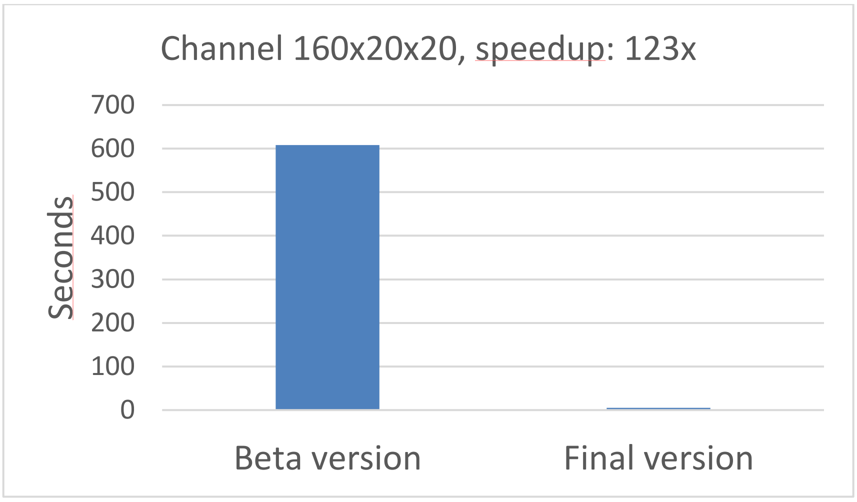


Figure 4.15: LatticeGen runtimes for channel 160x20x20

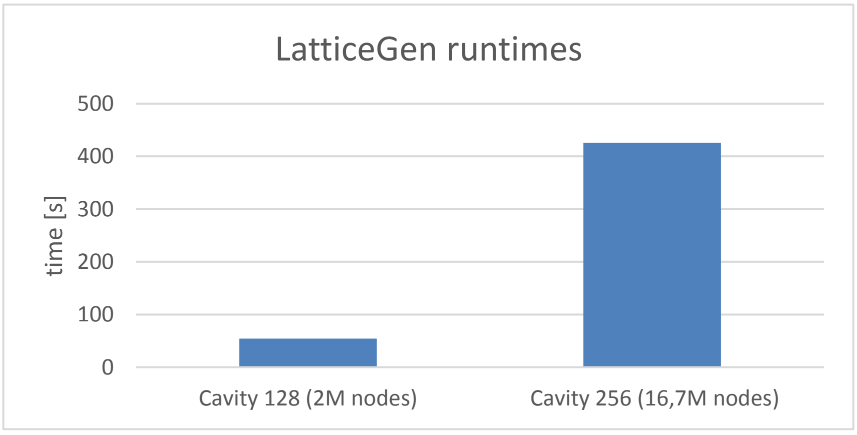


Figure 4.16: LatticeGen runtimes (M2 machine)

4.6 Memory usage of 3D LBM solver

Memory usage of the LBM solver is a very important issue for practical applications. If there is a need to simulate high Reynold number problems, there are three options to achieve it – eq. (4.2). First option is to increase velocity, however LBM has a limitation for maximum Mach number – exceeding the limit (Mach < 0.3) usually leads to the divergence of the simulation. Mach number depends only on the value of the velocity – eq. (4.3). Therefore we cannot reach high Reynold number by increasing velocity. Second option is to decrease viscosity, but it will also cause divergence of the simulation. Therefore the only one way to increase Re and keep the simulation stable is increasing problem size (Nc).

$$Re = \frac{UcNc}{\nu} \tag{4.2}$$

$$Ma = \sqrt{3}\mathrm{Uc} \tag{4.3}$$

To simulate bigger lattices, obviously we need bigger memory resources. There is a couple of ways to achieve that:

- Use better GPU
- Use multiple GPUs
- Optimize the code

The best option is to optimize the code, because it will give us double profit when we will also use better/multiple GPUs.

First step in reducing memory usage is thoroughly analysing all the arrays allocated on GPU. Therefore I have prepared relevant calculation sheet, which will be helpful to analyse the memory usage and then to implement changes which will decrease memory consumption- Table 4.4.

Lid Driven Cavity 256					
n	m	h	NumNodes	NumConns	bcCount
256	256	256	16777216	1963008	390152
Float size	int size	unsigned long long size			
4	4	8			
Original version					
size in B	size in KB	size in MB	Type	Name	Size
67108864	65536	64	int	fluid_d	numNodes
67108864	65536	64	FLOAT_TYPE	coordX_d	numNodes
67108864	65536	64	FLOAT_TYPE	coordY_d	numNodes
67108864	65536	64	FLOAT_TYPE	coordZ_d	numNodes
7852032	7668	7,48828125	int	bcNodeIdX_d	numConns
7852032	7668	7,48828125	int	bcNodeIdY_d	numConns
7852032	7668	7,48828125	int	bcNodeIdZ_d	numConns
7852032	7668	7,48828125	int	latticeId_d	numConns
7852032	7668	7,48828125	int	bcType_d	numConns
7852032	7668	7,48828125	int	bcBoundId_d	numConns
7852032	7668	7,48828125	FLOAT_TYPE	bcX_d	numConns
7852032	7668	7,48828125	FLOAT_TYPE	bcY_d	numConns
7852032	7668	7,48828125	FLOAT_TYPE	bcZ_d	numConns
67108864	65536	64	FLOAT_TYPE	rho_d	numNodes
67108864	65536	64	FLOAT_TYPE	u1_d	numNodes
67108864	65536	64	FLOAT_TYPE	v1_d	numNodes
67108864	65536	64	FLOAT_TYPE	w1_d	numNodes
67108864	65536	64	FLOAT_TYPE	u_prev_d	numNodes
67108864	65536	64	FLOAT_TYPE	v_prev_d	numNodes
67108864	65536	64	FLOAT_TYPE	w_prev_d	numNodes
67108864	65536	64	FLOAT_TYPE	rho_prev_d	numNodes
1275068416	1245184	1216	FLOAT_TYPE	f_d	19*NumNodes
1275068416	1245184	1216	FLOAT_TYPE	fColl_d	19*NumNodes
1275068416	1245184	1216	FLOAT_TYPE	fprev_d	19*NumNodes
1275068416	1245184	1216	FLOAT_TYPE	temp19a_d	19*NumNodes
1275068416	1245184	1216	FLOAT_TYPE	temp19b _d	19*NumNodes
67108864	65536	64	FLOAT_TYPE	tempA_d	NumNodes
67108864	65536	64	FLOAT_TYPE	tempB_d	NumNodes
67108864	65536	64	FLOAT_TYPE	u_d	NumNodes
67108864	65536	64	FLOAT_TYPE	v_d	NumNodes
67108864	65536	64	FLOAT_TYPE	w_d	NumNodes
134217728	131072	128	unsigned long long	bcMask_d	NumNodes
1560608	1524,03125	1,488311768	int	bcIdxCollapsed_d	bcCount
3121216	3048,0625	2,976623535	unsigned long long	bcMaskCollapsed_d	bcCount
28090944	27432,5625	26,78961182	FLOAT_TYPE	qCollapsed_d	18*bcCount
1207959552	1179648	1152	FLOAT_TYPE	q_d	18*NumNodes
1275068416	1245184	1216	FLOAT_TYPE	f1_d	19*NumNodes
67108864	65536	64	int	bcIdx_d	NumNodes
1207959552	1179648	1152	int	stream_d	18*NumNodes
TOTAL MEMORY					
		10978,64908	MB		

Table 4.4: Memory usage for Lid-driven cavity 256, single precision (beta version of 3D LBM solver)

The calculations presented in Table 4.4 have been validated by comparing them with measured memory usage. There is a comparison of 4 options of the solver:

Option	Residuals type	Stream_d data type
0	Beta version	
1	MaxRelDiff	int
2	MaxRellDiff	bool
3	MaxMacroDiff	int
4	MaxMacroDiff	bool

Table 4.5: 4 Options of LBM solver, analysed in memory usage section

All of those options will be described in further part of this section. Validation proved, that estimation of memory usage prepared by author of this thesis is correct – maximum difference between measured and estimated value is less than 1,5%.

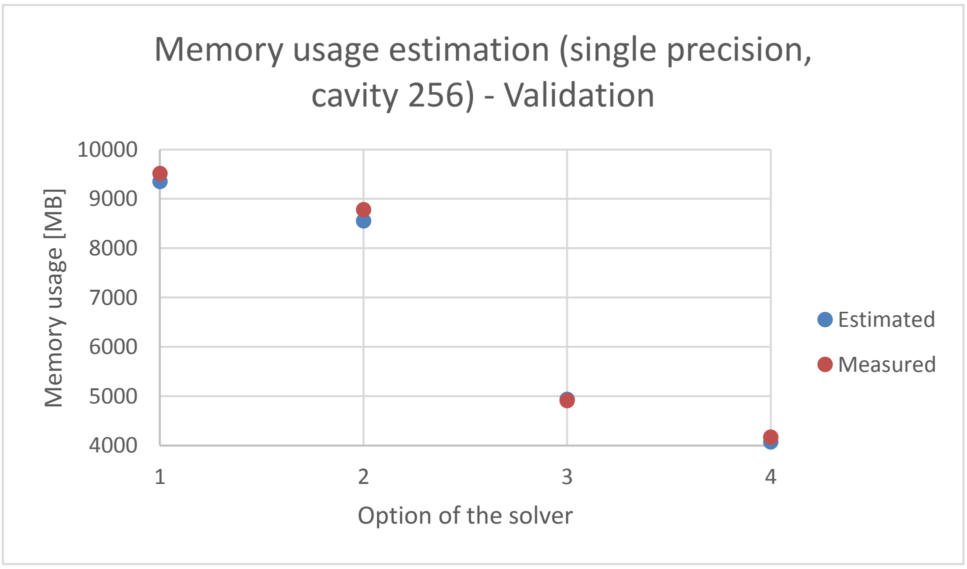


Figure 4.17: Memory usage estimation (single precision, cavity 256) - Validation

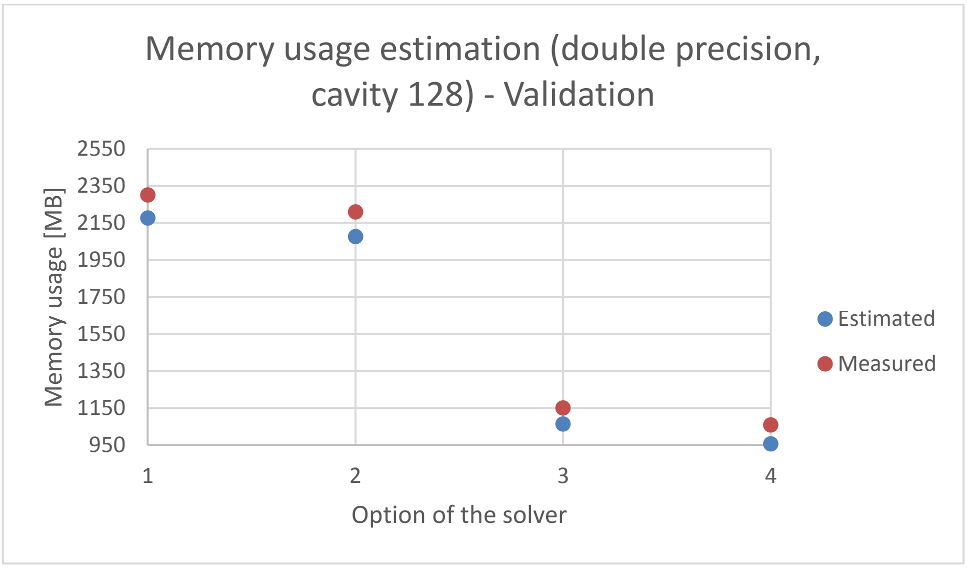


Figure 4.18: Memory usage estimation (double precision, cavity 128) - Validation

At first glance on the Table 4.4, we can see that arrays used to calculate residuals are the most memory consuming. Firstly implemented residuals was adapted from the previous, 2D version of the solver. Authors of that solver implemented L2 norm of distribution function as the residuals. There has been also implemented second type of residuals – maximum relative difference of distribution function – eq. (4.4). The main disadvantage of it, is size of the distribution function, which is equal to $19 * NoOfNodes$. Secondly, residuals based on distribution function are not easy to analyse by the user. More intuitive values for the CFD engineer to analyse are macroscopic values. Moreover, each node has 4 macroscopic values, instead of 19 values of distribution function. Therefore, there has been implemented another residuals, based on macroscopic values - (4.5)

$$MaxFdRelDiff = Max(\frac{f_n - f_{n-1}}{f_1}) \tag{4.4}$$

$$MaxMacroDiff\_u = Max(u_n - u_{n-1}) \tag{4.5}$$

Using this type of residuals, we do not need temp19a_d, temp19b_d, fprev_d arrays, which were the biggest arrays. First two arrays were used as a temporary arrays, necessary to implement reduction algorithms on GPU (e.g. finding maximum value of the array). Implementing Macro residuals, we can calculate each of 4 residuals separately, so we can reduce the size of the two temporary arrays 19x. We also do not need to store each of 19 distribution functions per node, we just need 4 macro values per each node.

In the beta version of the solver, with implemented all three types of the residuals, there have been allocated all the GPU arrays, even though they were not necessary - Table 4.4. Therefore the next step in memory optimisation was to only allocate necessary arrays. To achieve it, in the initialisation step, there will be checked which type of residual is chosen by the user, and basing on this information, appropriate arrays will be allocated on GPU.

Another noticed memory issue, is the stream_d array. This array contains only two values: 0 or 1. It is used to determine whether streaming is allowed or not, for particular direction of particular molecule. To store two values, we need only 1 bit, however in 2D version of the solver, and then in my 3D version, adapted directly from 2D version, data type of stream_d is an integer (4 bytes = 32 bits). As we can remark, each element of stream_d array was unnecessarily 32x bigger. Stream_d array has 18*NumNodes elements, so the difference in size is quite noticeable.

Implementation of all the changes described above, gave us huge memory saving - Table 4.6. Using M1 machine, which has 12GB of VRAM, now we can simulate lattices with 50M nodes, instead of 20M nodes, using the beta version of the solver (Table 4.4). That memory optimisation lets us to simulate lid-driven cavity with Re=1000 using M1.

Lid Driven Cavity 256					
n	m	h	NumNodes	NumConns	bcCount
256	256	256	16777216	1963008	390152
Float size	int size	unsigned long long size			
4	4	8			
MaxMacroDiff, bool Stream_d					
size in B	size in KB	size in MB	Type	Name	Size
67108864	65536	64	int	fluid_d	numNodes
67108864	65536	64	FLOAT_TYPE	coordX_d	numNodes
67108864	65536	64	FLOAT_TYPE	coordY_d	numNodes
67108864	65536	64	FLOAT_TYPE	coordZ_d	numNodes
7852032	7668	7,48828125	int	bcNodeIdX_d	numConns
7852032	7668	7,48828125	int	bcNodeIdY_d	numConns
7852032	7668	7,48828125	int	bcNodeIdZ_d	numConns
7852032	7668	7,48828125	int	latticeId_d	numConns
7852032	7668	7,48828125	int	bcType_d	numConns
7852032	7668	7,48828125	int	bcBoundId_d	numConns
7852032	7668	7,48828125	FLOAT_TYPE	bcX_d	numConns
7852032	7668	7,48828125	FLOAT_TYPE	bcY_d	numConns
7852032	7668	7,48828125	FLOAT_TYPE	bcZ_d	numConns
67108864	65536	64	FLOAT_TYPE	rho_d	numNodes
67108864	65536	64	FLOAT_TYPE	u1_d	numNodes
67108864	65536	64	FLOAT_TYPE	v1_d	numNodes
67108864	65536	64	FLOAT_TYPE	w1_d	numNodes
67108864	65536	64	FLOAT_TYPE	u_prev_d	numNodes
67108864	65536	64	FLOAT_TYPE	v_prev_d	numNodes
67108864	65536	64	FLOAT_TYPE	w_prev_d	numNodes
67108864	65536	64	FLOAT_TYPE	rho_prev_d	numNodes
1275068416	1245184	1216	FLOAT_TYPE	f_d	19*NumNodes
1275068416	1245184	1216	FLOAT_TYPE	fColl_d	19*NumNodes
67108864	65536	64	FLOAT_TYPE	tempA_d	NumNodes
67108864	65536	64	FLOAT_TYPE	tempB_d	NumNodes
67108864	65536	64	FLOAT_TYPE	u_d	NumNodes
67108864	65536	64	FLOAT_TYPE	v_d	NumNodes
67108864	65536	64	FLOAT_TYPE	w_d	NumNodes
134217728	131072	128	unsigned long long	bcMask_d	NumNodes
1560608	1524,03125	1,488311768	int	bcIdxCollapsed_d	bcCount
3121216	3048,0625	2,976623535	unsigned long long	bcMaskCollapsed_d	bcCount
67108864	65536	64	int	bcIdx_d	NumNodes
301989888	294912	288	bool	stream_d	18*NumNodes
TOTAL MEMORY					
		4071,859467	MB		

Table 4.6: Memory usage for Lid-driven cavity 256, single precision (The lowest memory consumption version of 3D LBM solver (4th option))

Figure 4.19 and Figure 4.20 compare memory usage by different options of the solver. We can notice that the difference between 0th and 4th option is quite big, memory usage (in MB) is almost three times smaller.

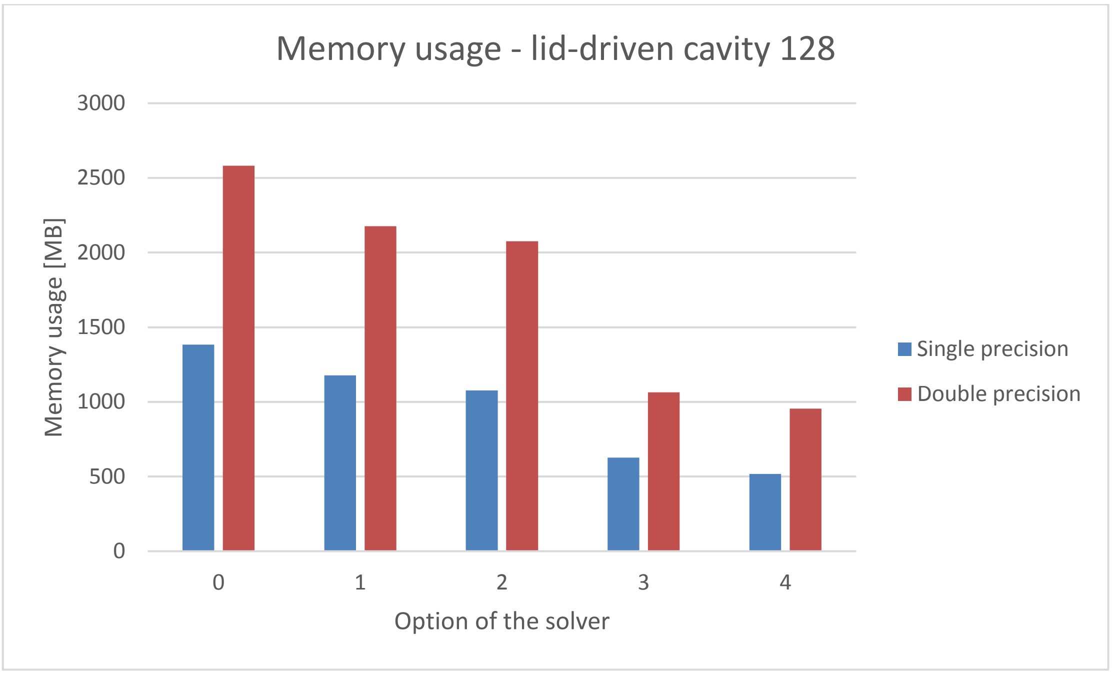


Figure 4.19: Memory usage - lid-driven cavity 128

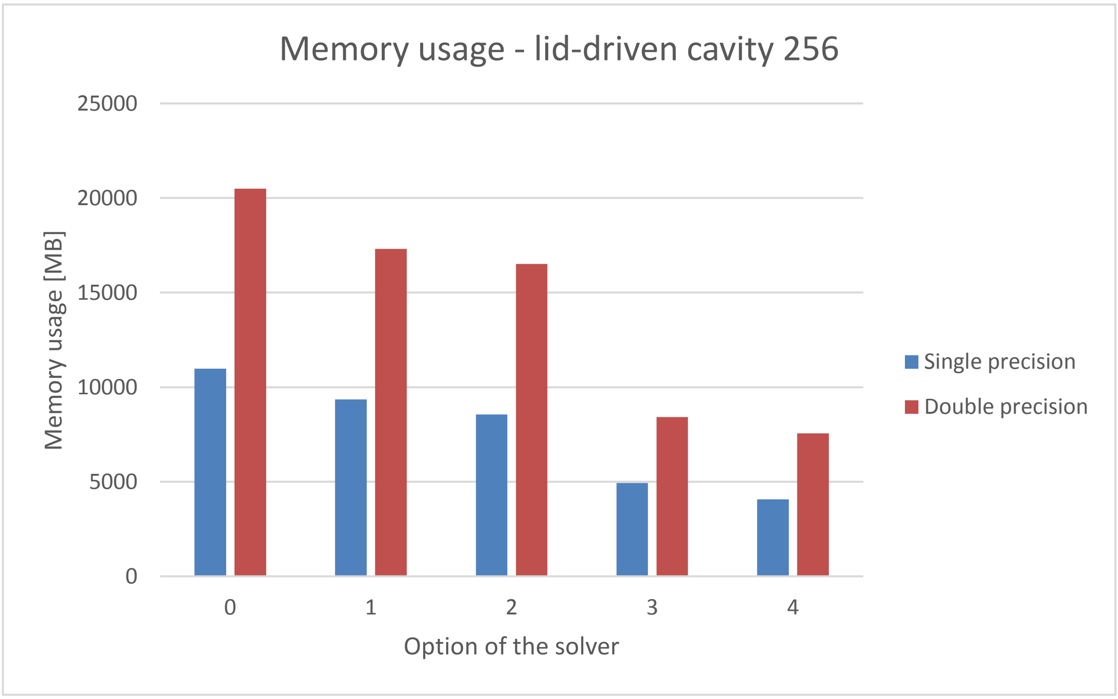
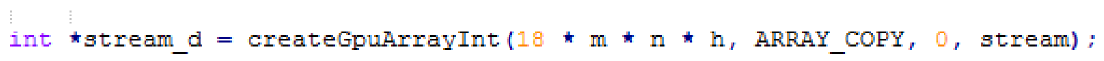


Figure 4.20: Memory usage - lid-driven cavity 256

4.7 Performance of 3D LBM solver

Main reason for developing the code using the GPU and CUDA was to provide good performance of the solver. Therefore, after developing the first, working and validated version of the solver, I should analyse and improve the performance of the 3D LBM CUDA solver.

My first idea to improve the performance of the code was changing an array designed for storing information about allowed directions for streaming – array stream_d:



```
int *stream_d = createGpuArrayInt(18 * m * n * h, ARRAY_COPY, 0, stream);
```

Figure 4.21: First version of stream_d array

As we can see above, datatype of this array is int. However, each element of this array can be only 0 (streaming permitted) or 1 (streaming allowed). Int datatype contains 64 bits (8 bytes), while we just need 1 bit per each element of this array, to store necessary information. At first glance, it was just a memory optimisation problem. However, this array is stored in global memory, stream_d is also accessed by every thread in each iteration. As we know, global memory is slow type of the memory, therefore, reading 64x unnecessarily bigger variables could noticeably slow down the solver. I have changed the datatype of that array to bool, then I have compared runtimes of the streaming substep - Figure 4.22, Figure 4.23.

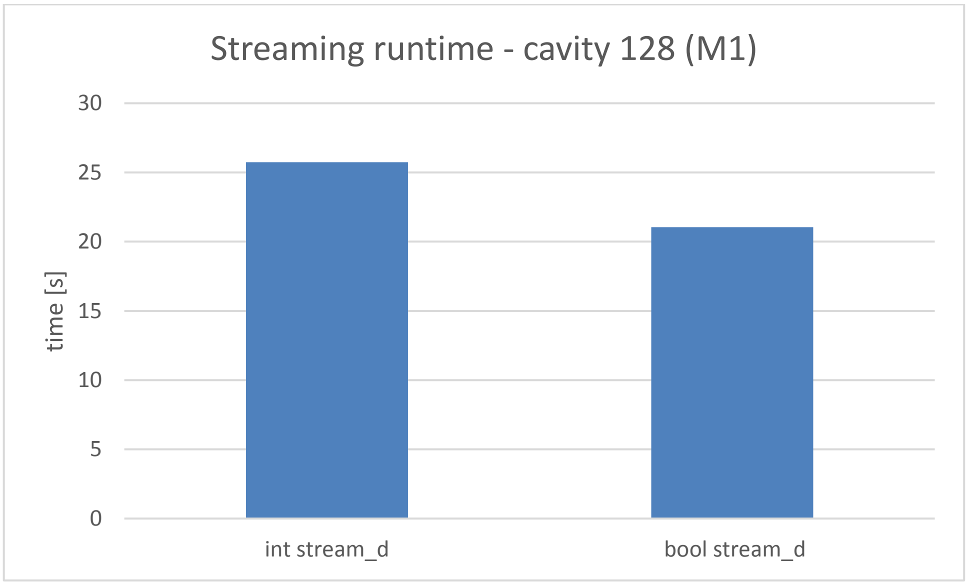


Figure 4.22: Streaming runtime - cavity 128

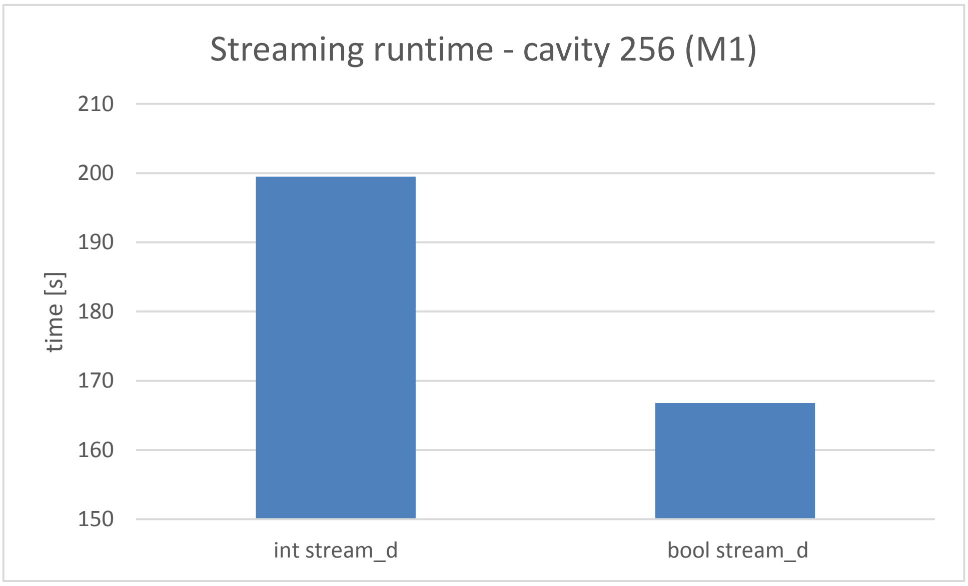


Figure 4.23: Streaming runtime - cavity 256

As we can see in the Figure 4.22 and Figure 4.23, speedup of this change is equal to 1,2, so it is quite noticeable improvement of the code. Besides performance issues, it also decreases memory usage, what lets us to simulate bigger lattices.

Second thought for performance improvement were residuals. Basically, this substep of the LBM solver is unnecessary, it is also required for monitoring the simulation while the solver is already running. Residuals let us determine whether the simulation is diverged or converged, during the simulation, without waiting until the specified number of iterations will be executed. It is really valuable feature of the solver, in real life it can save a lot of money and time, while we can terminate simulation, which will not produce any relevant result.

Figure 4.24 presents time consumption of every substep of the main loop of the solver. We can notice, that the residuals are the most time consuming part. This plot was prepared with the basic version of residuals – L2, directly adapted from 2D version of the code. They are based on distribution function and they are showing L2 norm between actual and previous step.

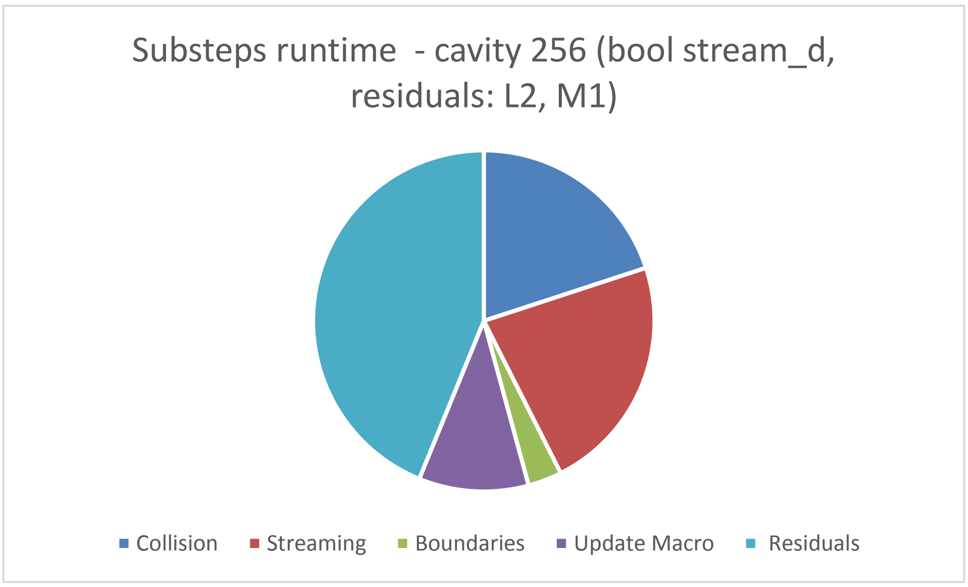


Figure 4.24: Substeps runtime - cavity 256 (residuals: L2)

L2 residuals were sufficient to determine simulation divergence, however they were difficult to analyse. Therefore, Aguilar [23] suggested to implement new type of residuals maximum relative difference of distribution function– eq. (4.4). This type of residuals was easier to analyse, however it was even more time consuming that the L2 residuals – Figure 4.25. This result was caused by more complex calculations and reduction operation – finding maximum value through all the directions of all nodes $(19 * m * n * h)$.

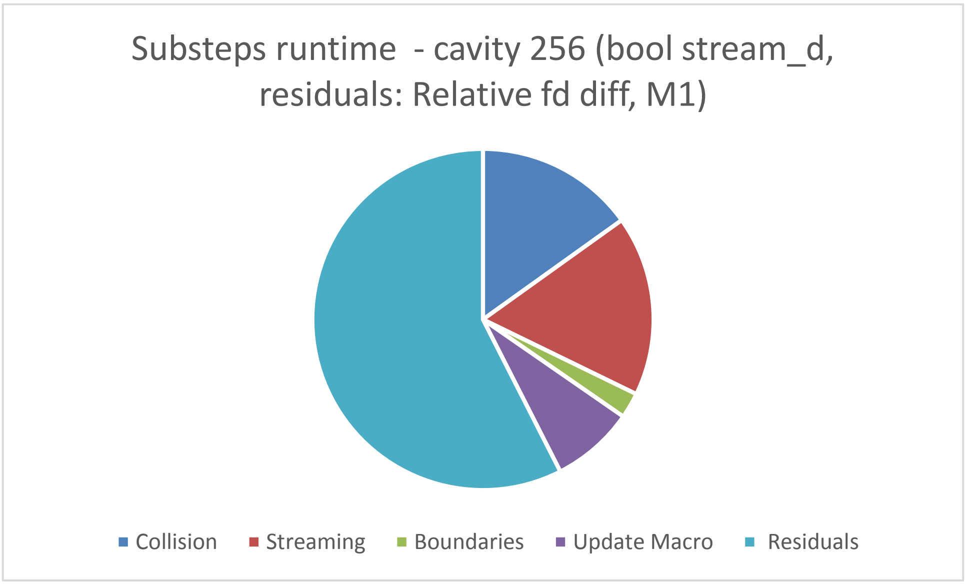


Figure 4.25: Substeps runtime - cavity 256 (residuals: Relative fd diff)

Those slow runtimes of the residuals forced us to implement one more type of residuals. The new residuals are based on macroscopic values – eq. (4.5). Each node contains information only about 4 macroscopic values – u, v, w, rho, therefore, it should be faster to calculate this type of residuals instead of those based on distribution function. Finally, Figure 4.26 presents that the macro diff residuals are much faster than the 2 previous types of the residuals. This performance improvement is caused mainly by smaller data necessary to process, to calculate these residuals.

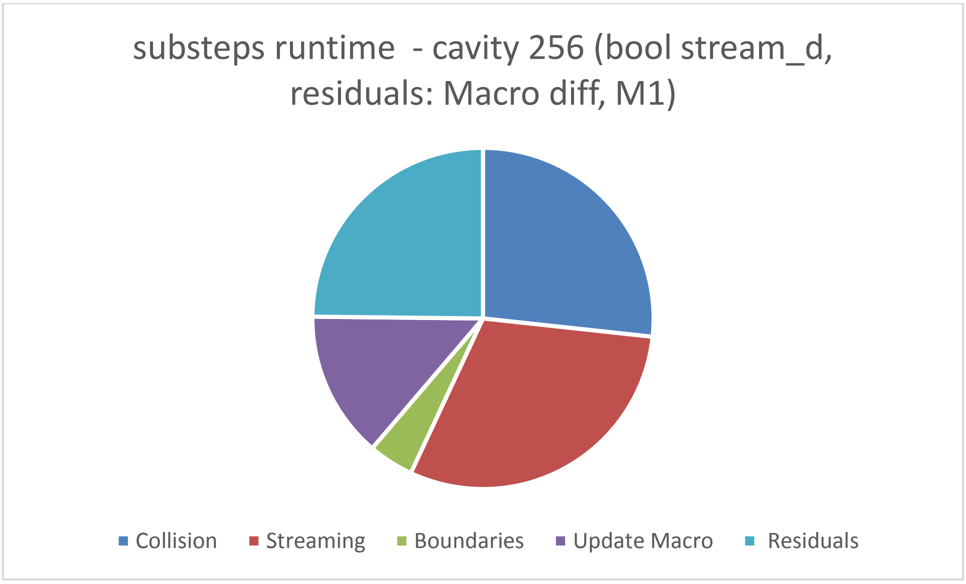


Figure 4.26: Substeps runtime - cavity 256 (residuals: Macro diff)

Figure 4.27 and Figure 4.28 show comparison of previously described three types of residuals. As we can see, for bigger problem cases (Figure 4.28), macro diff dominance over other types are much bigger than for the smaller cavity (Figure 4.27). For cavity 256, macro diff residuals are almost 3x faster than the second result L2. For smaller problem case (cavity 128), this dominance is not such impressive, but macro diffs are still the fastest type of residuals.

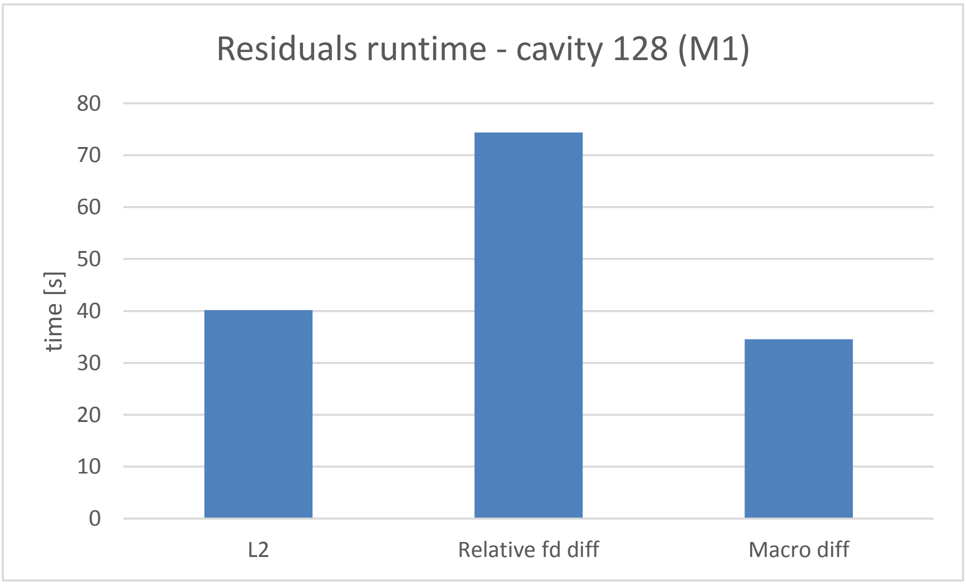


Figure 4.27: Residuals runtime - cavity 128

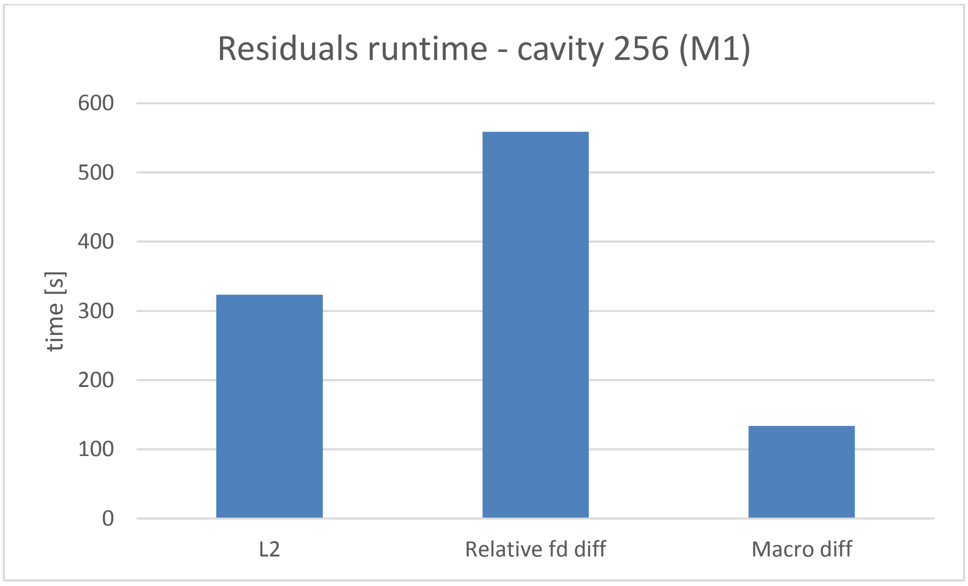


Figure 4.28: Residuals runtime - cavity 256

As I mentioned before, residuals are not necessary for LBM solver, to produce correct results, they are used only for monitoring the simulation, during its execution. Therefore, theoretically we do not need them if we are totally sure, that our simulation has to work properly, until the final iteration. I have prepared runtimes comparison of the simulations for 3 types of residuals, however those runtimes have subtracted runtimes of the residuals. Theoretically, those times should be perfectly the same, because counted time includes only exactly the same operations. The only one difference was residuals, but they were subtracted from the overall time. However, Figure 4.29 presents, that the runtimes are different, for different residuals. At first glance, it looked very strange, almost impossible, therefore I have tried to discover how it is possible. The difference between those simulations was different amount of allocated global memory, but it should not make any differences in runtimes. However, I finally found information [37], that different areas of GPU global memory can have different bandwidth. This information from NVIDIA engineer was about GTX 970, not Tesla, which are used in this comparison. However Tesla GPU has probably quite similar architecture, therefore the access to the global memory, allocated in different places, can be different.

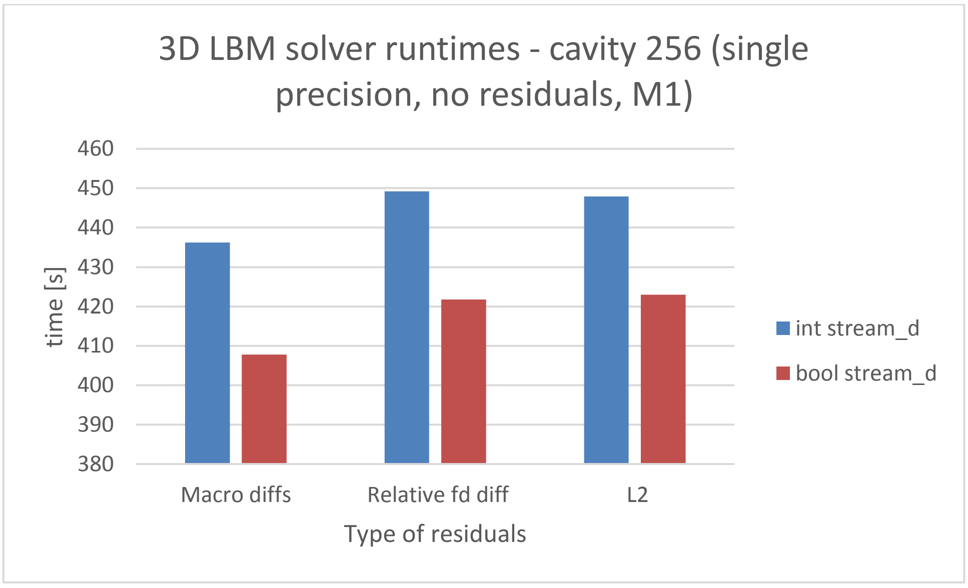


Figure 4.29: Solver runtimes - cavity 256

Until now, we have used only runtime to determine the performance of the solver. Using runtimes are only correct to compare performance of code, using exactly the same size of input data. When we want to compare performance of the solver for different lattice sizes, we should use another metric – million nodes updates per second (MLUPS). Therefore, this metric show us, how many nodes have been updated in one second, so results of this comparison are independent from the problem size. Figure 4.30 and Figure 4.31 present comparison of performance, measured in MLUPS, for cavity 128 and 256. We can notice, for all types of residuals, bigger problem case means better performance, what is quite normal for every parallel code, where the bigger problem means bigger advantage from the parallelisation.

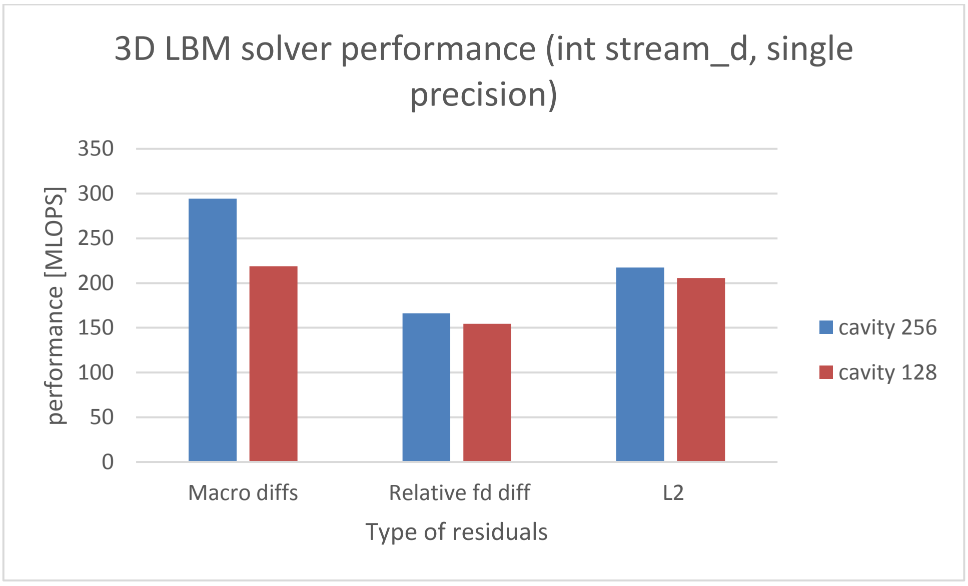


Figure 4.30: Solver performance - cavity 128/256 comparison (int stream_d)

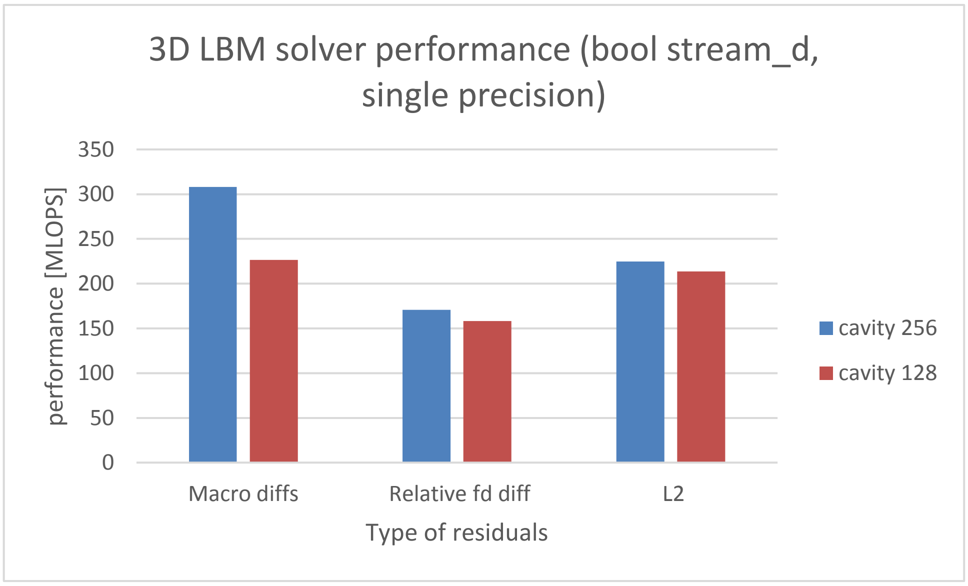


Figure 4.31: Solver performance - cavity 128/256 comparison (bool stream_d)

Figure 4.32 presents comparison of the solver run on two different machines – M1 and M2. As we can see, the difference is quite big, around 50%. Therefore, we can notice, how important is the hardware in the HPC, even for CUDA, where theoretically we have hundreds of thousands threads. However, each GPU has different clock of the SMs, etc. So the difference between various GPUs are quite noticeable, as we can see in the Figure 4.32.

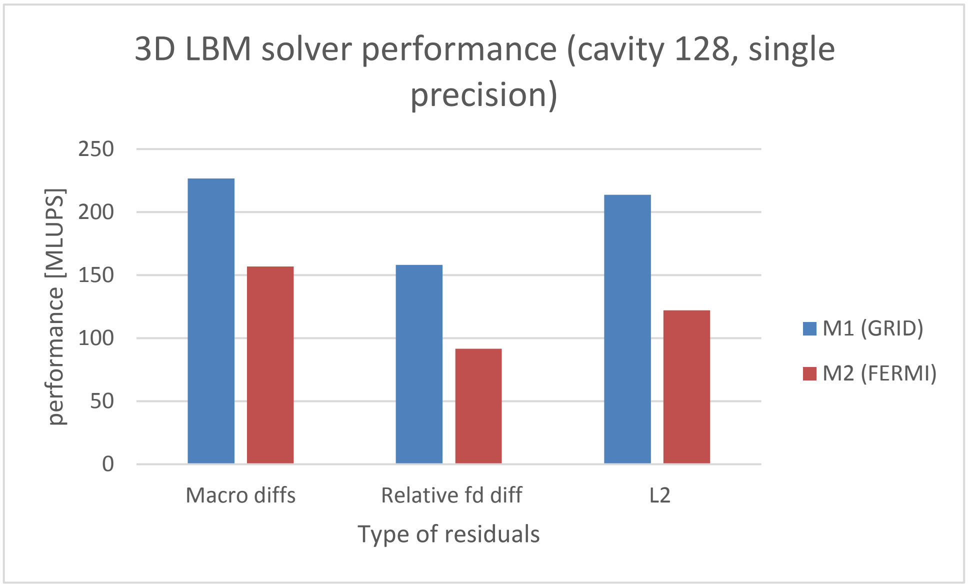


Figure 4.32: Solver performance - M1/M2 comparison

Someone could say, that the most important performance metrics for each parallel code is speedup. However, it is only useful for CPU parallel codes, when the parallelisation is done in classic way – parallel code is using a couple of threads/CPUs, but single threaded execution is still usable. However CUDA is really complicated and complex parallel platform, we are using hundreds of thousands threads, which are allocated automatically on SMs, we can specify only the maximum number of threads, not the number of concurrent threads. Second issue is the performance of single GPU thread, which is much slower than CPU thread – the same code executed on single CPU thread and single GPU thread would be thousands times faster on the CPU. Therefore, it is meaningless to show any speedup for the CUDA programs, while the single thread GPU is totally unusable for any calculations. We could for example calculate the speedup using single threaded CPU code and CUDA code.

However it is also meaningless to compare codes run on totally different devices, it would be similar to calculate speedup of calculations on the paper and on the computer – two totally different things. Therefore, it is almost impossible to prepare reasonable analysis of the performance, we could compare for example another CUDA LBM solvers, but each of those solvers were executed on different hardware, so this comparison would be also not really accurate.

5 Conclusion and outlook

The main objective of this thesis was to develop working 3D version of lattice generator and then, 3D LBM CUDA solver. Those programs have been developed basing on existing, two dimensional versions [12, [13, [14].

Firstly, there has been written an extensive literature review, which was helpful to familiarize with many issues, necessary to further work on this project. The literature review describes generally all necessary aspects of the LBM solver, described previously by others. There have been also generally discussed the HPC subject, history and evolution of the HPC and finally GPU shares in the HPC.

Next part of this thesis – methodology, explains the methods which will be used to fulfil the main objectives – developing the 3D versions of lattice generator and LBM solver. In this chapter, there has been described in details both the physics of the solver and the software side – CUDA platform, which will be used to develop the three dimensional LBM solver.

Finally, section 4 – Results and discussion, describes all the work performed in order to fulfil the objectives of this thesis. Firstly, there have been prepared basic, but working and validated version of both programs – lattice generator and LBM CUDA solver. Then, both the codes have been optimised, in order to simulate bigger problem cases, what was necessary to simulate high Reynold number flows. This project has been developed in cooperation with Aguilar [23], who was responsible for the physics of the LBM solver. He also performed successful verification and validation of the code, what was necessary to make sure, that those programs are out of any bugs.

Even though the developed software is working correctly for the 3D problem cases, what have been validated, there is still a lot of work to do, in order to improve this project. Firstly, at this moment, we can handle only simple geometries – BC faces have to be parallel to one of the three main planes of the 3D space (XY, XZ, YZ). This limitation is caused by the lattice generator, however the code is prepared for further development - implementation of the sloping BC faces does not require any restructuring of the code. Second task worth to do in the future is rebuilding the code of the LBM solver. Even though the code is fully working, it requires a cleaning up. Most time of working on this thesis was spent on debugging the code, then when I finally had the working 3D code, I spent my time on memory and performance optimisation. Therefore, the code could be much more readable and the structure of the code should also be easier for maintenance. However this project was quite challenging, therefore there is still a lot work, which can be perform by another students in the future.

REFERENCES

[1] Joel Ducoste. An Overview of computational Fluid Dynamics. Ghent University, 2008.

[2] Zhang Q., Liu Y., Chen T. Simulations of methane partial oxidation by CFD coupled with detailed chemistry at industry operating conditions. Article in Chemical engineering science, 13.03.2016.

[3] Pate L. CFD: a tool to design jet engine internal cooling system. Washington, D.C.: AIAA,1998.

[4] Rubio-Jimanez, C.A., Hernandez-Guerrero A., Lorenzini-Gutierrez D. CFD study of constructal microchannel networks for liquid-cooling of electronic devices. Elsevier Ltd, 2016.

[5] Bartesaghi S., Colombo G. Embedded CFD Simulation for Blood Flow. Article in Computer-Aided Design & Applications, 2013, Vol. 10 Issue 4, p685-699.

[6] A.A. Mohamad. Lattice Boltzmann method. Springer, 2011.

[7] Carlos P. Sosa. HPC: Past, Present and Future. University of Minnesota, 2011.

[8] Prometeus GmbH. URL http://www.top500.org. Accessed 05.04.2016

[9] Cray. URL: www.cray.com. Accessed: 25.07.2015

[10] Irene Moulitsas. “High Performance Computing” lecture slides, Cranfield University 2016.

[11] Salvatore Filippone. “Small scale parallel programming” Lecture slides, Cranfield University, 2016.

[12] Tamás István Józsa. Parallelization of lattice Boltzmann method using CUDA platform. Master’s thesis, Cranfield University, 2014.

[13] Ádám Koleszár. Optimisation of 2D lattice Boltzmann method using CUDA. Master’s thesis, Cranfield University 2015.

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

[15] Tom-Robin Teschner. Development of a two dimensional fluid solver based on the lattice Boltzmann method. Master's thesis, Cranfield University, 2013.

[16] Máté Tibor Szőke. Efficient implementation of a 2D lattice Boltzmann solver using modern parallelisation techniques. Master's thesis, Cranfield university 2014.

[17] Almasi G., Hargrove P., Tanase G., Zheng Y., UPC Collectives Library 2.0. IBM T.J. Watson Research Center, 2011.

[18] Alistair Revell. GPU Implementation of Lattice Boltzmann Method with Immersed Boundary: observations and results.The Oxford e-Research Centre Many-Core Seminar Series. 5th June 2013.

[19] NVIDIA. URL http://www.nvidia.co.uk/object/cuda-parallel-computing-uk.html Accessed 05.05.2016

[20] NVIDIA. URL https://blogs.nvidia.com/blog/2012/07/02/new-top500-list-4x-more-gpu-supercomputers/ Accessed 25.07.2016

[21] Michael Galloy's webpage. URL http://michaelgalloy.com/2013/06/11/ cpu-vs-gpu-performance.html. Accessed 20.07.2016.

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

[23] Alfonso Aguilar. Lattice Boltzmann solver development for 3d flows. Master's thesis, Cranfield University, 2016.

[24] Chen S. and Doolen G. Lattice Boltzmann method for Fluid Flows. IBM Research Division, 1998.

[25] Hecht and Harting. Implementation of on-site velocity boundary conditions for D3Q19 lattice Boltzmann simulations. An IOP and SISSA journal

[26] Bao Y. B. and Meskas J., Lattice Boltzmann Method for Fluid Simulations. April 2011

[27] NVIDIA. URL http://docs.nvidia.com/. Accessed 05.08.2016

[28] NVIDIA. URL https://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/. Accessed 06.08.2016

[29] The University of Texas at Austin. URL http://www.cs.utexas.edu/~fussell/courses/cs384g/projects/final/artifacts_f08/mjeong/cuda-raytracer_files/gridofthreadblocks.JPG. Accessed 06.08.2016

[30] Calvin College. URL https://cs.calvin.edu/courses/cs/374/CUDA/CUDA-Thread-Indexing-Cheatsheet.pdf. Accessed 06.08.2016

[31] Luitjens J. and Rennich S. CUDA Warps and Occupancy. GPU Computing Webinar, 7.12.2011.

[32] University of Oxford. URL https://people.maths.ox.ac.uk/gilesm/cuda/lecs/lec3-2x2.pdf. Accessed 07.08.2016. Accessed 07.08.2016

[33] Wikipedia. URL http://en.wikipedia.org. Accessed 09.08.2016

[34] STAR-CD version 3.26 user guide. CD-adapco, 2005.

[35] H.-C Ku, R.-S Hirsh, and T.-D Taylor. A pseudospectral method for solution of the three-dimensional incompressible Navier-Stokes equations. Journal of Computational Physics, 70(2):439–462, 1987.

[36] 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.

[37] PC perspective. URL http://www.pcper.com/reviews/Graphics-Cards/NVIDIA-Discloses-Full-Memory-Structure-and-Limitations-GTX-970. Accessed 10.08.2016

[38] Caltech Center for Advanced Computing Research. URL http://www.cacr.caltech.edu/~slombey/asci/vtk/vtk_formats.simple.html. Accessed 12.08.2016

APPENDICES

Appendix A Sample CUDA codes

A.1 CUDA kernel 1

```
__global__ void gpu_max256(FLOAT_TYPE *A, FLOAT_TYPE *B, int size) {
	int tid = threadIdx.x;
	int bid = blockIdx.y * gridDim.x + blockIdx.x;
	int gid = bid * blockDim.x * 2 + tid;

	extern __shared__ float temp[];
	temp[tid] = (gid < size) ? A[gid] : 0.0;

	temp[tid] += (gid + blockDim.x < size) ? A[gid + blockDim.x] : 0.0;
	__syncthreads();
	if (tid < 128)
		temp[tid] =
				(temp[tid] >= temp[tid + 128]) ? temp[tid] : temp[tid +
128];
	__syncthreads();
	if (tid < 64)
		temp[tid] = (temp[tid] >= temp[tid + 64]) ? temp[tid] : temp[tid + 64];
	__syncthreads();

	if (tid < 32) {
		temp[tid] = (temp[tid] >= temp[tid + 32]) ? temp[tid] : temp[tid + 32];
		__syncthreads();
		temp[tid] = (temp[tid] >= temp[tid + 16]) ? temp[tid] : temp[tid + 16];
		__syncthreads();
		temp[tid] = (temp[tid] >= temp[tid + 8]) ? temp[tid] : temp[tid + 8];
		__syncthreads();
		temp[tid] = (temp[tid] >= temp[tid + 4]) ? temp[tid] : temp[tid + 4];
		__syncthreads();
		temp[tid] = (temp[tid] >= temp[tid + 2]) ? temp[tid] : temp[tid + 2];
		__syncthreads();
		temp[tid] = (temp[tid] >= temp[tid + 1]) ? temp[tid] : temp[tid + 1];
	}

	if (tid == 0) {
		B[bid] = temp[0];
	}
```

A.2 CUDA kernel 2

```
__global__ void gpu_abs_sub(FLOAT_TYPE *A, FLOAT_TYPE *B, FLOAT_TYPE *C,
                int size, bool *divergence) {

        int blockId = blockIdx.x + blockIdx.y * gridDim.x;
        int ind = blockId * (blockDim.x * blockDim.y) + (threadIdx.y * blockDim.x)
                        + threadIdx.x;
        if (ind < size) {
                if(A[ind]!=A[ind]||B[ind]!=B[ind]) {
                        *divergence=true;
                }
                C[ind] = abs(A[ind] - B[ind]);

        }

}
```

A.3 CUDA program (Iterate3D.cu)

```
#include <stdio.h>                      // printf();
#include <math.h>                       // need to compile with -lm
#include <stdlib.h>                     // for calloc();
#include <stdbool.h>                    // Include for bool type variables!
#include <string.h>                     // String operations
#include <time.h>                       // time functions
#include <errno.h>
#include "GpuFunctions.h"       // GPU kernels
#include "ShellFunctions.h"     // For convenience
#include "FilesReading.h"       // For reading files
#include "FilesWriting.h"       // For writing files e.g. tecplot
#include "CellFunctions.h"      // For cell modifications
#include "ComputeResiduals.h"   // residuals
#include "LogWriter.h"
#include "Iterate.h"
#include "ArrayUtils.h"
#include "Check.h"
#include "cuda.h"
#include "GpuSum.h"

int Iterate3D(InputFilenames *inFn, Arguments *args) {
        // Time measurement: declaration, begin
        clock_t tStart = clock();

        FILE* logFile;               // file for log
        char autosaveFilename[768];  // autosave filename
        char outputFilename[768];    // initial data will be written to this file
        char finalFilename[768];     // final data will be written to this file
        char logFilename[768];       // path of the .log file
        char residualsFilename[768]; // path of the residuals file
        char timeFilename[768];      // path of time measurement file
        bool firstIter = true;
        bool *d_divergence;
        int AuxMacroDiff = 1;
        FLOAT_TYPE r = -1.0;
        logFilename[0] = '\0';
        residualsFilename[0] = '\0';
        timeFilename[0] = '\0';

        if (strlen(inFn->result)) {
                strcat(logFilename, inFn->result);
                strcat(residualsFilename, inFn->result);
                strcat(timeFilename, inFn->result);
        }
        strcat(logFilename, "lbmsolver.log");
        strcat(residualsFilename, "residuals.dat");
        strcat(timeFilename, "runtimes.dat");

        int autosaveIt = 1; // autosave i variable, will be incremented after every autosave
        int numNodes, numConns; // This will store the number of lines of the read files
        FLOAT_TYPE delta;          // grid spacing
        int n, m, h;                 // number of nodes in the x, y and z directions
        FLOAT_TYPE maxInletCoordY; // maximum inlet coordinate in y
        FLOAT_TYPE minInletCoordY; // minimum inlet coordinate in y
        FLOAT_TYPE maxInletCoordZ; // maximum inlet coordinate in z
        FLOAT_TYPE minInletCoordZ; // minimum inlet coordinate in z
        int numInletNodes;         // number of inlet nodes
        FLOAT_TYPE uMaxDiff = -1, vMaxDiff = -1, wMaxDiff = -1, rhoMaxDiff = -1;
        int *nodeIdX, *nodeIdY, *nodeIdZ, *nodeType, *bcNodeIdX, *bcNodeIdY,
                *bcNodeIdZ, *latticeId, *bcType, *bcBoundId;
        FLOAT_TYPE *nodeX, *nodeY, *nodeZ, *bcX, *bcY, *bcZ;

        FLOAT_TYPE taskTime[9];
        int i;
        for (i = 0; i < 9; ++i) {
                taskTime[i] = 0.0;
        }

        clock_t tInstant1, tInstant2; // Time measurement points, universal
        clock_t tIterStart, tIterEnd; // Time measurement points: main loop

        // cuda time measurement variables
        cudaEvent_t start, stop;
        float cudatime;
        CHECK(cudaEventCreate(&start));
        CHECK(cudaEventCreate(&stop));

        numNodes = readNodeFile(inFn->node, &nodeIdX, &nodeIdY, &nodeIdZ, &nodeX,
                &nodeY, &nodeZ, &nodeType, args->TypeOfProblem);
        if (numNodes == 0) {
                printf("NODES NOT FOUND in file\n");
                return 2;
        }

        int *fluid_d = createGpuArrayInt(numNodes, ARRAY_COPY, 0, nodeType);
        FLOAT_TYPE *coordX_d = createGpuArrayFlt(numNodes, ARRAY_COPY, 0., nodeX);
        FLOAT_TYPE *coordY_d = createGpuArrayFlt(numNodes, ARRAY_COPY, 0., nodeY);
        FLOAT_TYPE *coordZ_d = createGpuArrayFlt(numNodes, ARRAY_COPY, 0., nodeZ);

        numConns = readConnFile(inFn->bc, &bcNodeIdX, &bcNodeIdY, &bcNodeIdZ,
                &latticeId, &bcType, &bcX, &bcY, &bcZ, &bcBoundId,
                args->TypeOfProblem);
        if (numConns == 0) {
                printf("NEIGHBOURING NOT FOUND in file\n");
                return 2;
        }

        int *bcNodeIdX_d = createGpuArrayInt(numConns, ARRAY_COPY, 0, bcNodeIdX);
        int *bcNodeIdY_d = createGpuArrayInt(numConns, ARRAY_COPY, 0, bcNodeIdY);
        int *bcNodeIdZ_d = createGpuArrayInt(numConns, ARRAY_COPY, 0, bcNodeIdZ);
        int *latticeId_d = createGpuArrayInt(numConns, ARRAY_COPY, 0, latticeId);
        int *bcType_d = createGpuArrayInt(numConns, ARRAY_COPY, 0, bcType);
        int *bcBoundId_d = createGpuArrayInt(numConns, ARRAY_COPY, 0, bcBoundId);
        FLOAT_TYPE *bcX_d = createGpuArrayFlt(numConns, ARRAY_COPY, 0., bcX);
        FLOAT_TYPE *bcY_d = createGpuArrayFlt(numConns, ARRAY_COPY, 0., bcY);
        FLOAT_TYPE *bcZ_d = createGpuArrayFlt(numConns, ARRAY_COPY, 0., bcZ);

        m = getLastValue(nodeIdY, numNodes);
        n = getLastValue(nodeIdX, numNodes);
        h = getLastValue(nodeIdZ, numNodes);
        delta = getGridSpacing(nodeIdX, nodeIdY, nodeX, numNodes);

        numInletNodes = getNumInletNodes(bcType, latticeId, numConns,
                args->TypeOfProblem);
        maxInletCoordY = getMaxInletCoordY(bcType, latticeId, bcY, delta, numConns,
                args->TypeOfProblem);
        minInletCoordY = getMinInletCoordY(bcType, latticeId, bcY, delta, numConns,
                args->TypeOfProblem);
        maxInletCoordZ = getMaxInletCoordZ(bcType, latticeId, bcZ, delta, numConns,
                args->TypeOfProblem);
        minInletCoordZ = getMinInletCoordZ(bcType, latticeId, bcZ, delta, numConns,
                args->TypeOfProblem);

        writeInitLog(logFilename, args, delta, m, n, h, numInletNodes,
                maxInletCoordY, minInletCoordY, maxInletCoordZ, minInletCoordZ);
        logFile = fopen(logFilename, "a");
        // In case of no autosave
        sprintf(autosaveFilename, "NOWHERE!");

        initConstants3D(args, maxInletCoordY, minInletCoordY, maxInletCoordZ,
                minInletCoordZ, delta, m, n, h);

        dim3 tpb(THREADS, THREADS);                                         // THREADS/block
        dim3 bpg1((int) (sqrt(m * n * h) / THREADS) + 1,
                (int) (sqrt(m * n * h) / THREADS) + 1);       // blocks/grid   MxNxH
        dim3 bpg18((int) (sqrt(18 * m * n * h) / THREADS) + 1,
                (int) (sqrt(18 * m * n * h) / THREADS) + 1);  // blocks/grid 18MxNxH
        dim3 bpg19((int) (sqrt(19 * m * n * h) / THREADS) + 1,
                (int) (sqrt(19 * m * n * h) / THREADS) + 1);  // blocks/grid 19MxNxH
        dim3 bpgBC((int) (sqrt(numConns) / THREADS) + 1,
                (int) (sqrt(numConns) / THREADS) + 1);    // blocks/grid N_BC

        // residuals
        FLOAT_TYPE *norm = createHostArrayFlt(args->iterations, ARRAY_ZERO);
        FLOAT_TYPE *dragSum = createHostArrayFlt(args->iterations, ARRAY_ZERO);
        FLOAT_TYPE *liftSum = createHostArrayFlt(args->iterations, ARRAY_ZERO);
        FLOAT_TYPE *latFSum = createHostArrayFlt(args->iterations, ARRAY_ZERO);

        fprintf(logFile, "\n:::: Initializing ::::\n");
        printf("\n:::: Initializing ::::\n");
        CHECK(cudaEventRecord(start, 0));

        FLOAT_TYPE *u, *v, *w, *rho;

        int InitialCondLoadingErr = -1;
        if (args->UseInitialCondFromFile) {
                InitialCondLoadingErr = readInitConditionsFile(inFn->InitialConditions,
                        numNodes, n, m, h, &u, &v, &w, &rho);
        } else {
                u = createHostArrayFlt(m * n * h, ARRAY_ZERO);
                v = createHostArrayFlt(m * n * h, ARRAY_ZERO);
                w = createHostArrayFlt(m * n * h, ARRAY_ZERO);
                rho = createHostArrayFlt(m * n * h, ARRAY_ZERO);
        }

        FLOAT_TYPE *rho_d;
        if (InitialCondLoadingErr)
                rho_d = createGpuArrayFlt(m * n * h, ARRAY_FILL, args->rho);
        else
                rho_d = createGpuArrayFlt(m * n * h, ARRAY_COPY, 0, rho);
        FLOAT_TYPE *u1_d, *v1_d, *w1_d;
        if (args->inletProfile == NO_INLET) {
                if (InitialCondLoadingErr) {
                        u1_d = createGpuArrayFlt(m * n * h, ARRAY_FILL, args->u);
                        v1_d = createGpuArrayFlt(m * n * h, ARRAY_FILL, args->v);
                        w1_d = createGpuArrayFlt(m * n * h, ARRAY_FILL, args->w);
                } else {
                        u1_d = createGpuArrayFlt(m * n * h, ARRAY_COPY, 0, u);
                        v1_d = createGpuArrayFlt(m * n * h, ARRAY_COPY, 0, v);
                        w1_d = createGpuArrayFlt(m * n * h, ARRAY_COPY, 0, w);
                        printf("Initial conditions loaded from file\n");
                }
        }
        if (args->inletProfile == INLET) {
```

```
                printf(
                        "Inlet profile is not currently available! Please initiate Inlet profile
from file!\n");
                return 0;

        }
        FLOAT_TYPE *u_prev_d, *v_prev_d, *w_prev_d, *rho_prev_d;
        if (args->TypeOfResiduals == MacroDiff) {
                u_prev_d = createGpuArrayFlt(m * n * h, ARRAY_ZERO);
                v_prev_d = createGpuArrayFlt(m * n * h, ARRAY_ZERO);
                w_prev_d = createGpuArrayFlt(m * n * h, ARRAY_ZERO);
                rho_prev_d = createGpuArrayFlt(m * n * h, ARRAY_ZERO);
        }


        FLOAT_TYPE *f_d = createGpuArrayFlt(19 * m * n * h, ARRAY_ZERO);
        FLOAT_TYPE *fColl_d = createGpuArrayFlt(19 * m * n * h, ARRAY_ZERO);
        FLOAT_TYPE *f1_d, *fprev_d;
        if (args->TypeOfResiduals == FdRelDiff) {
                fprev_d = createGpuArrayFlt(19 * m * n * h, ARRAY_ZERO);
        }
        FLOAT_TYPE *temp19a_d, *temp19b_d;
        if (args->TypeOfResiduals != MacroDiff) {
                temp19a_d = createGpuArrayFlt(19 * m * n * h, ARRAY_ZERO);
                temp19b_d = createGpuArrayFlt(19 * m * n * h, ARRAY_ZERO);
        }
        FLOAT_TYPE *tempA_d = createGpuArrayFlt(m * n * h, ARRAY_ZERO);
        FLOAT_TYPE *tempB_d = createGpuArrayFlt(m * n * h, ARRAY_ZERO);

        int *mask = createHostArrayInt(m * n * h, ARRAY_ZERO);
        unsigned long long *bcMask = createHostArrayLongLong(m * n * h, ARRAY_ZERO);
        int *bcIdx = createHostArrayInt(m * n * h, ARRAY_ZERO);

        FLOAT_TYPE *u_d = createGpuArrayFlt(m * n * h, ARRAY_CPYD, 0, u1_d);
        FLOAT_TYPE *v_d = createGpuArrayFlt(m * n * h, ARRAY_CPYD, 0, v1_d);
        FLOAT_TYPE *w_d = createGpuArrayFlt(m * n * h, ARRAY_CPYD, 0, w1_d);

        bool *stream = createHostArrayBool(18 * m * n * h, ARRAY_FILL, 1);
        FLOAT_TYPE *q = createHostArrayFlt(18 * m * n * h, ARRAY_FILL, 0.5);

        int bcCount = initBoundaryConditions3D(bcNodeIdX, bcNodeIdY, bcNodeIdZ, q,
                bcBoundId, nodeType, bcX, bcY, bcZ, nodeX, nodeY, nodeZ, latticeId,
                stream, bcType, bcMask, bcIdx, mask, delta, m, n, h, numConns,
                args->boundaryType);
        unsigned long long *bcMask_d = createGpuArrayLongLong(m * n * h, ARRAY_COPY,
                0, bcMask);
        int *bcIdxCollapsed_d = createGpuArrayInt(bcCount, ARRAY_ZERO);
        unsigned long long *bcMaskCollapsed_d = createGpuArrayLongLong(bcCount,
                ARRAY_ZERO);

        FLOAT_TYPE *qCollapsed_d;
        if (args->boundaryType == CURVED)
                qCollapsed_d = createGpuArrayFlt(18 * bcCount, ARRAY_ZERO);

        dim3 bpgB((int) (sqrt(bcCount) / THREADS) + 1,
                (int) (sqrt(bcCount) / THREADS) + 1); // blocks/grid

        int *bcIdx_d = createGpuArrayInt(m * n * h, ARRAY_COPY, 0, bcIdx);

        collapseBc3D(bcIdx, bcIdxCollapsed_d, bcMask, bcMaskCollapsed_d, q,
                qCollapsed_d, mask, m, n, h, bcCount, args->boundaryType);

        bool *stream_d = createGpuArrayBool(18 * m * n * h, ARRAY_COPY, 0, stream);

        CHECK(cudaMemcpy(u, u_d, SIZEFLT(m*n*h), cudaMemcpyDeviceToHost));
        CHECK(cudaMemcpy(v, v_d, SIZEFLT(m*n*h), cudaMemcpyDeviceToHost));
        CHECK(cudaMemcpy(w, w_d, SIZEFLT(m*n*h), cudaMemcpyDeviceToHost));
        CHECK(cudaMemcpy(rho, rho_d, SIZEFLT(m*n*h), cudaMemcpyDeviceToHost));
        CHECK(cudaEventRecord(stop, 0));
        CHECK(cudaEventSynchronize(stop));
        CHECK(cudaEventElapsedTime(&cudatime, start, stop));
```

```
taskTime[T_INIT] += cudatime / 1000;

fclose(logFile);
writeNodeNumbers(logFilename, numNodes, numConns, bcCount);
logFile = fopen(logFilename, "a");

void *hostArrays[] = { nodeIdX, nodeIdY, nodeIdZ, nodeX, nodeY, nodeZ,
        nodeType, bcNodeIdX, bcNodeIdY, bcNodeIdZ, latticeId, bcType, bcX,
        bcY, bcZ, bcBoundId, u, v, w, rho, mask, bcMask, bcIdx, stream, q,
        norm, dragSum, liftSum, latFSum };
void *gpuArrays[] =
{ coordX_d, coordY_d, coordZ_d, fluid_d, bcNodeIdX_d, bcNodeIdY_d,
bcNodeIdZ_d, latticeId_d, bcType_d, bcX_d, bcY_d, bcZ_d,
bcBoundId_d, u_d, v_d, w_d, rho_d, u1_d, v1_d, w1_d, f_d,
fColl_d, temp19a_d, temp19b_d, tempA_d, tempB_d,
bcBoundId_d, bcMaskCollapsed_d, bcIdx_d, bcIdxCollapsed_d,
stream_d, qCollapsed_d }; //drag_d, lift_d, latF_d,

fprintf(logFile, "\n:::: Initialization done! ::::\n");

printf("Initialization took %f seconds\n", taskTime[T_INIT]);

// Write Initialized data
switch (args->outputFormat) {
case CSV:
        sprintf(outputFilename, "%sInitialData.csv", inFn->result);
        break;
case TECPLOT:
        sprintf(outputFilename, "%sInitialData.dat", inFn->result);
        break;
case PARAVIEW:
        sprintf(outputFilename, "%sInitialData.vti", inFn->result);
        break;
}
tInstant1 = clock(); // Start measuring time
WriteResults3D(outputFilename, nodeType, nodeX, nodeY, nodeZ, u, v, w, rho,
        nodeType, n, m, h, args->outputFormat);
tInstant2 = clock();
taskTime[T_WRIT] += (FLOAT_TYPE) (tInstant2 - tInstant1) / CLOCKS_PER_SEC;

printf("\nInitialized data was written to %s\n", outputFilename);

////////////////// ITERATION ///////////////////////

fprintf(logFile, "\n:::: Start Iterations ::::\n");
printf("\n:::: Start Iterations ::::\n");

printf("%d is the number of iterations \n", args->iterations);

tIterStart = clock(); // Start measuring time of main loop
size_t free, total;

cuMemGetInfo(&free, &total);
printf("^^^^ Free : %llu Mbytes \n",
        (unsigned long long) free / 1024 / 1024);

printf("^^^^ Total: %llu Mbytes \n",
        (unsigned long long) total / 1024 / 1024);

printf("^^^^ %f%% free, %f%% used\n", 100.0 * free / (double) total,
        100.0 * (total - free) / (double) total);
int iter = 0;
while (iter < args->iterations) {
        CHECK(cudaThreadSynchronize());
        CHECK(cudaEventRecord(start, 0)); // Start measuring time
        ////////////// COLLISION ///////////////
        switch (args->collisionModel) {
        case BGKW:
                gpuCollBgkw3D<<<bpg1, tpb>>>(fluid_d, rho_d, u_d, v_d, w_d, f_d,
                        fColl_d);
                break;
```

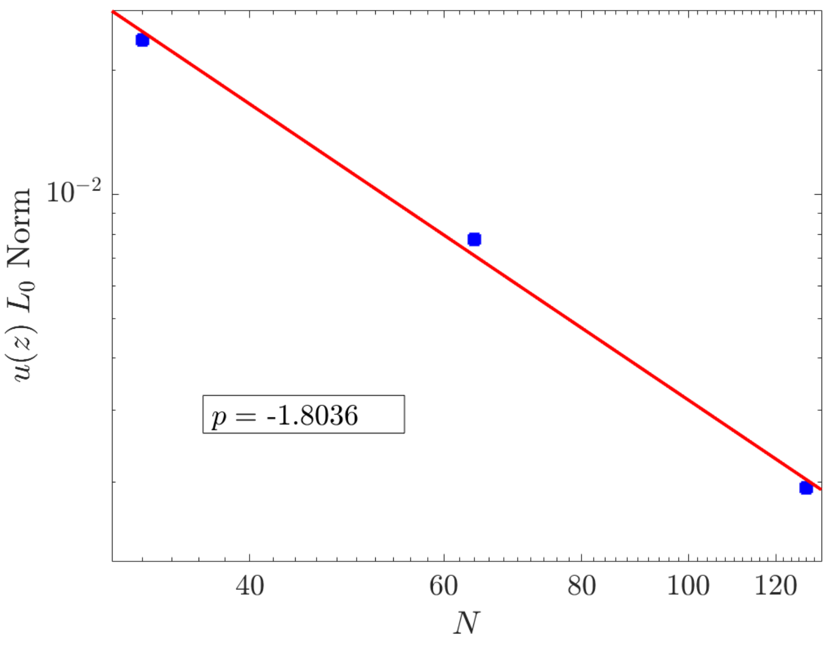
```
		case TRT:
			printf("TRT not implemented in 3D go for MRT \n");
			//	gpuCollTrt<<<bpg1,tpb>>>(fluid_d, rho_d, u_d, v_d, w_d, f_d,
fColl_d);
			break;

		case MRT:
			gpuCollMrt3D<<<bpg1, tpb>>>(fluid_d, rho_d, u_d, v_d, w_d, f_d,
				fColl_d);
			break;

		}

		CHECK(cudaEventRecord(stop, 0));
		CHECK(cudaEventSynchronize(stop));
		CHECK(cudaEventElapsedTime(&cudatime, start, stop));
		taskTime[T_COLL] += cudatime;

		////////////// STREAMING ///////////////
		CHECK(cudaThreadSynchronize());
		CHECK(cudaEventRecord(start, 0));

		gpuStreaming3D<<<bpg1, tpb>>>(fluid_d, stream_d, f_d, fColl_d);

		CHECK(cudaEventRecord(stop, 0));
		CHECK(cudaEventSynchronize(stop));
		CHECK(cudaEventElapsedTime(&cudatime, start, stop));
		taskTime[T_STRM] += cudatime;

		// make the host block until the device is finished with foo
		CHECK(cudaThreadSynchronize());

		// check for error
		cudaError_t error = cudaGetLastError();
		if (error != cudaSuccess) {
			// print the CUDA error message and exit
			printf("CUDA error: %s\n", cudaGetErrorString(error));
			exit(-1);
		}

		////////////// BOUNDARIES ///////////////
		CHECK(cudaThreadSynchronize());
		CHECK(cudaEventRecord(start, 0));

		gpuBcInlet3D<<<bpgB, tpb>>>(bcIdxCollapsed_d, bcMaskCollapsed_d, f_d,
			u1_d, v1_d, w1_d, bcCount);
		switch (args->bcwallmodel) {
		case SIMPLE:
			gpuBcSimpleWall3D<<<bpgB, tpb>>>(bcIdxCollapsed_d,
				bcMaskCollapsed_d, f_d, fColl_d, qCollapsed_d, bcCount);

			break;
		case COMPLEX:
			gpuBcComplexWall3D<<<bpgB, tpb>>>(bcIdxCollapsed_d,
				bcMaskCollapsed_d, f_d, fColl_d, qCollapsed_d, bcCount);

			break;
		}
		gpuBcOutlet3D<<<bpgB, tpb>>>(bcIdxCollapsed_d, bcMaskCollapsed_d, f_d, u_d, v_d,
w_d, rho_d, bcCount);
		gpuBcPeriodic3D<<<bpgB, tpb>>>(bcIdxCollapsed_d, bcMaskCollapsed_d,
f_d,bcCount);
		gpuBcSymm3D<<<bpgB, tpb>>>(bcIdxCollapsed_d, bcMaskCollapsed_d, f_d,bcCount);

		CHECK(cudaEventRecord(stop, 0));
		CHECK(cudaEventSynchronize(stop));
		CHECK(cudaEventElapsedTime(&cudatime, start, stop));
		taskTime[T_BNDC] += cudatime;

		// UPDATE VELOCITY AND DENSITY
```

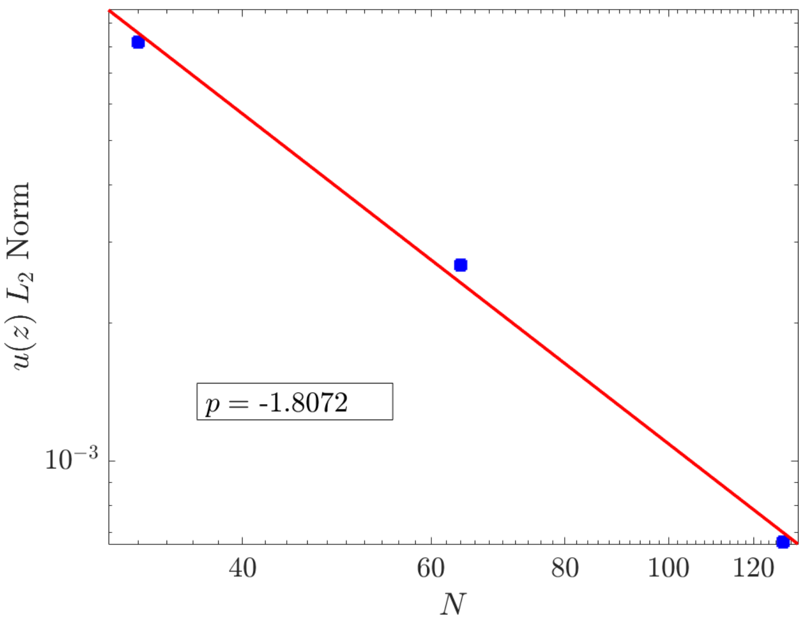
```
        CHECK(cudaThreadSynchronize());
        CHECK(cudaEventRecord(start, 0));

        gpuUpdateMacro3D<<<bpg1, tpb>>>(fluid_d, rho_d, u_d, v_d, w_d,
            bcBoundId_d, coordX_d, coordY_d, coordZ_d, f_d, args->g,bcMask_d,args-
>UpdateInltOutl);
        tInstant2 = clock();
        CHECK(cudaEventRecord(stop, 0));
        CHECK(cudaEventSynchronize(stop));
        CHECK(cudaEventElapsedTime(&cudatime, start, stop));
        taskTime[T_MACR] += cudatime;

        // COMPUTE RESIDUALS
        if (AuxMacroDiff * args->ShowMacroDiff == iter + 1) {
            CHECK(cudaThreadSynchronize());
            CHECK(cudaEventRecord(start, 0));

            if (args->TypeOfResiduals == L2) {
            r = computeResidual3D(f_d, fColl_d, temp19a_d, temp19b_d, m, n,h);
            } else {
                if (args->TypeOfResiduals == FdRelDiff) {
                    if (firstIter) {
                    firstIter = false;
                    f1_d = createGpuArrayFlt(19 * n * m * h, ARRAY_CPYD,
0,f_d);

                    }
                    r = computeNewResidual3D(f_d, fprev_d, f1_d, temp19a_d,
                        temp19b_d, m, n, h);
                    CHECK(cudaFree(fprev_d));
                    fprev_d = createGpuArrayFlt(19 * n * m * h, ARRAY_CPYD,
0,

                        f_d);

                } else {
                    bool h_divergence = false;
                    CHECK(cudaMalloc(&d_divergence,sizeof(bool)));

    CHECK(cudaMemcpy(d_divergence,&h_divergence,sizeof(bool),cudaMemcpyHostToDevice));
                    gpu_abs_sub<<<bpg1, tpb>>>(u_d, u_prev_d, tempA_d,n * m
* h, d_divergence);
                    uMaxDiff = gpu_max_h(tempA_d, tempB_d, n * m * h);
                    gpu_abs_sub<<<bpg1, tpb>>>(v_d, v_prev_d, tempA_d,n * m
* h, d_divergence);
                    vMaxDiff = gpu_max_h(tempA_d, tempB_d, n * m * h);
                    gpu_abs_sub<<<bpg1, tpb>>>(w_d, w_prev_d, tempA_d,n * m
* h, d_divergence);
                    wMaxDiff = gpu_max_h(tempA_d, tempB_d, n * m * h);
                    gpu_abs_sub<<<bpg1, tpb>>>(rho_d, rho_prev_d, tempA_d,n
* m * h, d_divergence);

    CHECK(cudaMemcpy(&h_divergence,d_divergence,sizeof(bool),cudaMemcpyDeviceToHost));
                    CHECK(cudaFree(d_divergence));
                    if (h_divergence) {
                        fprintf(stderr, "\nDIVERGENCE!\n");
                        break;
                    }
                    rhoMaxDiff = gpu_max_h(tempA_d, tempB_d, n * m * h);
                    if (abs(uMaxDiff) < args->StopCondition[0]
                    && abs(vMaxDiff) < args->StopCondition[1]
                    && abs(wMaxDiff) < args->StopCondition[2]
                    && abs(rhoMaxDiff) < args->StopCondition[3]) {
                        printf("simulation converged!\n");
                        break;
                    }
                    writeMacroDiffs(iter + 1, uMaxDiff, vMaxDiff, wMaxDiff,
                        rhoMaxDiff);
                    CHECK(cudaFree(u_prev_d));
                    CHECK(cudaFree(v_prev_d));
                    CHECK(cudaFree(w_prev_d));
                    CHECK(cudaFree(rho_prev_d));
                u_prev_d = createGpuArrayFlt(n * m * h, ARRAY_CPYD, 0, u_d);
```

```
                                        v_prev_d = createGpuArrayFlt(n * m * h, ARRAY_CPYD, 0, v_d);
                                        w_prev_d = createGpuArrayFlt(n * m * h, ARRAY_CPYD, 0, w_d);
                                        rho_prev_d = createGpuArrayFlt(n * m * h, ARRAY_CPYD, 0,rho_d);
                                        }
                                }

                                if (abs(r) < args->StopCondition[0]) {
                                        printf("simulation converged!\n");
                                        break;
                                }
                                if (r != r) {
                                        fprintf(stderr, "\nDIVERGENCE!\n");
                                        break;
                                }

                                CHECK(cudaEventRecord(stop, 0));
                                CHECK(cudaEventSynchronize(stop));
                                CHECK(cudaEventElapsedTime(&cudatime, start, stop));
                                taskTime[T_RESI] += cudatime;

                                AuxMacroDiff++;

                        }
                        norm[iter] = r;
                        if (args->TypeOfResiduals == MacroDiff) {
                                printf(
                                        "Iterating... %d/%d (%3.1f %%) Max macro diffs: u= %.10f v=
%.10f w= %.10f rho= %.10f \r",
                                        iter + 1, args->iterations,(FLOAT_TYPE) (iter + 1) * 100
                                        / (FLOAT_TYPE) (args->iterations), uMaxDiff,
                                        vMaxDiff, wMaxDiff, rhoMaxDiff);
                        } else {
                                printf("Iterating... %d/%d (%3.1f %%)  residual="FLOAT_FORMAT" \r",
                                        iter + 1, args->iterations,(FLOAT_TYPE) (iter + 1) * 100
                                        / (FLOAT_TYPE) (args->iterations), r);
                        }

                        iter++; // update loop variable

                        ////////////// Autosave ///////////////

                        if (iter == (args->autosaveEvery * autosaveIt)) {
                                autosaveIt++;
                                if (iter > args->autosaveAfter) {
                                        printf("autosave\n\n");
                                        //////////// COPY VARIABLES TO HOST ////////////////
                                CHECK(cudaMemcpy(u, u_d, SIZEFLT(m*n*h), cudaMemcpyDeviceToHost));
                                CHECK(cudaMemcpy(v, v_d, SIZEFLT(m*n*h), cudaMemcpyDeviceToHost));
                                CHECK(cudaMemcpy(w, w_d, SIZEFLT(m*n*h), cudaMemcpyDeviceToHost));
                                CHECK(cudaMemcpy(rho, rho_d, SIZEFLT(m*n*h), cudaMemcpyDeviceToHost));

                                        switch (args->outputFormat) {
                                        case CSV:
                                sprintf(autosaveFilename, "%sautosave_iter%05d.csv",inFn->result, iter);
                                                break;
                                        case TECPLOT:
                                sprintf(autosaveFilename, "%sautosave_iter%05d.dat",inFn->result, iter);
                                                break;
                                        case PARAVIEW:
                                sprintf(autosaveFilename, "%sautosave_iter%05d.vti",inFn->result, iter);
                                                break;
                                        }
                                        tInstant1 = clock(); // Start measuring time
                                        WriteResults3D(autosaveFilename, nodeType, nodeX, nodeY, nodeZ,
                                                u, v, w, rho, nodeType, n, m, h, args->outputFormat);
                                        tInstant2 = clock();
                                        taskTime[T_WRIT] += (FLOAT_TYPE) (tInstant2 - tInstant1)
                                                / CLOCKS_PER_SEC;
                                }
                        }
                }       ////////////// END OF MAIN WHILE CYCLE! ///////////////
```

```
        tIterEnd = clock(); // End measuring time of main loop
        taskTime[T_ITER] = (FLOAT_TYPE) (tIterEnd - tIterStart) / CLOCKS_PER_SEC;

        clock_t tEnd = clock();
        taskTime[T_OALL] = (FLOAT_TYPE) (tEnd - tStart) / CLOCKS_PER_SEC; // Calculate elapsed t
        taskTime[T_COLL] /= 1000;
        taskTime[T_STRM] /= 1000;
        taskTime[T_BNDC] /= 1000;
        taskTime[T_MACR] /= 1000;
        taskTime[T_RESI] /= 1000;

        fclose(logFile);
        writeEndLog(logFilename, taskTime);
        writeTimerLog(timeFilename, taskTime);
        if (args->TypeOfResiduals != MacroDiff) {
                writeResiduals(residualsFilename, norm, dragSum, liftSum, m * n * h,
                        args->iterations);
        }
        // Write final data
        CHECK(cudaMemcpy(u, u_d, SIZEFLT(m*n*h), cudaMemcpyDeviceToHost));
        CHECK(cudaMemcpy(v, v_d, SIZEFLT(m*n*h), cudaMemcpyDeviceToHost));
        CHECK(cudaMemcpy(w, w_d, SIZEFLT(m*n*h), cudaMemcpyDeviceToHost));
        CHECK(cudaMemcpy(rho, rho_d, SIZEFLT(m*n*h), cudaMemcpyDeviceToHost));

        switch (args->outputFormat) {
        case CSV:
                sprintf(finalFilename, "%sFinalData.csv", inFn->result);
                break;
        case TECPLOT:
                sprintf(finalFilename, "%sFinalData.dat", inFn->result);
                break;
        case PARAVIEW:
                sprintf(finalFilename, "%sFinalData.vti", inFn->result);
                break;
        }

        WriteResults3D(finalFilename, nodeType, nodeX, nodeY, nodeZ, u, v, w, rho,
                nodeType, n, m, h, args->outputFormat);

        WriteLidDrivenCavityMidLines3D(nodeX, nodeY, nodeZ, u, w, n, m, h, args->u);
        WriteChannelCrossSection3D(nodeX, nodeY, nodeZ, u, v, w, n, m, h, args->u);

        // Write information for user
        printf("\n\nLog was written to %s\n", logFilename);
        printf("Last autosave result can be found at %s\n", autosaveFilename);
        printf("residuals were written to %s\n", residualsFilename);
        printf("Profiling results were written to %s\n", timeFilename);
        printf("Final results were written to %s\n", finalFilename);

        cudaEventDestroy(start);
        cudaEventDestroy(stop);

        freeAllHost(hostArrays, sizeof(hostArrays) / sizeof(hostArrays[0]));
        freeAllGpu(gpuArrays, sizeof(gpuArrays) / sizeof(gpuArrays[0]));

        return 0;
}
```

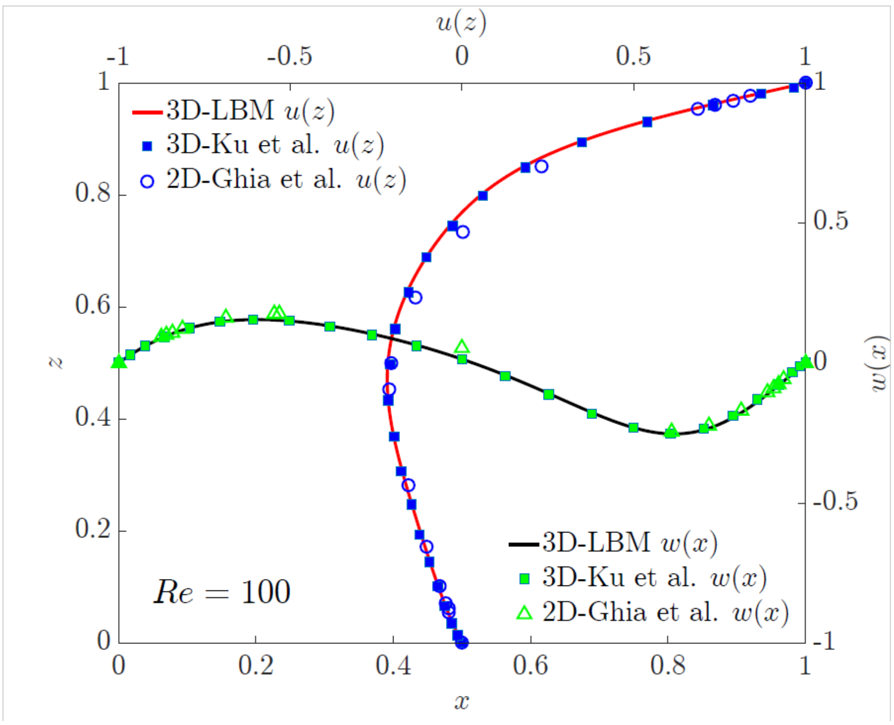
Appendix B Lid-driven cavity – Verification



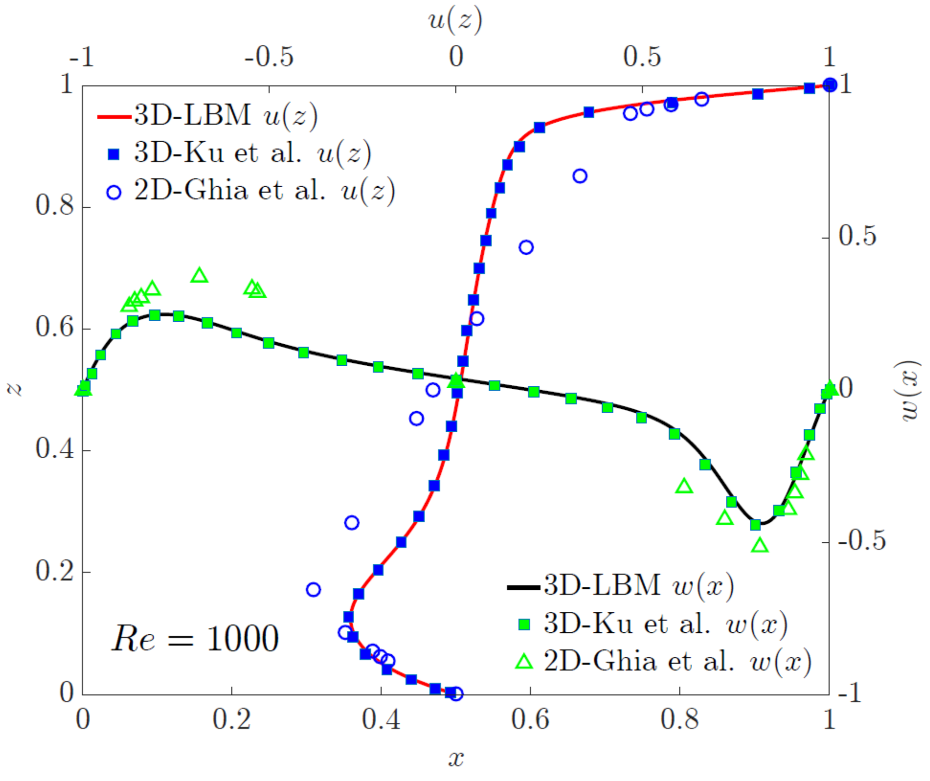


Verification of Lid-driven cavity [23]

Appendix C Lid-driven cavity – Validation

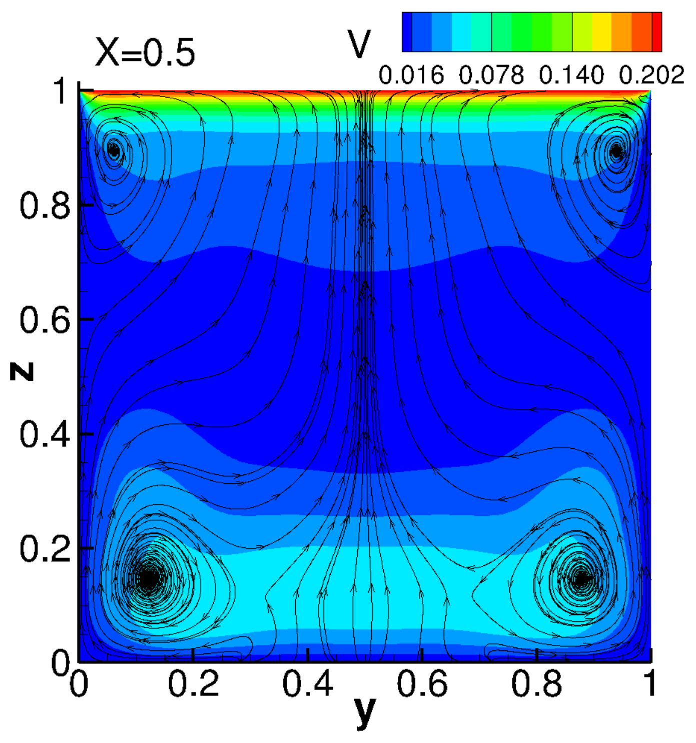


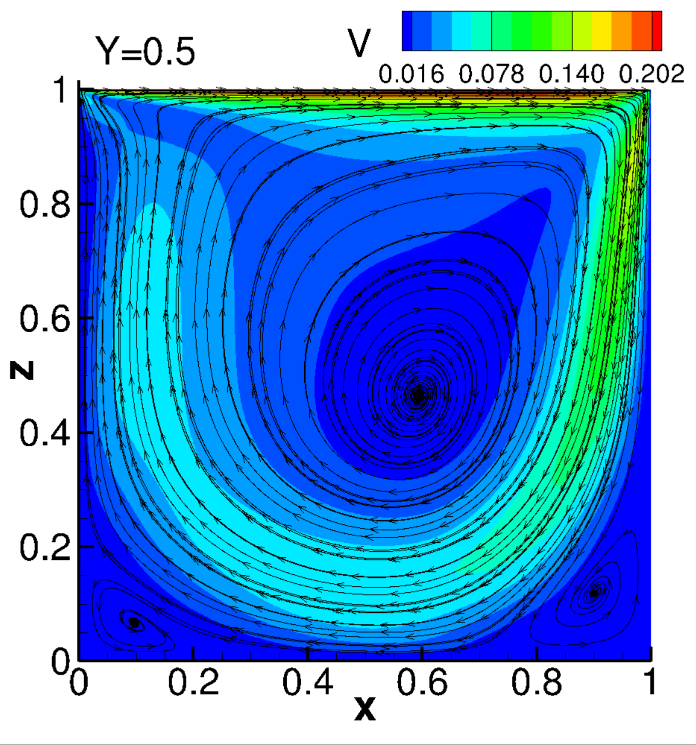
Validation of Lid-driven cavity (Re=100) [23, [35[36]

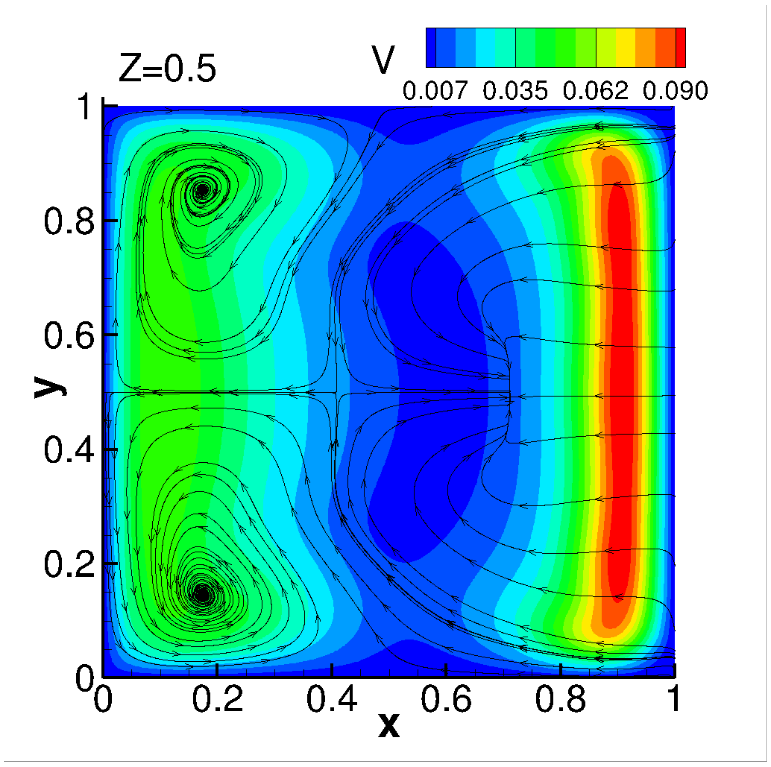


Validation of Lid-driven cavity (Re=1000) [23[35[36]

Appendix D Lid-driven cavity (Re=1000) – results







Cross sections of lid-driven cavity simulation (Re=1000) [23]