Cranfield UNIVERSITY

School of Aerospace, Transport and Manufacturing

MSc Thesis

Optimisation of 2D lattice Boltzmann method using CUDA

Ádám Koleszár

Supervisor:
Dr. Irene Moulitsas

Abstract

In this thesis the development of a 2D lattice Boltzmann CUDA implementation was continued. The previous version of the program were thoroughly profiled both in regards of speed and memory. The computational steps of the method were examined and improved one by one. Every change was validated with the comparison of the old and new results, and later with unit tests. Apart from algorithmic changes the program was also improved in regards of coding quality and maintainability. Usability was enhanced by command-line arguments and extensive documentation of the functions and structures used.

The improved code proved to be on average 30x faster than the original making it easier to use on larger datasets or running it on slightly less powerful hardware. The results were validated against the result from the original code on the usual benchmark geometries. All of the implemented collision models provided the same convergence as the previous results with very small differences due to changes to code structure, well within the numerical accuracy of single precision floating point numbers.

Contents

Nomenclature

λ	mean free path
Kn	Knudsen number
Ma	Mach number
Re	Reynolds number
ν	kinematic viscosity
Ω	collision matrix
ω	collision frequency
ρ	density
τ	relaxation time
$\vec{a}$	acceleration vector
$\vec{F}$	force vector
$\vec{g}$	gravitational forces
$\vec{p}$	momentum vector
$\vec{v}$	velocity vector
$C_{1,2}$	two-body collision operator
f	distribution function
L	characteristic length
m	mass
p	pressure

t	time
v_0	flow velocity
v_s	speed of sound
v_t	peculiar speed

List of Figures

List of Tables

Chapter 1

Introduction

Fluid dynamics is an essential part of applied physics. Most industries get in contact with some kind of fluid mechanics, either through pneumatic machines, pipe systems, vehicles on road, on water or in the air. At some point of the engineering of these machines considerations had to be made, which involves either a lot of measuring and experimentation or a lot of computation and simulation. As the computational power became more available and more reliable throughout the years simulating real life phenomena on computers has became a more viable and in most cases, cheaper option than building machines like wind tunnels.

There are several commercial software packages that can do these kind of simulations, but all of them has their limitation, because every numerical method has its boundaries where they can work well and result in accurate solutions. Most of them uses the Navier-Stokes equations, which is a classical approach, that describes the fluid as a continuum. Another approach is quantum physics, that can describe the motion of very small particles, but it is unnecessary and impractical for industry scale problems. A great median way is the lattice Boltzmann methods [1], that is a statistical physics approach more related to thermodynamics.

The method has proved to be a valuable tool in several different scenarios where the Navier-Stokes methods were simply not enough, like modelling multiphase flows, or small capillaries. The LBM is also liked by software developers as well, because it can highly benefit from parallel computation as in the algorithm the nodes have very limited interaction with neighbouring nodes, making it easier to deal with them in parallel.

To actually compute in parallel one needs several computing cores which very few supercomputers could afford until dawn of multi-core processors. Nowadays a typical desktop computer or even a smart phone has two or even four processing units, making parallel computation easily achievable. But to

increase the amount of data that can be computed in parallel one needs as many cores as possible. In recent years the usage of GPU related applications increased. They have different architecture than the normal CPUs, and they are specifically designed for parallel computation. With the introduction of better programming languages and video card that are not solely designed for gaming and graphical application, computing on GPUs are almost as feasible option as using traditional clusters.

By combining the power of the GPU and the lattice Boltzmann method, a CFD application was created by Tamas Istvan Jozsa[25] last year. The program is able to handle usual 2D fluid dynamics problems, but solves them much faster than a serial application. The previous work created a proof of concept application, which was validated against simulated and measured data, and performed well, but it is still in a proof of concept stage, meaning that is has hidden potential that can be observed and improved upon.

The program was based on two C++ implementation of the lattice Boltzmann method that were also developed on university grounds[2][42]. By combining the better parts of the two program, a C implementation was created by Jozsa and Szoke, to start their separate work. A GPU implementation and a PGAS UPC implementation[40]. Both directions has their advantages and their flaws.

1.1 Objective of the the thesis

The author will continue to work on the GPU code in the hope that by knowing the underlying hardware better, he can gain speed-up or more accuracy from the algorithm, making it a better solution for the problem.

To achieve this, throughout investigation and profiling of the previous model is needed. With these we can identify the model's weak points and improve them one-by-one possibly gaining better performance. Because the author is mainly a software developer, and has limited knowledge about the underlying physics, he will focus on the software aspect of this project, by introducing some practices that are well-known in software development circles, such as testing, clean code and documentation.

1.2 Structure of the thesis

In the following literature review (chapter 2), the reader will find detailed description of the methods used in CFD, a brief history of high performance computing, with comprehensive description of references that this work is

based on. This will be followed by an introduction of CUDA which is the tool/platform we used for this thesis.

In the problem formulation and methodology chapter (3) the reader will will be introduced to the details of the computational methods used, namely the lattice Boltzmann method. This chapter will also introduce a guide to programming with CUDA and other tools used for this project. In order for the method to work one needs prepared data that is a mesh in our case, that describes the geometry and the boundary conditions of our problem. Generating a mesh is out of the scope of this thesis, but the reader can get a general introduction to that topic as well. It is followed by a quick discussion on engineering software and the authors role in the development of this project.

The chapter of results (4) will elaborate on all the work that has been done to improve the solver solution and all the results achieved. This document will close with a conclusion chapter (5) and appendices.

Chapter 2

Literature review

In this section I will describe the general types of simulations and their challenges regarding CFD and then will write about the computational aspects and methods and architectures that are used to solve these problems.

2.1 Computational Fluid dynamics

Fluid dynamics comprises of three fundamental equation, just like most classical dynamics problem:[23]

- Conservation of mass
- Newton's second law ($F = ma$)
- Conservation of energy

So to describe a system one should make a model of it with its necessary simplifications and go through these principles to get equations that will describe the movement of the fluid. It is important to note that while the method is similar there are various kinds of fluid thus they need their own special model.

CFD is technique that uses the power of the computer to simulate flows and compute data with numerical methods. There are several approaches to this, depending on the problem at hand. There are various kinds of flows that can be measured and used. They are dependent on the material, the temperature, the geometry. Different fluids behave differently in the same environment. Taking all that into account we cannot have a good-for-all solution without sacrificing accuracy or using irrational amount of computing power. That is why it is easier to examine the fluid, categorize, make certain

assumptions and choose the fastest or the most convenient algorithm to solve it.

There are some main categories that can help with creating a model. Fluids can be incompressible and compressible. Of course there is no such thing as a totally incompressible fluid, but from a model creation point of view we can put up boundaries where our model will work. With incompressible fluids we usually talk about normal circumstances, no extreme temperature or pressure. Within these boundaries several fluids behave as incompressible which helps a lot with defining continuity.

To further categorize flows in terms of applicable methods, Gad-el-Hak [9] made one in regards to the Knudsen number (2.1), which is a dimensionless quantity that can describe the relation between the mean free path in the fluid (λ), the average distance a particle can travel before colliding with another, and the characteristic length of the flow (L), that can be actual dimensions or more precisely the scale of the gradient of the density $\frac{\partial \rho}{\partial y}$.

$$\mathrm{Kn} = \frac{\lambda}{L} \tag{2.1}$$

Calculating the Knudsen number can help us choose our method. For $Kn|0$ we can describe the fluid as continuum and use the Navier-Stokes equations (2.5, 2.6). For greater numbers the fluid goes through slip-flow and transition regimes until for numbers greater than $\sim$ 10 we can talk about the free-flow model[9].

There are two other dimensionless numbers that are also helping in describing certain flows. The Reynolds number (2.2) represents the ratio between inertial and viscous forces, where v_0 is the characteristic velocity and ν is the kinematic viscosity.

$$\mathrm{Re} = \frac{v_0 L}{\nu} \tag{2.2}$$

The Mach number (2.3) is the ratio of flow velocity (v_0) and the speed of sound (v_s) in the fluid, which can also represent a dynamic measure of compressibility.

$$\mathrm{Ma} = \frac{v_0}{v_s} \tag{2.3}$$

2.1.1 Macroscopic scale

Macroscopic scale means that the fluid is basically treated as a whole. It has macroscopic quantities like pressure, density, speed of flow. The computation of this model can be done with the Navier-Stokes equations which describe

the behaviour of the fluid like Newton's second theorem . It describes the dynamics of a volume of fluid without the actual volume dimension, so the mass becomes density.

$$\vec{F} = m\vec{a} = m\frac{d\vec{p}}{dt} \tag{2.4}$$

$$\nabla\vec{v} = 0 \tag{2.5}$$

$$\rho\frac{d\vec{v}}{dt} = \mu\Delta\vec{v} + \vec{g} - \nabla p \tag{2.6}$$

Meaning that change in flow speed is equal to the friction forces related to viscosity, gravitational forces and forces coming from the change in pressure. Equation 2.5 refers to the continuity, that there is no source or sink in the system.

2.1.2 Microscopic scale

Although it is relatively easy to use, it cannot describe every phenomena that can occur in a flow properly, like turbulence. To understand more about the physics behind fluid one can go down further to the molecular level, and draw up the equations of the individual particle dynamics. Classical physics doesn't work on this level, quantum physics does. The dynamics of atoms and molecules, their relation to each other can be describe with the Hamiltonian equations [1].

2.1.3 Mesoscopic scale

In most cases the molecular description of the fluid is expensive and not even necessary. A better trade-off between the two method is the Boltzmann equation, that describes the fluid from the statistical physics point of view. The velocity of the particles becomes probabilities and the system can be explained with a distribution function $f(\vec{x}, \vec{v}, t)$, which tells us how many particles have a velocity around $\vec{c}$ at given space and time. This way we don't have to examine every particle. We can divide the space up to small volumes and compute the velocities and the collisions. When the distribution function is known the macroscopic values can be computed as well using the Maxwell-Boltzmann equations.

2.2 CFD in use

Making a complete list of modern CFD usages is quite impossible. Every industry or discipline that works with fluid and flows at some point used some kind of CFD to measure or simulate real-life phenomenon so here is a small list which includes but not limited to

- Studying scouring around dams[36], sub-sea pipelines[45], or spur dikes[18] to protect hydraulic structures, or around shipwrecks to study wreck formations[41]
- In nuclear power plants there are several places and scenarios where CFD can be used. Like studying the feedwater heater[5], simulating hydrogen distribution during accidents[44], or modelling pool fire scenarios[13]
- In automobile development, like studying flow and catalysis in the neutraliser[27], examining effect of curved pipes in turbochargers[26], or noise reduction[19]

2.3 Lattice Boltzmann method

The LBM is a computational method based on the Boltzmann equation that describes the fluid with probabilities and the above mentioned Maxwell-Boltzmann equation is its equilibrium solution. Unlike traditional methods that solve conservation equations for macroscopic values the LBM models the fluid as particles that are constantly moving and colliding with each other. It is capable of modelling complex boundary conditions and microscopic interactions.

It also used in quite different fields. One of the most notable field for LBM are the simulation of multiphase flows whether it be GPGPU implementation[4], the development of a cascaded LBM[30] or moving of a droplet and its deformation[20] it is a widely used method. Most of its popularity comes from its ability to work with systems that need more information to simulate than standard flows. Another example is reactive flows where the concentration of materials is also changing, like in solid oxide fuel cells[3] or to model bacterial chemotaxis[46].

It is also easier to make a parallel algorithm of the LBM because of the nature of its steps. Its collision step can be computed in the cells itself, they don't need data from neighbouring cells. Depending on the boundary conditions and profiles used the boundary treatment can also be done within

the cells. The only part that need information form its neighbours is the streaming step. That is why there were several approach to create parallel LBM algorithm on different platforms. It can be implemented using several GPUs[28], using OpenMP and CUDA[47], Fortran-90[6], or MPI[22].

2.4 High performance computing

Humanity always had problems beyond one man's capabilities so the biggest invention was society where people worked together toward a common goal by distributing work. The notion can be applied to the computers as well. If one cannot solve a task in a large enough time let's use more, and distribute the work among them.

HPC means to solve a computational problem on many-core machines. Depending on the tasks different kind of distribution is used. Grid computing enables geographically disperse machines to work together on different or the same task, cluster computing enables closely coupled machines to work like a single system. Another common term for clusters is supercomputers. These machines are custom built group servers connected together with a high throughput network. They are capable of running computationally heavy tasks in parallel reducing computation time effectively.

2.4.1 History

Most of the early supercomputers can be linked to Seymour Cray who joined Control Data Corporation (CDC) to build supercomputers in 1958. In 1960 they completed the CDC 1604 which was one of the first supercomputers that used only solid state materials. Four years later, he and his team completed the CDC 6600, that switched from germanium transistors to silicon ones, that enabled more speed but with the price of overheating, so the machine had a refrigeration unit as well. Its speed-up was not only because of the new materials used, the design of the system also helped the increasing the computational power by distributing some of the work to peripheral elements. In 1962 Ferranti and Manchester University built the Atlas, which was 4 timer faster than the IBM 7094 and it was the first to apply virtual memory and paging, and the first modern operating system, the Atlas Supervisor.

In 1972 Cray left CDC and founded his own company and in 1976 he completed the Cray 1, which was the most successful supercomputer. It featured vector processors with chaining capabilities and used integrated circuits. Until 1990s the fastest supercomputers came from his company, having more processors and more memory, which needed more powerful cooling as well.

For example the Cray 2 was submerged in liquid coolant.

The 1990s was the era of massively parallel supercomputers. Until that time the machines gained speed-up from higher frequency processors. The massively parallel machines employed more and more processing units that were connected with fast network using different topologies from machine to machine. This era was also the dawn of Message Passing Interface (MPI), that offered a standardised way to pass data between connected nodes, and is available for various programming languages.

The list of the most powerful supercomputers can be inspected on `http://www.top500.org` since 1993. Figure 2.1 shows the average power of the summary of the top 500 machine, the most powerful machine and the 500th machine over the years.

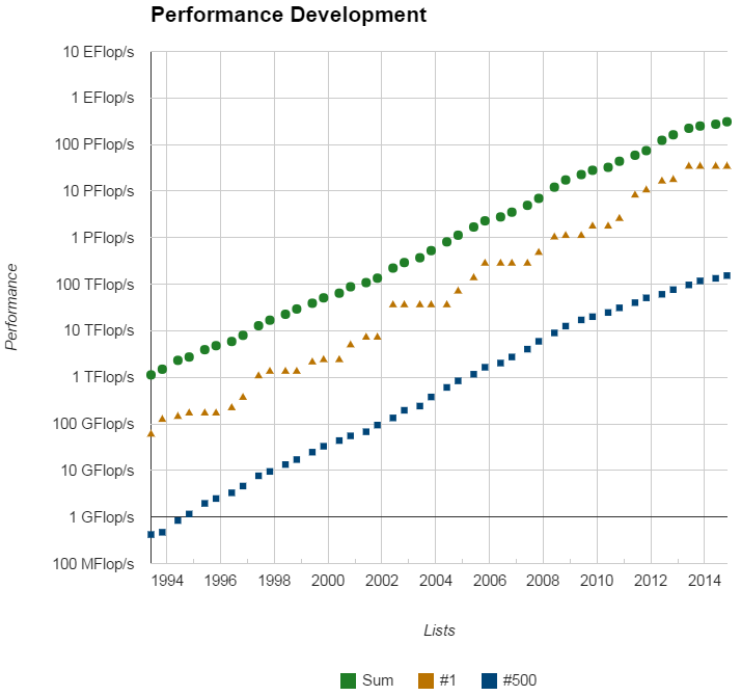


Figure 2.1: Increase in computing power since 1993[15]

2.5 Graphical processing unit

In short GPU, is device in computers that handles the graphical display of information of the computer. Whether it is displaying text, graphical user interfaces or 3D games, the GPU became an essential part of modern computers or even electronic devices like smart phones. But this wasn't always the case.

2.5.1 GPU history

The history of the graphical units starts in the 1970s[37]. The first device was merely just a pass-through between the CPU and the screen, by converting the data into serial bit-mapped video output. The CDP1861 from RCA was one of the first devices that was able to output NTSC signal. Its successor the TIA was integrated into the Atari 2600 and was able to handle sound effects and input controllers as well.

Later based on the Motorola's MC6845 video address generator IBM and Commodore quickly followed and it became part of the first personal computers. In the professional field Intel's iSBX 275 Video Graphics Controller Multimode Board had a stunning 256x256 resolution with 8 colours and it had 32kB of video memory that was sufficient enough to draw graphical primitives like lines, arcs, circles and rectangles.

ATI was founded in 1985 under the name of Array Technology Inc and came out with their first Color Emulation Card that featured 16kB memory and could drive a monochrome monitor. It was shipped with Commodore Computers hence the huge success of the chip.

With the arrival of colour monitors, the lack of standardisation become an issue. ATI, NEC and six other graphics chip company founded the Video Electronics Standards Association (VESA). As with any other computing device the numbers started to grow. New chips featured more memory, could display bigger resolutions and more colour. Many new companies were created in that era.

In 1992 the OpenGL 1.0 API was released by SGI and soon became the standard interface for 2D and 3D graphics, add used by the gaming industry as well. The purpose of the interface was to bridge the gap between the graphical hardware and the programmers, and make 3D programming easier by providing an abstraction layer. However Microsoft started developing their own API as well, the Direct3D, without the intention of fully supporting OpenGL on their systems and gaining the developers favour against various proprietary APIs.

Another rival for OpenGL was Glide, developed by 3dfx Interactive, who

was also famous for its cutting edge Voodoo series accelerator cards. These cards were supplementary devices alongside graphic cards and they enabled 3D graphical computation while the GPU processed only 2D graphics. Later they sell the devices in pairs, providing their own graphics solution. They also developed an efficient way to communicate between the cards (namely the SLI technology) which later became the communication interface between nVidia cards as well. In the early 2000s 3dfx lost its fame and popularity and with them Glide API also disappeared.

After the end of 3dfx two major manufacturer remained, namely nVidia and ATI and their competition is still ongoing on the market of commercial graphic cards.

2.5.2 Graphics pipeline

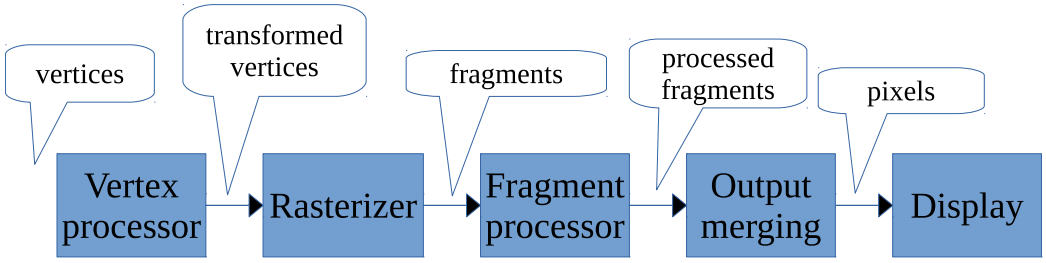


Figure 2.2: OpenGL graphics pipeline

Figure 2.2 shows a brief depiction of the operation pipeline in the GPU. With the introduction of the vertex and fragment processors the GPUs become programmable. These small programs are called shaders. At the beginning they could only be programmed by assembly language but later development enable easier to use C like languages (e.g. cG[1]). This enabled the programmers and graphics designers to write their own graphical programs or something more general.

2.5.3 General purpose GPU

It was not until the 21st century that these devices became mainstream and their improvements gave birth to a new kind of computing. With the introduction of shaders and the availability of floating point computation, programmers could use the graphical hardware for things other than just graphical display and 3D computation. While these small programs were

[1] `https://developer.nvidia.com/cg-toolkit`

designed to compute graphic related things like shadows, surfaces and lightnings, their 2D/3D matrix computation options made them feasible to compute general matrix related tasks as well. A great example for its usage was a SPECT image attenuation correction algorithm implemented on a GPU[24]. They transported their data as a texture and let the GPU do the different rotation and interpolation algorithms achieving a great speed compared to commercial processors.

Even in the era of devices that only handled integer and low precision floating point arithmetic, some research was already put into the usage of the GPU, like non-linear diffusion computation using 12bit arithmetic[31] and matrix-matrix multiplication with 8bit arithmetic[10]. With the introduction of the single precision floating point arithmetic (32bit), more scientific application could benefit from the device, and because it promised faster computation many of them concentrated their effort developing their existing algorithms on the GPU and compare it to the CPU. S. Tomov et al. used it to benchmark probability based computations like Monte-Carlo methods on the Ising model[38].

Even though it had the power and the opportunity to become a new computational platform, its design was very hard to comprehend and program, so apart from some research and small successes it could never take hold. At least not until more general programming languages and devices were invented.

With introduction of CUDA a new era of high-performance computing started. With it's C/C++ language bindings, and their own compiler it enabled the user to program their algorithms easily, and run parallel codes on commercial devices, not just high-end supercomputers. The rival ATI also developed their own computational platform and API (Stream), but it could not take hold. ATI abandoned their proprietary glsapi and started supporting the emerging open-source solution: OpenCL[2].

OpenCL is an open source API with goal of a general programming interface for heterogeneous systems, consisting of various numbers of computing elements (CPUs, GPUs). Later nVidia also supported OpenCL through their drivers but their own solution is inevitably always runs faster on their devices.

Although OpenCL is a more general approach for the problem of GPGPU programming it also means it takes a lot more effort to develop a platform independent solution with it, as it needs a lot of initialisation steps. NVidia's CUDA can be simpler to develop because it only needs to know a certain set of hardwares.

[2] `https://www.khronos.org/opencl/`

Chapter 3

Problem formulation and methodology

3.1 The Boltzmann Equation

As we stated above the lattice Boltzmann method is a statistical mesoscopic approach to various fluid dynamics and thermodynamics problem. It numerically computes the solution for the Boltzmann equation, but let's start first with the basics. According the Newton's second law the following dynamic equations can be written for a simple particle that moves. Equation (3.1) tells about the momentum ($\vec{p_i}$) of the particle, where i denotes that it is the i-th particle in a given volume. The momentum equals the mass (m) times the velocity of the particle at point $\vec{x_i}$. The second equation (3.2) describe the effect of the forces ($\vec{F_i}$) on the i-th particle.

$$m\frac{d\vec{x_i}}{dt} = \vec{p_i} \tag{3.1}$$

$$\frac{d\vec{p_i}}{dt} = \vec{F_i} \tag{3.2}$$

Solving these equations for every particle could lead to an expensive calculation with low information density. To overcome this, a probability density function $f(\vec{x_i}, \vec{p_i}, t)$ were introduced, that describes what is the probability of finding a particle with the momentum of $\vec{p_i}$ in a given point in space ($\vec{x_i}$) and time (t). With this we get the Boltzmann equation (3.3), where $C_{1,2}$ is the so-called two-body collision operator, that describes the interaction particles, while the left hand side of the equation is the change in the distribution function.

$$(\partial_t + \frac{\vec{p}}{m}\partial_{\vec{x}} + \vec{F}\partial_{\vec{p}})f(\vec{x_i}, \vec{p_i}, t) = C_{1,2} \tag{3.3}$$

In order to get the collision operator one must know the two-body distribution function ($f_{1,2}$) as well, which represent the probability of having two particles at the same time and space. Luckily the probabilities are independent for both particles so $f_{1,2} = f_1 f_2$[39], which makes the Boltzmann equation easier to solve.

Knowing the distribution function enables us to get the macroscopic properties as well, and with the Chapmen-Enskog procedure one can also derive the Navier-Stokes equations from the Boltzmann equation, meaning that there a clear connection between the micro and macroscopic descriptions. To get the macroscopic variables one should integrate the distribution function, where u is the macroscopic velocity of the flow and ρe is the energy density.

$$m \int f d\vec{v} = \rho \tag{3.4}$$

$$m \int f \vec{v} d\vec{v} = \rho \vec{u} \tag{3.5}$$

$$m \int f \frac{v^2}{2} d\vec{v} = \rho e \tag{3.6}$$

The distribution function can be separated to equilibrium and non-equilibrium parts $f = f^{eq} + f^{neq}$, where after the characteristic relaxation time τ_r, the non-equilibrium part disappears, leading to the well-know Maxwell-Boltzmann distribution in D dimension

$$f^{eq} = \rho (2\pi v_T^2)^{-D/2} e^{-c^2/2v_T^2} \tag{3.7}$$

where $\vec{c} = \vec{v} - \vec{u}$ is the molecules relative speed to the fluid, called the peculiar speed, and v_T is related to the temperature that can be also derived from the distribution.

$$< v_T^2 >= \int v^2 f(v) dv = \frac{k_B T}{m} \tag{3.8}$$

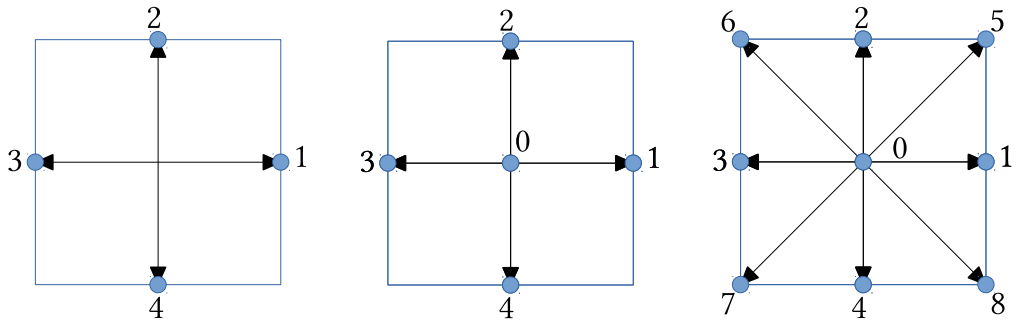
3.2 The Lattice Boltzmann Method

Our intention with the lattice Boltzmann method is to discretize the Boltzmann equation into nodes and lattices in space and time and solve the equation for the individual elements. In order to do so we need to know or at least approximate the two-body collision operator. Next we need to define how cells effect each other by their collisions. Then we need to deal with the boundaries of our problem space as they have different behaviour than

neighbouring cells. And at last but not least we can compute our macroscopic variables from the distribution function which can then be fed into the collision computation again forming a loop where every cycle represents a discrete moment in time.

3.2.1 Lattice arrangements

By discrete space we mean a mesh that contains cells, where we can deal with the particle as distribution function (more about it later in section 3.3). Depending on that we have a 2D or a 3D problem we can define simple and more complex lattice arrangements or in other words different neighbour relations for the cells. For our model we used the D2Q9 arrangements. The notation covers the dimension of the problem (number after D) and the number of neighbouring cells that we want to take into account when propagating our probabilities after the collision step (number after Q). It is 9 in our case because we count in the starting cell itself as well. Figures 3.1 and 3.2 show the usual arrangements used in 2D and 3D models.



(a) D2Q4 (b) D2Q5 (c) D2Q9

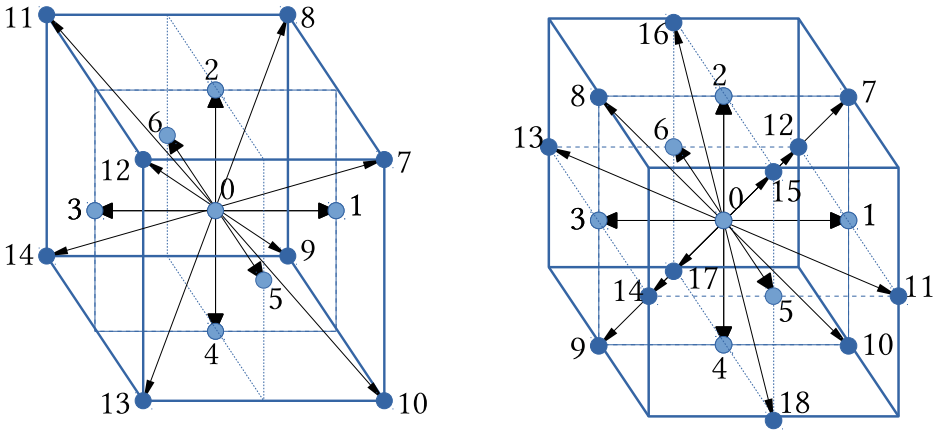
Figure 3.1: 2D lattice arrangements

3.2.2 Collision models

The first step of the LBM is the collision step. We simulate the collision of the particles and their effect on the distribution function. Our program has three models implemented.

BGKW model

Discovered by P.L. Bhatnagar, E.P. Gross and M. Krook in 1954[34], simultaneously with Welander, hence the abbreviation. It is a simplified model,



(a) D3Q15 (b) D3Q19

Figure 3.2: 3D lattice arrangements

an approximation to the two-body collision operator, where ω is the collision frequency and τ is the relaxation time that is related to kinematic viscosity by (3.10) in case of the D2Q9 setup.

$$\Omega_{BGKW} = \omega(f^{eq} - f) = \frac{1}{\tau}(f^{eq} - f) \tag{3.9}$$

$$\omega_{D2Q9} = \frac{1}{3\nu + \frac{1}{2}} \tag{3.10}$$

Substituting it into the Boltzmann equation and discretising it will lead to the following equation.

$$f_i = (1 - \omega)f_i + \omega f_i^{eq} \tag{3.11}$$

TRT model

The TRT model is an abbreviation for two-relaxation-time and is developed by Ginzburg[14]. It is based on the fact that the distribution function can be split into symmetric and asymmetric parts, where f_{-i} is the distribution function in the opposite direction of f_i.

$$f_i^s = \frac{1}{2}(f_i + f_{-i}) \tag{3.12}$$

and

$$f_i^a = \frac{1}{2}(f_i - f_{-i}) \tag{3.13}$$

leading to following equation

$$f_i = f_i - \frac{1}{2}(\omega_s + \omega_a) f_i^{s,eq} - \frac{1}{2}(\omega_s - \omega_a) f_i^{a,eq} \tag{3.14}$$

where ω_a can be computed from ω_s, for D2Q9 it is chosen to minimise the viscosity dependence on slip velocity:

$$\omega_a = \frac{8(2 - \omega_s)}{8 - \omega_s} \tag{3.15}$$

MRT model

The MRT model is an abbreviation for multi-relaxation-time[29], which works by generalising the collision operator using a collision matrix ($\mathbf{\Omega}$).

$$f_i = f_i - \mathbf{\Omega}(f_i - f_i^{eq}) \tag{3.16}$$

It can be converted to momentum space, where $\mathbf{M}$ is the mapping between velocity and momentum space, $\mathbf{S}$ is the relaxation matrix, $\mathbf{m}$ and $\mathbf{m}^{eq}$ are vectors of moments, so that $\mathbf{m} = \mathbf{M}\mathbf{f}$. All of them can be calculated for the D2Q9 setup.

$$f_i = f_i - \mathbf{M}^{-1}\mathbf{S}(\mathbf{m} - \mathbf{m}^{eq}) \tag{3.17}$$

3.2.3 Streaming

The change that is computed in the collision step will affect neighbouring cells as well. The streaming step computes how these changes propagate to other nodes. As can be seen on figure 3.3 the distribution functions propagate to the neighbour in the direction of the lattice.

3.2.4 Boundary conditions

Bounce-back model

The bounce-back model is usually used on solid stationary or moving boundaries, like walls. The idea is that the particles bounce-back from these walls, whether elastically or not is dependent on the model. It is common practice to place the wall in the middle of lattices (shown on figure 3.4). After the steaming step f_4, f_7 and f_8 are known, so applying the conservation of mass and momentum leads to $f_2 = f_4$, $f_5 = f_7$ and $f_6 = f_8$.

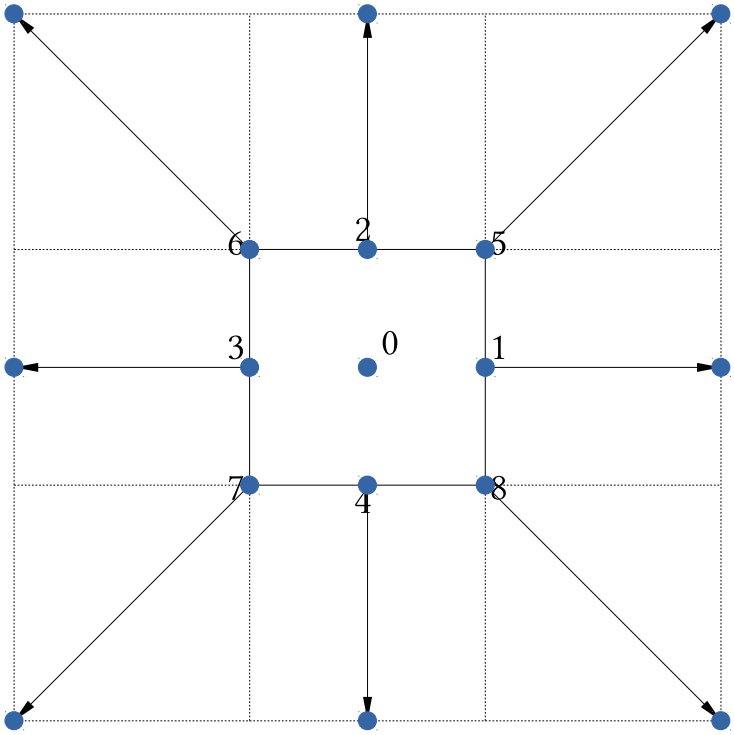


Figure 3.3: Streaming step

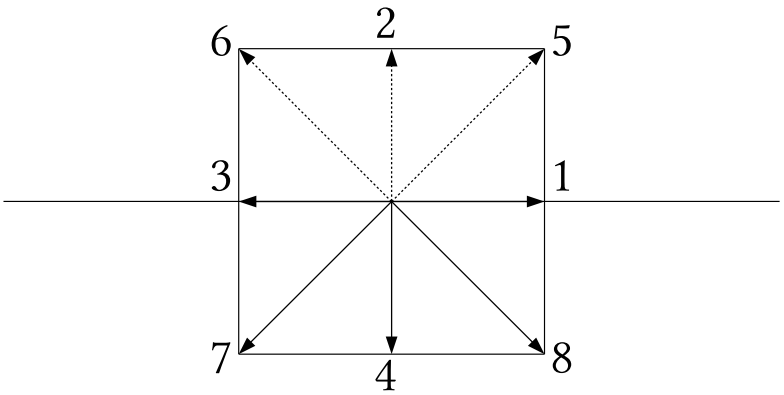


Figure 3.4: Bounce-back scheme

Inlet conditions

A method to deal with inlets and outlets is coming from Zue and He[35] which is coming from the following known equations (if the inlet velocities

are known), in our example on the west boundary:

$$\rho = \sum_{k=0}^{8} f_k \tag{3.18}$$

$$\rho u = f_1 + f_5 + f_8 - f_6 - f_3 - f_7 \tag{3.19}$$

$$\rho v = f_2 + f_5 + f_6 - f_7 - f_4 - f_8 \tag{3.20}$$

$$f_1 - f_1^{eq} = f_3 - f_3^{eq} \tag{3.21}$$

Substituting the equilibrium distribution functions from the BGKW model and rearranging the equations yields the four unknowns we need to compute for the boundary:

$$\rho = \frac{1}{1-u}(f_0 + f_2 + f_4 + 2(f_3 + f_6 + f_7)) \tag{3.22}$$

$$f_1 = f_3 + \frac{2}{3}\rho u \tag{3.23}$$

$$f_5 = f_7 - \frac{1}{2}(f_2 - f_4) + \frac{1}{6}\rho u + \frac{1}{2}\rho v \tag{3.24}$$

$$f_8 = f_6 + \frac{1}{2}(f_2 - f_4) + \frac{1}{6}\rho u - \frac{1}{2}\rho v \tag{3.25}$$

The unknown lattice directions can be computed similarly for boundaries on other sides as well.

Outlet conditions

In case of outlets in most cases the output velocities are not known. The common practice is to use the open boundary condition, which is an extrapolation of the distribution functions on the boundary using values from neighbouring nodes. We can use first or second order extrapolation depending on which one is more stable. For example, second order on the east boundary:

$$f_{3,n} = 2f_{3,n-1} - f_{3,n-2} \tag{3.26}$$

$$f_{6,n} = 2f_{6,n-1} - f_{6,n-2} \tag{3.27}$$

$$f_{7,n} = 2f_{7,n-1} - f_{7,n-2} \tag{3.28}$$

There is another approach that is similar to the inlet conditions which can be used if the output pressure is known (and thus the density). An example

for the east boundary would yield the following:

$$\rho_{out}u = -\rho_{out} + f_0 + f_2 + f_4 + 2(f_1 + f_5 + f_8) \tag{3.29}$$

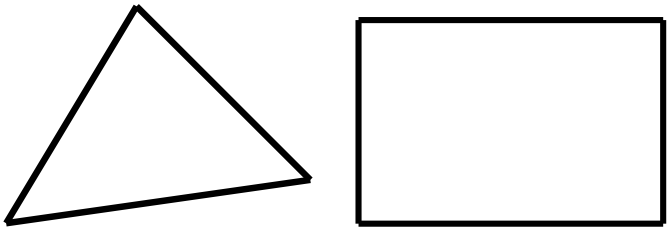
$$f_3 = f_1 - \frac{2}{3}\rho_{out}u \tag{3.30}$$

$$f_7 = f_5 + \frac{1}{2}(f_2 - f_4) - \frac{1}{6}\rho_{out}u \tag{3.31}$$

$$f_6 = f_8 - \frac{1}{2}(f_2 - f_4) - \frac{1}{6}\rho_{out}u \tag{3.32}$$

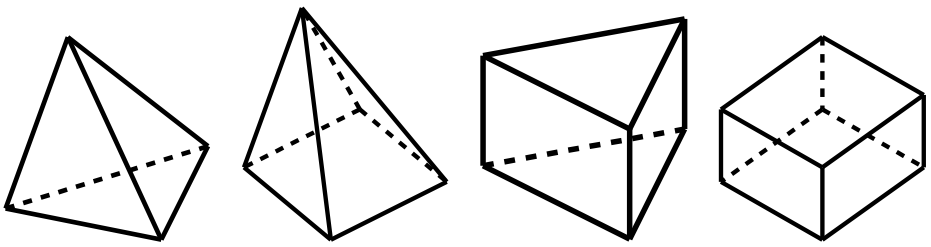
3.3 Mesh generation

In computational methods to solve partial differential equations (PDE) a discrete representation of the space is needed. This is the mesh that divides the space into small parts where the equations can be approximated. In 2D there two types of meshes regarding the shape of the elements: triangular and quadrilateral (Figure 3.5). In 3D they can be tetrahedral, quadrilateral pyramids, triangular prisms and hexahedrons.



(a) triangle (b) quadrilateral

Figure 3.5: 2D shapes for mesh



(a) triangle (b) pyramid (c) prism (d) hexahedron

Figure 3.6: 3D shapes for mesh

Meshes can be also classified into structured and unstructured grids. Structured grids (quadrilateral and hexahedral) has regular connectivity,

meaning that the storage arrangement of these grid also defines their neighbourhood relationships. While unstructured grid (usually triangular and tetrahedral) need to store information about the connection of the elements separately.

For reasonable size grids a mesh generator is needed, that can divide the space up according specified parameters. For this thesis we used the same in-house mesh generator software as in the previous theses[25][40] that was made by G. Abbruzzese[2]. It is a C++ program that can generate equidistant meshes for geometries defined in Pointwise[1]. Several geometries were already defined with the software, so creating these was not part of this work.

3.3.1 Used meshes

To validate the changes made during the optimisation of the code we chose three geometry, that are the basic examples in CFD but are quite representative. These are the channel (Fig. 3.7) which simulates a simple tube in 2D. The sudden expansion (Fig. 3.8) of the tube into a larger one. And the lid-driven cavity (Fig. 3.9) that simulates a small rectangular space which connects to a flow on its upper side. The dashed lines represent boundaries that have inward or outward flow, the thick lines are the walls and the arrows represent the direction of the flow.

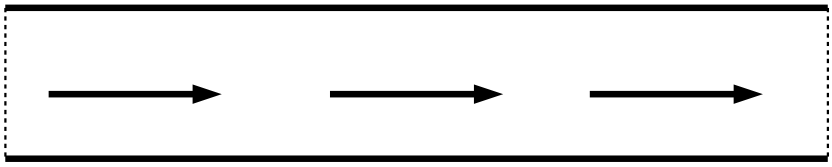


Figure 3.7: Channel

3.4 CUDA programming

The ease of using the CUDA platform is that it is integrated into the C/C++ language and because it has its own compiler that has similar functionality and syntax as the GNU C compiler (GCC[2]). Most of the options that available in GCC are available in NVCC (nVidia CUDA compiler) as well. The development suite is also capable of linking against non-GPU code as well,

[1] `http://www.pointwise.com/`
[2] `https://gcc.gnu.org/`

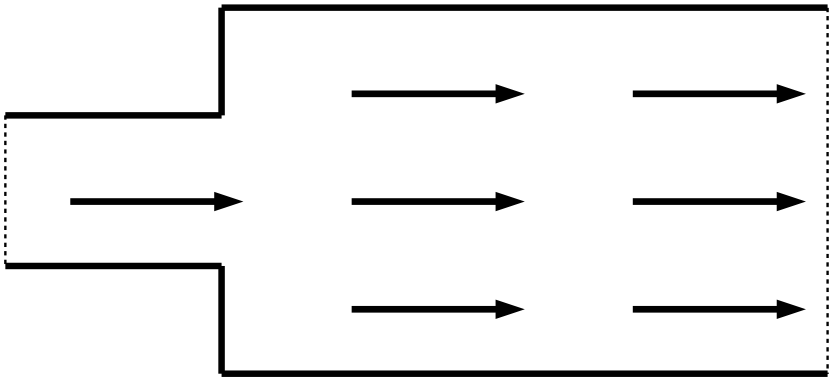


Figure 3.8: Expansion

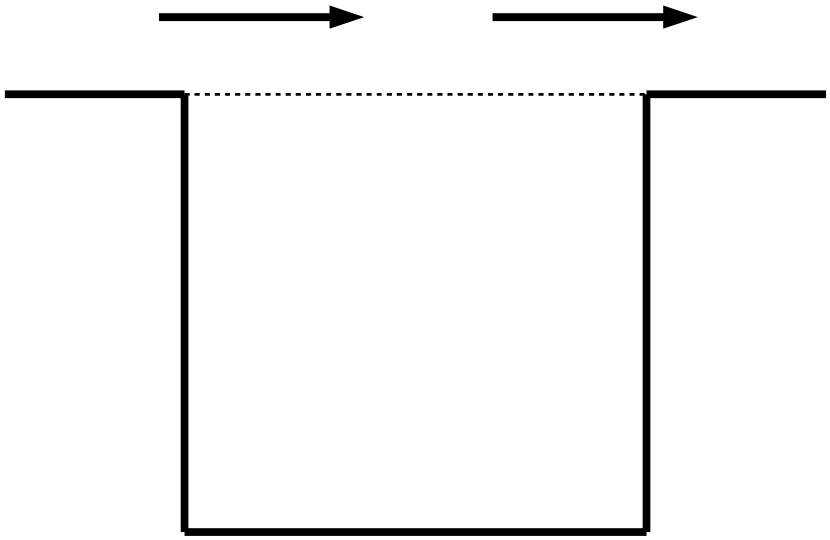


Figure 3.9: Lid-driven cavity

to achieve this NVCC delegates the CPU code to the platform dependent C compiler: GCC on Linux and MSVC[3] on Windows, making CUDA code easy to integrate into existing code bases.

The CUDA API divides the code into three different parts. The *host* part consist of source files and codes running on the CPU, using the memory chips on the motherboard. The *device* part consist of code run and called on the GPU. These parts are not accessible from the host side. The last part is a the bridge between the two, the *kernels*. They are special functions that are called by the CPU but run on the GPU. They also have additional parameters that are predefining the problem size and are annotated with

[3]Microsoft Visual C/C++

triangle brackets: `<<<.,.>>>`.

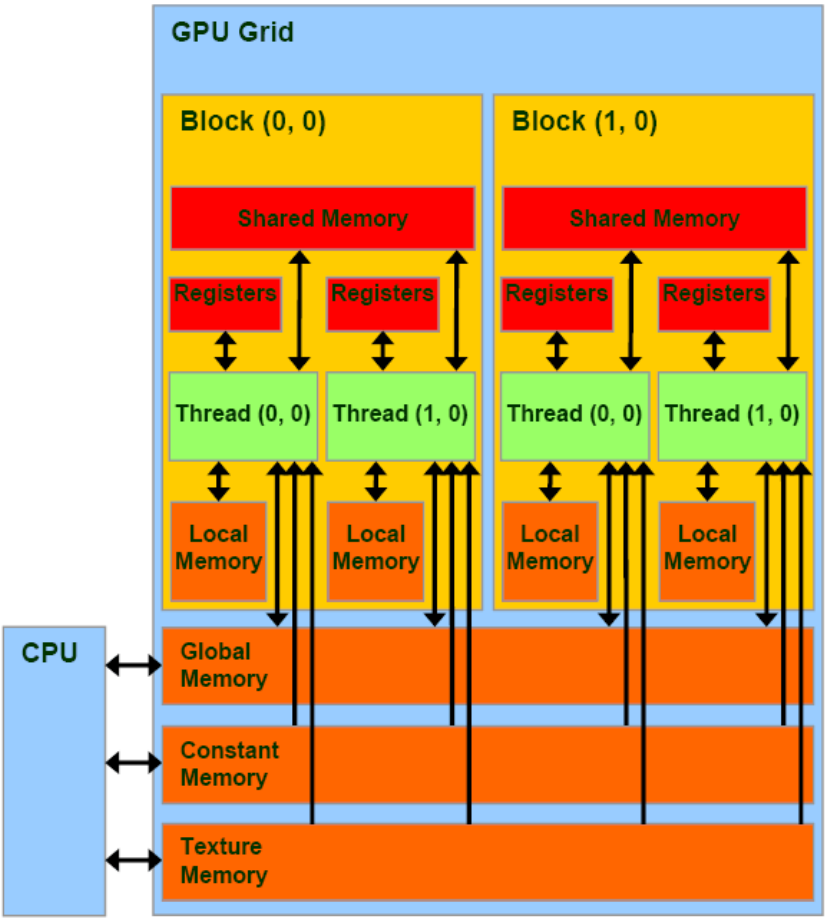


Figure 3.10: CUDA threads and memory types

Other than calling conventions and separate domains the CUDA API also defines different type of memories that the device can use. Each of them has their own advantage in a given situation. Figure 3.10 show a basic diagram about the thread hierarchy and the different memory types of the GPU.

3.4.1 CUDA sample code

The most trivial task that can easily show how to use CUDA parallelisation instead of loops is the adding of two vectors. In serial code this can be achieved by going through all the elements of the two vector and adding them one-by-one as figure 3.11 shows.

With CUDA the approach is a little bit different. Like most parallel cases the intention is to find independent elements of the dataset and work with

```
for (i=0; i<n; ++i)
  vC[i] = vA[i] + vB[i];
```

Figure 3.11: Adding vectors in serial code

large chunk of them at a time in parallel, meaning that in one cycle the same operation is done on multiple elements. As was written above, the CUDA divides the operations into *threads*. In order to add the two vectors we need a thread for every element of the vector. For a lot of elements the threads need to be grouped together into *blocks* that will form a *grid*. This is needed because the device can only handle a finite amount of threads.

```
__global__ void gpuAdd(float *vA, float *vB,
                       float *vC, int size)
{
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  if (i < size)
  {
    vC[i] = vA[i] + vB[i];
  }
}
```

(a) Kernel function

```
dim3 threads(256);
dim3 blocks((n-1)/threads+1);

gpuAdd<<<blocks, threads>>>(vA, vB, vC, n);
```

(b) Call kernel from host code

Figure 3.12: Parallel vector add with CUDA

To add vectors with CUDA we will need a kernel, and we need to call the kernel from the host code, specifying the amount of threads. Figure 3.12 shows us how to do it. The CUDA API defines the `dim3` type that is a struct that holds 3 integer values to define the dimensions of our problem. Because we are only dealing with a 1D vector we don't need define higher dimensions for our threads and blocks. The kernel code itself is very similar to our serial code in regards of the actual computation but it needs some additional information.

The threads behave as individual processes, they don't know anything about each other, only if we explicitly introduce synchronisation or shared

variables. They only know their own ID which we can use to identify the element the thread needs to work on. Because computer are representing everything in binary data, the number of threads used are usually a power of 2, but if we want to deal with arrays of various sizes, we need to protect ourselves from invalid memory reads. That is why the kernel needs to know the array size, so that threads that outside of the boundaries won't do any harm, just idle.

3.4.2 Shared memory

The previous example used only the global memory for adding two vectors mainly because the operation does not need other elements of the vector. In cases where a thread needs data from elsewhere, like neighbouring cells, the usage of shared memory can boost performance. The shared memory is accessible from blocks of threads, so if every threads loads part of the data into the shared memory (in parallel), then for the remaining of the operation the threads only need to access the more faster and smaller shared memory, instead of loading slowly from global memory.

3.4.3 Constant and texture memory

A part of global memory is reserved and called constant and texture memory to indicate their different use. The constant memory, as its name suggests is for constant values, that are needed by the kernels or the device functions. Its size is quite small, but it has faster access than global memory.

The texture memory is remnant from the graphical computations (the GPU is mainly used for). It is allocated on the same chip as global memory, but it supports different data access. While the global memory is mainly for large amount of serial data (arrays), the access logic and caching of the texture memory was designed for higher dimension data. It speeds up accessing data that are spatially close, making it a better solution for algorithms that use neighbouring cells data for computation.

3.5 Engineers and programmers

There is a certain aspect that needs to be discussed regarding engineering software solutions. Namely the difference between to the disciplines and the resulting products. People with engineering background learn mathematics, physics and other natural sciences mainly. They acquire skills to plan az experiment, to measure and design real life objects, machines etc. Most of

them learn programming out of necessity to simulate real life phenomena or to post-process results with the computer. Their main programming languages are fortran and C for computation and other script languages such as python to post process data. Their main goal is a quick, simple and working solution in order to get a certain task done.

Programmers have different interests and goals regarding a program. They also learn mathematics (most likely not the same topics), so that can be a common point but other than that, their education is different from the engineers. They think of a software in a more abstract way, by classifying and planning the tasks at hand before the actual implementation. During their studies they also concentrate more on the quality of the software. Their primary goal is to have a software that is open to further development, correct and possibly more understandable by others making teamwork easier.

Despite the differences for engineering problems such as this lattice Boltzmann solver, both disciplines needed for a good solution. Engineers needed for their knowledge and experience in physical phenomena and experiments and programmers needed to create a software solution that can withstand time and work as flawlessly as possible.

3.6 Clean code

Clean code is usually meaning an easily understandable code. Several good practices need to be employed in order for it to become one. If a code is clean it also means that is free of obstructing elements preventing modification and expansion. A source code can be hectic, unpredictable and hard to comprehend no matter the amount of documentation it is accompanied with.

So first things first, the code should document itself. To achieve this one should employ a naming technique that will tell exactly what that variable is for, or what that function does. We should also avoid using multiple responsibility. If a function does more than its name suggests that means it obscures us from understanding it easily, and can also lead to undesired consequences if someone else try to reuse that particular code.

This and other practices can be found in many software development books, but probably to most notable of the is Robert C. Martin's Clean code book[32], and educational video series. It is mostly aimed at object oriented programming but its practices can be useful for any developer. It describes the way badly written codes tend to work after some time, and several techniques to eliminate these problems from the design phase of software development. It gives heavy emphasis on agile software development as well, as he was one of the few that created the Agile Manifesto[11], that

describes briefly the method that focus heavily on user-developer interaction and quickly responding to changes of plans.

3.6.1 Test-driven development

One of the key technique of agile software development is TDD. The process is the following. We have a class that we want to implement and test. If skeleton of the class is already available we first create unit tests for its method with the intention that they should fail, as they not implemented properly yet. Than we implement the methods to satisfy the previously created tests. If they pass we can go on to the next method and so on.

By using this technique we can make sure that our methods are tested from the beginning and by running all the test every time something is changed we can always rely on our tests that the new functionality does not break the old one.

3.6.2 Refactoring

The technique of refactoring[33] is often used together with test-driven development. The main idea of refactoring is that we don't change functionality, we just change form for the sake of readability and maintainability. The simplest way to refactor your code is to change variable and function names, so that they instantly describe the functionality related to them. A common technique is to use shorter names for small scope variable and bigger, more revealing ones to globally available variables. I t is quite the opposite for function names, as they need short names if they are bigger and general (e.g. `init()`), to mark their significance with the perfect one word, and smaller one-liners, and utility functions get longer names that should clearly and precisely describe their purpose, (e.g. `increaseMatrixElementWeight()`).

Another technique is separating functionalities. The goal is the single responsibility so that every function should have only one purpose and only one effect. This also means that the C language way of defining output parameters are not desired but of course can rarely be eliminated. If your function returns more than one value, split into separate functions so that each of them return only one value. This also helps maintaining large code bases as source files tend to get bigger and bigger, and inevitably more unreadable.

Another effect of large code and catch-it-all functions is that they invite code duplication. If such a moment occurs when you have a functionality that's already written, but in the middle of another function, instead of copying it, you should make it a separate function and call from both sides, making maintaining easier.

There are several other techniques that can help and encourage a better design, but most of them are developed for object oriented software design. So we should take out what we can, because it can improve a C code as well.

3.7 Used development tools

3.7.1 Revision control

Unfortunately the first thing a programmer will do after developing a software is forgetting how he or she did it. Usually it is also problematic to even remember when certain features were introduced. Without help it can become a difficult task to follow something back to its sources. For these reasons it is essential to use some kind of system that helps keeping up with our changes and provides a way to roll back to previous solutions.

These tools are called revision control systems. Their job is to keep track of the changes done to the code. The developers responsibility is to "save" their work at certain points and annotate them properly, so either them or others can follow up on what has been done. Software developing teams usually have a guideline on how to annotate changes and how often should they do it.

A one man project might not need the services of these systems, but for the author it was an essential tool to keep track of the different speed-up attempts and their results even after several other changes. For this task the author used GIT[4], which is a well-known and used system in *nix environments and now also on Windows, as the Visual Studio Team Explorer platform[5] supports it.

It is capable of keeping track of changes on the user's local machine building a local repository, but is also able to connect to a centralised repository so that development teams can share their work. These features helped the open-source community to share work for a long time now.

3.7.2 cuda-gdb

The CUDA debugger[7] is very similar to the GNU debugger[6] (gdb) that comes with the GNU C Compiler (GCC) package. It is included in the Linux version of the nVidia compiler (NVCC) package. It recognises all the

[4] `https://git-scm.com/`

[5] `https://www.visualstudio.com/en-us/products/team-explorer-everywhere-vs.aspx`

[6] `http://www.gnu.org/software/gdb`

command that gdb, and lot others related to debugging on the device. It is very useful when the program throws CUDA related exceptions, because normal tools cannot understand them and cannot trace back the source of the exceptions.

3.7.3 cuda-memcheck

The CUDA memory checker[8] is also part of the CUDA Toolkit, and is used to discover memory related errors. Its services are similar to the commonly used valgrind[7] memory checker. It gave us great help in finding memory leaks and invalid addressing in the code that don't produce imminent problems but on the long run they can cause serious memory corruption issues.

[7] `http://valgrind.org/`

Chapter 4

Results and discussion

4.1 Machine configuration

To use the computing power of the GPU of course we need a computer that has one. Throughout the work on this thesis two different configuration was used. One of them was an nVidia TESLA based machine on the university, and the other one was a less powerful commercial nVidia GPU card in a desktop computer. Details of the machines used, can be observed in Table 4.1.

	TESLA	GTX 460
CPU	16x Intel Xeon @2.4GHz	4x Intel Core2 @2.4GHz
Memory	24GB	6GB
GPU	TESLA C2050	GeForce GTX 460
Compute capability	2.0	2.1
CUDA cores	448	336
GPU clock rate	1150MHz	1350MHz
GPU memory	3GB	1GB
GPU mem. cl. rate	1500MHz	1800MHz
OS	Ubuntu 14.04 64bit	Windows 7 SP1 64bit

Table 4.1: Configuration of used machines

4.2 Code review and profiling

The source code was handed over by Józsa[25] in two separate directories, one for the single precision and one for the double precision source files.

There was an instruction in the *main.cpp* file regarding the compilation of the program and a sample configuration file, with sample node and boundary conditions data file for a lid-driven cavity.

The code was written in C/CUDA. It had a main file (*Iterate.cu*) that held all the computation algorithms and several helper file that took care of the IO operations, memory allocations and the residual computations.

The C programming language was chosen because it is a more low level language which makes it more suitable to write performance oriented hardware related code. It also makes it harder to maintain and to hand over to other developers. However together with the thesis[25], acting as the primary documentation, and with relevant comments in the code it was understandable.

Dealing with legacy code is not an easy task. Several books tell about the issues and their resolution[12], mostly stating that the main problem with legacy code is the missing though process, that helped the program developed in the first place. But it can be also beneficial as the code can be seen from different perspective by another developer.

Although this particular code base can hardly be called a legacy code, code written by others always have some of the characteristics of a legacy code. Having not seen the original C++ programs [2][42] also helped with thinking out of box regarding the improvement of the algorithms of Józsa's code[25].

4.2.1 Profiling

In order to even talk about the speed-up of the method we should have a look what we can start with. First the performance of the inherited GPU program needs to be measured. The data presented in the thesis cannot be used because the code changed slightly since the submission of it. Luckily the program had internal time measurement routines so to measure its speed it is enough to run the program with different parameters.

As can be seen on Figure 4.1 the program has parts that are run only once and there are 6 steps that is run in a loop. Figure 4.2 shows the relative runtime for the steps in the loop. The outer ring shows the time with the residual computation, and the inner ring without it for a better view of the results. Apart from the residual computation that is run on the CPU instead of the GPU the most time consuming part is either the handling of the boundary conditions or the collision model depending on the model chosen, because the other steps use up the same amount of time for all three run. The simplest and thus the fastest model is the BGKW model. The second is the TRT and the most time consuming method is the MRT method that

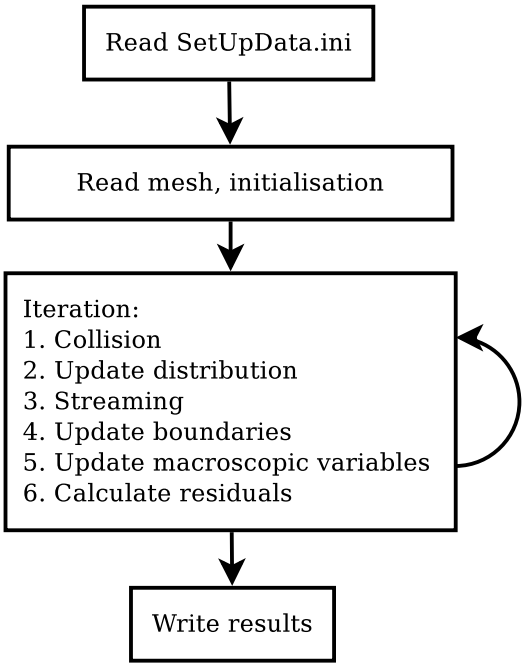


Figure 4.1: Program flowchart

uses matrix multiplications to approximate the collision operator.

Figure 4.3 shows the overall runtime of the algorithm on the expansion geometry using different mesh sizes. The program was run with 5000 iterations, excluding the slowest residual computation step, displayed on a logarithmic scale for better a view of the initialisation step as it is rather small compared to the runtime of the loop.

4.2.2 Data representation

The original code stores data in two kind of "dimension". Most of the geometry data and macroscopic values are stored in node dimension $(m * n)$ arrays, while distribution function and related coefficients are stored in lattice dimension $(9 * m * n)$, where m representing the number of rows and n the number of columns.

All data are stored as row major arrays. Data for the lattices are stored group together in 9 row major arrays, so neighbouring lattices for the same node are store $m * n$ element away from each other.

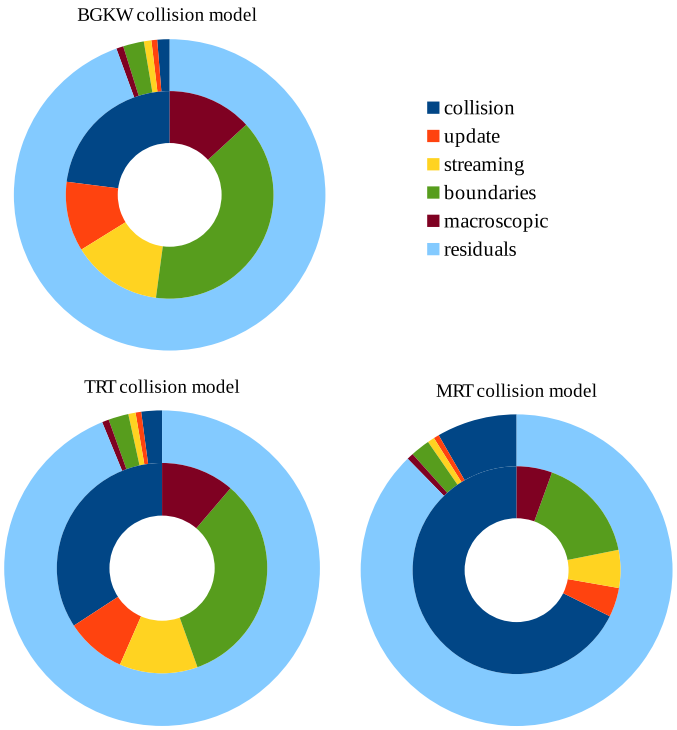


Figure 4.2: Runtime of the different operations

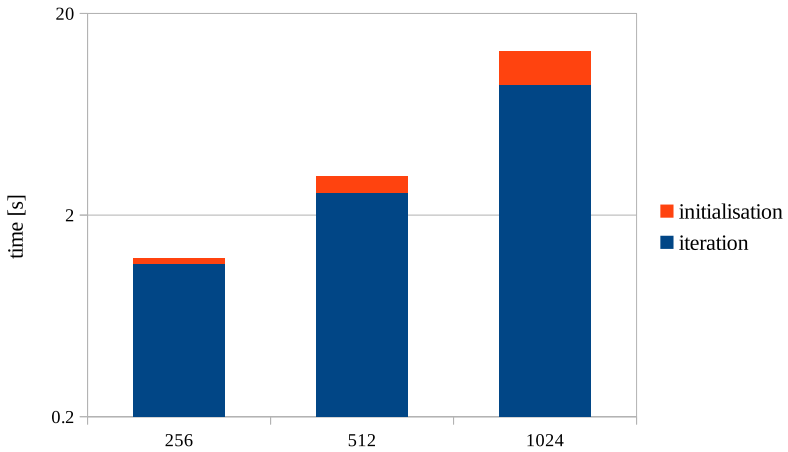


Figure 4.3: Overall runtime of the algorithm on different mesh width

4.2.3 Memory management

Table 4.2 shows the approximate amount of memory used up by the program in regards to the mesh size, with exact values for a 512x512 lid-driven cavity mesh (Fig. 3.9), assuming 4byte long integer and floating point values (single precision). The sizes are calculated from the mesh size.

data	size	size of lid
constants	928 B	928 B
input nodes	$5 * m * n$	5 MB
input BC	$7 * n_{BC}$	168 kB
index arrays	$7 * m * n$	7 MB
BC index arrays	$3 * 9 * m * n$	27 MB
macroscopic values	$7 * m * n$	7 MB
distribution func.	$2 * 9 * m * n$	18 MB
temporary data	$5 * 9 * m * n$	45 MB
summary	$\sim 110 * m * n$	~110 MB

Table 4.2: Memory allocation

The program allocates these amount of memory on both the CPU and the GPU side, and then copies back-and-forth between them during the steps. Table 4.3 shows the amount of memory copied between the CPU and the GPU.

step	size	size of lid
initial copy	$104 * m * n$	104 MB
initialisation	$44 * m * n$	44 MB
residuals	$20 * m * n$	20 MB
autosave	$4 * m * n$	4 MB
final write	$4 * m * n$	4 MB

Table 4.3: Memory copy

Memory for the residuals and autosave steps are in the main loop, but the latter only gets called at given intervals, so our main concern is computing the residuals which is not only a CPU operation it needs data from the GPU at every iteration slowing the code down. A possible solution to the problem is the implementation an effective GPU reduction algorithm (more about it in section 4.3.8).

4.3 Optimisation

After the profiling of the program we could start working on optimisations. For this task we had to make sure that the changes to the code won't effect the outcome, just the performance. In order measure speed-up we used 5000 iterations with the same dimensionless velocity and viscosity values, namely:

- $u = 0.05$
- $v = 0.0$
- $\nu = 0.01$

For the value check at the first version we used visualised images to compare the results, but later a comparing functions was implemented into the program to compare results using matrix norm (section 4.6.3).

4.3.1 Initialisation

The GPU code had several unnecessary arrays and back and forth copying between the device and the host. Elimination of these memory operation did have some favourable effect on the initialisation time, but it was clearly not enough because most of these code was probably already eliminated during compile time, as the C compiler can optimise the code as such that it removes unused variables.

The bigger speed-up was gained during the initialisation of the boundary conditions. The loop that initialised this part was run on the CPU and it looped through all the nodes on the mesh, even though it only had to modify it at the boundaries. Changing the loop to only go through the boundary conditions turned out to be quite successful as the number of boundary elements compared to the whole mesh is significantly smaller.

mesh size	nodes	conditions	b. nodes	ratio
258x15	3870	1610	534	41.6%
514x28	14392	3224	1072	22.4%
1026x54	55404	6446	2146	11.6%

Table 4.4: Loop size comparison on meshes for channel

Table 4.4, 4.5 and 4.6 shows the difference in the loop size. The number of nodes is the dimension of the mesh. The number of conditions are the lattices that fall on a boundary and the last number is the number of nodes that are on the boundary. Because the initialisation was reduced to the number of

```
for i=0..N
  for j=0..M
    for k=0..Nbc
      if Bc(i,j) exists
        for l=0..8
          set bc_id(i,j,l)
          set q(i,j,l)
          set boundary_id(i,j)
```

(a) old behaviour

```
for i=0..Nbc
  set boundary_id
  for j=0..8
    set bc_id(bcx(i),bcy(i),j)
    set q(bcx(i),bcy(i),j)
```

(b) new behaviour

Figure 4.4: Initialisation

mesh size	nodes	conditions	b. nodes	ratio
258x41	10578	1760	584	16.64%
514x79	40606	3530	1174	8.7%
1026x156	160056	7062	2352	4.41%
2050x310	635500	14124	4706	2.22%

Table 4.5: Loop size comparison on meshes for expansion

mesh size	nodes	conditions	b. nodes	ratio
255x256	65280	3032	1008	4.64%
514x515	264710	6140	2044	2.32%
1026x1027	1053702	12284	4092	1.16%

Table 4.6: Loop size comparison on meshes for lid-driven cavity

boundary conditions the last column represent the ratio between the number of nodes and the number of boundary conditions.

Figure 4.5 shows the running time of the initialisation for the old and the new program, for every geometry in the order of channel, expansion and lid-driven cavity. The diagram uses a logarithmic scale so that differences are more visible. Figure 4.6 shows the actual speed-up gained from the change for every geometry. The lines show a quadratic nature because of the changing

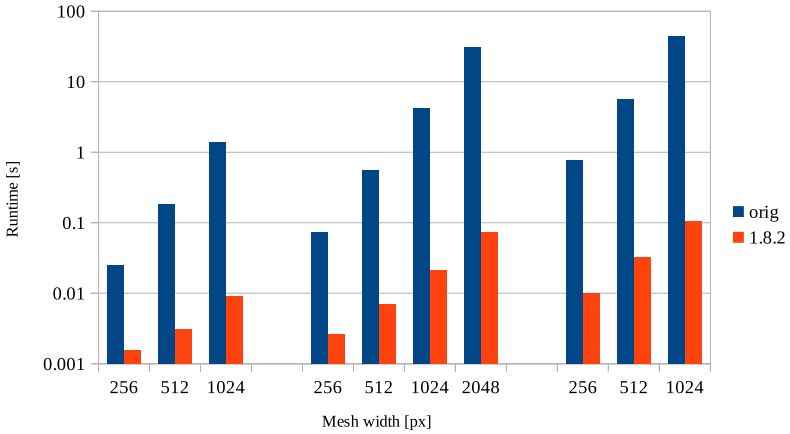


Figure 4.5: Channel, expansion and lid-driven cavity initialisation runtimes

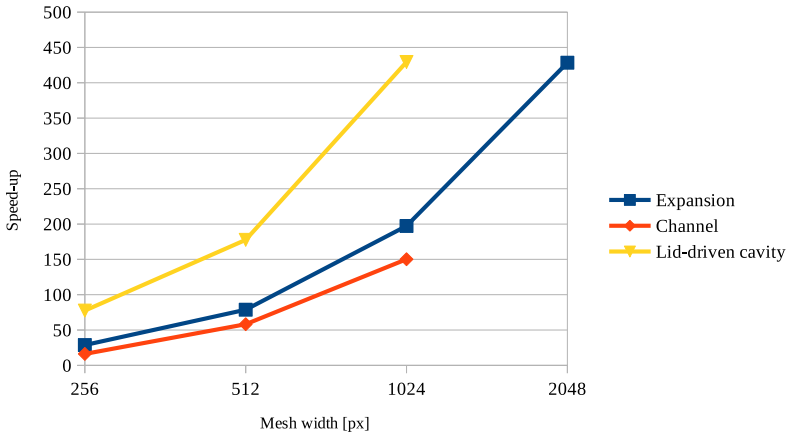


Figure 4.6: Channel, expansion and lid-driven cavity initialisation speed-up

difference between the number of nodes and the boundary lattices.

4.3.2 Combining the update and streaming step

In the update step the computed changes in the previous step gets written into the nodes where fluid is present. In the streaming step the very same nodes get overwritten by their neighbouring values (Figure 4.7). Although these steps are not the most computationally heavy ones, both operation works on almost the same data so they can be combined, successfully eliminating an unnecessary walk-through of the matrix and a kernel initialisation. This change resulted in an average **1.78x** speed-up of the two operations.

```
update_kernel:
  if fluid[i]
    f[i] = fColl[i]
-----------------
streaming_kernel:
  if fluid[i]
    if stream[i]
      f[i] = fColl[neighbour(i)]
```

(a) old code

```
if fluid[i]
  if lattice[i]>0 and stream[i]
    f[i] = fColl[neighbour(i)]
  else
    f[i] = fColl[i]
```

(b) new code

Figure 4.7: Update and streaming step combination

4.3.3 Reducing threads

In the inherited code, all kernels were called with $9*M*N$ amount of threads regardless of the information they were dealing with. Most of the steps don't need that much threads to run. Especially the boundary conditions step (more about it in section 4.3.7). All the other kernels needs more threads but not as much they are currently using now.

We were able to change the kernels to use only $M*N$ amount of threads for every task. If a tasks has to deal with all the 9 lattice directions, a `for` loop was introduced to go through the directions. This provided speed-up for every kernel because it also reduced the number of idle threads.

The results of these changes were not measured separately, because they were introduced together with the next change.

4.3.4 Unfolding kernels

Although the CUDA system enables us to write algorithm just like we would do with normal C/C++ code on the GPU, it doesn't mean that it will work the same as the CPU. For this it is generally considered bad practice to use too much branches and variable length loops in the kernel code. Everything that is known at compile time makes the work of the compiler easier and results in a more optimised program.

By actually writing down every instruction that are done in a loop the threads can synchronise their jobs better, saving time. Luckily there are at most nine instructions in the loops, so we did not have to write hundreds of lines, with this but the speed-up is relevant.

Unfolding macroscopic computation

```
rho[i] = 0.0;
u[i] = 0.0;
v[i] = 0.0;
for (j=0; j<9; ++j)
{
  rho[i] += f[i+j*size];
  u[i] += f[i+j*size]*cx[j];
  v[i] += f[i+j*size]*cy[j];
}
```

(a) old code

```
r = u = v = 0.0;
r = f[i] + f[i+size] + f[i+2*size] + ...
u = f[i+size] - f[i+3*size] + f[i+5*size] + ...
v = f[i+2*size] - f[i+4*size] + f[i+5*size] + ...
```

(b) new code

Figure 4.8: Unfolding the macroscopic kernel

Figure 4.8 shows how computation of the macroscopic values were unfolded. Calculating the density is straightforward, for every node it is the sum of the distribution function over the lattices. Calculating the velocity is quite similar but it only contains their respective elements in that direction. The small arrays (`cx[]` and `cy[]`) are representing this containing `[0,1,-1]` to indicate which lattice matters in that particular direction. The elimination of this array look-up by unfolding the loop, helps the GPU to compute it faster.

Figure 4.9 and 4.10 shows the runtimes and the speed-up of the macroscopic computation step using as a result of the unfolding. As can be seen the change halved the computation time.

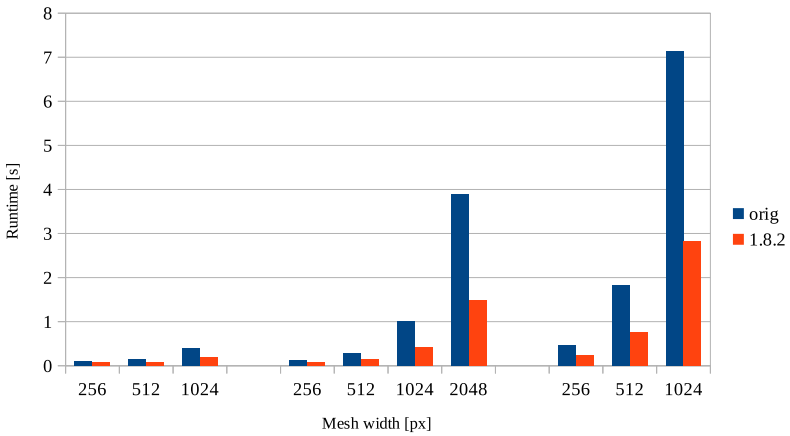


Figure 4.9: Channel, expansion and lid-driven cavity unfolded macroscopic runtimes

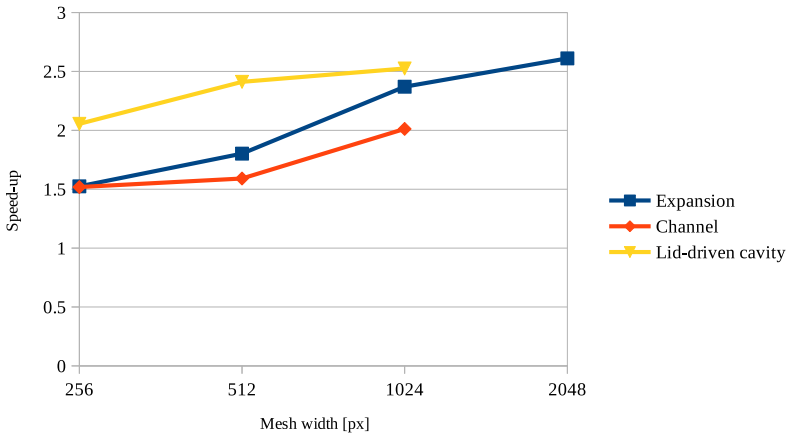


Figure 4.10: Channel, expansion and lid-driven cavity unfolded macroscopic speed-up

Unfolding the BGKW model

It is also beneficial to introduce temporary variables for values that are read from other arrays if they are needed in more than one computation in a given kernel. If we read them every time from the arrays the compiler will consider it as a value that can change during execution, so it will need to reach for it every time in the global memory. By storing them in the threads' local memory at the start of the computation, they can be reached faster.

Similar changes were introduced in the BGKW collision model step (Fig-

```
Feq_d[ind] = Rho_d[ind_s]*w_d[ind_c]*( 1.0 +
  3.0*(U_d[ind_s]*cx_d[ind_c] + V_d[ind_s]*cy_d[ind_c])
+ 4.5*(U_d[ind_s]*cx_d[ind_c] + V_d[ind_s]*cy_d[ind_c])
* (U_d[ind_s]*cx_d[ind_c] + V_d[ind_s]*cy_d[ind_c])
- 1.5*(U_d[ind_s] * U_d[ind_s] + V_d[ind_s]* V_d[ind_s]);

METAF_d[ind] = omega_d*Feq_d[ind]+(1.0-omega_d)*F_d[ind];
```

(a) old code

```
__device__ FLOAT_TYPE feqc(FLOAT_TYPE u, FLOAT_TYPE uc,
           FLOAT_TYPE v, FLOAT_TYPE vc,
           FLOAT_TYPE rho, FLOAT_TYPE w)
{
FLOAT_TYPE vec = u*uc + v*vc;
return rho * w * (1. + 3.*vec + 4.5*vec*vec -
                  1.5*(u*u + v*v));
}

[...]

u = U_d[ind];
v = V_d[ind];
r = Rho_d[ind];
METAF_d[ind]      = omega_d * feqc(u,  0, v,  0, r, 4./9.) +
                    (1.0-omega_d) * F_d[ind];
METAF_d[ind+1*ms] = omega_d * feqc(u,  1, v,  0, r, 1./9.) +
                    (1.0-omega_d) * F_d[ind+1*ms];
...
```

(b) new code

Figure 4.11: Unfolding the BGKW collision model

ure 4.11), where the same directional arrays are needed. To ease readability the computation was refactored into a separate device-only function. By default the CUDA compiler tries to make these functions `inline`, eliminating the overhead of the function call, so do not be afraid to introduce new functions for the sake of readability.

Figure 4.12 and 4.13 display the result of the function refactor and the unfolding in the BGKW model. The change halved the runtime here as well but as can be seen with lid-driven cavity it also neared its maximum speed-

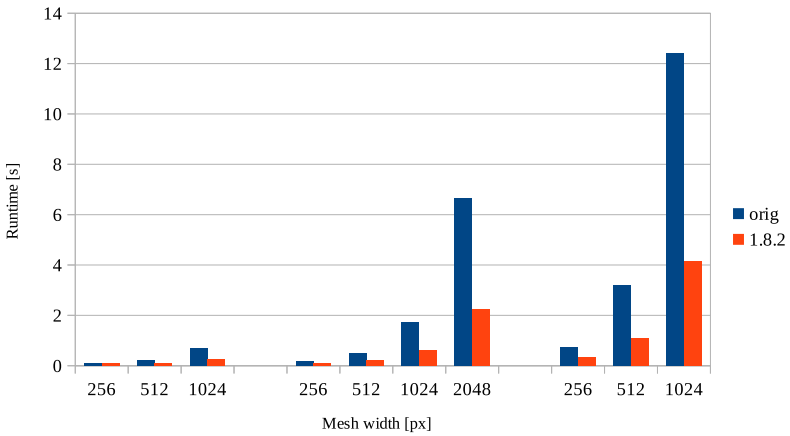


Figure 4.12: Channel, expansion and lid-driven cavity unfolded collision runtimes

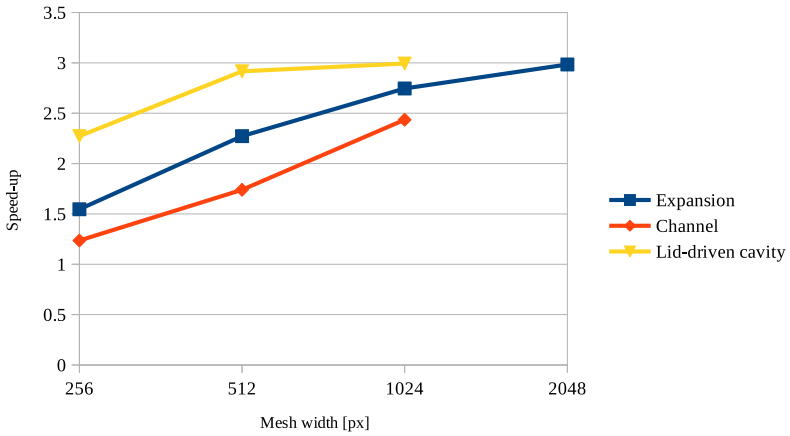


Figure 4.13: Channel, expansion and lid-driven cavity unfolded collision speed-up

up with the 1024x1024 mesh, which could forecast a performance drop with larger meshes.

Unfolding the streaming step

The streaming step also used an external array in constant memory to compute the step in the array to reach the designated lattice in neighbouring nodes. Removing the use of this array and reducing the number of threads from $9 * m * n$ to $m * n$ made possible to get more than 2x speed-up.

As can be seen on 4.14 and 4.15 the speed-up of the significantly larger lid-driven cavity neared its maximum again. It could be due to the fact that this step has the most inter-node data transfer, thus slowing the method down with larger sizes. It would also be a perfect candidate to use texture memory to store the distribution function in this step as it supports better memory access for data that uses lots of neighbouring data in 2D or 3D space.

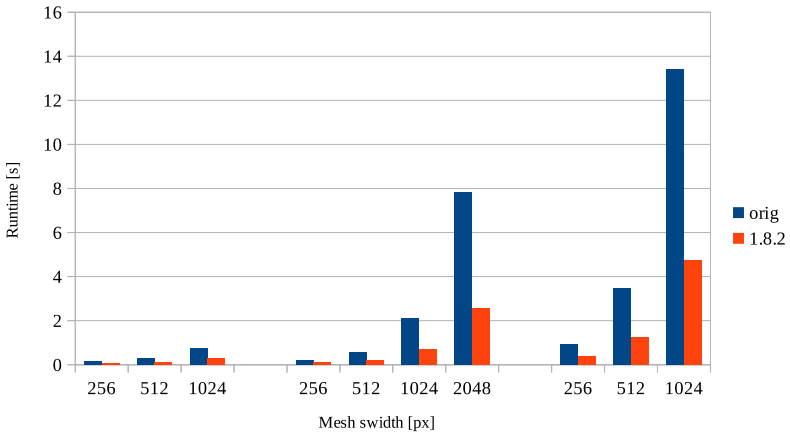


Figure 4.14: Channel, expansion and lid-driven cavity unfolded streaming runtimes

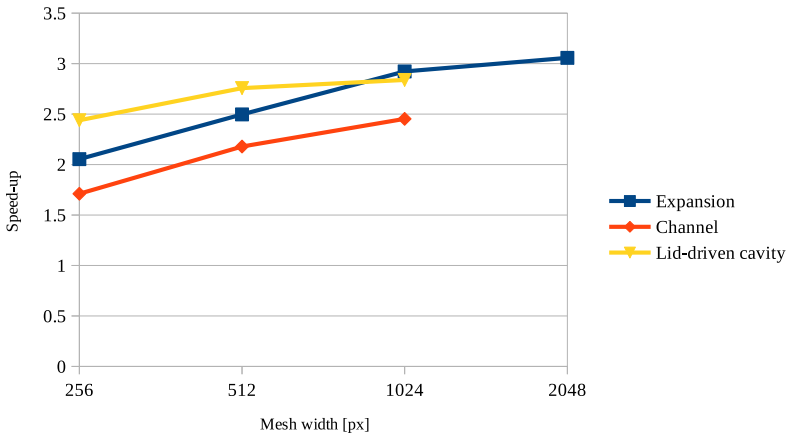


Figure 4.15: Channel, expansion and lid-driven cavity unfolded streaming speed-up

4.3.5 Combining the TRT steps

The TRT collision model is similar to the BGKW model in a way that they use the same formula to compute the equilibrium distribution function, thus it can reuse the same device function that was introduced in the BGKW model. It also helps combining the two parts of the model, because the first part just computed the equilibrium values, and the second part computed the actual distribution function from the macroscopic values. These functions used an intermediary array to transfer data between each other, which also falls out with the combination the steps, saving on memory operations.

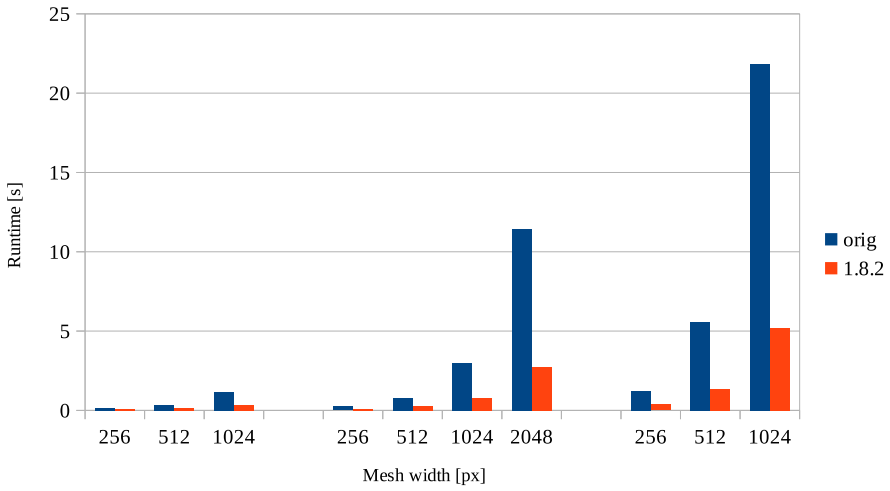


Figure 4.16: Channel, expansion and lid-driven cavity combined TRT runtimes

Figure 4.16 shows the running time of the algorithm with different geometries. The time is clearly dependent on the number of nodes, as it takes significantly longer time to compute for the square-like lid-driven cavity than for the long and narrow channel. Figure 4.17 shows the speed-up gained from the change. For the larger meshes the speed-up is bigger at first but doesn't improve that much with the size, but for the smallest amount of nodes (channel geometry) we could more speed-up with the increase of the size.

4.3.6 Combining the MRT steps

The MRT collision model in the code is divided into two steps and uses 3 full size $(9 * M * N)$ intermediary arrays to store temporary data for the

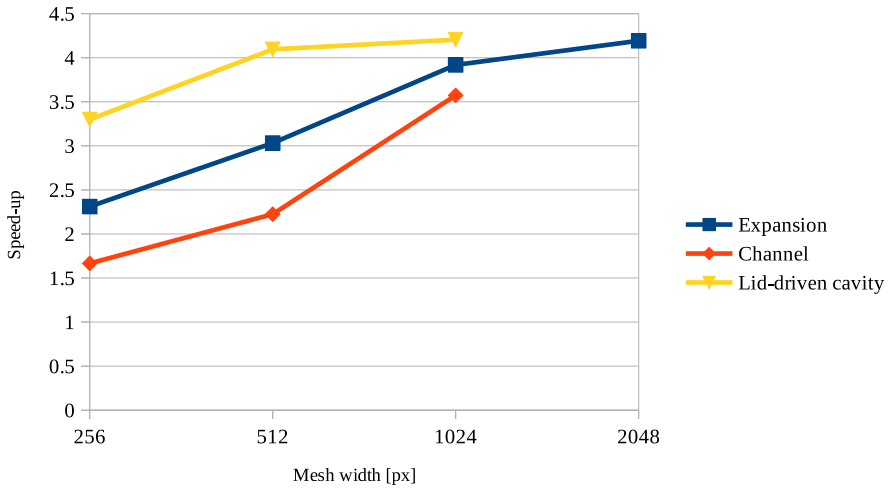


Figure 4.17: Channel, expansion and lid-driven cavity combined TRT speed-up

computation. On closer inspection of the code we realised that the additional arrays can be completely removed if the two steps are combined.

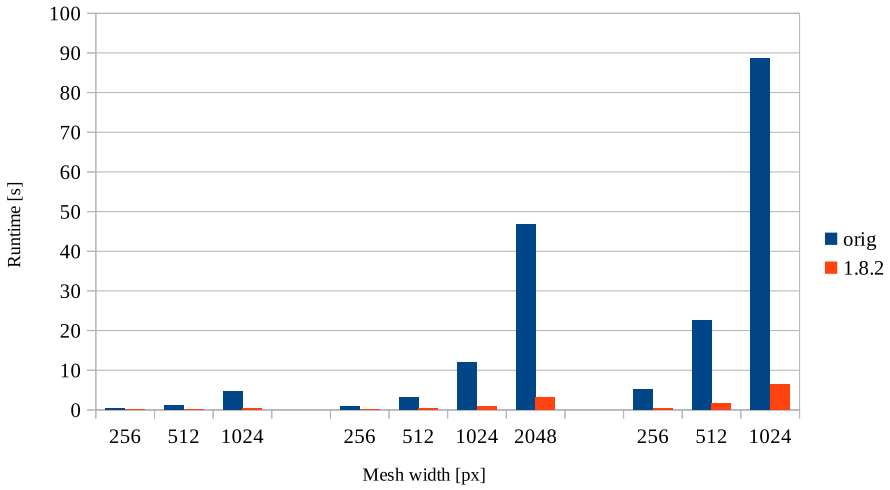


Figure 4.18: Channel, expansion and lid-driven cavity combined MRT runtimes

The method itself goes through all the fluid nodes and compute the probabilities for the next steps from the velocity and density values. It goes through all directions on every step.By combining the two steps and convert-

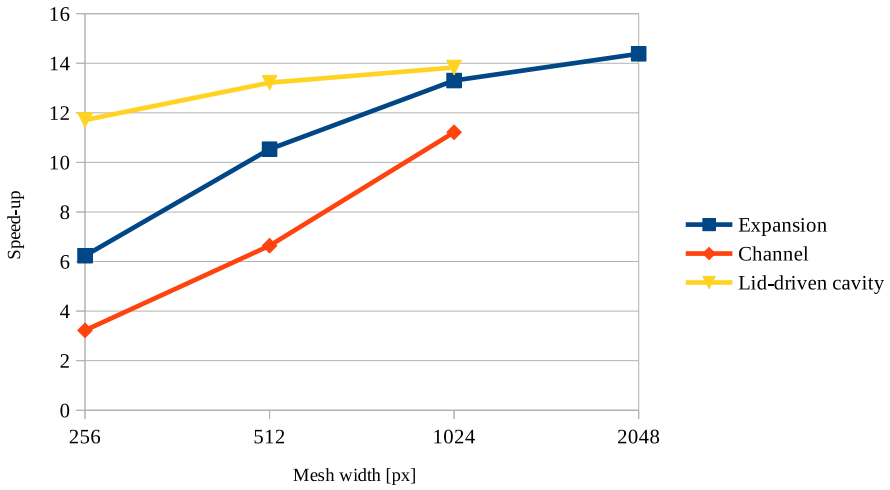


Figure 4.19: Channel, expansion and lid-driven cavity combined MRT speed-up

ing the full size global arrays to local small 9 element arrays we can save a kernel call and several unnecessary global memory read.

The summation of the directions are still done in `for` loops, but these alone could make this approach to take almost the same time as the BGKW method making it a better alternative. Figure 4.18 and 4.19 shows the run-times and speed-up gained with these changes. As could be seen before this collision model is the most computationally heavy because as the others only relied on certain combinations of already computed input values, the MRT method uses matrix multiplications to get the distribution function. We could also see that this model gained the biggest speed-up which is due to the fact that the most power can come out of the GPU if it is heavily used for computation not memory operations.

4.3.7 Rearrange boundary conditions

Investigating the code revealed that the boundary condition computing steps use resources the most wastefully. As explained in the initialisation section (4.3.1) the amount of nodes taking part in the boundary conditions is much less than number of nodes, so storing them in full size arrays and evaluating them with $9 * M * N$ threads is completely unnecessary. These functions also use full size arrays to contain very sparse boolean like data.

Boundary conditions bitmask

To overcome these issues we introduced two new arrays that can store all the necessary information and has even less elements that the actual number of boundary conditions because they are bound to the nodes, not the directions in the nodes saving us a large amount of space and time.

To combine these arrays we used an array for the ids of the nodes taking part in the boundary conditions and another one that stores all the other information in a bitmask fashion. Bitmask is a good and space saving method for storing several properties in one integer. It is not suitable to store text or actual number, but it is very useful for true/false properties or enumerated options.

48	32	16	0
boundary ID	- CF 0	8 7 6 5	4 3 2 1

Table 4.7: Boundary conditions bitmask

Table 4.7 show the structure of the boundary condition bitmask. The first raw is positions of the bits that goes from `0` to `63`. The first 16 bit contains the eight directions of the lattices, numbered from 1 to 8 as can be seen on figure 3.1c. Each lattice takes up 2 bits, which means they can have four values:

- **00**: no boundary condition
- **01**: wall
- **10**: inlet
- **11**: outlet

On the 16-17th bit there is the boundary condition for the node, or the 0th lattice in the middle. The 18th bit represents if the node is fluid or solid. The 19th bit is for corner node indication. The next four bits are not in use at the moment. The last 8 bits can hold the ID of the boundary we want for lift/drag computation. We can define up to 256 boundaries for the computation but it is highly unlikely that we would need that many.

The initialisation step takes care of matching the boundary conditions to the nodes, then checks them for corner conditions. A boundary node is considered cornered if it contains lattices that have different boundary types, like wall and inlet. These nodes and their corner lattice will be considered wall for the rest of the computation, but they also need to be marked as

```
int flyId = 0;
int i,j;
for (i=0; i<n*m; ++i)
{
    if (mask[i])
    {
        bcIdxCollapsed[flyId] = bcIdx[i];
        bcMaskCollapsed[flyId] = bcMask[i];
        for (j=0; j<8; ++j)
        {
            qCollapsed[flyId*8+j] = q[i*8+j];
        }
        flyId++;
    }
}
```

Figure 4.20: Compressing boundary condition arrays

corner, because some of the boundary condition computations treat these nodes differently.

Figure 4.20 shows how the data compressing is done after the initialisation of the boundary conditions. The `mask[]` array is filled with zeros and ones during the previous step where every data is stored in $m * n$ size arrays, signing if there is boundary condition data related to the node or not. Then only nodes with actual data are stored in the collapsed arrays, effectively reducing storage and runtime on the boundary conditions step. We tried to implement this functionality on the GPU as well, as there is working example for it in different C++ template libraries, but we could not overcome some race condition issues, and since the host code is in C not C++ we cannot use the template libraries.

Speed-up

Figure 4.21 shows the time used up by the BC steps altogether (wall, inlet, outlet). The is presented on a logarithmic scale to make is more visible. As mentioned before the previous solution went over all the lattices for these computations, but the new algorithm only walks through the nodes that take part in the boundary. The number differences between the lattices ($9 * m * n$) and the number of boundary nodes can be seen in tables 4.4, 4.5 and 4.6. Obviously since the lid-driven cavity has the largest and most square-shaped mesh it has the smallest ratios between those numbers so it comes as no

surprise that the biggest speed-up is realised with this geometry. The speed-up values can be seen on figure 4.22.

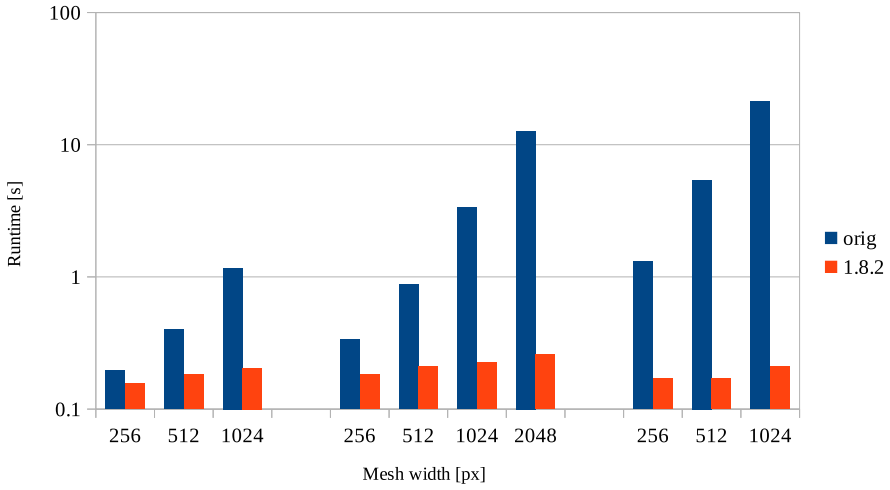


Figure 4.21: Channel, expansion and lid-driven cavity boundary conditions runtimes

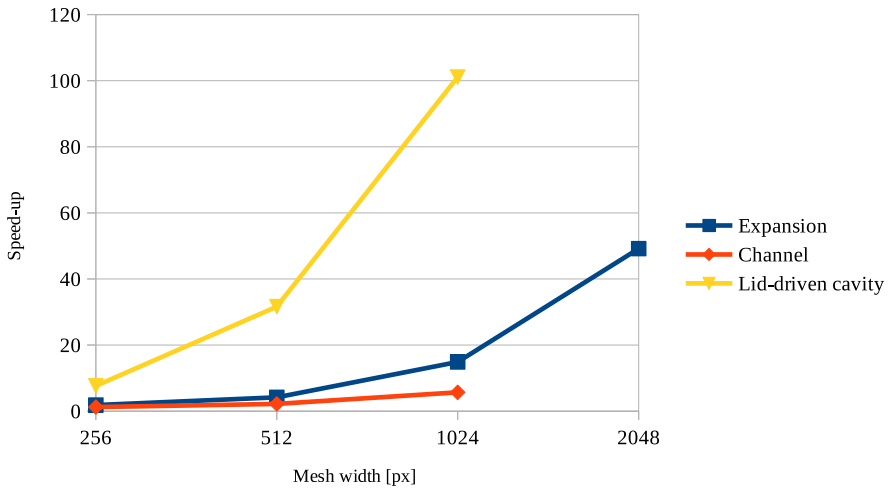


Figure 4.22: Channel, expansion and lid-driven cavity boundary conditions speed-up

4.3.8 Computing residuals on the GPU

The computation of the residuals step consist of a norm computing of the distribution function comparing it to the results in the previous step to have a handle on convergence. It also contains computation of the lift and drag forces alongside the chosen boundary. Both of these steps require the summation of the values in the nodes and lattices. This is called a reduction task, meaning that we need get a single data from large amount of data. These steps are considered the slowest in parallel computing and need the most care and knowledge to implement.

Luckily, for the CUDA platform, a short presentation[16] is available since the introduction of the platform that presents a fast and clever way of dealing with reduction on the GPU by introducing every small programming trick that it can. The example is showed on the summation of an array which is the very task we need to perform under this step.

```
id, threadId, blockId
shared[threadId] = input[id]
sync
for (s=1; s<block; s*=2)
  if (threadId % 2 == 0)
    shared[threadId] += shared[threadId+s];
  sync
if (threadId == 0)
  result[blockId] = shared[0]
```

Figure 4.23: Pseudo code for sum on GPU

The guide start from a very simple parallel version of the reduction (figure 4.23). Every thread stores an element from the array in the shared memory then synchronise. Then in the loop every thread tries to work on two elements resulting in one, so they use less and less resources (in our case threads) with every step and stores the results in place of one of the input elements. This approach is almost slower than running a serial version on the CPU as it leads to divergent branching on the GPU meaning that the threads running in parallel have to do completely different things which is very inefficient.

Changing the stride (step size between elements) helps creating non-divergent branches, but as the algorithm is heavily reliant on shared memory, this approach can lead to bank conflicts. Bank conflicts happen when many threads try to access the same element in shared memory, slowing its access time. Introducing sequential addressing, so that resulting elements are

```
for (s=block/2; s>0; s>>=1)
  if (threadId < s)
    shared[threadId] += shared[threadId+s];
  sync
```

Figure 4.24: Pseudo code for sequential addressing

stored after one other, rather than spread around the array eliminates bank conflicts (figure 4.24).

```
shared[threadId] = input[id] + input[id+block]
```

Figure 4.25: Pseudo code for first reduction during load

In this solution every thread stores a value in shared memory but only half of them starts working on reducing the elements. By storing the first reduction in shared memory instead of the input elements we can start with half the threads but all of them will do some computation leading to better efficiency (figure 4.25). The rest of the guide contains optimisation steps like completely unrolling the inner loop and using templates to make it work with variable thread numbers. Unfortunately not all steps of the guide could be realised because either the change of the platform or the compilers since the publishing of the guide. The presented solution also contained usage of C++ templates, which we wanted to avoid since ours is a C/CUDA solution, so the presented algorithm was only partially implemented.

Figure 4.26 and 4.27 shows the runtime and speed-up of the changes made to the residual computation. The runtime chart is again presented with logarithmic scale to make the difference more visible. As can be seen changing serial CPU implementation to a GPU one proved to be quite successful even though it is a reduction task. The smaller geometries (channel and expansion) benefited nicely from the speed-up but the lid-driven cavity shows some decline in performance as the number of nodes growing probably to fact that the GPU reduction needs CPU management as well, because larger array cannot be reduced in one go, as all blocks produce one result element. More element means more blocks and more run of the summation algorithm on GPU resulting in more CPU-GPU memory transfer as well.

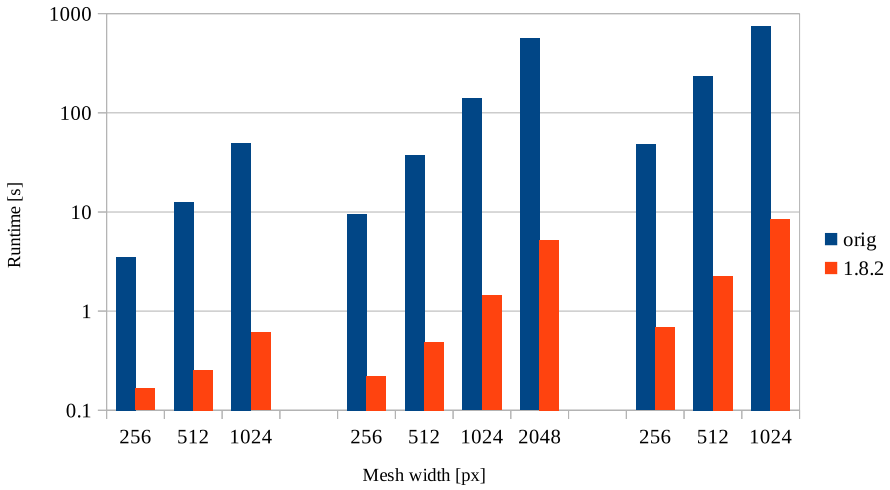


Figure 4.26: Channel, expansion and lid-driven cavity residual computation runtimes

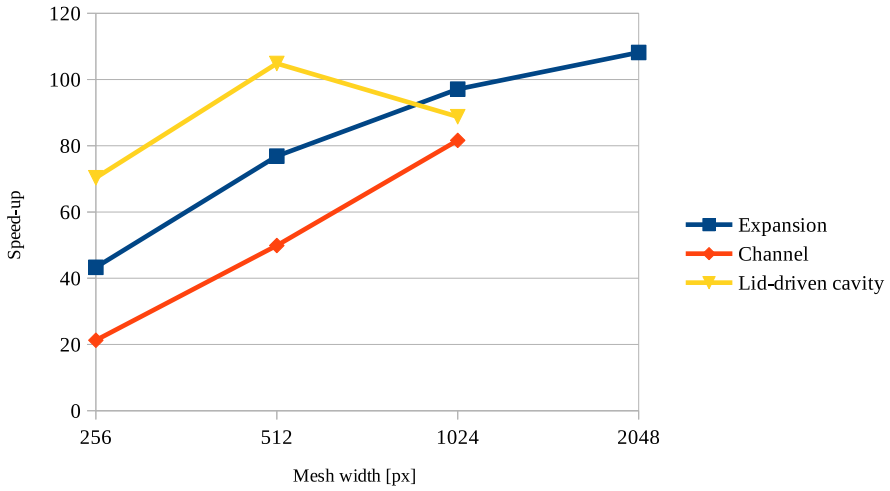


Figure 4.27: Channel, expansion and lid-driven cavity residual computation speed-up

4.3.9 Splitting the residual and the lift/drag computation

The residual computation step contains a norm computing for the distribution function's matrix to follow up on the changes during the steps and inspect convergence. It also computes the lift and drag force on the chosen

boundary, however if we don't need that information we should not spend time on it. That is why the two steps was separated, so if the configuration does not indicate lift/drag computation the program will only compute the norm.

Another change here concerns IO operations. Previously every iteration printed out the residuals into a file, meaning that every iteration spent a small but unnecessary amount time writing to files. By introducing an array to contain these values we can deal with the output at the end of the programs run. The only drawback of this approach is that the program will consume more memory, since it will need an array for the residuals in the size of the iterations which can be quite big for longer runs.

This change is merely a small feature added to the program, to save some time if some results are not needed, it has just a slight effect on runtimes and performance, as computing the residuals is the harder part because it involves a 9 times larger array, subtraction and a square operation while computing the lift/drag forces is just a summation of the arrays. That is why this change can gain around **2.3x** on small geometries and only around **1.3x** on large ones.

4.4 Other changes

4.4.1 Makefile

The very first addition to the code was a Makefile. In *nix environments it is a commonly used tool to manage compilation and linking of the source code into libraries or executable. With proper scripting Makefile also makes other program making task easier e.g. running tests, generating documentation, handling installation.

For this project the following Makefile targets were added. To enable compiling the program with single and double precision floating point capability, the flag `FLOAT_TYPE=USE_SINGLE` or `FLOAT_TYPE=USE_DOUBLE` can be added to the `make` command.

- **debug:** compiles the source files with debug symbol that are essential for finding bugs in the code
- **release:** compiles the whole project without intermediary object files (more about it in section 4.4.5)
- **test:** compiles the code and runs the unit tests (more about it in section 4.6.1)

Example:

```
$ make FLOAT_TYPE=USE_DOUBLE release
```

4.4.2 Compiler optimisation

If we are building an application with heavy emphasis on speed and performance, we will definitely run into issues with compiler optimisations. The problem with such functionality is that it is kind of an unknown variable. Sometimes it speeds things up, but other times it can lead to decreased performance as well. There are good practices on how to write fast code, that takes memory operations into consideration, but some of these techniques are also included in the compilers, we just don't know which ones, until we read their extensive documentation.

We tried some good practices to speed up our project, but it is very likely that we haven't thought of all of them. That is why we also tried turning on compiler optimisations and see if that helped. We used the `nvcc` provided `-O3` option to maximise optimisation. The results showed that it could not improve our program performance-wise, but it remained in the Makefile in case it helps later, either by a new compiler, or by improved functions.

4.4.3 Floating point precision

The original code contained two different directories for the single and double precision floating point computations. In C++ this problem could be easily overcome by using templates which enable defining the same functions for different data types, and compiling the ones that are actually used. Unfortunately C does not have this functionality so more ancient method is used: preprocessor directives. By defining the `USE_SINGLE` or `USE_DOUBLE` keywords at compile time, the C preprocessor can substitute the `FLOAT_TYPE` keyword inside the code with the proper floating point type. As can be read above these keyword can also be defined in the `make` command for easier compilation.

In the "CPU world" double precision is much more used and common than in GPUs where it only appeared in the last decade. Since graphical operations and transformations don't rely too much on double precision it was not really needed until more advanced effects and the introduction of GPGPU computing.

For CUDA capable devices double precision is an extra that only works on a subset of scalar processors on a given device, so algorithms using double precision is expected to have lessened performance than single precision.

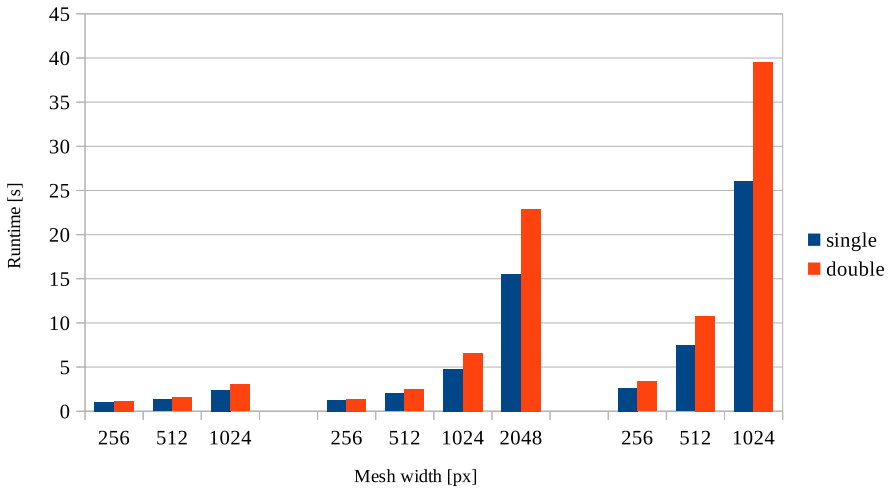


Figure 4.28: Channel, expansion and lid-driven cavity single/double overall runtimes

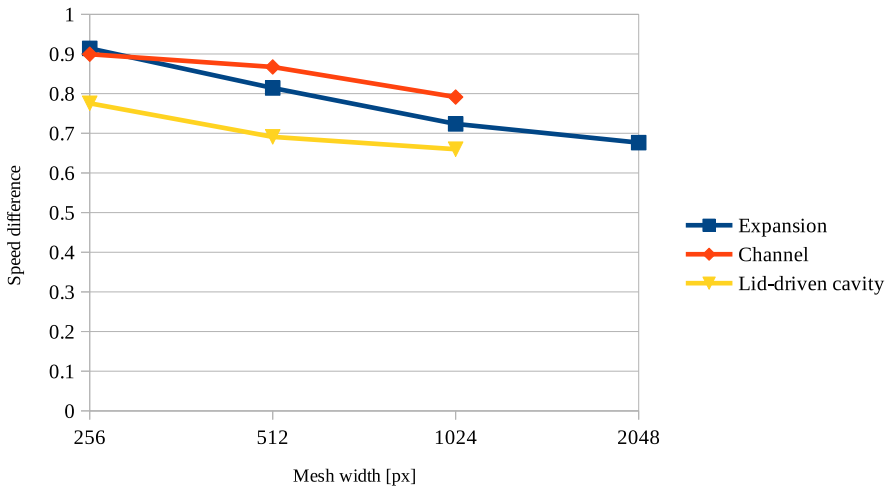


Figure 4.29: Channel, expansion and lid-driven cavity single/double speed difference

Figure 4.28 shows the running times of the whole program with single and double precision. The performance difference can be clearly seen on figure 4.29 as well, showing that higher precision has a cost, so if the computation doesn't require higher precision it is more economical to run in single precision.

4.4.4 Command line arguments

For easier usage command line arguments were added to control various expects of the program that include all the parameters that can be also defined in the initialisation file and some new command like running tests, comparing results etc. For this purpose the argtable2[17] open-source[1] library was used.

The full list of commands:

```
$ ./lbmsolver -h
Usage: ./lbmsolver  [-ht] [-f <file>] [-n <file>] [-b <file>]
                    [-o <file>] [-u <u>] [-v <v>] [-r <rho>]
                    [-s <nu>] [--inlet=[yes|no|pulsatile]]
                    [-c [BGKW|TRT|MRT]] [--curved]
                    [-l [yes|second|first]] [-i <N>]
                    [--every=<N>] [--after=<N>]
                    [--format=[paraview|tecplot]] [-d <id>]
Usage: ./lbmsolver  compare <file> [<file>]

-h, --help                      Print help options
-f, --initfile=<file>           Initialisation from file
-t, --test                      Run unit tests
-n, --node=<file>               Node file
-b, --bc=<file>                 Boundary conditions file
-o, --output=<file>             Output directory
-u, --uavg=<u>                  mean U (x velocity)
-v, --vavg=<v>                  mean V (y velocity)
-r, --rho=<rho>                 Density
-s, --viscosity=<nu>            Viscosity
--inlet=[yes|no|pulsatile]      Inlet profile (default: no)
-c, --collision=[BGKW|TRT|MRT]  Collision model (default: BGKW)
--curved                        Curved boundaries
-l, --outlet=[yes|second|first] Outlet profile (default: second)
-i, --iter=<N>                  Number of iterations
--every=<N>                     Autosave every <N> iterations
--after=<N>                     Autosave after the <N>th iteration
--format=[paraview|tecplot]     Output format (default: paraview)
-d, --draglift=<id>             Calculate drag/lift on <id>
```

[1] GNU Library General Public License

4.4.5 File separation

Reading and maintaining source files larger than 2-300 lines is hard. It gets easier over time, because of the brain's ability of pattern recognition, meaning that after a time one can recognize parts of code from their looks, like reading a map, but it doesn't become maintainable. This is a problem more related to group projects, but it should still be taken into consideration.

For this reason the main source file (*Iteration.cu*) was split into several smaller parts encapsulating the different operations of the main loop. To do this a new compiler flag was introduced in CUDA SDK 5.0 that enables the compiler to create and link separate object files with CUDA code in them, but it comes with a cost in runtime. So while it is easier to handle development-wise it is slower. A workaround to the problem is adding a shell script that concatenates the source files and a release target in the Makefile to call this script and create the executable without creating object files. Of course this way the whole project needs to be compiled again if a single file changes, but this feature is only needed for release versions.

In the beginning of the optimisations of this project this change could meant around 1.15x speed-up because of the more compact code and the lack of library linking, but as the code evolved and more and more function became faster this change meant less and less on overall performance. As of the last version of the code (v1.8.2) compiling the program using this technique, together with the `-O3` compiler optimisation only resulted in an average of **1.019x** speed-up, which is quite insignificant but measurable.

4.4.6 Windows compatibilty

Although it was never in scope for the program to run in a Windows environment, some unforeseen issues with the university TESLA machine made it necessary to run the code in Windows as well, on the desktop computer (section 4.1). Luckily it did not become a huge task as it depended on very small amount of architecture related libraries. The differences included are directory creation and some macros.

Our external dependencies were also easy to convert, because both of them were implemented using ANSI C. The CuTest framework had only one source file, but the Argtable2 library needed a separate build on Windows. The only difference was that it cannot use the Regex libraries, so such instances needed a small change in the code as well.

In the code this was realised by introducing some preprocessor directives (figure 4.30), that separates the Windows and Linux code. On Windows the nVidia compiler uses the Visual C++ compiler suite to build the host code,

```
#if defined(_WIN32) || defined(_WIN64)
  [...] (Windows code)
#else
  [...] (Linux code)
#endif
```

Figure 4.30: Preprocessor directive for different OS

so some kind of Visual Studio environment is required, but the actual IDE is not used, because to build the program we can use that same Makefile, with similar but slight modifications. In order to do that GNU Make is needed on Windows as well. It can be either the POSIX compliant Cygwin[2] version or the Windows compliant MinGW[3] version.

4.5 Final code

The final version of the code contains all the changes above and it proved to be a successful optimisation. The speed-up is not really comparable to the change between a serial and a parallel algorithm but it is significant, as can be seen on figure 4.32, which were measured using the BGKW collision model. The lid-driven cavity which is the usual benchmark setup for CFD application has the almost maximised its speed-up with the largest dimension. Larger meshes expected to behave more poorly but still much faster than with the previous version. The others still have potential so enlarging their dimension could reveal higher performance difference.

Figure 4.31 shows the overall running times for the original and the newer application. For 5000 iterations which is not quite enough for the lid-driven cavity to reach its equilibrium state, the time difference is significant. For larger dimension we have an average of **30x** speed-up which could mean that a half hour run can be done in one minute.

4.5.1 Memory management

With the introduction of the combined and compressed boundary condition representation and the combination of the collision models we could save up to half the time that was needed for the lid-driven cavity. The ratio can differ for different dimensions, and the fact that we store the residuals until

[2] `https://www.cygwin.com`
[3] `http://www.mingw.org`

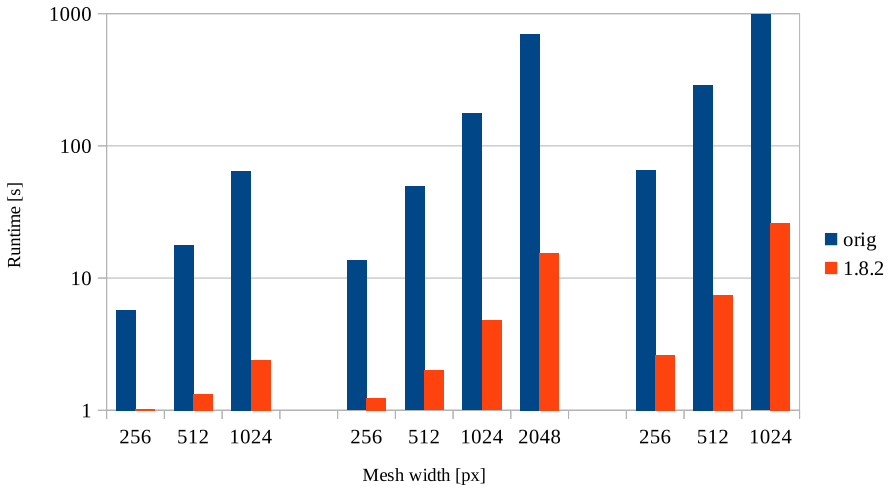


Figure 4.31: Channel, expansion and lid-driven cavity overall runtimes

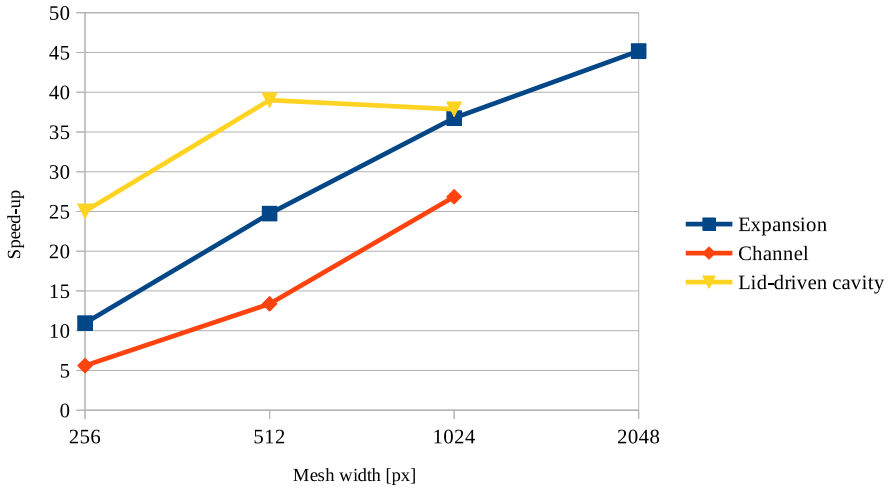


Figure 4.32: Channel, expansion and lid-driven cavity overall speed-up

the end of the program's run, makes it variable. Table 4.8 shows an example with 5000 iterations.

4.5.2 Profiling

With the latest version we should see how the ratios between the different steps changed after optimisations. As can be seen on 4.33 the ratio between the steps are normalised and the residual computation can be fitted into the

data	size	size of lid
constants	233 B	233 B
input nodes	$5 * m * n$	5 MB
input BC	$7 * n_{BC}$	168 kB
index arrays	$3 * m * n$	3 MB
BC arrays	$10 * nBCn$	80kB
macroscopic values	$7 * m * n$	7 MB
distribution func.	$2 * 9 * m * n$	18 MB
temporary data	$2 * 8 * m * n$	16 MB
residuals	$3 * n_{iter}$	60 kB
summary	$\sim 51 * m * n$	∼51.7 MB

Table 4.8: Memory allocation

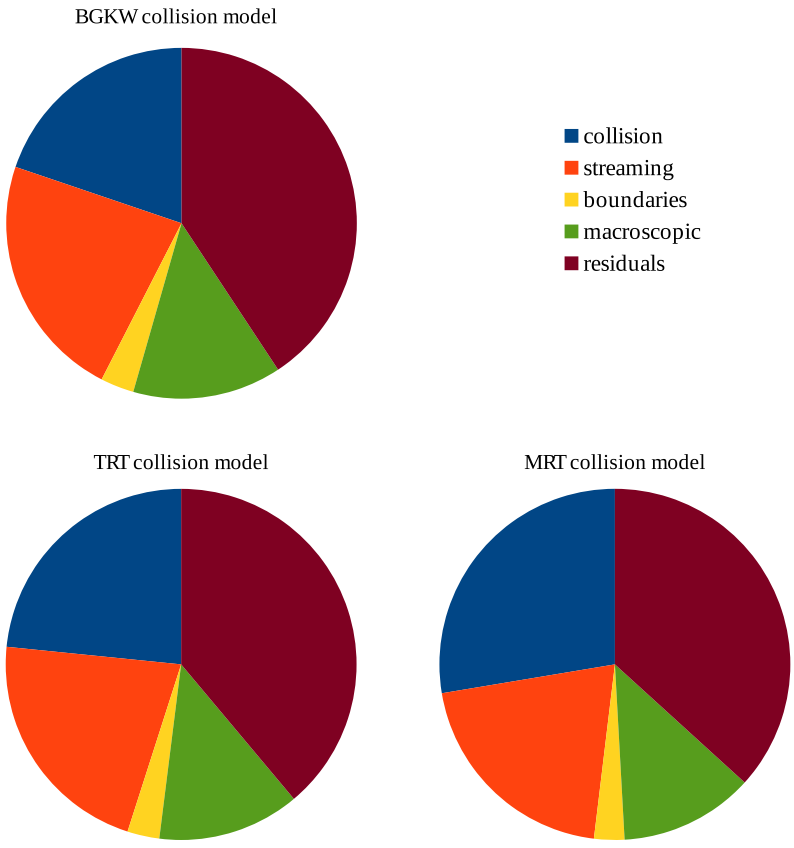


Figure 4.33: Profiling of the final version

chart thanks to the parallelisation. The normalisation can be seen with the collision models. The order of speed is still same between them but they are quite near to each other meaning that it won't have a serious drawback on performance if we chose the computationally more heavy MRT model instead of the BGKW.

4.6 Validation and verification

The previous model[25] was validated and verified extensively against measured and simulated data and it proved to be a good implementation of the lattice Boltzmann method. This work mainly focused on performance improvement and optimisation, and as such it didn't changed the verified and validated model. For this reason it is sufficient to compare this model's results against the previous one for which the built-in comparator (section 4.6.3) and the unit tests (section 4.6.1) are responsible.

4.6.1 Unit tests

Unit tests as their name suggests are for testing the units of the software. Units can be practically any part of the code that is testable in any way, but we usually mean classes and functions by a unit. There are several testing tools for object oriented programming that enable the testing of classes by usually implementing a tester class that calls the objects' methods and check their result against predefined criteria.

C is of course, not an object oriented language so its units should be their functions. For this task we found the CuTest[21] framework to be light-weight enough to easily integrate it into the project. It provides some structures to define test cases and test suits and also provide some checker functions for the basic data types.

By writing unit tests for the existing functions of the program we could ensure that all the optimisation steps will keep their behaviour and their accuracy. It was also helpful to found smaller bugs introduced by other changes. Even if we don't strictly follow the workflow of TDD it is still beneficial to run the unit tests after every change. More advanced continuous integration systems do this automatically every time a new code is introduced to the repository.

The functionality of CuTest was also extended slightly. In test frameworks for object oriented code it is customary to deal with allocations and freeing on the test case level, which is usually implemented by a `setUp()` and a `tearDown()` functions, which are not present in CuTest, so it is the test

```
* make test
cuda-memcheck --leak-check full ./lbmsolver --test
========= CUDA-MEMCHECK

=========== Test GpuComputeResiduals ============
....PASS

=========== Test gpuComputeResidMask ============
....PASS

[...]

====== Test compare gpu_update_* functions ======
....PASS

SUMMARY
.........................

OK (27 tests)

========= LEAK SUMMARY: 0 bytes leaked in 0 allocations
========= ERROR SUMMARY: 0 errors
```

Figure 4.34: Sample output from the unit tests

case's responsibility to clean up after itself. This should not a big problem, but in case of failure of a test, the framework is designed to jump back to the start of the test case, write the failure message and return, without calling the rest of the test case. This could lead to unintentional memory leak. To alleviate this problem a `cleanup()` callback function was added to the test cases that run after the case has ended even if it was a failure.

By introducing this change we were able to run memory leak check (*cuda-memcheck*) together with the unit tests checking leaks made by the actual program. Figure 4.34 shows a sample output of the unit tests. The dots represent the amount of value checks in a case.

4.6.2 Functional testing

Although it helps a lot at development, unit tests are not enough. Especially when the software system consists of several modules that can have any kind of dependency with each other. These combined systems have a

tendency to crash if the compatibility and integration of the components are not tested, because even though a module works perfectly on its own (unit test are passed), it does not guarantee that another component cannot broke it by introducing something that the developers did not think of. To avoid or minimise these problems proper functional and integration testing is essential.

Our software however is simple in a way that it does not need to interact with other components yet, but to ensure good quality several things still need to be tested on the application level. These usually include black or grey box testing, where the testers do not know the underlying implementation, so just run program with different kind of input data and compare the results to the expected values.

Some of these tests can also be automated but they usually contain some manual testing as well. Due to the size of this project we chose the later case. This process needs planning and proper test documentation, for others to use it. For our case the code documentation contains some test cases.

4.6.3 Result comparison

Although it is not strictly fall into the category of functional or unit testing, we implemented a method that can compare the results of the application to the previous versions of the program. By this, we can further ensure that despite the changes the code can produce the same, previously validated results.

```
Comparing results
Reading files.........done
Comparing results...
     array |            diff |      diff/nodes
-------------------------------------------------
    fluids |               0 |               0
         x |               0 |               0
         y |               0 |               0
         u |     0.000172762 |     2.64644e-09
         v |     0.000125096 |     1.91627e-09
   vel_mag |     0.000167395 |     2.56423e-09
       rho |      0.00248409 |     3.80522e-08
  pressure |     0.000834911 |     1.27895e-08
```

Figure 4.35: Sample output from result comparison

Figure 4.35 shows a sample output of the result comparison that checks all 7 arrays that are in the result file. These are coordinate, velocity, density, and pressure values. The program reads both results from file then runs a norm computation on them using the GPU functions we used in the residual computation step, and compare those values. Then prints out the square root of the difference.

Comparison checks were also included into the unit tests to ensure compatibility on the function level, by comparing results from old and new functions, to further guarantee that we are dealing with the same but improved program.

4.6.4 Validation results

Reordering some of the operations in the iteration steps lead to some difference in the results so a thorough check was needed to ensure that introduced numerical difference is just a minor floating point computation error and not some kind of mistake that starts small but grows as the iterations continue. The comparison was carried out on all collision model with the 256 wide lid-driven cavity mesh running 500,000 iterations saving results every 10000 iterations, and on the sudden expansion and the channel doing 200,000 iterations. Then the results were compared to each other with the same norm as in the residual step.

Figure 4.36, 4.37 and 4.38 shows the differences on the channel case with all the collision models. As can be seen the changes in the code did introduce a slight difference, but these values are for the difference between norm of the velocities. Averaging this on the nodes this means only 10^{-10} which is well within the realm of the numerical error of single floating point precision. The rest of the validation cases can be observed in appendix A.

4.7 Documentation

As was already discussed in section 3.6 the code should be written in a way that the code should document itself, but if the development did not start this way one can only clean up things on the go, since there is rarely time for a complete rewrite of the code and is often unwise, as it is probably not worth the risk and the time.

But sometimes a well chosen function or variable is not enough to clear things up for the future readers of the code. It is especially in the case of performance oriented codes, since use little low level tricks to speed things up,

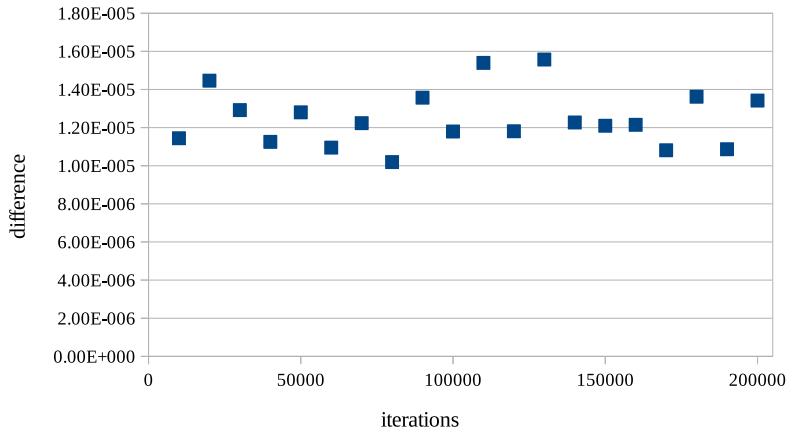


Figure 4.36: Differences in the result for the channel, using BGKW model

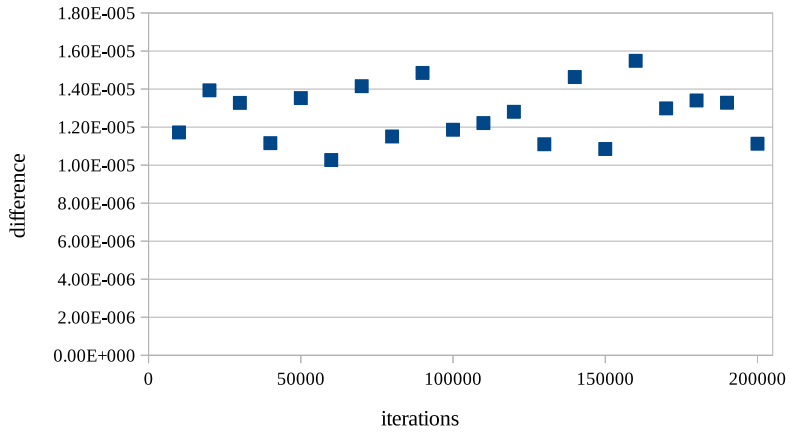


Figure 4.37: Differences in the result for the channel, using TRT model

that might need substantial amount of focus from the reader to understand. For these cases documenting the code will help the readers further.

For this we used the very well known Doxygen framework[43], which is a tool that can generate documentation in various formats from the code itself. All it needs is a special syntax to mark up the code comments. With it all classes and functions can be documented and cross linked making it a valuable document for future developers.

The raw documentation can be found in the code above the function declarations, but it can be also read as HTML pages and as a standalone document thanks to its ability to generate LaTeX files that can be compiled into a PDF document. Such documentation of the code can be read in

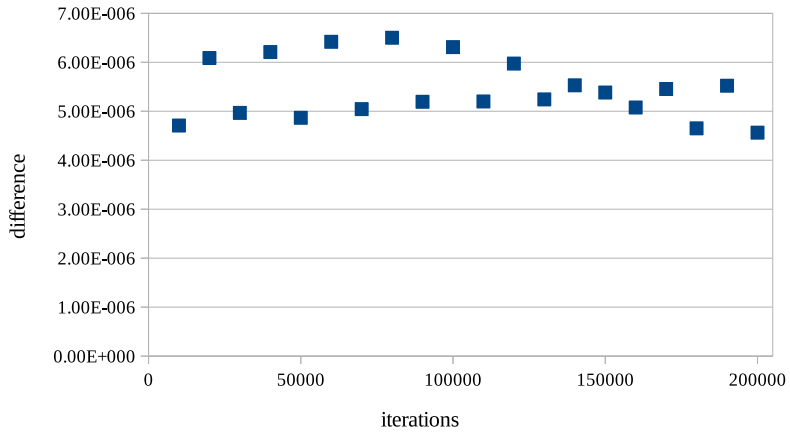


Figure 4.38: Differences in the result for the channel, using MRT model

appendix B. The online documentation is reachable here: `http://public.cranfield.ac.uk/e102081/LBM`.

4.7.1 Doxygen sample

Doxygen recognises more than syntax, like @javadoc[4] so it is easy to use for developers who are coming from Java backgrounds as well. Figure 4.39 shows a sample on how to write comments for a function to document its behaviour.

All the tags (words that starts with @) carry a special meaning that will define the place and layout of the information in the generated documentation. The `@brief` tag specifies a brief description of the function while the `@param` tag describes the parameters of the function. With the help of a LaTeX compiler equations can be added to the documentation as well. Doxygen will compile and convert them to images to display on the generated HTML pages.

Figure 4.40 shows an example on how the page looks with default layout settings. Doxygen is also capable of understanding the mark-down syntax which is usually used by Linux *README* files, that describes the program. Our project also has a *README* file that is included into Doxygen, so it can be read in the project's documentation as well.

[4] `http://www.oracle.com/technetwork/articles/java/index-jsp-135444.html`

```
/**
 * @brief BGKW collision model
 * @details Computation for every lattice :
 * \f[ f^{eq} = \rho w (1 + 3(u c_x + v c_y) +
       \frac{9}{2}(u c_x + v c_y)^2 -
       \frac{3}{2}(u^2 + v^2)) \f]
 * \f[ f_{coll} = \omega f^{eq} + (1-\omega) f^{eq} \f]
 * @todo doc: insert name or reference, explain method
 *
 * @param[in]  Fluid_d fluid condition
 * @param[in]  Rho_d density
 * @param[in]  U_d,V_d velocity vectors (x,y)
 * @param[out] METAF_d distribution function
 * @param[in]  F_d distribution function from boundary step
 * @ingroup solver
 */
__global__
void gpu_bgk_unf(int* Fluid_d, FLOAT_TYPE* Rho_d,
                 FLOAT_TYPE* U_d, FLOAT_TYPE* V_d,
                 FLOAT_TYPE* METAF_d, FLOAT_TYPE* F_d);
```

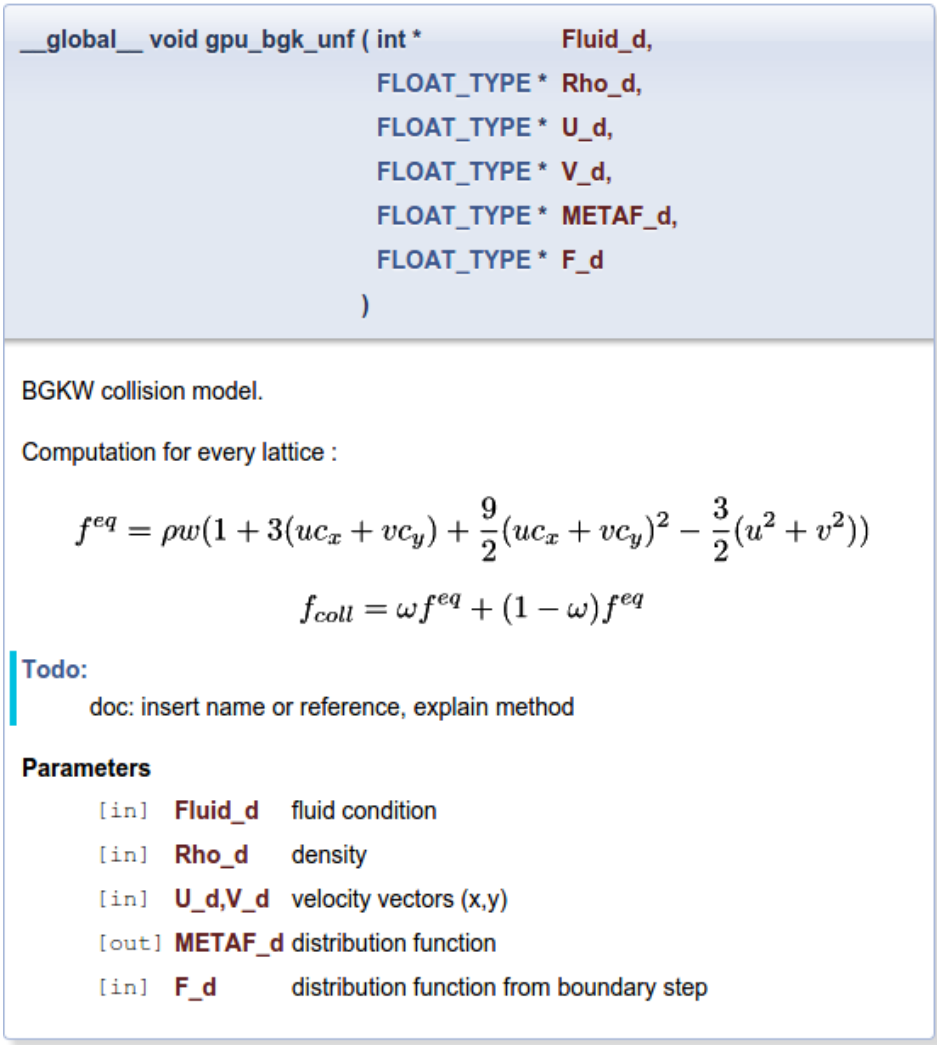
Figure 4.39: Doxygen commenting sample

4.7.2 Documentation structure

As mentioned above the first page of the documentation is the *README* file. It holds information about the building and usage of the program. After that comes the change log that describes how to project changed in the last couple of months. Chapter 3 describes the test cases. They are mostly the documentation of the unit tests. It is indicated if some of tests are intended to run manually. The next chapter is *todo* list, that gathers all the notes in the source that has the `@todo` tag.

Next chapter lists the deprecated functions. These function are mostly the old functions. They could be removed, but they are left there for comparing results and to show from what the new functions evolved. Next comes different indexes, which are a table of contents for the documentation of the functions.

Lastly comes the actual documentation. It is separated into modules, structures and files. As this project has very few defined structure, most of the documentation lies within the file documentation. Doxygen allows to describe several modules, and also support grouping functions into modules.



```
__global__ void gpu_bgk_unf ( int *         Fluid_d,
                              FLOAT_TYPE *  Rho_d,
                              FLOAT_TYPE *  U_d,
                              FLOAT_TYPE *  V_d,
                              FLOAT_TYPE *  METAF_d,
                              FLOAT_TYPE *  F_d
                            )
```

BGKW collision model.

Computation for every lattice :

$$f^{eq} = \rho w(1 + 3(uc_x + vc_y) + \frac{9}{2}(uc_x + vc_y)^2 - \frac{3}{2}(u^2 + v^2))$$

$$f_{coll} = \omega f^{eq} + (1 - \omega) f^{eq}$$

Todo:
doc: insert name or reference, explain method

Parameters

[in]	**Fluid_d**	fluid condition
[in]	**Rho_d**	density
[in]	**U_d,V_d**	velocity vectors (x,y)
[out]	**METAF_d**	distribution function
[in]	**F_d**	distribution function from boundary step

Figure 4.40: Doxygen html documentation sample

Our project has only one module, the solver, which is just the grouping of the functions used during the iterations. At the end of the document there is an index to help the reader find a specific variable or keyword if needed.

Chapter 5

Conclusion

As was mentioned previously this work was aimed at the optimisation of lattice Boltzmann implementation[25] using the power of the GPU. The goal was to improve performance and make the program become a better subject for software development making emphasis on coding and functional quality.

First a literature review was carried out to gather information and the brief history about the methods used along with some examples of what these solutions are capable of. Next we got some history lesson about supercomputer and the evolution of GPU usage and GPGPU computing. As the theories behind the applied methods become clearer the work on the model could start.

The used techniques and methods were discussed and we presented our way of handling the software development issues as well pointing out the differences between science and software development project emphasising the need of collaboration and patience to one another. Because as more and more science project need simulations and heavy computations it is inevitable that the engineers and software developers should work together toward a common goal.

After laying down our methods and our goals we could start working on the lattice Boltzmann implementation. The model was profiled using the built in time measurement techniques and every method was evaluated regarding its memory and processing power usage. Every step of the model had been revised, and improved reaching various amounts of speed-up.

There was a serious effort to make the code fully parallel, by only using the GPU for computation saving memory transfers between the host system (CPU) and the GPU. This effort was partially successful as the initialisation of the data structure used during the iteration is still computed on the CPU, but luckily every step that takes place in the main loop are now fully using the power of the GPU. For every step various CUDA techniques were employed

squeezing more and more power out of the GPU. Some changes were more successful than other, but every step benefited from the improvements.

On the software development side some changes were introduced as well which does not add to the performance but certainly helps with future development of the program. We introduced a revision control system to follow the work done and enable the code to become a team project. Proper building tools were added to ease compiling the code on different systems. Although it is not popular among engineers as they like fixed configuration files better, but in software development it is always a good idea to add command line arguments.

To ensure quality and the same results as the previous model functional and unit tests were included in the project making changing the code easier for future developers as well. By comparing our results with the previous model we can safely say that the same qualities apply for every part of the code as before. None of the methods used changed regarding accuracy and convergence.

During the works a documentation was also formulated using the popular format of writing the documentation inside the source code and generating the documents in various formats. The document version of this work is available in the appendix, while the web page format can be uploaded to any kind of HTML web service for easier access.

5.1 Outlook

This work was focused on performance improvement and cleaner code but the work is far from done. As new techniques emerge earlier practices need to be re-evaluated and revised to improve performance or accuracy or to add more features to the software.

This program could certainly benefit from adding new computational methods and a generalised model for it to be able to become a better and faster tool in the hands of engineers and sciences. For this it is suggested to revise all the algorithms used and choose suitable ones or improve their computational quality.

In order to generalise the model the supporting of differently shaped or even unstructured meshes could widen the program's usage spectrum. Adding 3D model handling could probably be the most important improvement this code could get. But if we stay close to the individual steps in the iteration it is worth looking into the usage of texture memory, especially in the streaming step, because as it was mentioned before it has better access time with data that has some sort of spatiality (like reaching neighbouring

cells). The usage of shared memory in some steps were also investigated but proved to be less of an option, because it is usually useful if threads work on other threads' data, which is not really the case in most of the steps.

It is also worth mentioning that the reintroduction of a C++ host could also be beneficial as it enables newer programming techniques and easier models. It also enables the usage of a vast variety of libraries and tools that could help improving some methods. The main concern against C++ was performance and the lack of full support from the GPU side. But the CUDA environment already supporting most of the language and its compiler mainly regards the GPU code as a C++ code, so the introduction of an object oriented design or at least some template related techniques won't affect the performance that much.

5.2 Final thoughts

In conclusion the work on this project was fun and challenging at the same time. It was fun because pushing the envelope an squeezing more power out of the hardware is always fun and rewarding, and it was a challenge because a lot of theories and methods need to be learnt to comprehend the function and operation of this model. I hope those who come after me would find the opportunity to work on it just as good as I did.

Glossary

API Application Programming Interface. 10–12, 22–24

BC Boundary conditions. 34, 48, 60

BGKW Bhatnagar–Gross–Krook-Welander collision model. 19, 31, 40, 41, 44, 46, 58, 61

CDC Control Data Corporation. 8

CFD Computational Fluid Dynamics. 2, 4, 7, 21, 58

CPU Central Processing Unit. 2, 10, 12, 22, 30, 31, 34, 35, 38, 50, 51, 54, 69

CUDA Compute Unified Device Architecture. 3, 8, 12, 21–24, 41, 50, 54, 69, 71

GPGPU General Purpose Graphical Processing Unit. 7, 12, 54, 69

GPU Graphical Processing Unit. 2, 8, 10–12, 22, 23, 25, 30, 31, 34, 35, 38, 46, 48, 50, 51, 54, 64, 69–71

HPC High performance computing. 8

HTML Hyper-text markup language. 65, 66, 70

IO Input/Output. 31, 53

LBM Lattice Boltzmann Method. 1, 7, 8, 15

MPI Message Passing Interface. 8, 9

MRT Multi-relaxation-time collision model. 17, 31, 44, 46, 61

OS Operating system. 30

PDE Partial differential equation. 20

PDF Portable document format. 65

PGAS Partitioned global address space. 2

SLI Scan line interleave. 11

SPECT Single photon emission computed tomography. 12

TDD Test-driven development. 27, 61

TIA Television Interface Adapter. 10

TRT Two-relaxation-time collision model. 16, 31, 44

UPC Unified parallel C. 2

VESA Video Electronics Standards Association. 10

References

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

[2] Gennaro Abbruzzese. Development of a 2d lattice boltzmann code. Master's thesis, Cranfield University, 2013.

[3] Pietro Asinari, Michele Calì Quaglia, Michael R. von Spakovsky, and Bhavani V. Kasula. Direct numerical calculation of the kinematic tortuosity of reactive mixture flow in the anode layer of solid oxide fuel cells by the lattice boltzmann method. *Journal of Power Sources*, 170, 2007.

[4] A Banari, C Janssen, ST Grilli, and M Krafczyk. Efficient gpgpu implementation of a lattice boltzmann model for multiphase flows with high density ratios. *COMPUTERS & FLUIDS*, 93, 2014.

[5] Pay-Chung Chen Chin-Ho Lee Chin-Cheng Huang, Jen-Sheng Hsieh. Flow analysis and flow-induced vibration evaluation for low-pressure feedwater heater of a nuvlear power plant. *International Journal of pressure vessels and piping*, 85, 2008.

[6] E.T. Coon, M.L. Porter, and Q. Kang. Taxila lbm: A parallel, modular lattice boltzmann framework for simulating pore-scale flow in porous media. *Computational Geosciences*, 18, 2014.

[7] NVIDIA Corporation. Cuda-gdb user manual. `http://docs.nvidia.com/cuda/cuda-gdb`, 2015. online; accessed: 05–07-2015.

[8] NVIDIA Corporation. Cuda-memcheck user manual. `http://docs.nvidia.com/cuda/cuda-memcheck`, 2015. online; accessed: 05–07-2015.

[9] Mohamed Gad el Hak. The fluid mechanincs of microdevices - the freeman scholar lecture. *Journal of Fluids Engineering*, 121, 1999.

[10] D. McAllister ES Larsen. Fast matrix multiplies using graphics hardware. In *The international conference for high performance computing and communications*, 2001.

[11] Kent Beck et al. Agile manifesto. `http://agilemanifesto.org`, 2001. online, accessed: 25–07–2015.

[12] Michael C. Feathers. *Working effectively with Legacy Code*. Prentice Hall, first edition, 2004.

[13] Y.M. Ferng and C.H. Lin. Investigation of appropriate mesh size and solid angle number for cfd simulating the characteristics of pool fires with experiments assessment. *Nuclear Engineering and Design*, 240, 2010.

[14] Irina Ginzburg. Equilibrium-type and link-type lattice boltzmann models for generic advection and anisotropic-dispersion equation. *Advances in Water Resources*, 28, 2005.

[15] Prometeus GmbH. Top 500 supercomputer sites. `http://top500.org`, 2014. online; accessed: 15–06–2015.

[16] Mark Harris. Optimising parallel reduction in cuda. `http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf`, 2007. online; accessed: 05–07–2015.

[17] Stewart Heitmann. Argtable: Ansi c command line parser. `http://argtable.sourceforge.net`, 1998. online; accessed: 05–07–2015.

[18] Abdollah Ardeshir Seyyed Hadi Hosseini Hojat Karami, Hossein Basser. Verification of numerical study of scour around spur dikes using experimental data. *Water and Environment Journal*, 28.1, 2014.

[19] Keng D. Hseuh, Sanjay Abhyankar, Sanjeeva Addala, Anant Kamat, Chun Wu, and Mike P. Haffey Road NVH Dept. Application of computational fluid dynamics (cfd) to automobile wind noise source minimization. *Journal of the Acoustical Society of America*, 97, 1995.

[20] Takaji Inamuro, Nobuharu Konishi, and Fumimaru Ogino. A galilean invariant model of the lattice boltzmann method for multiphase fluid flows using free-energy approach. *Computer Physics Communications*, 129, 2000.

[21] Asim Jalis. Cutest: C unit testing framework. `http://cutest.sourceforge.net`, 2003. online; accessed: 05–07–2015.

[22] Bohumir Jelinek, Mohsen Eshraghi, Sergio Felicelli, and John F. Peters. Large-scale parallel lattice boltzmann–cellular automaton model of two-dimensional dendritic growth. *Computer Physics Communications*, 185, 2014.

[23] Gérard Degrez Erik Dick-Roger Grundmann Jan Vierendeels John D. Anderson Jr, Joris Degroote. *Computational Fluid Dynamics*. Springer, third edition, 2009.

[24] Bin Li Tianfang Li Zhengrong Liang Junhai Wen, Zigang Wang.

[25] Tamás István Józsa. Parallelization of lattice boltzmann method using cuda platform. Master's thesis, Cranfield University, 2014.

[26] S Kikuchi, N Yamasaki, and A Yamagata. Effect of a curved duct upstream on performance of small centrifugal compressors for automobile turbochargers. *JOURNAL OF THERMAL SCIENCE*, 22, 2013.

[27] O. Klenov and A. Noskov. Computational fluid dynamics in the development of catalytic reactors. *Catalysis in Industry*, 3, 2011.

[28] Qinjian Li, Chengwen Zhong, Kai Li, Guangyong Zhang, Xiaowei Lu, Qing Zhang, Kaiyong Zhao, and Xiaowen Chu. A parallel lattice boltzmann method for large eddy simulation on multiple gpus. *Computing*, 96, 2014.

[29] Chuguang Zheng Liang Wang, Zhaoli Guo. Multi relaxation time lattice boltzmann model for axisymmetric flows. *Computers and Fluids*, 39, 2010.

[30] D Lycett-Brown and KH Luo. Multiphase cascaded lattice boltzmann method. *COMPUTERS & MATHEMATICS WITH APPLICATIONS*, 67, 2014.

[31] R. Strzodka M. Rumpf. Nonlinear diffusion in graphics hardware. In *Proceedings of EG/IEEE TCVG Symposium on Visualization*, 2001.

[32] Robert C. Martin. *Clean code, A handbook of agile software craftsmanship*. Prentice Hall, first edition, 2009.

[33] John Brant William Opdyke Don Roberts Martin Fowler, Kent Beck. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[34] M. Krook P.L. Bhatnagar, E.P. Gross. A model for collision processes in gases. i. small amplitude processes in charged and neutral one-component systems. *Physical Review*, 94, 1954.

[35] Xiaoyi He Qisu Zou. On pressure and velocity boundary conditions for the lattice boltzmann bgk model. *Physics of Fluids*, 9, 1997.

[36] Bohui Pang Sherong Zhang and Gaohui Wang. A new formula based on computational fluid dynamics for estimating maximum depth of scour by jets from overflow dams. *Journal of Hydroinformatics*, 16.5, 2014.

[37] Graham Singer. The history of the modern graphics processor. `http://www.techspot.com/article/650-history-of-the-gpu`, 2013. online; accessed: 30–05–2015.

[38] Robert Bennett Gordon Smith John Spiletic Stanimire Tomov, Michael McGuigan. Benchmarking and implementation of probability-based simulations on programmable graphics cards. *Computers and graphics*, 29(1), 2005.

[39] Sauro Succi. *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond*. Oxford University Press, 2001.

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

[41] R. Quinn TAG Smyth. The role of computational fluid dynamics in understanding shipwreck site formation processes. *Journal of Archaeological Science*, 45, 2014.

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

[43] Dimitri van Heesch. Doxygen documentation generation tool. `http://www.stack.nl/~dimitri/doxygen/index.html`, 1997. online, accessed: 19–07–2015.

[44] D.C. Visser, M. Houkema, N.B. Siccama, and E.M.J. Komen. Validation of a fluent cfd model for hydrogen distribution in a containment. *Nuclear Engineering and Design*, 245, 2012.

[45] K.C.S. Kwok M.M. Liu Y. Zhang, M. Zhao. Cfd-dem analysis of the onset scour around subsea pipelines. *Applied Mathematical Modelling*, November, 2014.

[46] Zhifeng Yan and Markus Hilpert. A multiple-relaxation-time lattice-boltzmann model for bacterial chemotaxis: Effects of initial concentration, diffusion, and hydrodynamic dispersion on traveling bacterial bands. *Bulletin of Mathematical Biology*, 76, 2014.

[47] Yu Ye, Kenli Li, Yan Wang, and Tan Deng. Parallel computation of entropic lattice boltzmann method on hybrid cpu–gpu accelerated system. *Computers and Fluids*, 110, 2015.

Appendix A

Validation results

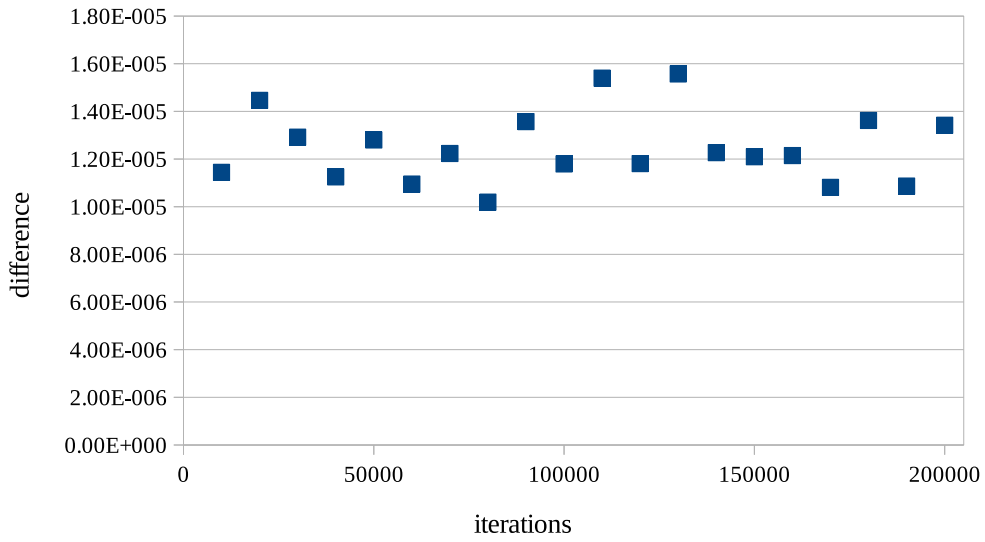


Figure A.1: Channel flow, BGKW model, difference

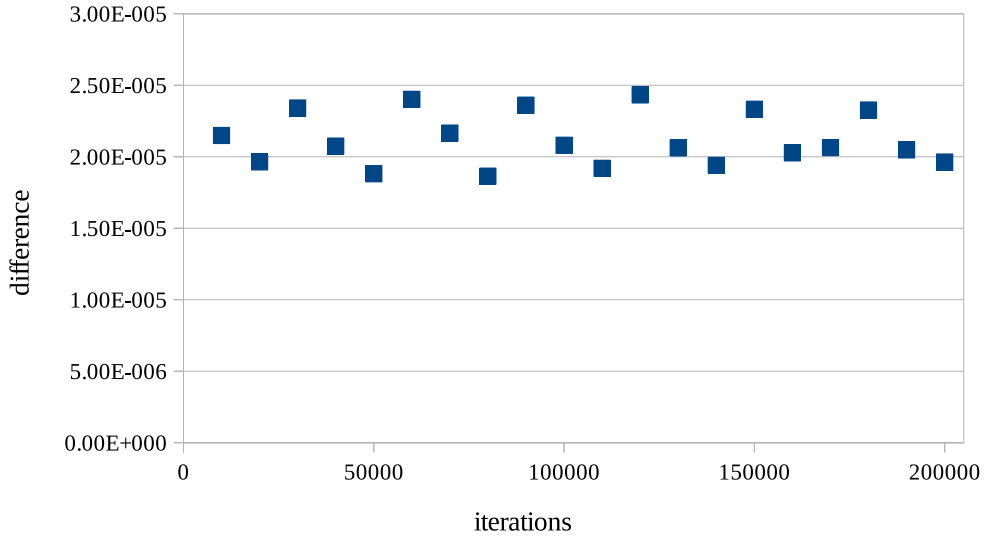


Figure A.2: Sudden expansion, BGKW model, difference

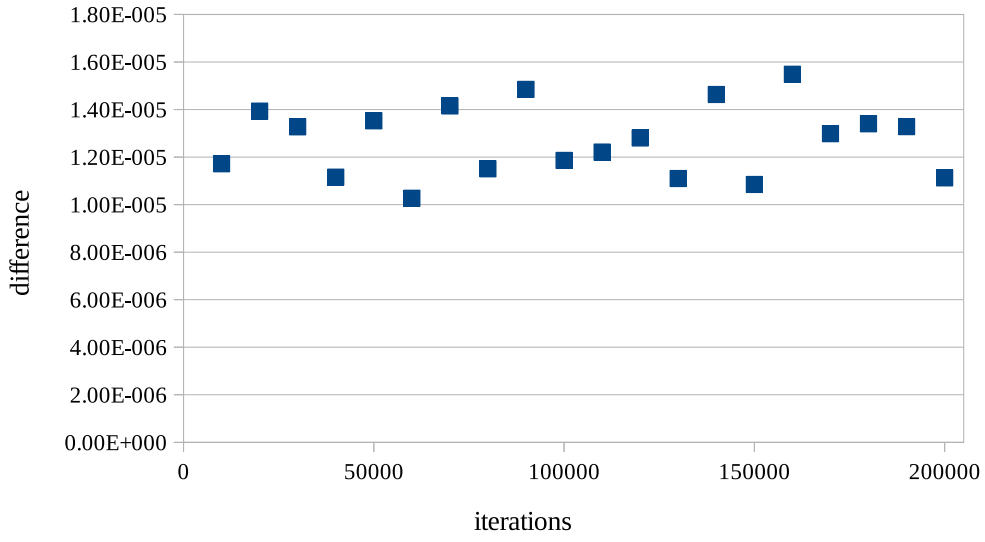


Figure A.3: Channel flow, TRT model, difference

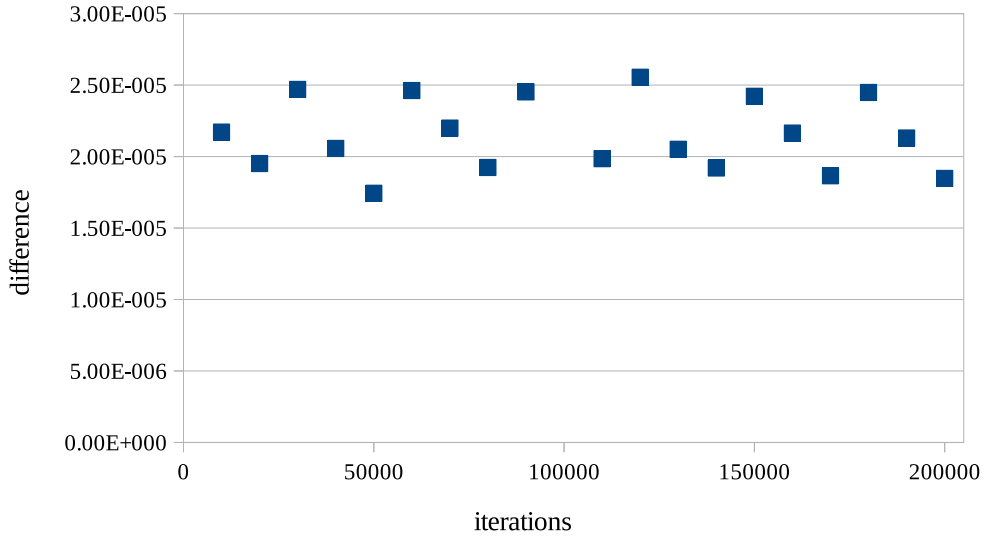


Figure A.4: Sudden expansion, TRT model, difference

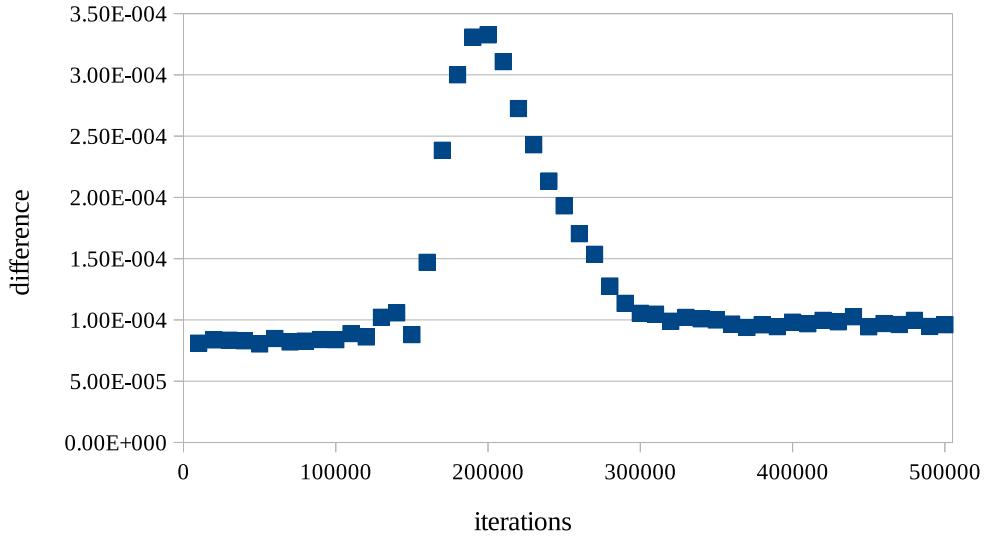


Figure A.5: Lid-driven cavity, TRT model, difference

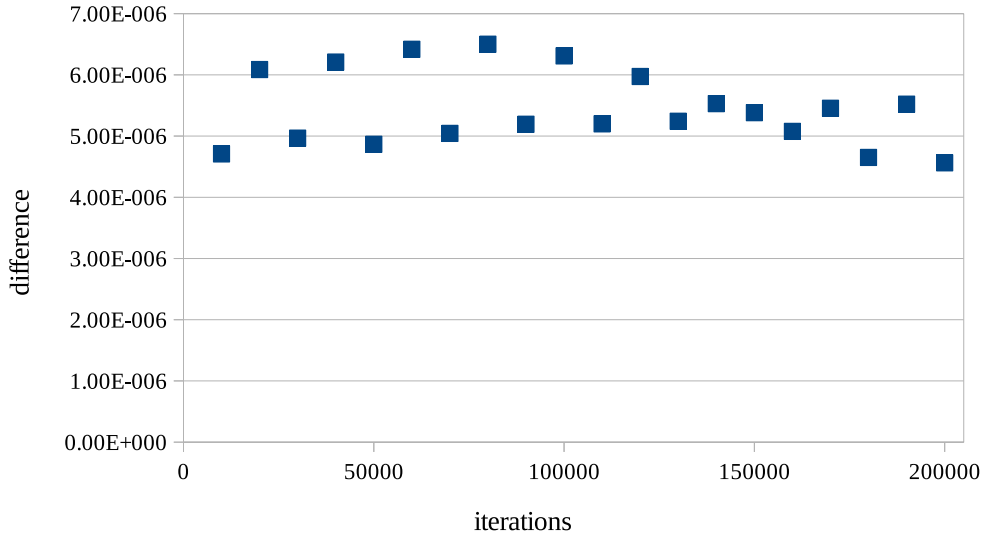


Figure A.6: Channel flow, MRT model, difference

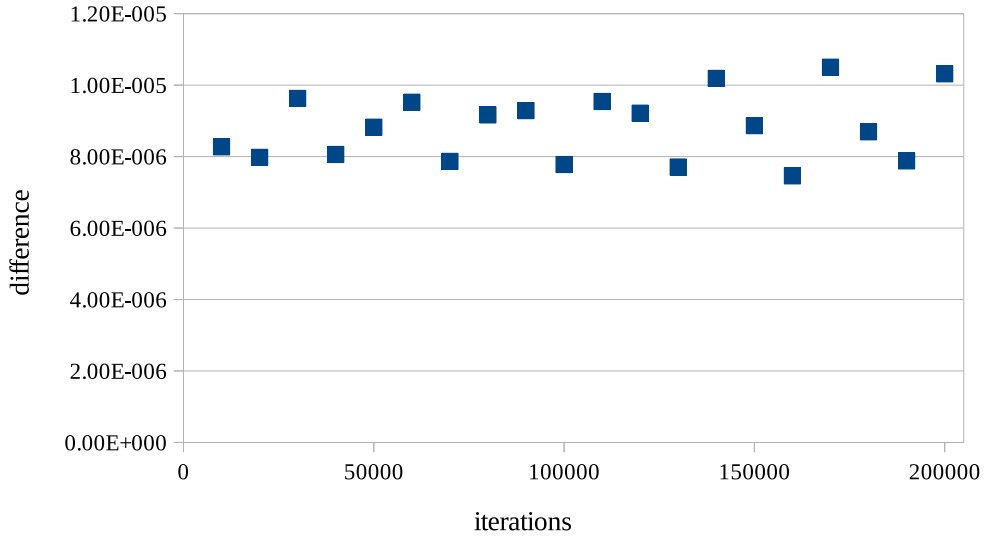


Figure A.7: Sudden expansion, MRT model, difference

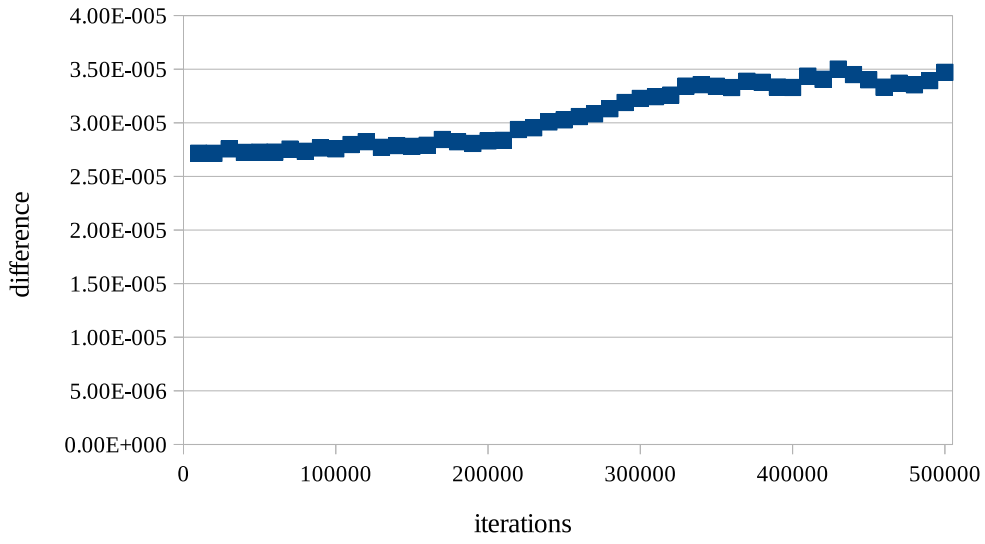


Figure A.8: Lid-driven cavity, MRT model, difference

Appendix B

Documentation

CUDA LBM Solver

1.8.2

Generated by Doxygen 1.8.6

Sun Aug 9 2015 22:46:54

Contents

Chapter 1

Main Page

Cranfield University 2015

This software is a Lattice Boltzmann solver for simple geometries.

Build

To build the program use the provided Makefile. By default it will create object files and build the code with single precision (float)

```
$ make
```

To build with double precision

```
$ make FLOAT_TYPE=USE_DOUBLE
```

To build a more compact (and slightly faster) version without object files

```
$ make release
```

To create Doxygen documentation (HTML, LaTeX)

```
$ make doc
```

To create Doxygen documentation and pdf

```
$ made latexdoc
```

To run unittests

```
$ make test
```

To create a debug build

```
$ make debug
```

To clean the project

```
$ make clean
```

Build on Windows

The code is prepared to be built in Windows environment although success is not guaranteed. To build it you will need GNU make. You can use the Windows compliant MinGW version or the POSIX compliant Cygwin version, both should work without other changes the same way on Linux.

- Visual C++ compiler suite (added to PATH) / also work with the free Express edition
- CUDA Toolkit (added to PATH)
- GNU make (added to PATH)

Dependencies

CuTest

CuTest is a single file ANSI C unittest framework, which compiles together with the code so it does not need any special treatment.

Argtable 2

Argtable is an ANSI C command line parser, whick use CMAKE for building configuration thus can be built on any environment. On windows it does not support regex so some parameters are defined as normal string parameters on windows. To build it, download and setup CMAKE than generate the project according to your environment and build it.

The repository contains already built version of the library for 64bit linux and 64bit Windows.

Running the program

Input files

The program needs at least 3 files to run if no parameters passed to the executable

- SetUpData.ini
- Mesh/D2node.dat
- Mesh/BCconnectors.dat

Input parameters

To get the full list of parameters

```
$ ./lbmsolver -h
Usage: ./lbmsolver  [-ht] [-f <file>] [-n <file>] [-b <file>] [-o <file>] [-u <u>] [-v <v>] [-r <rho>] [-s <nu
Usage: ./lbmsolver  compare <file> [<file>]
```

short	long	description
-h	–help	Print help options
-f	–initfile=<file>	Initialisation from file (default: SetUpData.ini)

-t	–test	Run unit tests
-n	–node=<file>	Node file (default: Mesh/D2node.dat)
-b	–bc=<file>	Boundary conditions file (default: Mesh/BCconnectors.dat)
-o	–output=<file>	Output directory (default: Results)
-u	–uavg=<u>	Mean U (x velocity)
-v	–vavg=<v>	Mean V (y velocity)
-r	–rho=<rho>	Density
-s	–viscosity=<nu>	Viscosity
	–inlet=[yes|no|pulsatile]	Inlet profile (default: no)
-c	–collision=[BGKW|TRT|MRT]	Collision model (default: BGKW)
	–curved	Curved boundaries
-l	–outlet=[yes|second|first]	Outlet profile (default: second)
-i	–iter=<N>	Number of iterations (default: 1000)
	–every=<N>	Autosave after every <N> iterations (default: 0)
	–after=<N>	Start autosaving after the <N>th iteration (default: 1000)
	–format=[paraview|tecplot]	Output format (default: paraview)
-d	–draglift=<id>	Calculate drag/lift on <id> boundary (default: 0)

If an init file is passed with -f all other parameters are discarded.

Compare files

To compare results in the default place with previous ones just run

```
$ ./lbmsolver compare <path to result file>
```

To compare files at path different than the default

```
$ ./lbmsolver compare <path to result file1> <path to result file2>
```

Authors

Adam Koleszar

Tamas Istvan Jozsa

Mate Tibor Szoke

Chapter 2

Changelog

Notable changes to the CUDA LBM Solver project

[1.8.2] - 2015-08-09

Added

- Add combined, faster TRT kernel

Changed

- rename variable and function names

Fixes

- Fix TRT collision model
- Fix wall condition

[1.8.1] - 2015-07-25

Added

- Add support for building on Windows
- Add resource freeing to failing tests (teardown)
- Add more unittests
- Add result check for CUDA API calls

Changed

- Split residual step into norm and drag/lift computation

Fixed

- Fix failing unittests
- FIx unittests not checking anything

[1.8.0] - 2015-07-20

Added

- Add doc and latexdoc target to Makefile
- Add Doxygen documentation comments
- Add new struct for input parameters
- Add this changelog

Changed

- Refactor variable names
- Separate headers
- Separate input handling from main

[1.7.1] - 2015-07-10

Added

- O3 to release target (to little effect)
- Add comparing unittests for boundary conditions

Fixed

- Fix outlet handling in macro step

[1.7.0] - 2015-07-06

Added

- Add command line interface

Fixed

- Fix problem with float/double switch

Changed

- Makefile options for float/double

[1.6.1] - 2015-07-03

Changed

- Combined MRT steps
- Rearrange MRT collision model to use less resurces

Added

- Add release target to Makefile
- Add debug target to Makefile

[1.6.0] - 2015-07-02

Changed

- Unfold kernels for sum, streaming and collision

Added

- Add utility functions for unittests

[1.5.0] - 2015-06-29

Changed

- Rearranged boundary conditions to speed up process and save memory

Added

- Add unittests for init and boundary conditions
- Add option to compare result files

[1.4.0] - 2015-06-24

Added

- GPU residual computation using GPU reduce (sum)
- Add unittests

Changed

- refactor init loop

[1.3.0] - 2015-06-16

Changed

- Generalise float type (use macros)

Removed

- Remove gpu_update_f and integrate it into gpu_streaming

[1.2.0] - 2015-06-16

Changed

- Move GPU constants to initialisation

Added

- Add final results write

[1.1.0] - 2015-06-15

Removed

- Removed binaries and meshes from repository

Added

- Add header files

Changed

- Move GPU functions to separate source files
- Speed up initialisation by rearranging the init loop

[1.0.1] - 2015-06-11

Fixed

- Fix typo in Makefile

[1.0] - 2015-06-11

Added

- Create repository
- Fork lbm-solver
- Add Makefile

Chapter 3

Test List

Global handleArguments (int argc, char ∗∗argv, InputFilenames ∗inFn, Arguments ∗args)

MANUAL: In order to test argument handling try some of these cases

- run with -h, should display help
- run without arguments, should work as written in SetUpData.ini
- run with -f <ini> and any other argument, should work as written in SetUpData.ini
- run with -o <folder> should output results to given folder
- run with -i <n> should do iterations n times
- run with -n <file> with non-existent file, should print out error message
- try using long argument options as well, should work as the short version
- run in compare mode with one file given, should compare Results/FinalData.csv to given file
- run in compare mode with two files, should compare the 2 given files

Global runTestCompareOutlet (CuTest ∗tc, OutletProfile op)

Prepare boundary conditions for the lid driven cavity

- Run the function
- Check the sum of all the arrays between the two algorithms

Global runTestGpuBcWall (CuTest ∗tc, BoundaryType boundaryType)

Prepare boundary conditions for the lid driven cavity

- Run the function
- Check the sum of all the arrays against predefined values

Global runTestGpuBoundaries2 (CuTest ∗tc, BoundaryType boundaryType)

Prepare boundary conditions for the lid driven cavity

- Run the function
- Check the sum of all the arrays against predefined values

Global runTestGpuInit1 (InletProfile inlt, FLOAT_TYPE u0, FLOAT_TYPE v0)

Unittest for gpu_init_1

- Prepare arrays
- call function
- check the sum of the arrays against predefined values
- test for all inlet profiles

Global testCompareGpuStream (CuTest ∗tc)

Prepare boundary conditions for the lid driven cavity

- Run the function

- Check the sum of all the arrays between the two algorithms

Global testCompareInlet (CuTest *tc)

Prepare boundary conditions for the lid driven cavity

- Run the function
- Check the sum of all the arrays between the two algorithms

Global testCompareMrt (CuTest *tc)

Prepare boundary conditions for the lid driven cavity

- Run the function
- Check the sum of all the arrays between the two algorithms

Global testCompareTrt (CuTest *tc)

Prepare boundary conditions for the lid driven cavity

- Run the function
- Check the sum of all the arrays between the two algorithms

Global testCompareUpdateMacro (CuTest *tc)

compare results from previous runs

Global testCompareWall (CuTest *tc)

Prepare boundary conditions for the lid driven cavity

- Run the function
- Check the sum of all the arrays against predefined values

Global testComputeDragLift (CuTest *tc)

Prepare arrays

- call function
- check the sum of the arrays against predefined values

Global testComputeResidual (CuTest *tc)

Prepare arrays

- call function
- check the sum of the arrays against predefined values

Global testGpuBcInlet (CuTest *tc)

Prepare boundary conditions for the lid driven cavity

- Run the function
- Check the sum of all the arrays against predefined values

Global testGpuBcOutlet (CuTest *tc)

Prepare boundary conditions for the lid driven cavity

- Run the function
- Check the sum of all the arrays against predefined values

Global testGpuBoundaries1 (CuTest *tc)

Prepare boundary conditions for the lid driven cavity

- Run the function
- Check the sum of all the arrays against predefined values

Global testGpuBoundaries3 (CuTest *tc)

Prepare boundary conditions for the lid driven cavity

- Run the function
- Check the sum of all the arrays against predefined values

Global testGpuComputeResidMask (CuTest *tc)

Prepare arrays

- call function
- check the sum of the arrays against predefined values

Global testGpuComputeResiduals (CuTest *tc)

Prepare arrays

- call function
- check the sum of the arrays against predefined values

Global testGpuCondCopy (CuTest *tc)

Allocate arrays and fill them with random values

- call function
- check the sum of the arrays against predefined values

Global testGpuInitInletProfile (CuTest *tc)

Prepare arrays for inlet profile

- call function
- check the sum of the arrays against predefined values

Global testGpuSquareSubstract (CuTest *tc)

Allocate arrays and fill them with fixed values

- call function
- check the sum of the arrays against predefined values

Global testGpuSum (CuTest *tc)

Allocate arrays and fill them with fixed values

- call function
- check the sum of the arrays against predefined values

Global testGpuSum256 (CuTest *tc)

Allocate arrays and fill them with fixed values

- call function
- check the sum of the arrays against predefined values

Global testGpusumHost (CuTest *tc)

Allocate arrays and fill them with fixed values

- call function
- check the sum of the arrays against predefined values

Global testGpuUpdateMacro (CuTest *tc)

Prepare arrays for lid driven cavity

- call function
- check the sum of the arrays against predefined values

Global testGpuUpdateMacro_ (CuTest *tc)

Prepare arrays for lid driven cavity

- call function
- check the sum of the arrays against predefined values

Global testGpuUpdateNew (CuTest *tc)

Prepare arrays for lid driven cavity

- call function
- check the sum of the arrays against predefined values

Chapter 4

Todo List

Global feqc (FLOAT_TYPE u, FLOAT_TYPE uc, FLOAT_TYPE v, FLOAT_TYPE vc, FLOAT_TYPE rho, FLOAT_TYPE w)

doc: insert name or reference, explain method

Global gpu_boundaries1 (int ∗fluid_d, int ∗boundary_d, int ∗bcId_d, FLOAT_TYPE ∗f_d, FLOAT_TYPE ∗u0_d, FLOAT_TYPE ∗v0_d, int ∗corner_d)

code: complete with y velocity

Global gpu_boundaries3 (int ∗fluid_d, int ∗bcId_d, FLOAT_TYPE ∗f_d, FLOAT_TYPE ∗u0_d, FLOAT_TYPE ∗v0_d, int ∗corner_d)

code: fill north-side outlet

Global gpu_update_macro (int ∗fluid_d, FLOAT_TYPE ∗rho_d, FLOAT_TYPE ∗u_d, FLOAT_TYPE ∗v_d, int ∗bcId_d, int ∗boundaryId_d, FLOAT_TYPE ∗drag_d, FLOAT_TYPE ∗lift_d, FLOAT_TYPE ∗coordX_d, FLOAT_TYPE ∗coordY_d, FLOAT_TYPE ∗f_d)

code: probably should handle outlet on other sides

Global gpu_update_new (int ∗fluid_d, FLOAT_TYPE ∗rho_d, FLOAT_TYPE ∗u_d, FLOAT_TYPE ∗v_d, int ∗bcMask_d, FLOAT_TYPE ∗drag_d, FLOAT_TYPE ∗lift_d, FLOAT_TYPE ∗coordX_d, FLOAT_TYPE ∗coordY_d, FLOAT_TYPE ∗f_d)

code: probably should handle outlet on other sides

Global gpuBcInlet (int ∗bcIdx_d, int ∗bcMask_d, FLOAT_TYPE ∗f_d, FLOAT_TYPE ∗u0_d, FLOAT_TYPE ∗v0_d, int size)

code: simplify code even though it will change accuracy

Global gpuBcOutlet (int ∗bcIdx_d, int ∗bcMask_d, FLOAT_TYPE ∗f_d, FLOAT_TYPE ∗u0_d, FLOAT_TYPE ∗v0_d, int size)

code: simplify code even though it will change accuracy

Global gpuUpdateMacro (int ∗fluid_d, FLOAT_TYPE ∗rho_d, FLOAT_TYPE ∗u_d, FLOAT_TYPE ∗v_d, int ∗bcMask_d, FLOAT_TYPE ∗drag_d, FLOAT_TYPE ∗lift_d, FLOAT_TYPE ∗coordX_d, FLOAT_TYPE ∗coordY_d, FLOAT_TYPE ∗f_d)

code: probably should handle outlet on other sides

File TestIterate.cu

create a command-line argument for test meshes

Chapter 5

Deprecated List

Global CompDataConn (int *NumInletNodes, FLOAT_TYPE *MaxInletCoordY, FLOAT_TYPE *MinInletCoordY, int *BCconn0, int *BCconn1, int *BCconn2, int *BCconn3, FLOAT_TYPE *BCconn4, FLOAT_TYPE *BCconn5, int *BCconn6, int *NumConn, FLOAT_TYPE *Delta)

use getNumInletNodes, getMaxInletCoordY and getMinInletCoordY instead

Global CompDataNode (FLOAT_TYPE *Delta, int *m, int *n, int *Nodes0, int *Nodes1, FLOAT_TYPE *Nodes2, FLOAT_TYPE *Nodes3, int *Nodes4, int *NumNodes)

use getLastValue and getGridSpacing instead

Global ComputeResiduals (int *boundaryId, FLOAT_TYPE *f, FLOAT_TYPE *fColl, FLOAT_TYPE *drag, FLOAT_TYPE *lift, FLOAT_TYPE *residuals, int *m, int *n, int computeDragLift)

use GpuComputeResidMask instead

Global gpu_bgk (int *fluid_d, FLOAT_TYPE *fEq_d, FLOAT_TYPE *rho_d, FLOAT_TYPE *u_d, FLOAT_TYPE *v_d, FLOAT_TYPE *fColl_d, FLOAT_TYPE *f_d)

use gpuCollBgkw

Global gpu_boundaries1 (int *fluid_d, int *boundary_d, int *bcId_d, FLOAT_TYPE *f_d, FLOAT_TYPE *u0_d, FLOAT_TYPE *v0_d, int *corner_d)

use gpuBcInlet

Global gpu_boundaries2 (int *fluid_d, FLOAT_TYPE *Fneighbours_d, FLOAT_TYPE *fColl_d, int *bcId_d, FLOAT_TYPE *Q_d, FLOAT_TYPE *f_d)

use gpuBcWall

Global gpu_boundaries3 (int *fluid_d, int *bcId_d, FLOAT_TYPE *f_d, FLOAT_TYPE *u0_d, FLOAT_TYPE *v0_d, int *corner_d)

use gpuBcOutlet

Global gpu_cond_copy (FLOAT_TYPE *A, FLOAT_TYPE *B, int *cond, int value, int size)

use gpu_cond_copy_mask

Global gpu_init (int *corner_d, FLOAT_TYPE *rho_d, FLOAT_TYPE *Q_d, int *boundary_d, FLOAT_TYPE *coordY_d, int *stream_d, int *bcId_d, FLOAT_TYPE *u_d, FLOAT_TYPE *v_d, FLOAT_TYPE *u0_d, FLOAT_TYPE *v0_d)

use gpu_init_1 and initBoundaryConditions instead

Global gpu_init_1 (FLOAT_TYPE *rho_d, FLOAT_TYPE *u0_d, FLOAT_TYPE *v0_d, FLOAT_TYPE *u_d, FLOAT_TYPE *v_d, FLOAT_TYPE *coordY_d, int size)

use createGpuArrayFlt and gpuInitInletProfile

Global gpu_init_8 (int *stream_d, FLOAT_TYPE *Q_d, int size)

use createGpuArrayFlt instead

Global gpu_mrt1 (int *fluid_d, FLOAT_TYPE *rho_d, FLOAT_TYPE *u_d, FLOAT_TYPE *v_d, FLOAT_TYPE *f_d, FLOAT_TYPE *mEq_d, FLOAT_TYPE *m_d)

use gpuCollMrt

Global gpu_mrt2 (int *fluid_d, FLOAT_TYPE *collision_d, FLOAT_TYPE *m_d, FLOAT_TYPE *mEq_d, FLOAT_TYPE *fColl_d, FLOAT_TYPE *f_d)

use gpuCollMrt

Global gpu_streaming (int *fluid_d, int *stream_d, FLOAT_TYPE *f_d, FLOAT_TYPE *fColl_d)

use gpuStreaming

Global gpu_trt1 (int *fluid_d, FLOAT_TYPE *fEq_d, FLOAT_TYPE *rho_d, FLOAT_TYPE *u_d, FLOAT_TYPE *v_d)

use gpuCollTrt

Global gpu_trt2 (int *fluid_d, FLOAT_TYPE *fEq_d, FLOAT_TYPE *f_d, FLOAT_TYPE *fColl_d)

use gpuCollTrt

Global gpu_update_macro (int *fluid_d, FLOAT_TYPE *rho_d, FLOAT_TYPE *u_d, FLOAT_TYPE *v_d, int *bcId_d, int *boundaryId_d, FLOAT_TYPE *drag_d, FLOAT_TYPE *lift_d, FLOAT_TYPE *coordX_d, FLOAT_TYPE *coordY_d, FLOAT_TYPE *f_d)

use gpuUpdateMacro

Global gpu_update_new (int *fluid_d, FLOAT_TYPE *rho_d, FLOAT_TYPE *u_d, FLOAT_TYPE *v_d, int *bcMask_d, FLOAT_TYPE *drag_d, FLOAT_TYPE *lift_d, FLOAT_TYPE *coordX_d, FLOAT_TYPE *coordY_d, FLOAT_TYPE *f_d)

use gpuUpdateMacro

Global GpuComputeResiduals (int *boundaryId_d, FLOAT_TYPE *f_d, FLOAT_TYPE *fColl_d, FLOAT_TYPE *drag_d, FLOAT_TYPE *lift_d, FLOAT_TYPE *temp9a_d, FLOAT_TYPE *temp9b_d, FLOAT_TYPE *tempA_d, FLOAT_TYPE *tempB_d, FLOAT_TYPE *residuals, int *m, int *n, int computeDragLift)

use GpuComputeResidMask

Chapter 6

Module Index

6.1 Modules

Here is a list of all modules:

Chapter 7

Data Structure Index

7.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 8

File Index

8.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 9

Module Documentation

9.1 Solver

CUDA LBM Solver functions.

Files

- file Iterate.h

  The solver itself.

Functions

- __global__ void gpuBcInlet (int *bcIdx_d, int *bcMask_d, FLOAT_TYPE *f_d, FLOAT_TYPE *u0_d, FLOAT_TYPE *v0_d, int size)

  Inlet boundary conditions.
- __global__ void gpuBcWall (int *bcIdx_d, int *bcMask_d, FLOAT_TYPE *f_d, FLOAT_TYPE *fColl_d, FLOAT_TYPE *Q_d, int size)

  Wall boundary conditions.
- __global__ void gpuBcOutlet (int *bcIdx_d, int *bcMask_d, FLOAT_TYPE *f_d, FLOAT_TYPE *u0_d, FLOAT_TYPE *v0_d, int size)

  Outlet boundary conditions.
- __global__ void gpuCollBgkw (int *fluid_d, FLOAT_TYPE *rho_d, FLOAT_TYPE *u_d, FLOAT_TYPE *v_d, FLOAT_TYPE *f_d, FLOAT_TYPE *fColl_d)

  BGKW collision model (Bhatnagar–Gross–Krook-Welander)
- __global__ void gpuCollTrt (int *fluid_d, FLOAT_TYPE *rho_d, FLOAT_TYPE *u_d, FLOAT_TYPE *v_d, FLOAT_TYPE *f_d, FLOAT_TYPE *fColl_d)

  TRT collision model (two-relaxation-time)
- __global__ void gpuCollMrt (int *fluid_d, FLOAT_TYPE *rho_d, FLOAT_TYPE *u_d, FLOAT_TYPE *v_d, FLOAT_TYPE *f_d, FLOAT_TYPE *fColl_d)

  MRT collision model (multiple-relaxation-time)
- __global__ void gpuStreaming (int *fluid_d, int *stream_d, FLOAT_TYPE *f_d, FLOAT_TYPE *fColl_d)

  Streaming.
- __global__ void gpuUpdateMacro (int *fluid_d, FLOAT_TYPE *rho_d, FLOAT_TYPE *u_d, FLOAT_TYPE *v_d, int *bcMask_d, FLOAT_TYPE *drag_d, FLOAT_TYPE *lift_d, FLOAT_TYPE *coordX_d, FLOAT_TYPE *coordY_d, FLOAT_TYPE *f_d)

  Update macroscopic values using bitmask.

9.1.1 Detailed Description

CUDA LBM Solver functions. Steps:

1. collision model
2. streaming
3. boundary conditions
4. macroscopic values

9.1.2 Function Documentation

9.1.2.1 __global__ void gpuBcInlet (int * *bcIdx_d*, int * *bcMask_d*, FLOAT_TYPE * *f_d*, FLOAT_TYPE * *u0_d*, FLOAT_TYPE * *v0_d*, int *size*)

Inlet boundary conditions.

Computes the effect of the inlet conditions, using ... North inlet conditions (with $v_0 = 0$):

$$f_4 = f_2$$

$$f_7 = f_5 - u_0(\frac{1}{6}(f_0 + f_1 + f_3) - \frac{1}{3}(f_2 + f_5 + f_6))$$

$$f_8 = f_6 + u_0(\frac{1}{6}(f_0 + f_1 + f_3) + \frac{1}{3}(f_2 + f_5 + f_6))$$

West inlet conditions:

$$f_1 = f_3 + \frac{2}{3}\frac{u_0}{1-u_0}(f_0 + f_2 + f_4 + 2(f_3 + f_6 + f_7))$$

$$f_5 = f_7 + \frac{1}{6}\frac{u_0}{1-u_0}(f_0 + f_2 + f_4 + 2(f_3 + f_6 + f_7))$$

$$f_8 = f_6 + \frac{1}{6}\frac{u_0}{1-u_0}(f_0 + f_2 + f_4 + 2(f_3 + f_6 + f_7))$$

The function goes through the boundary conditions and only act on lattices having inlet conditions that are not corner nodes. In the corners, wall conditions apply.

Parameters

in	*bcIdx_d*	boundary condition indices
in	*bcMask_d*	boundary conditions bitmask
out	*f_d*	distribution function
in	*u0_d,v0_d*	input velocity vectors (x,y)
in	*size*	number of boundary conditions

Todo code: simplify code even though it will change accuracy

Todo code: compute inlet on other sides

9.1.2.2 __global__ void gpuBcOutlet (int * *bcIdx_d*, int * *bcMask_d*, FLOAT_TYPE * *f_d*, FLOAT_TYPE * *u0_d*, FLOAT_TYPE * *v0_d*, int *size*)

Outlet boundary conditions.

Computes the effect of the outlet conditions, using ... East outlet conditions:

$$f_3 = f_1 - \frac{2}{3}\frac{u_0}{1-u_0}(f_0 + f_2 + f_4 + 2(f_1 + f_5 + f_8))$$

$$f_6 = f_8 - \frac{1}{6}\frac{u_0}{1-u_0}(f_0 + f_2 + f_4 + 2(f_1 + f_5 + f_8))$$

$$f_7 = f_5 - \frac{1}{6}\frac{u_0}{1-u_0}(f_0 + f_2 + f_4 + 2(f_1 + f_5 + f_8))$$

Second order open boundary east outlet conditions

$$f_1 = 2f_1^{(-1)} - f_1^{(-2)}$$

$$f_5 = 2f_5^{(-1)} - f_5^{(-2)}$$

$$f_8 = 2f_8^{(-1)} - f_8^{(-2)}$$

First order open boundary east outlet conditions

$$f_1 = f_1^{(-1)}$$

$$f_5 = f_5^{(-1)}$$

$$f_8 = f_8^{(-1)}$$

Parameters

in	*bcIdx_d*	boundary condition indices
in	*bcMask_d*	boundary conditions bitmask
out	*f_d*	distribution function
in	*u0_d,v0_d*	input velocity vectors (x,y)
in	*size*	number of boundary conditions

Todo code: simplify code even though it will change accuracy

Todo code: fill north-side outlet

Todo code: fill west-side outlet

Todo code: fill south-side outlet

Todo code: fill north-side outlet

Todo code: fill west-side outlet

Todo code: fill south-side outlet

Todo code: fill north-side outlet

Todo code: fill west-side outlet

Todo code: fill south-side outlet

9.1.2.3 __global__ void gpuBcWall (int * *bcIdx_d*, int * *bcMask_d*, FLOAT_TYPE * *f_d*, FLOAT_TYPE * *fColl_d*, FLOAT_TYPE * *Q_d*, int *size*)

Wall boundary conditions.

Computes the effect of the wall conditions. For straight wall the computation we use the half-way bounce-back method: $f_{opp(i)} = f_i$, e.g. on an east-side wall:

$$f_3 = f_1; f_6 = f_8; f_7 = f_5$$

If the boundaries are not straight the following interpolation is done using q vector as defined in initBoundary-Conditions, where $f_{c(i)}$ means the same lattice in the neighbouring node in the given direction (more about this in gpuStreaming):

$$f_{opp(i)} = 2q_i f_i^{coll} + (1-2q_i) f_{c(i)}^{coll}, q_i < \frac{1}{2}$$

$$f_{opp(i)} = \frac{1}{2q_i} f_i^{coll} + \frac{2q_i - 1}{2q_i} f_{opp(i)}^{coll}, q_i >= \frac{1}{2}$$

Parameters

in	*bcIdx_d*	boundary condition indices
in	*bcMask_d*	boundary conditions bitmask
out	*f_d*	distribution function
in	*fColl_d*	distribution function from the collison step
in	*Q_d*	grid center ratio array
in	*size*	number of boundary conditions

9.1.2.4 __global__ void gpuCollBgkw (int * *fluid_d*, FLOAT_TYPE * *rho_d*, FLOAT_TYPE * *u_d*, FLOAT_TYPE * *v_d*, FLOAT_TYPE * *f_d*, FLOAT_TYPE * *fColl_d*)

BGKW collision model (Bhatnagar–Gross–Krook-Welander)

Computation for every lattice :

$$f_i^{eq} = \rho w(1 + 3(uc_{xi} + vc_{yi}) + \frac{9}{2}(uc_{xi} + vc_{yi})^2 - \frac{3}{2}(u^2 + v^2))$$

$$f_i^{coll} = \omega f_i^{eq} + (1-\omega) f_i^{eq}$$

Parameters

in	*fluid_d*	fluid condition
in	*rho_d*	density
in	*u_d,v_d*	velocity vectors (x,y)
in	*f_d*	distribution function from boundary step
out	*fColl_d*	distribution function

9.1.2.5 __global__ void gpuCollMrt (int * *fluid_d*, FLOAT_TYPE * *rho_d*, FLOAT_TYPE * *u_d*, FLOAT_TYPE * *v_d*, FLOAT_TYPE * *f_d*, FLOAT_TYPE * *fColl_d*)

MRT collision model (multiple-relaxation-time)

The MRT collision model computes new values for the distribution function in momentum space and uses the following formula:

$$f_i^{coll} = f_i - \mathbf{M}^{-1}\mathbf{S}(\mathbf{m} - \mathbf{m}^{eq}) = f_i - \mathbf{M}^{-1}\mathbf{S}(\mathbf{Mf} - \mathbf{m}^{eq})$$

$$mathbf{fm} = (\rho, e, \varepsilon, j_x, q_x, j_y, q_y, p_{xx}, p_{xy})^T$$

$$m_0^{eq} = \rho$$

$$m_1^{eq} = -2\rho + 3(j_x^2 + j_y^2)$$
$$m_2^{eq} = \rho - 3(j_x^2 + j_y^2)$$
$$m_3^{eq} = j_x = \rho u_x = \sum_i f_i^{eq} c_{ix}$$
$$m_4^{eq} = -j_x$$
$$m_5^{eq} = j_y = \rho u_y = \sum_i f_i^{eq} c_{iy}$$
$$m_6^{eq} = -j_y$$
$$m_7^{eq} = j_x^2 + j_y^2$$
$$m_8^{eq} = j_x j_y$$

For more details check A. A. Mohamad - Lattice Boltzmann Method (book)

Parameters

in	*fluid_d*	fluid condition
in	*rho_d*	density
in	*u_d,v_d*	velocity vectors (x,y)
in	*f_d*	distribution function from boundary step
out	*fColl_d*	distribution function

9.1.2.6 __global__ void gpuCollTrt (int * *fluid_d,* FLOAT_TYPE * *rho_d,* FLOAT_TYPE * *u_d,* FLOAT_TYPE * *v_d,* FLOAT_TYPE * *f_d,* FLOAT_TYPE * *fColl_d*)

TRT collision model (two-relaxation-time)

Computation for every lattice :

$$f_i^{eq} = \rho w(1 + 3(uc_{xi} + vc_{yi}) + \frac{9}{2}(uc_{xi} + vc_{yi})^2 - \frac{3}{2}(u^2 + v^2))$$
$$f_i^{coll} = f_i - \frac{1}{2}\omega(f_i + f_i^{-1} - f_i^{eq} - f_i^{eq,-1}) - \frac{1}{2}\omega_a(f_i - f_i^{-1} - f_i^{eq} + f_i^{eq,-1})$$
$$f_i^{-1} = f_{opp(i)}$$

Parameters

in	*fluid_d*	fluid condition
in	*rho_d*	density
in	*u_d,v_d*	velocity vectors (x,y)
in	*f_d*	distribution function from boundary step
out	*fColl_d*	distribution function

9.1.2.7 __global__ void gpuStreaming (int * *fluid_d,* int * *stream_d,* FLOAT_TYPE * *f_d,* FLOAT_TYPE * *fColl_d*)

Streaming.

Basically the distribution function propagates to neighbouring cells the following way

	i-4	i-3	i-2	i-1	i	i+1	i+2	i+3	i+4
j+4	f6				f2				f5
j+3		\			|			/	

j+2			\		|		/		
j+1				f6	f2	f5			
j	f3	-	-	f3	f0	f1	-	-	f1
j-1				f7	f4	f8			
j-2			/		|		\		
j-3		/			|			\	
j-4	f7				f4				f8

Parameters

fluid_d	fluid condition
stream_d	streaming array
f_d	distribution function
fColl_d	distribution function from the collison step

9.1.2.8 __global__ void gpuUpdateMacro (int * *fluid_d*, FLOAT_TYPE * *rho_d*, FLOAT_TYPE * *u_d*, FLOAT_TYPE * *v_d*, int * *bcMask_d*, FLOAT_TYPE * *drag_d*, FLOAT_TYPE * *lift_d*, FLOAT_TYPE * *coordX_d*, FLOAT_TYPE * *coordY_d*, FLOAT_TYPE * *f_d*)

Update macroscopic values using bitmask.

Computation of the macroscopic values for a node are the following:

$$\rho = \sum_{i=0}^{8} f_i$$

$$\vec{v} = (u,v) = \frac{1}{\rho}\sum_{i=1}^{8} f_i \vec{c}_i$$

$$F_{drag} = \frac{\rho}{15}(20-x)$$

$$F_{lift} = \frac{\rho}{15}(20-y)$$

See Also

for $\vec{c}_i$ see cx_d and cy_d in GpuConstants.h

Parameters

in	*fluid_d*	fluid condition
out	*rho_d*	density
out	*u_d,v_d*	velocity vectors (x,y)
in	*bcMask_d*	boundary conditions bitmask (see BcMacros.h)
out	*drag_d*	drag
out	*lift_d*	lift
in	*coordX_d,coordY_d*	node coordinates x,y
in	*f_d*	distribution function

Todo code: probably should handle outlet on other sides

Chapter 10

Data Structure Documentation

10.1 Arguments Struct Reference

Input parameters.

`#include <Arguments.h>`

Data Fields

- FLOAT_TYPE u

  velocity x component
- FLOAT_TYPE v

  velocity y component
- FLOAT_TYPE rho

  density
- FLOAT_TYPE viscosity

  viscosity
- InletProfile inletProfile

  inlet profile
- OutletProfile outletProfile

  outlet profile
- CollisionModel collisionModel

  collision model
- BoundaryType boundaryType

  boundary type
- OutputFormat outputFormat

  output format
- int iterations

  number of iterations
- int autosaveEvery

  autosave every n iteration
- int autosaveAfter

  autosave after nth iteration
- int boundaryId

  boundary ID for drag/lift calculation

10.1.1 Detailed Description

Input parameters.

The documentation for this struct was generated from the following file:

- include/Arguments.h

10.2 InputFilenames Struct Reference

Input filenames.

`#include <Arguments.h>`

Data Fields

- char init [512]
  init file name
- char node [512]
  node file name
- char bc [512]
  bc file name
- char result [512]
  results directory name
- char comp [512]
  filename to compare
- char final [512]
  filename to compare to

10.2.1 Detailed Description

Input filenames.

The documentation for this struct was generated from the following file:

- include/Arguments.h

Chapter 11

File Documentation

11.1 Arguments.cu File Reference

Command line parameter handling.

```
#include <string.h>
#include <regex.h>
#include "Arguments.h"
#include "FilesReading.h"
#include "argtable2.h"
```

Functions

- InletProfile getInletProfile (const char *input, InletProfile defaultValue)
  Get inlet profile from command line parameter.
- OutletProfile getOutletProfile (const char *input, OutletProfile defaultValue)
  Get outlet profile from command line parameter.
- CollisionModel getCollisionModel (const char *input, CollisionModel defaultValue)
  Get collision model from command line parameter.
- BoundaryType getBoundaryType (int input, BoundaryType defaultValue)
  Get boundary type from command line parameter.
- OutputFormat getOutputFormat (const char *input, OutputFormat defaultValue)
  Get output format from command line parameter.
- ArgResult handleArguments (int argc, char **argv, InputFilenames *inFn, Arguments *args)
  Parse command line arguments.

11.1.1 Detailed Description

Command line parameter handling.

Author

Adam Koleszar (adam.koleszar@gmail.com)

11.1.2 Function Documentation

11.1.2.1 BoundaryType getBoundaryType (int *input,* BoundaryType *defaultValue*)

Get boundary type from command line parameter.

Parameters

input	command line parameter
defaultValue	default value

Returns

chosen boundary type

11.1.2.2 CollisionModel getCollisionModel (const char * input, CollisionModel defaultValue)

Get collision model from command line parameter.

Parameters

input	command line parameter
defaultValue	default value

Returns

chosen collision model

11.1.2.3 InletProfile getInletProfile (const char * input, InletProfile defaultValue)

Get inlet profile from command line parameter.

Parameters

input	command line parameter
defaultValue	default value

Returns

chosen inlet profile

11.1.2.4 OutletProfile getOutletProfile (const char * input, OutletProfile defaultValue)

Get outlet profile from command line parameter.

Parameters

input	command line parameter
defaultValue	default value

Returns

chosen outlet profile

11.1.2.5 OutputFormat getOutputFormat (const char * input, OutputFormat defaultValue)

Get output format from command line parameter.

Parameters

input	command line parameter
defaultValue	default value

Returns

chosen output format

11.1.2.6 ArgResult handleArguments (int *argc*, char ** *argv*, InputFilenames * *inFn*, Arguments * *args*)

Parse command line arguments.

Parameters

argc	number of command line arguments
argv	command line arguments
inFn	input filenames
args	input parameters

Returns

task that need to be done, or error

Test MANUAL: In order to test argument handling try some of these cases

- run with -h, should display help
- run without arguments, should work as written in SetUpData.ini
- run with -f <ini> and any other argument, should work as written in SetUpData.ini
- run with -o <folder> should output results to given folder
- run with -i <n> should do iterations n times
- run with -n <file> with non-existent file, should print out error message
- try using long argument options as well, should work as the short version
- run in compare mode with one file given, should compare Results/FinalData.csv to given file
- run in compare mode with two files, should compare the 2 given files

11.2 ArrayUtils.cu File Reference

Memory allocation wrappers for host and gpu arrays.

```
#include <stdlib.h>
#include <stdio.h>
#include <cuda.h>
#include <time.h>
#include <assert.h>
#include "ArrayUtils.h"
```

Functions

- FLOAT_TYPE getRandom (unsigned long *seed)

  Get a uniform random number.
- __device__ FLOAT_TYPE getRandomDev (unsigned long seed)

Get a uniform random number on GPU.

- int * createGpuArrayInt (int length, ArrayOption op, int fill, int *copy)

  Create 1D int array on the device.
- FLOAT_TYPE * createGpuArrayFlt (int length, ArrayOption op, FLOAT_TYPE fill, FLOAT_TYPE *copy)

  Create 1D float/double array on the device.
- int * createHostArrayInt (int length, ArrayOption op, int fill, int *copy)

  Create 1D int array on the host.
- FLOAT_TYPE * createHostArrayFlt (int length, ArrayOption op, FLOAT_TYPE fill, FLOAT_TYPE *copy)

  Create 1D float/double array on the device.
- int ** create2DHostArrayInt (int width, int height)

  Allocate 2D integer array.
- FLOAT_TYPE ** create2DHostArrayFlt (int width, int height)

  Allocate 2D floating point array.
- int *** create3DHostArrayInt (int width, int height, int depth)

  Allocate 3D integer array.
- FLOAT_TYPE *** create3DHostArrayFlt (int width, int height, int depth)

  Allocate 3D floating point array.
- bool *** create3DHostArrayBool (int width, int height, int depth)

  Allocate 3D boolean array.
- __global__ void gpuArrayFillInt (int *array_d, int size)

  GPU kernel to fill int array with given value.
- __global__ void gpuArrayFillFlt (FLOAT_TYPE *array_d, int size)

  GPU kernel to fill float/double array with given value.
- __global__ void gpuArrayFillRandom (FLOAT_TYPE *array_d, unsigned long seed, int size)

  GPU kernel to fill float/double array with random value.
- void hostArrayFillInt (int *array_h, int fill, int size)

  Fill int array with given value.
- void hostArrayFillFlt (FLOAT_TYPE *array_h, FLOAT_TYPE fill, int size)

  Fill float/double array with given value.
- void hostArrayFillRandom (FLOAT_TYPE *array_h, int size, FLOAT_TYPE r)

  Fill float/double array with random value.
- void freeAllHost (void **as, int n)

  Free all host arrays.
- void freeAllGpu (void **as, int n)

  Free all GPU arrays.

Variables

- __constant__ FLOAT_TYPE fill_fd

  value to fill into floating point array
- __constant__ int fill_id

  value to fill into integer array

11.2.1 Detailed Description

Memory allocation wrappers for host and gpu arrays.

Author

Adam Koleszar (adam.koleszar@gmail.com)

11.2.2 Function Documentation

11.2.2.1 FLOAT_TYPE** create2DHostArrayFlt (int *width*, int *height*)

Allocate 2D floating point array.

Note

not used

Parameters

in	*width*	array width
in	*height*	array height

Returns

pointer to the allocated array

11.2.2.2 int** create2DHostArrayInt (int *width*, int *height*)

Allocate 2D integer array.

Note

not used

Parameters

in	*width*	array width
in	*height*	array height

Returns

pointer to the allocated array

11.2.2.3 bool*** create3DHostArrayBool (int *width*, int *height*, int *depth*)

Allocate 3D boolean array.

Note

not used

Parameters

in	*width*	array width
in	*height*	array height
in	*depth*	array depth

Returns

pointer to the allocated array

11.2.2.4 FLOAT_TYPE*** create3DHostArrayFlt (int *width*, int *height*, int *depth*)

Allocate 3D floating point array.

Note

not used

Parameters

in	*width*	array width
in	*height*	array height
in	*depth*	array depth

Returns

pointer to the allocated array

11.2.2.5 int*** create3DHostArrayInt (int *width*, int *height*, int *depth*)

Allocate 3D integer array.

Note

not used

Parameters

in	*width*	array width
in	*height*	array height
in	*depth*	array depth

Returns

pointer to the allocated array

11.2.2.6 FLOAT_TYPE* createGpuArrayFlt (int *length*, ArrayOption *op* = ARRAY_NONE, FLOAT_TYPE *fill* = 0, FLOAT_TYPE * *copy* = NULL)

Create 1D float/double array on the device.

Parameters

length	length of the array
op	options (default: none)
fill	value to fill array (default: 0)
copy	host array to copy into the new array (default: NULL)

Returns

new array

11.2.2.7 int* createGpuArrayInt (int *length*, ArrayOption *op* = ARRAY_NONE, int *fill* = 0, int * *copy* = NULL)

Create 1D int array on the device.

Parameters

length	length of the array
op	options (default: none)
fill	value to fill array (default: 0)
copy	host array to copy into the new array (default: NULL)

Returns

new array

11.2.2.8 FLOAT_TYPE* createHostArrayFlt (int *length*, ArrayOption *op* = ARRAY_NONE, FLOAT_TYPE *fill* = 0, FLOAT_TYPE * *copy* = NULL)

Create 1D float/double array on the device.

Parameters

length	length of the array
op	options (default: none)
fill	value to fill array (default: 0)
copy	array to copy into the new array (default: NULL)

Returns

new array

11.2.2.9 int* createHostArrayInt (int *length*, ArrayOption *op* = ARRAY_NONE, int *fill* = 0, int * *copy* = NULL)

Create 1D int array on the host.

Parameters

length	length of the array
op	options (default: none)
fill	value to fill array (default: 0)
copy	array to copy into the new array (default: NULL)

Returns

new array

11.2.2.10 void freeAllGpu (void ** *as*, int *n*)

Free all GPU arrays.

Parameters

as	array of arrays
n	number of arrays

11.2.2.11 void freeAllHost (void ** *as*, int *n*)

Free all host arrays.

Parameters

as	array of arrays
n	number of arrays

11.2.2.12 FLOAT_TYPE getRandom (unsigned long * *seed*)

Get a uniform random number.

Parameters

seed	seed for the random number (initialise with time(NULL))

Returns

uniform random number

11.2.2.13 __device__ FLOAT_TYPE getRandomDev (unsigned long *seed*)

Get a uniform random number on GPU.

Parameters

seed	seed for the random number (initialise with time(NULL))

Returns

uniform random number

11.2.2.14 __global__ void gpuArrayFillFlt (FLOAT_TYPE * *array_d,* int *size*)

GPU kernel to fill float/double array with given value.

Parameters

array_d	array on device
size	size of the array

Warning

fill value should be set into variable fill_fd in constant memory

11.2.2.15 __global__ void gpuArrayFillInt (int * *array_d,* int *size*)

GPU kernel to fill int array with given value.

Parameters

array_d	array on device
size	size of the array

Warning

fill value should be set into variable fill_id in constant memory

11.2.2.16 __global__ void gpuArrayFillRandom (FLOAT_TYPE * *array_d,* unsigned long *seed,* int *size*)

GPU kernel to fill float/double array with random value.

Parameters

array_d	array on device
seed	seed for random generator (spiked with global id)
size	size of the array

11.2.2.17 void hostArrayFillFlt (FLOAT_TYPE * *array_h*, FLOAT_TYPE *fill*, int *size*)

Fill float/double array with given value.

Parameters

array_h	array on device
fill	value to fill
size	size of the array

11.2.2.18 void hostArrayFillInt (int * *array_h*, int *fill*, int *size*)

Fill int array with given value.

Parameters

array_h	array on device
fill	value to fill
size	size of the array

11.2.2.19 void hostArrayFillRandom (FLOAT_TYPE * *array_h*, int *size*, FLOAT_TYPE *r* = `1.0`)

Fill float/double array with random value.

Parameters

array_h	array on device
size	size of the array
r	range of random number

11.3 GpuCollision.cu File Reference

Collision model.

```
#include "GpuFunctions.h"
#include "GpuConstants.h"
```

Functions

- __global__ void gpu_bgk (int *fluid_d, FLOAT_TYPE *fEq_d, FLOAT_TYPE *rho_d, FLOAT_TYPE *u_d, FLOAT_TYPE *v_d, FLOAT_TYPE *fColl_d, FLOAT_TYPE *f_d)

  BGKW collision model.
- __device__ FLOAT_TYPE feqc (FLOAT_TYPE u, FLOAT_TYPE uc, FLOAT_TYPE v, FLOAT_TYPE vc, FLOAT_TYPE rho, FLOAT_TYPE w)

  Compute the equilibrum distribution without the collision frequency.
- __global__ void gpuCollBgkw (int *fluid_d, FLOAT_TYPE *rho_d, FLOAT_TYPE *u_d, FLOAT_TYPE *v_d, FLOAT_TYPE *f_d, FLOAT_TYPE *fColl_d)

BGKW collision model (Bhatnagar–Gross–Krook-Welander)

- __global__ void gpuCollTrt (int *fluid_d, FLOAT_TYPE *rho_d, FLOAT_TYPE *u_d, FLOAT_TYPE *v_d, FLOAT_TYPE *f_d, FLOAT_TYPE *fColl_d)

  TRT collision model (two-relaxation-time)
- __global__ void gpu_trt1 (int *fluid_d, FLOAT_TYPE *fEq_d, FLOAT_TYPE *rho_d, FLOAT_TYPE *u_d, FLOAT_TYPE *v_d)

  TRT collision model (step 1)
- __global__ void gpu_trt2 (int *fluid_d, FLOAT_TYPE *fEq_d, FLOAT_TYPE *f_d, FLOAT_TYPE *fColl_d)

  TRT collision model (step 2))
- __global__ void gpuCollMrt (int *fluid_d, FLOAT_TYPE *rho_d, FLOAT_TYPE *u_d, FLOAT_TYPE *v_d, FLOAT_TYPE *f_d, FLOAT_TYPE *fColl_d)

  MRT collision model (multiple-relaxation-time)
- __global__ void gpu_mrt1 (int *fluid_d, FLOAT_TYPE *rho_d, FLOAT_TYPE *u_d, FLOAT_TYPE *v_d, FLOAT_TYPE *f_d, FLOAT_TYPE *mEq_d, FLOAT_TYPE *m_d)

  MRT collision model (step 1)
- __global__ void gpu_mrt2 (int *fluid_d, FLOAT_TYPE *collision_d, FLOAT_TYPE *m_d, FLOAT_TYPE *mEq_d, FLOAT_TYPE *fColl_d, FLOAT_TYPE *f_d)

  MRT collision model (step 2)

11.3.1 Detailed Description

Collision model.

Author

Adam Koleszar (adam.koleszar@gmail.com)

11.3.2 Function Documentation

11.3.2.1 __device__ FLOAT_TYPE feqc (FLOAT_TYPE *u,* FLOAT_TYPE *uc,* FLOAT_TYPE *v,* FLOAT_TYPE *vc,* FLOAT_TYPE *rho,* FLOAT_TYPE *w*)

Compute the equilibrum distribution without the collision frequency.

...

Todo doc: insert name or reference, explain method

Parameters

u,v	velocity
uc,vc	velocity component, see: cx_d cy_d
rho	density
w	lattice weight, see: w_d

Returns

equlibrum distribution function

11.3.2.2 __global__ void gpu_bgk (int * *fluid_d,* FLOAT_TYPE * *fEq_d,* FLOAT_TYPE * *rho_d,* FLOAT_TYPE * *u_d,* FLOAT_TYPE * *v_d,* FLOAT_TYPE * *fColl_d,* FLOAT_TYPE * *f_d*)

BGKW collision model.

Deprecated use gpuCollBgkw

Parameters

in	*fluid_d*	fluid condition
out	*fEq_d*	distribution function without frequency (unused)
in	*rho_d*	density
in	*u_d,v_d*	velocity vectors (x,y)
out	*fColl_d*	distribution function
in	*f_d*	distribution function from boundary step

latticeId

NOTE rho ∗ w ∗ (1 + 3(u∗ex + v∗ey) + 4.5(u∗ex + v∗ey)^2 - 1.5(u^2 + v^2))

NOTE collision_freq∗Feq + (1-coll_freq)∗F

11.3.2.3 __global__ void gpu_mrt1 (int ∗ *fluid_d*, FLOAT_TYPE ∗ *rho_d*, FLOAT_TYPE ∗ *u_d*, FLOAT_TYPE ∗ *v_d*, FLOAT_TYPE ∗ *f_d*, FLOAT_TYPE ∗ *mEq_d*, FLOAT_TYPE ∗ *m_d*)

MRT collision model (step 1)

Deprecated use gpuCollMrt

Parameters

in	*fluid_d*	fluid condition
in	*rho_d*	density
in	*u_d,v_d*	velocity vectors (x,y)
in	*f_d*	distribution function from boundary step
out	*m_d,mEq_d*	intermediary arrays

11.3.2.4 __global__ void gpu_mrt2 (int ∗ *fluid_d*, FLOAT_TYPE ∗ *collision_d*, FLOAT_TYPE ∗ *m_d*, FLOAT_TYPE ∗ *mEq_d*, FLOAT_TYPE ∗ *fColl_d*, FLOAT_TYPE ∗ *f_d*)

MRT collision model (step 2)

Deprecated use gpuCollMrt

Parameters

in	*fluid_d*	fluid condition
out	*fColl_d*	distribution function
in	*f_d*	distribution function from boundary step
in	*m_d,mEq_d*	intermediary arrays
out	*collision_d*	intermediary arrays (unused)

11.3.2.5 __global__ void gpu_trt1 (int ∗ *fluid_d*, FLOAT_TYPE ∗ *fEq_d*, FLOAT_TYPE ∗ *rho_d*, FLOAT_TYPE ∗ *u_d*, FLOAT_TYPE ∗ *v_d*)

TRT collision model (step 1)

Deprecated use gpuCollTrt

Parameters

in	*fluid_d*	fluid condition
out	*fEq_d*	distribution function without frequency
in	*rho_d*	density
in	*u_d,v_d*	velocity vectors (x,y)

11.3.2.6 __global__ void gpu_trt2 (int * *fluid_d*, FLOAT_TYPE * *fEq_d*, FLOAT_TYPE * *f_d*, FLOAT_TYPE * *fColl_d*)

TRT collision model (step 2))

Deprecated use gpuCollTrt

Parameters

in	*fluid_d*	fluid condition
in	*fEq_d*	distribution function without frequency
out	*fColl_d*	distribution function
in	*f_d*	distribution function from boundary step

11.4 include/Arguments.h File Reference

This file contains the acceptable arguments for the solver.

```
#include "FloatType.h"
```

Data Structures

- struct Arguments

  Input parameters.
- struct InputFilenames

  Input filenames.

Enumerations

- enum InletProfile { INLET =1, NO_INLET, PULSATILE_INLET }

  Inlet profile options.
- enum OutletProfile { OUTLET =1, OUTLET_SECOND, OUTLET_FIRST }

  Outlet profile options.
- enum CollisionModel { BGKW =1, TRT, MRT }

  Collision models.
- enum BoundaryType { CURVED =1, STRAIGHT }

  Boundary types.
- enum OutputFormat { PARAVIEW =1, TECPLOT }

  Output formats.
- enum ArgResult {
  NORMAL =0, HELP, INIT, TEST,
  COMPARE, ERROR }

  Result from command line argument parsing.

Functions

- ArgResult handleArguments (int argc, char **argv, InputFilenames *inFn, Arguments *args)

  Parse command line arguments.

11.4.1 Detailed Description

This file contains the acceptable arguments for the solver.

Author

Adam Koleszar (adam.koleszar@gmail.com)

11.4.2 Enumeration Type Documentation

11.4.2.1 enum ArgResult

Result from command line argument parsing.

Enumerator

NORMAL normal execution, parameters read

HELP print help

INIT read values from init file

TEST run unit tests

COMPARE compare results

ERROR error happened

11.4.2.2 enum BoundaryType

Boundary types.

Enumerator

CURVED curved boundaries

STRAIGHT straight boundaries

11.4.2.3 enum CollisionModel

Collision models.

Enumerator

BGKW BGKW method.

TRT TRT method.

MRT MRT method.

11.4.2.4 enum InletProfile

Inlet profile options.

Enumerator

INLET inlet profile

NO_INLET no inlet profile

PULSATILE_INLET pulsatile inlet profile

Warning

not implemented

11.4.2.5 enum OutletProfile

Outlet profile options.

Enumerator

OUTLET outlet profile

OUTLET_SECOND open boundary second order

OUTLET_FIRST open boundary first order

11.4.2.6 enum OutputFormat

Output formats.

Enumerator

PARAVIEW paraview format (.csv)

TECPLOT tecplot format (.dat)

11.4.3 Function Documentation

11.4.3.1 ArgResult handleArguments (int *argc*, char ** *argv*, InputFilenames * *inFn*, Arguments * *args*)

Parse command line arguments.

Parameters

argc	number of command line arguments
argv	command line arguments
inFn	input filenames
args	input parameters

Returns

task that need to be done, or error

Test MANUAL: In order to test argument handling try some of these cases

- run with -h, should display help
- run without arguments, should work as written in SetUpData.ini
- run with -f <ini> and any other argument, should work as written in SetUpData.ini

- run with -o <folder> should output results to given folder
- run with -i <n> should do iterations n times
- run with -n <file> with non-existent file, should print out error message
- try using long argument options as well, should work as the short version
- run in compare mode with one file given, should compare Results/FinalData.csv to given file
- run in compare mode with two files, should compare the 2 given files

11.5 include/ArrayUtils.h File Reference

Memory allocation wrappers for host and gpu arrays.

```
#include <stdbool.h>
#include "FloatType.h"
```

Macros

- #define THREADS 256

  number of threads used for kernels
- #define SIZEINT(a) ((a)∗sizeof(int))

  wraps sizeof(int)
- #define SIZEFLT(a) ((a)∗sizeof(FLOAT_TYPE))

  wraps sizeof(FLOAT_TYPE)

Enumerations

- enum ArrayOption {
  ARRAY_NONE, ARRAY_ZERO, ARRAY_FILL, ARRAY_COPY,
  ARRAY_CPYD, ARRAY_RAND }

  Array creation options.

Functions

- FLOAT_TYPE getRandom (unsigned long ∗seed)

  Get a uniform random number.
- int ∗ createHostArrayInt (int length, ArrayOption op=ARRAY_NONE, int fill=0, int ∗copy=NULL)

  Create 1D int array on the host.
- int ∗ createGpuArrayInt (int length, ArrayOption op=ARRAY_NONE, int fill=0, int ∗copy=NULL)

  Create 1D int array on the device.
- FLOAT_TYPE ∗ createHostArrayFlt (int length, ArrayOption op=ARRAY_NONE, FLOAT_TYPE fill=0, FLOAT_TYPE ∗copy=NULL)

  Create 1D float/double array on the device.
- FLOAT_TYPE ∗ createGpuArrayFlt (int length, ArrayOption op=ARRAY_NONE, FLOAT_TYPE fill=0, FLOAT_TYPE ∗copy=NULL)

  Create 1D float/double array on the device.
- int ∗∗ create2DHostArrayInt (int width, int height)

  Allocate 2D integer array.
- FLOAT_TYPE ∗∗ create2DHostArrayFlt (int width, int height)

  Allocate 2D floating point array.
- int ∗∗∗ create3DHostArrayInt (int width, int height, int depth)

Allocate 3D integer array.

- FLOAT_TYPE *** create3DHostArrayFlt (int width, int height, int depth)

  Allocate 3D floating point array.

- bool *** create3DHostArrayBool (int width, int height, int depth)

  Allocate 3D boolean array.

- void hostArrayFillInt (int *array_h, int fill, int size)

  Fill int array with given value.

- __global__ void gpuArrayFillInt (int *array_d, int size)

  GPU kernel to fill int array with given value.

- void hostArrayFillFlt (FLOAT_TYPE *array_h, FLOAT_TYPE fill, int size)

  Fill float/double array with given value.

- __global__ void gpuArrayFillFlt (FLOAT_TYPE *array_d, int size)

  GPU kernel to fill float/double array with given value.

- void hostArrayFillRandom (FLOAT_TYPE *array_h, int size, FLOAT_TYPE r=1.0)

  Fill float/double array with random value.

- __global__ void gpuArrayFillRandom (FLOAT_TYPE *array_d, unsigned long seed, int size)

  GPU kernel to fill float/double array with random value.

- void freeAllHost (void **as, int n)

  Free all host arrays.

- void freeAllGpu (void **as, int n)

  Free all GPU arrays.

11.5.1 Detailed Description

Memory allocation wrappers for host and gpu arrays.

Author

Adam Koleszar (adam.koleszar@gmail.com)

11.5.2 Macro Definition Documentation

11.5.2.1 #define SIZEFLT(*a*) ((a)*sizeof(FLOAT_TYPE))

wraps sizeof(FLOAT_TYPE)

Parameters

a	number of floats/doubles

11.5.2.2 #define SIZEINT(*a*) ((a)*sizeof(int))

wraps sizeof(int)

Parameters

a	number of ints

11.5.3 Enumeration Type Documentation

11.5.3.1 enum ArrayOption

Array creation options.

Enumerator

ARRAY_NONE Simple allocation.
ARRAY_ZERO Allocation and memset to zero.
ARRAY_FILL Allocation and fill with given value.
ARRAY_COPY Allocation and copy from given array.
ARRAY_CPYD Allocation and copy from given GPU array.
ARRAY_RAND Allocation and fill with random values.

11.5.4 Function Documentation

11.5.4.1 FLOAT_TYPE** create2DHostArrayFlt (int *width*, int *height*)

Allocate 2D floating point array.

Note

not used

Parameters

in	*width*	array width
in	*height*	array height

Returns

pointer to the allocated array

11.5.4.2 int** create2DHostArrayInt (int *width*, int *height*)

Allocate 2D integer array.

Note

not used

Parameters

in	*width*	array width
in	*height*	array height

Returns

pointer to the allocated array

11.5.4.3 bool*** create3DHostArrayBool (int *width*, int *height*, int *depth*)

Allocate 3D boolean array.

Note

not used

Parameters

in	*width*	array width
in	*height*	array height
in	*depth*	array depth

Returns

pointer to the allocated array

11.5.4.4 FLOAT_TYPE*** create3DHostArrayFlt (int *width*, int *height*, int *depth*)

Allocate 3D floating point array.

Note

not used

Parameters

in	*width*	array width
in	*height*	array height
in	*depth*	array depth

Returns

pointer to the allocated array

11.5.4.5 int*** create3DHostArrayInt (int *width*, int *height*, int *depth*)

Allocate 3D integer array.

Note

not used

Parameters

in	*width*	array width
in	*height*	array height
in	*depth*	array depth

Returns

pointer to the allocated array

11.5.4.6 FLOAT_TYPE* createGpuArrayFlt (int *length*, ArrayOption *op* = ARRAY_NONE, FLOAT_TYPE *fill* = 0, FLOAT_TYPE * *copy* = NULL)

Create 1D float/double array on the device.

Parameters

length	length of the array
op	options (default: none)
fill	value to fill array (default: 0)
copy	host array to copy into the new array (default: NULL)

Returns

new array

11.5.4.7 int* createGpuArrayInt (int *length*, ArrayOption *op* = ARRAY_NONE, int *fill* = 0, int * *copy* = NULL)

Create 1D int array on the device.

Parameters

length	length of the array
op	options (default: none)
fill	value to fill array (default: 0)
copy	host array to copy into the new array (default: NULL)

Returns

new array

11.5.4.8 FLOAT_TYPE* createHostArrayFlt (int *length*, ArrayOption *op* = ARRAY_NONE, FLOAT_TYPE *fill* = 0, FLOAT_TYPE * *copy* = NULL)

Create 1D float/double array on the device.

Parameters

length	length of the array
op	options (default: none)
fill	value to fill array (default: 0)
copy	array to copy into the new array (default: NULL)

Returns

new array

11.5.4.9 int* createHostArrayInt (int *length*, ArrayOption *op* = ARRAY_NONE, int *fill* = 0, int * *copy* = NULL)

Create 1D int array on the host.

Parameters

length	length of the array
op	options (default: none)
fill	value to fill array (default: 0)
copy	array to copy into the new array (default: NULL)

Returns

new array

11.5.4.10 void freeAllGpu (void ∗∗ *as*, int *n*)

Free all GPU arrays.

Parameters

as	array of arrays
n	number of arrays

11.5.4.11 void freeAllHost (void ∗∗ *as*, int *n*)

Free all host arrays.

Parameters

as	array of arrays
n	number of arrays

11.5.4.12 FLOAT_TYPE getRandom (unsigned long ∗ *seed*)

Get a uniform random number.

Parameters

seed	seed for the random number (initialise with time(NULL))

Returns

uniform random number

11.5.4.13 __global__ void gpuArrayFillFlt (FLOAT_TYPE ∗ *array_d*, int *size*)

GPU kernel to fill float/double array with given value.

Parameters

array_d	array on device
size	size of the array

Warning

fill value should be set into variable fill_fd in constant memory

11.5.4.14 __global__ void gpuArrayFillInt (int ∗ *array_d*, int *size*)

GPU kernel to fill int array with given value.

Parameters

array_d	array on device
size	size of the array

Warning

fill value should be set into variable fill_id in constant memory

11.5.4.15 __global__ void gpuArrayFillRandom (FLOAT_TYPE ∗ *array_d*, unsigned long *seed*, int *size*)

GPU kernel to fill float/double array with random value.

Parameters

array_d	array on device
seed	seed for random generator (spiked with global id)
size	size of the array

11.5.4.16 void hostArrayFillFlt (FLOAT_TYPE * *array_h*, FLOAT_TYPE *fill*, int *size*)

Fill float/double array with given value.

Parameters

array_h	array on device
fill	value to fill
size	size of the array

11.5.4.17 void hostArrayFillInt (int * *array_h*, int *fill*, int *size*)

Fill int array with given value.

Parameters

array_h	array on device
fill	value to fill
size	size of the array

11.5.4.18 void hostArrayFillRandom (FLOAT_TYPE * *array_h*, int *size*, FLOAT_TYPE *r* = `1.0`)

Fill float/double array with random value.

Parameters

array_h	array on device
size	size of the array
r	range of random number

11.6 include/BcMacros.h File Reference

Macros to manipulate boundary conditions bitmask.

Macros

- #define BC_WALL_E 0x00001

  BC: east (1) wall.
- #define BC_INLT_E 0x00002

  BC: east (1) inlet.
- #define BC_OUTL_E 0x00003

  BC: east (1) outlet.
- #define BC_WALL_N 0x00004

  BC: north (2) wall.
- #define BC_INLT_N 0x00008

  BC: north (2) inlet.

- #define BC_OUTL_N 0x0000C

  BC: north (2) outlet.
- #define BC_WALL_W 0x00010

  BC: west (3) wall.
- #define BC_INLT_W 0x00020

  BC: west (3) inlet.
- #define BC_OUTL_W 0x00030

  BC: west (3) outlet.
- #define BC_WALL_S 0x00040

  BC: south (4) wall.
- #define BC_INLT_S 0x00080

  BC: south (4) inlet.
- #define BC_OUTL_S 0x000C0

  BC: south (4) outlet.
- #define BC_WALL_NE 0x00100

  BC: north-east (5) wall.
- #define BC_INLT_NE 0x00200

  BC: north-east (5) inlet.
- #define BC_OUTL_NE 0x00300

  BC: north-east (5) outlet.
- #define BC_WALL_NW 0x00400

  BC: north-west (6) wall.
- #define BC_INLT_NW 0x00800

  BC: north-west (6) inlet.
- #define BC_OUTL_NW 0x00C00

  BC: north-west (6) outlet.
- #define BC_WALL_SW 0x01000

  BC: south-west (7) wall.
- #define BC_INLT_SW 0x02000

  BC: south-west (7) inlet.
- #define BC_OUTL_SW 0x03000

  BC: south-west (7) outlet.
- #define BC_WALL_SE 0x04000

  BC: south-east (8) wall.
- #define BC_INLT_SE 0x08000

  BC: south-east (8) inlet.
- #define BC_OUTL_SE 0x0C000

  BC: south-east (8) outlet.
- #define BC_WALL_B 0x10000

  BC: main (0) wall.
- #define BC_INLT_B 0x20000

  BC: main (0) inlet.
- #define BC_OUTL_B 0x30000

  BC: main (0) oulet.
- #define BC_CORNER 0x40000

  BC: corner.
- #define BC_FLUID 0x80000

  BC: fluid.
- #define BC_NONE 0x0

  BC: type none (00)
- #define BC_WALL 0x1

BC: type wall (01)

- #define BC_INLT 0x2

  BC: type inlet (10)
- #define BC_OUTL 0x3

  BC: type oulet (11)
- #define BC_ALL 0x3

  BC: type all/any (11)
- #define BC_E(i) ((i)<< 0)

  BC: set type to east.
- #define BC_N(i) ((i)<< 2)

  BC: set type to north.
- #define BC_W(i) ((i)<< 4)

  BC: set type to west.
- #define BC_S(i) ((i)<< 6)

  BC: set type to south.
- #define BC_NE(i) ((i)<< 8)

  BC: set type to north-east.
- #define BC_NW(i) ((i)<<10)

  BC: set type to north-west.
- #define BC_SW(i) ((i)<<12)

  BC: set type to south-west.
- #define BC_SE(i) ((i)<<14)

  BC: set type to south-east.
- #define BC_B(i) ((i)<<16)

  BC: set type to main.
- #define BC_C(i) ((i)<<18)

  BC: set type to corner.
- #define BC_F(i) ((i)<<19)

  BC: set type to fluid.
- #define BC_MASK(type, dir) ((type) << (((dir)-1) * 2))

  BC: set type to specified direction.
- #define BND_ID_ALL 0xFF000000

  BC: boundary id all.
- #define BOUND_ID(i) ((i)<<24)

  BC: set boundary id for lift/drag.

11.6.1 Detailed Description

Macros to manipulate boundary conditions bitmask.

Author

Adam Koleszar (adam.koleszar@gmail.com)

The bitmask looks the following way

```
| 30 | 28 | 26 | 24 | 22 | 20 | 18 | 16 | 14 | 12 | 10 |  8 |  6 |  4 |  2 |  0 |
|-------------------|---------|----|----|----|----|----|----|----|----|----|----|----|
| boundary ID (8bit)|  empty  | FC | M0 | SE | SW | NW | NE |  S |  W |  N |  E |
```

11.6.2 Macro Definition Documentation

11.6.2.1 #define BC_B(*i*) ((i)<<16)

BC: set type to main.

Parameters

i	BC type

11.6.2.2 #define BC_C(*i*) ((i)<<18)

BC: set type to corner.

Parameters

i	true/false

11.6.2.3 #define BC_E(*i*) ((i)<< 0)

BC: set type to east.

Parameters

i	BC type

11.6.2.4 #define BC_F(*i*) ((i)<<19)

BC: set type to fluid.

Parameters

i	true/false

11.6.2.5 #define BC_MASK(*type, dir*) ((type) << (((dir)-1) * 2))

BC: set type to specified direction.

Parameters

type	BC type
dir	direction

Returns

bitmask

11.6.2.6 #define BC_N(*i*) ((i)<< 2)

BC: set type to north.

Parameters

i	BC type

11.6.2.7 #define BC_NE(*i*) ((i)<< 8)

BC: set type to north-east.

Parameters

i	BC type

11.6.2.8 #define BC_NW(*i*) ((i)<<10)

BC: set type to north-west.

Parameters

i	BC type

11.6.2.9 #define BC_S(*i*) ((i)<< 6)

BC: set type to south.

Parameters

i	BC type

11.6.2.10 #define BC_SE(*i*) ((i)<<14)

BC: set type to south-east.

Parameters

i	BC type

11.6.2.11 #define BC_SW(*i*) ((i)<<12)

BC: set type to south-west.

Parameters

i	BC type

11.6.2.12 #define BC_W(*i*) ((i)<< 4)

BC: set type to west.

Parameters

i	BC type

11.6.2.13 #define BOUND_ID(*i*) ((i)<<24)

BC: set boundary id for lift/drag.

Parameters

i	boundary ID

11.7 include/CellFunctions.h File Reference

This file contains initialisation for the MRT collision model.

```
#include "FloatType.h"
```

Functions

- void MRTInitializer (FLOAT_TYPE *velMomMap, FLOAT_TYPE *momCollMtx, FLOAT_TYPE omega)

  This function is responsible for the MRT initialization.

11.7.1 Detailed Description

This file contains initialisation for the MRT collision model.

Author

Istvan Tamas Jozsa (jozsait@gmail.com)

11.7.2 Function Documentation

11.7.2.1 void MRTInitializer (FLOAT_TYPE * *velMomMap,* FLOAT_TYPE * *momCollMtx,* FLOAT_TYPE *omega*)

This function is responsible for the MRT initialization.

The MRT collision model computes new values for the distribution function in momentum space and uses the following formula:

$$f_i = f_i - \mathbf{M}^{-1}\mathbf{S}(\mathbf{m} - \mathbf{m}^{eq})$$

$$\mathbf{S} = diag(1.0, 1.4, 1.4, 1.0, 1.2, 1.0, 1.2, \omega, \omega)$$

For more details check A. A. Mohamad - Lattice Boltzmann Method (book)

Parameters

out	*velMomMap*	mapping between velocity and momentum space $\mathbf{M}$
out	*momCollMtx*	collision matrix in momentum space $\mathbf{M}^{-1}\mathbf{S}$
	omega	collision frequency

11.8 include/Check.h File Reference

Check CUDA API calls.

Macros

- #define CHECK(err) check(err, __FILE__, __LINE__)

  Check macro for easier access.

Functions

- void check (cudaError err, const char *location, int lineno)

  Check return value of CUDA API calls.

11.8.1 Detailed Description

Check CUDA API calls.

Author

Adam Koleszar (adam.koleszar@gmail.com)

11.8.2 Macro Definition Documentation

11.8.2.1 #define CHECK(*err*) check(err, __FILE__, __LINE__)

Check macro for easier access.

Parameters

err	return value from API call

11.8.3 Function Documentation

11.8.3.1 void check (cudaError *err,* const char * *location,* int *lineno*)

Check return value of CUDA API calls.

Parameters

err	return value from API call
location	source file where check was called
lineno	line number where check was called

Warning

exits program if error occurs without cleanup

11.9 include/ComputeResiduals.h File Reference

Functions for residual computations and result comparing.

```
#include "FloatType.h"
```

Functions

- void ComputeResiduals (int *boundaryId, FLOAT_TYPE *f, FLOAT_TYPE *fColl, FLOAT_TYPE *drag, FLOAT_TYPE *lift, FLOAT_TYPE *residuals, int *m, int *n, int computeDragLift)

  This function compute the residuals.
- __host__ void GpuComputeResiduals (int *boundaryId_d, FLOAT_TYPE *f_d, FLOAT_TYPE *fColl_d, FLOAT_TYPE *drag_d, FLOAT_TYPE *lift_d, FLOAT_TYPE *temp9a_d, FLOAT_TYPE *temp9b_d, FLOAT_TYPE *tempA_d, FLOAT_TYPE *tempB_d, FLOAT_TYPE *residuals, int *m, int *n, int computeDragLift)

  Compute the residuals on the GPU.
- __host__ void GpuComputeResidMask (int *bcMask_d, FLOAT_TYPE *f_d, FLOAT_TYPE *fColl_d, FLOAT_TYPE *drag_d, FLOAT_TYPE *lift_d, FLOAT_TYPE *temp9a_d, FLOAT_TYPE *temp9b_d, FLOAT_TYPE *tempA_d, FLOAT_TYPE *tempB_d, FLOAT_TYPE *residuals, int *m, int *n, int computeDragLift)

  Compute the residual on the GPU for BC bitmask.
- __host__ FLOAT_TYPE computeResidual (FLOAT_TYPE *f_d, FLOAT_TYPE *fColl_d, FLOAT_TYPE *temp9a_d, FLOAT_TYPE *temp9b_d, int m, int n)

Compute residual on GPU.

- __host__ FLOAT_TYPE computeDragLift (int *bcMask_d, FLOAT_TYPE *dl_d, FLOAT_TYPE *tempA_d, FLOAT_TYPE *tempB_d, int m, int n, int boundaryId)

  Compute drag or lift on GPU.

11.9.1 Detailed Description

Functions for residual computations and result comparing.

Author

Adam Koleszar (adam.koleszar@gmail.com)

11.9.2 Function Documentation

11.9.2.1 __host__ FLOAT_TYPE computeDragLift (int * *bcMask_d*, FLOAT_TYPE * *dl_d*, FLOAT_TYPE * *tempA_d*, FLOAT_TYPE * *tempB_d*, int *m*, int *n*, int *boundaryId*)

Compute drag or lift on GPU.

Parameters

bcMask_d	BC bitmask
dl_d	drag or lift
tempA_d,tempB_d	temporary array for summation (size: MxN)
m	number of columns
n	number of rows
boundaryId	boundary to calculate drag/lift on

Returns

drag/lift

11.9.2.2 __host__ FLOAT_TYPE computeResidual (FLOAT_TYPE * *f_d*, FLOAT_TYPE * *fColl_d*, FLOAT_TYPE * *temp9a_d*, FLOAT_TYPE * *temp9b_d*, int *m*, int *n*)

Compute residual on GPU.

compute the norm of the difference of f and fColl

Parameters

f_d	distribution function
fColl_d	distribution function collision step
temp9a_d,temp9b_d	temporary array for summation (size: 9xMxN)
m	number of columns
n	number of rows

Returns

norm of the difference

11.9.2.3 void ComputeResiduals (int * *boundaryId*, FLOAT_TYPE * *f*, FLOAT_TYPE * *fColl*, FLOAT_TYPE * *drag*, FLOAT_TYPE * *lift*, FLOAT_TYPE * *residuals*, int * *m*, int * *n*, int *computeDragLift*)

This function compute the residuals.

Warning

Exits the whole program if the iteration diverges

Parameters

in	*boundaryId*	boundary types
in	*f*	distribution function
in	*fColl*	distribution function (previous step)
in	*drag*	drag
in	*lift*	lift
out	*residuals*	array for residuals and lift/drag
in	*m*	number of columns
in	*n*	number of rows
in	*computeDragLift*	boundary ID for drag/lift computation

Deprecated use GpuComputeResidMask instead

11.9.2.4 __host__ void GpuComputeResidMask (int * *bcMask_d*, FLOAT_TYPE * *f_d*, FLOAT_TYPE * *fColl_d*, FLOAT_TYPE * *drag_d*, FLOAT_TYPE * *lift_d*, FLOAT_TYPE * *temp9a_d*, FLOAT_TYPE * *temp9b_d*, FLOAT_TYPE * *tempA_d*, FLOAT_TYPE * *tempB_d*, FLOAT_TYPE * *residuals*, int * *m*, int * *n*, int *computeDragLift*)

Compute the residual on the GPU for BC bitmask.

Warning

Exits the whole program if the iteration diverges

Parameters

bcMask_d	BC bitmask
f_d	distribution function
fColl_d	distribution function (previous step)
drag_d	drag
lift_d	lift
temp9a_d,temp9b_d	temporary array for summation (size: 9xMxN)
tempA_d,tempB_d	temporary array for summation (size: MxN)
residuals	array for residuals and lift/drag
m	number of columns
n	number of rows
computeDragLift	boundary ID for drag/lift computation

11.9.2.5 __host__ void GpuComputeResiduals (int * *boundaryId_d*, FLOAT_TYPE * *f_d*, FLOAT_TYPE * *fColl_d*, FLOAT_TYPE * *drag_d*, FLOAT_TYPE * *lift_d*, FLOAT_TYPE * *temp9a_d*, FLOAT_TYPE * *temp9b_d*, FLOAT_TYPE * *tempA_d*, FLOAT_TYPE * *tempB_d*, FLOAT_TYPE * *residuals*, int * *m*, int * *n*, int *computeDragLift*)

Compute the residuals on the GPU.

Warning

Exits the whole program if the iteration diverges

Deprecated use GpuComputeResidMask

Parameters

in	*boundaryId_d*	boundary types
in	*f_d*	distribution function
in	*fColl_d*	distribution function (previous step)
in	*drag_d*	drag
in	*lift_d*	lift
in	*temp9a_d,temp9b_d*	temporary array for summation (size: 9xMxN)
in	*tempA_d,tempB_d*	temporary array for summation (size: MxN)
out	*residuals*	array for residuals and lift/drag
in	*m*	number of columns
in	*n*	number of rows
in	*computeDragLift*	boundary ID for drag/lift computation

11.10 include/FilesReading.h File Reference

File reading functions.

```
#include "FloatType.h"
#include "Arguments.h"
```

Functions

- int getNumberOfLines (const char *filename)

  Get the number of lines in a file.
- void readInitFile (const char *filename, Arguments *args)

  Read init file (usually SetUpData.ini)
- int readNodeFile (const char *filename, int **ni, int **nj, FLOAT_TYPE **nx, FLOAT_TYPE **ny, int **nf)

  Read node file (usually D2node.dat) Arrays supplied are allocated inside, freeing these arrays is the caller's responsibility.
- int readConnFile (const char *filename, int **ni, int **nj, int **dir, int **bc, FLOAT_TYPE **bcx, FLOAT_TYPE **bcy, int **id)

  Read boundary conditions file (usually BCconnectors.dat) Arrays supplied are allocated inside, freeing these arrays is the caller's responsibility.
- int readResultFile (const char *filename, FLOAT_TYPE ***results, int **fluid)

  Read result file (usually FinalData.csv)
- __host__ int compareFiles (const char *f1, const char *f2)

  Compare result files.
- int getLastValue (int *arr, int n)

  Gets the last value of the array Use it with nodeIdx to get number rows, and with nodeIdy to get number of columns.
- FLOAT_TYPE getGridSpacing (int *ni, int *nj, FLOAT_TYPE *nx, int n)

  Returns the grid spacing of the mesh.
- int getNumInletNodes (int *bc, int *dir, int n)

  Get number of inlet nodes.

- FLOAT_TYPE getMaxInletCoordY (int *bc, int *dir, FLOAT_TYPE *bcy, FLOAT_TYPE delta, int n)

  Get the maximum coordinate of the inlet in the Y direction.
- FLOAT_TYPE getMinInletCoordY (int *bc, int *dir, FLOAT_TYPE *bcy, FLOAT_TYPE delta, int n)

  Get the minimum coordinate of the inlet in the Y direction.
- void CompDataNode (FLOAT_TYPE *Delta, int *m, int *n, int *Nodes0, int *Nodes1, FLOAT_TYPE *Nodes2, FLOAT_TYPE *Nodes3, int *Nodes4, int *NumNodes)

  Compute values from node data.
- void CompDataConn (int *NumInletNodes, FLOAT_TYPE *MaxInletCoordY, FLOAT_TYPE *MinInletCoordY, int *BCconn0, int *BCconn1, int *BCconn2, int *BCconn3, FLOAT_TYPE *BCconn4, FLOAT_TYPE *BCconn5, int *BCconn6, int *NumConn, FLOAT_TYPE *Delta)

  Compute values from BC data.

11.10.1 Detailed Description

File reading functions.

Author

Adam Koleszar (adam.koleszar@gmail.com)

11.10.2 Function Documentation

11.10.2.1 __host__ int compareFiles (const char * *f1*, const char * *f2*)

Compare result files.

Note

Use it to compare FinalData.csv with previous result.

Parameters

f1,f2	files to compare

Returns

0 for identical files and 1 for different files

11.10.2.2 void CompDataConn (int * *NumInletNodes*, FLOAT_TYPE * *MaxInletCoordY*, FLOAT_TYPE * *MinInletCoordY*, int * *BCconn0*, int * *BCconn1*, int * *BCconn2*, int * *BCconn3*, FLOAT_TYPE * *BCconn4*, FLOAT_TYPE * *BCconn5*, int * *BCconn6*, int * *NumConn*, FLOAT_TYPE * *Delta*)

Compute values from BC data.

Deprecated use getNumInletNodes, getMaxInletCoordY and getMinInletCoordY instead

Parameters

out	*NumInletNodes*	number of inlet nodes
out	*MaxInletCoordY*	maximum inlet coordinate y

out	*MinInletCoordY*	minimum inlet coordinate y
in	*BCconn0*	node id x
in	*BCconn1*	node id y
in	*BCconn2*	lattice id (direction)
in	*BCconn3*	boundary type
in	*BCconn4*	BC coordinate x
in	*BCconn5*	BC coordinate y
in	*BCconn6*	BC ID for drag/lift
in	*NumConn*	number of BC
in	*Delta*	grid spacing

11.10.2.3 void CompDataNode (FLOAT_TYPE * *Delta*, int * *m*, int * *n*, int * *Nodes0*, int * *Nodes1*, FLOAT_TYPE * *Nodes2*, FLOAT_TYPE * *Nodes3*, int * *Nodes4*, int * *NumNodes*)

Compute values from node data.

Deprecated use getLastValue and getGridSpacing instead

Parameters

out	*Delta*	grid spacing
out	*m*	number of columns
out	*n*	number of rows
in	*Nodes0*	node id x
in	*Nodes1*	node id y
in	*Nodes2*	node coordinate x
in	*Nodes3*	node coordinate y
in	*Nodes4*	node type
in	*NumNodes*	number of nodes

11.10.2.4 FLOAT_TYPE getGridSpacing (int * *ni*, int * *nj*, FLOAT_TYPE * *nx*, int *n*)

Returns the grid spacing of the mesh.

Parameters

ni,nj	node ID (x,y)
nx	node coordinate x
n	number of nodes

Returns

grid spacing

11.10.2.5 int getLastValue (int * *arr*, int *n*)

Gets the last value of the array Use it with nodeIdx to get number rows, and with nodeIdy to get number of columns.

Parameters

arr	input array
n	array size

Returns

last element plus one

11.10.2.6 FLOAT_TYPE getMaxInletCoordY (int * *bc*, int * *dir*, FLOAT_TYPE * *bcy*, FLOAT_TYPE *delta*, int *n*)

Get the maximum coordinate of the inlet in the Y direction.

Parameters

bc	BC type (1: wall, 2: inlet, 3: outlet)
dir	direction
bcy	BC coordinate y
delta	grid spacing
n	number of BC nodes

Returns

maximum coordinate of the inlet in the Y direction

11.10.2.7 FLOAT_TYPE getMinInletCoordY (int * *bc*, int * *dir*, FLOAT_TYPE * *bcy*, FLOAT_TYPE *delta*, int *n*)

Get the minimum coordinate of the inlet in the Y direction.

Parameters

bc	BC type (1: wall, 2: inlet, 3: outlet)
dir	direction
bcy	BC coordinate y
delta	grid spacing
n	number of BC nodes

Returns

minimum coordinate of the inlet in the Y direction

11.10.2.8 int getNumberOfLines (const char * *filename*)

Get the number of lines in a file.

Parameters

in	*filename*	filename

Returns

number of lines

11.10.2.9 int getNumInletNodes (int * *bc*, int * *dir*, int *n*)

Get number of inlet nodes.

Parameters

bc	BC type (1: wall, 2: inlet, 3: outlet)
dir	direction
n	number of BC nodes

Returns

number of inlet nodes

11.10.2.10 int readConnFile (const char * *filename,* int ** *ni,* int ** *nj,* int ** *dir,* int ** *bc,* FLOAT_TYPE ** *bcx,* FLOAT_TYPE ** *bcy,* int ** *id*)

Read boundary conditions file (usually BCconnectors.dat) Arrays supplied are allocated inside, freeing these arrays is the caller's responsibility.

```
6   2   5
  \ | /
3 - 0 - 1
  / | \
7   4   8
```

Parameters

in	*filename*	filename
out	*ni,nj*	node ID (x,y)
out	*dir*	direction
out	*bc*	BC type (1: wall, 2: inlet, 3: outlet)
out	*bcx,bcy*	BC coordinate (x,y)
out	*id*	boundary ID (for drag/lift computation)

Returns

number of lines read

11.10.2.11 void readInitFile (const char * *filename,* Arguments * *args*)

Read init file (usually SetUpData.ini)

Parameters

in	*filename*	filename
out	*args*	input parameters

11.10.2.12 int readNodeFile (const char * *filename,* int ** *ni,* int ** *nj,* FLOAT_TYPE ** *nx,* FLOAT_TYPE ** *ny,* int ** *nf*)

Read node file (usually D2node.dat) Arrays supplied are allocated inside, freeing these arrays is the caller's responsibility.

Parameters

in	*filename*	filename
out	*ni,nj*	node ID (x,y)
out	*nx,ny*	node coordinate (x,y)
out	*nf*	node type (0: solid, 1:fluid)

Returns

number of lines read

11.10.2.13 int readResultFile (const char * *filename,* FLOAT_TYPE *** *results,* int ** *fluid*)

Read result file (usually FinalData.csv)

Parameters

in	*filename*	filename
out	*results*	coordinates (x,y), velocity (u,v), vel_mag, rho, pressure
out	*fluid*	node type (0: solid, 1:fluid)

Returns

number of lines read

11.11 include/FilesWriting.h File Reference

Functions for file writing.

```
#include "FloatType.h"
#include "Arguments.h"
```

Functions

- void WriteResults (char *OutputFile, FLOAT_TYPE *CoordX, FLOAT_TYPE *CoordY, FLOAT_TYPE *U, FLOAT_TYPE *V, FLOAT_TYPE *Rho, int *Fluid, int n, int m, OutputFormat outputFormat)

  Print results to output file.

11.11.1 Detailed Description

Functions for file writing.

Author

Istvan Tamas Jozsa (jozsait@gmail.com)

11.11.2 Function Documentation

11.11.2.1 void WriteResults (char * *OutputFile,* FLOAT_TYPE * *CoordX,* FLOAT_TYPE * *CoordY,* FLOAT_TYPE * *U,* FLOAT_TYPE * *V,* FLOAT_TYPE * *Rho,* int * *Fluid,* int *n,* int *m,* OutputFormat *outputFormat*)

Print results to output file.

Parameters

OutputFile	name of the outpu file
CoordX,CoordY	x,y coordinates
U,V	velocity x,y coordinates
Rho	density
Fluid	node type (0: solid, 1: fluid)
n	number of rows
m	number of columns
outputFormat	output file format

11.12 include/FloatType.h File Reference

File to select floating point precision.

Macros

- #define FLOAT_TYPE float
  floating point type (float/double)
- #define FLOAT_FORMAT "%f"
  floating point formatting string for scanf

11.12.1 Detailed Description

File to select floating point precision. Usage: make FLOAT_TYPE=USE_DOUBLE <target>

Default: USE_SINGLE

Author

Adam Koleszar (adam.koleszar@gmail.com)

11.13 include/GpuConstants.h File Reference

Header for variables stored in GPU constant memory.

Variables

- __constant__ InletProfile inletProfile_d
  inlet profile
- __constant__ BoundaryType boundaryType_d
  boundary type
- __constant__ OutletProfile outletProfile_d
  outlet profile
- __constant__ int dlBoundaryId_d
  boudary ID
- __constant__ int cx_d [9]
  velocity x unit vector components
- __constant__ int cy_d [9]
  velocity y unit vector components

- __constant__ int width_d

  number of columns
- __constant__ int height_d

  number of rows
- __constant__ int c_d [9]

  direction offset
- __constant__ int opp_d [9]

  opposite lattice offset
- __constant__ FLOAT_TYPE delta_d

  grid spacing
- __constant__ FLOAT_TYPE w_d [9]

  lattice weights
- __constant__ FLOAT_TYPE omega_d

  collision frequency for D2Q9 $\omega = \frac{1}{3\nu+0.5}$
- __constant__ FLOAT_TYPE omegaA_d

  assymetric collision frequency $\omega_a = \frac{8(2-\omega)}{8-\omega}$
- __constant__ FLOAT_TYPE rhoIn_d

  input density
- __constant__ FLOAT_TYPE uIn_d

  input velocity x
- __constant__ FLOAT_TYPE vIn_d

  input velocity y
- __constant__ FLOAT_TYPE minInletCoordY_d

  maximum inlet coordinate y
- __constant__ FLOAT_TYPE maxInletCoordY_d

  minimum inlet coordinate y
- __constant__ FLOAT_TYPE velMomMap_d [81]

  MRT constants: mapping between velocity and momentum space $\mathbf{M}$.
- __constant__ FLOAT_TYPE momCollMtx_d [81]

  MRT constants: collision matrix in momentum space $\mathbf{M}^{-1}\mathbf{S}$.

11.13.1 Detailed Description

Header for variables stored in GPU constant memory.

Author

Adam Koleszar (adam.koleszar@gmail.com)

These constants need to be extern in other files because they can only be declared in one source file. If other object files need these variable, just include this header and don't forget to compile them with -rdc=true

Velocity unit vector components (cx_d, cy_d)

```
 (-1,1)    (0,1)    (1,1)
         6    2    5
           \  |  /
(-1,0)  3 -(0,0)- 1  (1,0)
           /  |  \
         7    4    8
 (-1,-1)   (0,-1)   (1,-1)
```

Lattice weights (w_d)

```
(1/36)   (1/9)   (1/36)
       6   2   5
         \ | /
(1/9) 3 -(4/9)- 1 (1/9)
         / | \
       7   4   8
(1/36)   (1/9)   (1/36)
```

Opposite lattices (opp_d)

```
     (8)   (4)   (7)
        6   2   5
          \ | /
    (1) 3 -(0)- 1 (3)
          / | \
        7   4   8
     (5)   (2)   (6)
```

11.14 include/GpuFunctions.h File Reference

Header for all the kernels.

```
#include <cuda.h>
#include <stdio.h>
#include "FloatType.h"
#include "Arguments.h"
```

Functions

- __host__ void initConstants (Arguments *args, FLOAT_TYPE maxInletCoordY, FLOAT_TYPE minInletCoordY, FLOAT_TYPE delta, int m, int n)

  Initialise GPU constants.
- __global__ void gpu_init (int *corner_d, FLOAT_TYPE *rho_d, FLOAT_TYPE *Q_d, int *boundary_d, FLOAT_TYPE *coordY_d, int *stream_d, int *bcId_d, FLOAT_TYPE *u_d, FLOAT_TYPE *v_d, FLOAT_TYPE *u0_d, FLOAT_TYPE *v0_d)

  Initialise boundary conditions and macroscopic values.
- __global__ void gpu_init_1 (FLOAT_TYPE *rho_d, FLOAT_TYPE *u0_d, FLOAT_TYPE *v0_d, FLOAT_TYPE *u_d, FLOAT_TYPE *v_d, FLOAT_TYPE *coordY_d, int size)

  Initialise macroscopic values.
- __global__ void gpuInitInletProfile (FLOAT_TYPE *u0_d, FLOAT_TYPE *v0_d, FLOAT_TYPE *y_d, int size)

  Initialise initial velocity vector for inlet profile.
- __global__ void gpu_init_8 (int *stream_d, FLOAT_TYPE *Q_d, int size)

  Initialise stream and q arrays.
- __host__ int initBoundaryConditions (int *bcNodeIdX, int *bcNodeIdY, FLOAT_TYPE *q, int *bcBoundId, int *fluid, FLOAT_TYPE *bcX, FLOAT_TYPE *bcY, FLOAT_TYPE *nodeX, FLOAT_TYPE *nodeY, int *latticeId, int *stream, int *bcType, int *bcMask, int *bcIdx, int *mask, FLOAT_TYPE delta, int m, int n, int size)

  Initialise boundary conditions.
- __host__ void collapseBc (int *bcIdx, int *bcIdxCollapsed_d, int *bcMask, int *bcMaskCollapsed_d, FLOAT_TYPE *q, FLOAT_TYPE *qCollapsed_d, int *mask, int m, int n, int size)

  Collapse boundary condition arrays.
- __global__ void gpuBcInlet (int *bcIdx_d, int *bcMask_d, FLOAT_TYPE *f_d, FLOAT_TYPE *u0_d, FLOAT_TYPE *v0_d, int size)

  Inlet boundary conditions.
- __global__ void gpuBcWall (int *bcIdx_d, int *bcMask_d, FLOAT_TYPE *f_d, FLOAT_TYPE *fColl_d, FLOAT_TYPE *Q_d, int size)

Wall boundary conditions.

- __global__ void gpuBcOutlet (int *bcIdx_d, int *bcMask_d, FLOAT_TYPE *f_d, FLOAT_TYPE *u0_d, FLOAT_TYPE *v0_d, int size)

  Outlet boundary conditions.
- __global__ void gpu_convert (int *bcIdx_d, int *bcMask_d, int *fluid_d, int *boundary_d, int *corner_d, int *bcId_d, int size)

  Convert new bitmask to the old boundary condition arrays.
- __global__ void gpu_bgk (int *fluid_d, FLOAT_TYPE *fEq_d, FLOAT_TYPE *rho_d, FLOAT_TYPE *u_d, FLOAT_TYPE *v_d, FLOAT_TYPE *fColl_d, FLOAT_TYPE *f_d)

  BGKW collision model.
- __global__ void gpuCollBgkw (int *fluid_d, FLOAT_TYPE *rho_d, FLOAT_TYPE *u_d, FLOAT_TYPE *v_d, FLOAT_TYPE *f_d, FLOAT_TYPE *fColl_d)

  BGKW collision model (Bhatnagar–Gross–Krook-Welander)
- __global__ void gpuCollTrt (int *fluid_d, FLOAT_TYPE *rho_d, FLOAT_TYPE *u_d, FLOAT_TYPE *v_d, FLOAT_TYPE *f_d, FLOAT_TYPE *fColl_d)

  TRT collision model (two-relaxation-time)
- __global__ void gpu_trt1 (int *fluid_d, FLOAT_TYPE *fEq_d, FLOAT_TYPE *rho_d, FLOAT_TYPE *u_d, FLOAT_TYPE *v_d)

  TRT collision model (step 1)
- __global__ void gpu_trt2 (int *fluid_d, FLOAT_TYPE *fEq_d, FLOAT_TYPE *f_d, FLOAT_TYPE *fColl_d)

  TRT collision model (step 2))
- __global__ void gpuCollMrt (int *fluid_d, FLOAT_TYPE *rho_d, FLOAT_TYPE *u_d, FLOAT_TYPE *v_d, FLOAT_TYPE *f_d, FLOAT_TYPE *fColl_d)

  MRT collision model (multiple-relaxation-time)
- __global__ void gpu_mrt1 (int *fluid_d, FLOAT_TYPE *rho_d, FLOAT_TYPE *u_d, FLOAT_TYPE *v_d, FLOAT_TYPE *f_d, FLOAT_TYPE *mEq_d, FLOAT_TYPE *m_d)

  MRT collision model (step 1)
- __global__ void gpu_mrt2 (int *fluid_d, FLOAT_TYPE *collision_d, FLOAT_TYPE *m_d, FLOAT_TYPE *mEq_d, FLOAT_TYPE *fColl_d, FLOAT_TYPE *f_d)

  MRT collision model (step 2)
- __global__ void gpu_streaming (int *fluid_d, int *stream_d, FLOAT_TYPE *f_d, FLOAT_TYPE *fColl_d)

  Streaming.
- __global__ void gpuStreaming (int *fluid_d, int *stream_d, FLOAT_TYPE *f_d, FLOAT_TYPE *fColl_d)

  Streaming.
- __global__ void gpu_boundaries1 (int *fluid_d, int *boundary_d, int *bcId_d, FLOAT_TYPE *f_d, FLOAT_TYPE *u0_d, FLOAT_TYPE *v0_d, int *corner_d)

  Inlet boundary conditions.
- __global__ void gpu_boundaries2 (int *fluid_d, FLOAT_TYPE *Fneighbours_d, FLOAT_TYPE *fColl_d, int *bcId_d, FLOAT_TYPE *Q_d, FLOAT_TYPE *f_d)

  Wall boundary conditions.
- __global__ void gpu_boundaries3 (int *fluid_d, int *bcId_d, FLOAT_TYPE *f_d, FLOAT_TYPE *u0_d, FLOAT_TYPE *v0_d, int *corner_d)

  Outlet boundary conditions.
- __global__ void gpu_update_macro (int *fluid_d, FLOAT_TYPE *rho_d, FLOAT_TYPE *u_d, FLOAT_TYPE *v_d, int *bcId_d, int *boundaryId_d, FLOAT_TYPE *drag_d, FLOAT_TYPE *lift_d, FLOAT_TYPE *coordX_d, FLOAT_TYPE *coordY_d, FLOAT_TYPE *f_d)

  Update macroscopic values.
- __global__ void gpu_update_new (int *fluid_d, FLOAT_TYPE *rho_d, FLOAT_TYPE *u_d, FLOAT_TYPE *v_d, int *bcMask_d, FLOAT_TYPE *drag_d, FLOAT_TYPE *lift_d, FLOAT_TYPE *coordX_d, FLOAT_TYPE *coordY_d, FLOAT_TYPE *f_d)

  Update macroscopic values using bitmask.
- __global__ void gpuUpdateMacro (int *fluid_d, FLOAT_TYPE *rho_d, FLOAT_TYPE *u_d, FLOAT_TYPE *v_d, int *bcMask_d, FLOAT_TYPE *drag_d, FLOAT_TYPE *lift_d, FLOAT_TYPE *coordX_d, FLOAT_TYPE *coordY_d, FLOAT_TYPE *f_d)

  Update macroscopic values using bitmask.

11.14.1 Detailed Description

Header for all the kernels.

Author

Adam Koleszar (adam.koleszar@gmail.com)

11.14.2 Function Documentation

11.14.2.1 __host__ void collapseBc (int * *bcIdx*, int * *bcIdxCollapsed_d*, int * *bcMask*, int * *bcMaskCollapsed_d*, FLOAT_TYPE * *q*, FLOAT_TYPE * *qCollapsed_d*, int * *mask*, int *m*, int *n*, int *size*)

Collapse boundary condition arrays.

Parameters

in	*bcIdx*	boundary condition indices
out	*bcIdxCollapsed_d*	boundary condition indeces collapsed
in	*bcMask*	[description]
out	*bcMaskCollapsed_d*	[description]
in	*q*	grid center ratio array
out	*qCollapsed_d*	grid center ratio array collapsed
in	*mask*	mask for collapsing
in	*m*	number of columns
in	*n*	number of rows
in	*size*	number of boundary conditions

11.14.2.2 __global__ void gpu_bgk (int * *fluid_d*, FLOAT_TYPE * *fEq_d*, FLOAT_TYPE * *rho_d*, FLOAT_TYPE * *u_d*, FLOAT_TYPE * *v_d*, FLOAT_TYPE * *fColl_d*, FLOAT_TYPE * *f_d*)

BGKW collision model.

Deprecated use gpuCollBgkw

Parameters

in	*fluid_d*	fluid condition
out	*fEq_d*	distribution function without frequency (unused)
in	*rho_d*	density
in	*u_d,v_d*	velocity vectors (x,y)
out	*fColl_d*	distribution function
in	*f_d*	distribution function from boundary step

latticeId

NOTE rho * w * (1 + 3(u*ex + v*ey) + 4.5(u*ex + v*ey)^2 - 1.5(u^2 + v^2))

NOTE collision_freq*Feq + (1-coll_freq)*F

11.14.2.3 __global__ void gpu_boundaries1 (int * *fluid_d*, int * *boundary_d*, int * *bcId_d*, FLOAT_TYPE * *f_d*, FLOAT_TYPE * *u0_d*, FLOAT_TYPE * *v0_d*, int * *corner_d*)

Inlet boundary conditions.

Deprecated use gpuBcInlet

Parameters

in	*fluid_d*	fluid condition
in	*boundary_d*	node boundary type
in	*bcId_d*	lattice boundary type
out	*f_d*	distribution function
in	*u0_d,v0_d*	input velocity vectors (x,y)
in	*corner_d*	corner array

Todo code: complete with y velocity

Todo code: compute inlet on other sides

11.14.2.4 __global__ void gpu_boundaries2 (int * *fluid_d*, FLOAT_TYPE * *Fneighbours_d*, FLOAT_TYPE * *fColl_d*, int * *bcId_d*, FLOAT_TYPE * *Q_d*, FLOAT_TYPE * *f_d*)

Wall boundary conditions.

Deprecated use gpuBcWall

Parameters

in	*fluid_d*	fluid condition
out	*Fneighbours_d*	intermediary array for curved wall (unused)
in	*bcId_d*	lattice boundary type
out	*f_d*	distribution function
in	*Q_d*	grid center ratio array
in	*fColl_d*	distribution function from the collison step

11.14.2.5 __global__ void gpu_boundaries3 (int * *fluid_d*, int * *bcId_d*, FLOAT_TYPE * *f_d*, FLOAT_TYPE * *u0_d*, FLOAT_TYPE * *v0_d*, int * *corner_d*)

Outlet boundary conditions.

Deprecated use gpuBcOutlet

Parameters

in	*fluid_d*	fluid condition
in	*bcId_d*	lattice boundary type
out	*f_d*	distribution function
in	*u0_d,v0_d*	input velocity vectors (x,y)
in	*corner_d*	corner array

Todo code: fill north-side outlet

Todo code: fill west-side outlet

Todo code: fill south-side outlet

Todo code: fill north-side outlet

Todo code: fill west-side outlet

Todo code: fill south-side outlet

Todo code: fill north-side outlet

Todo code: fill west-side outlet

Todo code: fill south-side outlet

11.14.2.6 __global__ void gpu_convert (int * *bcIdx_d*, int * *bcMask_d*, int * *fluid_d*, int * *boundary_d*, int * *corner_d*, int * *bcId_d*, int *size*)

Convert new bitmask to the old boundary condition arrays.

Note

used only for validation

Parameters

in	*bcIdx_d*	boundary condition indices
in	*bcMask_d*	boundary conditions bitmask
out	*fluid_d*	fluid condition
out	*boundary_d*	node boundary type
out	*corner_d*	corner array
out	*bcId_d*	lattice boundary type
in	*size*	number of boundary conditions

11.14.2.7 __global__ void gpu_init (int * *corner_d*, FLOAT_TYPE * *rho_d*, FLOAT_TYPE * *Q_d*, int * *boundary_d*, FLOAT_TYPE * *coordY_d*, int * *stream_d*, int * *bcId_d*, FLOAT_TYPE * *u_d*, FLOAT_TYPE * *v_d*, FLOAT_TYPE * *u0_d*, FLOAT_TYPE * *v0_d*)

Initialise boundary conditions and macroscopic values.

Deprecated use gpu_init_1 and initBoundaryConditions instead

Parameters

corner_d	corner array
rho_d	density array
Q_d	grid center ratio array
boundary_d	node boundary type
coordY_d	node coordinates y
stream_d	streaming array
bcId_d	lattice boundary type
u_d,v_d	velocity vectors (x,y)
u0_d,v0_d	input velocity vectors (x,y)

4∗1.5∗Uavg∗ distance_from_min ∗ distance_from_max / length^2

11.14.2.8 __global__ void gpu_init_1 (FLOAT_TYPE * *rho_d*, FLOAT_TYPE * *u0_d*, FLOAT_TYPE * *v0_d*, FLOAT_TYPE * *u_d*, FLOAT_TYPE * *v_d*, FLOAT_TYPE * *coordY_d*, int *size*)

Initialise macroscopic values.

Deprecated use createGpuArrayFlt and gpuInitInletProfile

Parameters

rho_d	density
u0_d,v0_d	input velocity
u_d,v_d	velocity
coordY_d	node coordinates y
size	vector size (MxN)

NOTE 4∗1.5∗Uavg∗ distance_from_min ∗ distance_from_max / length^2

11.14.2.9 __global__ void gpu_init_8 (int ∗ *stream_d*, FLOAT_TYPE ∗ *Q_d*, int *size*)

Initialise stream and q arrays.

Deprecated use createGpuArrayFlt instead

Parameters

stream_d	streaming array
Q_d	grid center ratio array
size	vector size (8xMxN)

11.14.2.10 __global__ void gpu_mrt1 (int ∗ *fluid_d*, FLOAT_TYPE ∗ *rho_d*, FLOAT_TYPE ∗ *u_d*, FLOAT_TYPE ∗ *v_d*, FLOAT_TYPE ∗ *f_d*, FLOAT_TYPE ∗ *mEq_d*, FLOAT_TYPE ∗ *m_d*)

MRT collision model (step 1)

Deprecated use gpuCollMrt

Parameters

in	*fluid_d*	fluid condition
in	*rho_d*	density
in	*u_d,v_d*	velocity vectors (x,y)
in	*f_d*	distribution function from boundary step
out	*m_d,mEq_d*	intermediary arrays

11.14.2.11 __global__ void gpu_mrt2 (int ∗ *fluid_d*, FLOAT_TYPE ∗ *collision_d*, FLOAT_TYPE ∗ *m_d*, FLOAT_TYPE ∗ *mEq_d*, FLOAT_TYPE ∗ *fColl_d*, FLOAT_TYPE ∗ *f_d*)

MRT collision model (step 2)

Deprecated use gpuCollMrt

Parameters

in	*fluid_d*	fluid condition
out	*fColl_d*	distribution function
in	*f_d*	distribution function from boundary step
in	*m_d,mEq_d*	intermediary arrays

out	*collision_d*	intermediary arrays (unused)

11.14.2.12 __global__ void gpu_streaming (int * *fluid_d*, int * *stream_d*, FLOAT_TYPE * *f_d*, FLOAT_TYPE * *fColl_d*)

Streaming.

Deprecated use gpuStreaming

Parameters

fluid_d	fluid condition
stream_d	streaming array
f_d	distribution function
fColl_d	distribution function from the collison step

11.14.2.13 __global__ void gpu_trt1 (int * *fluid_d*, FLOAT_TYPE * *fEq_d*, FLOAT_TYPE * *rho_d*, FLOAT_TYPE * *u_d*, FLOAT_TYPE * *v_d*)

TRT collision model (step 1)

Deprecated use gpuCollTrt

Parameters

in	*fluid_d*	fluid condition
out	*fEq_d*	distribution function without frequency
in	*rho_d*	density
in	*u_d,v_d*	velocity vectors (x,y)

11.14.2.14 __global__ void gpu_trt2 (int * *fluid_d*, FLOAT_TYPE * *fEq_d*, FLOAT_TYPE * *f_d*, FLOAT_TYPE * *fColl_d*)

TRT collision model (step 2))

Deprecated use gpuCollTrt

Parameters

in	*fluid_d*	fluid condition
in	*fEq_d*	distribution function without frequency
out	*fColl_d*	distribution function
in	*f_d*	distribution function from boundary step

11.14.2.15 __global__ void gpu_update_macro (int * *fluid_d*, FLOAT_TYPE * *rho_d*, FLOAT_TYPE * *u_d*, FLOAT_TYPE * *v_d*, int * *bcId_d*, int * *boundaryId_d*, FLOAT_TYPE * *drag_d*, FLOAT_TYPE * *lift_d*, FLOAT_TYPE * *coordX_d*, FLOAT_TYPE * *coordY_d*, FLOAT_TYPE * *f_d*)

Update macroscopic values.

Deprecated use gpuUpdateMacro

Parameters

in	*fluid_d*	fluid condition
out	*rho_d*	density
out	*u_d,v_d*	velocity vectors (x,y)
in	*bcId_d*	lattice boundary type
in	*boundaryId_d*	boundary ID for lift/drag
out	*drag_d*	drag
out	*lift_d*	lift
in	*coordX_d,coordY_d*	node coordinates x,y
in	*f_d*	distribution function

Todo code: probably should handle outlet on other sides

11.14.2.16 __global__ void gpu_update_new (int * *fluid_d*, FLOAT_TYPE * *rho_d*, FLOAT_TYPE * *u_d*, FLOAT_TYPE * *v_d*, int * *bcMask_d*, FLOAT_TYPE * *drag_d*, FLOAT_TYPE * *lift_d*, FLOAT_TYPE * *coordX_d*, FLOAT_TYPE * *coordY_d*, FLOAT_TYPE * *f_d*)

Update macroscopic values using bitmask.

Deprecated use gpuUpdateMacro

Parameters

in	*fluid_d*	fluid condition
out	*rho_d*	density
out	*u_d,v_d*	velocity vectors (x,y)
in	*bcMask_d*	boundary conditions bitmask
out	*drag_d*	drag
out	*lift_d*	lift
in	*coordX_d,coordY_d*	node coordinates x,y
in	*f_d*	distribution function

Todo code: probably should handle outlet on other sides

11.14.2.17 __global__ void gpuInitInletProfile (FLOAT_TYPE * *u0_d*, FLOAT_TYPE * *v0_d*, FLOAT_TYPE * *y_d*, int *size*)

Initialise initial velocity vector for inlet profile.

This function only sets the x velocity (u) to the following profile

$$l = (y_{max}^{inlet} - y_{min}^{inlet})^2$$

$$u_0 = \frac{6}{l} * u_{in} * (y - y_{min}^{inlet}) * (y_{max}^{inlet} - y)$$

Parameters

u0_d,v0_d	initial velocity

y_d	node coordinates y
size	vector size (MxN)

11.14.2.18 __host__ int initBoundaryConditions (int * *bcNodeIdX*, int * *bcNodeIdY*, FLOAT_TYPE * *q*, int * *bcBoundId*, int * *fluid*, FLOAT_TYPE * *bcX*, FLOAT_TYPE * *bcY*, FLOAT_TYPE * *nodeX*, FLOAT_TYPE * *nodeY*, int * *latticeId*, int * *stream*, int * *bcType*, int * *bcMask*, int * *bcIdx*, int * *mask*, FLOAT_TYPE *delta*, int *m*, int *n*, int *size*)

Initialise boundary conditions.

Fills the bitmask with appropriate boundary conditions and counts the boundary conditions group by the nodes. Fills the mask array for collapseBc to collapse the MxN size arrays into Nbc size arrays. Nodes treated as corners if their lattices hold more than one kind of boundaries (e.g. inlet and wall). The corner node and the corner lattice than treated as wall, and the corner flag is set in the bitmask.

Streaming is enabled everywhere by default, but it is disabled on the opposite side of the boundary lattices.

The q vector contains the ratio between the grid spacing and the lattice distance from the center of the node. If all boundaries are straight lines q vector contains 0.5 everywhere. If the boundary doesn't go through the center of the node (curved conditions), the following interpolation is applied:

$$\delta_{grid} = |x_{0,0} - x_{1,0}|$$

$$q_{lat} = [0; 1; 1; 1; 1; \sqrt{2}; \sqrt{2}; \sqrt{2}; \sqrt{2}]$$

$$q = \frac{\sqrt{(x_{BC} - x_{node})^2 + (y_{BC} - y_{node})^2}}{\delta_{grid} q_{lat}}$$

See Also

BcMacros.h

Parameters

in	*bcNodeIdX,bcNodeIdY*	node ID (i,j)
out	*q*	grid center ratio array
in	*bcBoundId*	boundary ID for drag/lift computation
in	*fluid*	fluid condition
in	*bcX,bcY*	BC coordinates (x,y)
in	*nodeX,nodeY*	node coordinates (x,y)
in	*latticeId*	lattice id (0..8) array
out	*stream*	streaming array
in	*bcType*	boundary type
out	*bcMask*	boundary conditions bitmask
out	*bcIdx*	boundary condition indices
out	*mask*	mask for collapsing (needed by collapseBc)
in	*delta*	grid spacing
in	*m*	number of columns
in	*n*	number of rows
in	*size*	vector size (MxN)

Returns

number of boundary conditions

11.14.2.19 __host__ void initConstants (Arguments * *args*, FLOAT_TYPE *maxInletCoordY*, FLOAT_TYPE *minInletCoordY*, FLOAT_TYPE *delta*, int *m*, int *n*)

Initialise GPU constants.

See Also

GpuConstant.h for the constants

Parameters

args	input parameters
maxInletCoordY	maximum inlet coordinate y
minInletCoordY	minimum inlet coordinate y
delta	grid spacing
m	number of columns
n	number of rows

11.15 include/GpuSum.h File Reference

Functions for sum on GPU.

```
#include "FloatType.h"
#include <cuda.h>
```

Functions

- __global__ void gpu_sqsub (FLOAT_TYPE *A, FLOAT_TYPE *B, FLOAT_TYPE *C, int size)

  Compute square of the difference of two vectors: $(A-B)^2$.
- __global__ void gpu_sqsubi (int *A, int *B, FLOAT_TYPE *C, int size)

  Compute square of the difference of two vectors: $(A-B)^2$.
- __global__ void gpu_sum (FLOAT_TYPE *A, FLOAT_TYPE *B, int size)

  Sum of a vector.
- __global__ void gpu_sum256 (FLOAT_TYPE *A, FLOAT_TYPE *B, int size)

  Sum of a vector (for 256 threads only)
- __global__ void gpu_cond_copy (FLOAT_TYPE *A, FLOAT_TYPE *B, int *cond, int value, int size)

  Conditional vector copy.
- __global__ void gpu_cond_copy_mask (FLOAT_TYPE *A, FLOAT_TYPE *B, int *mask, int value, int size)

  Conditional vector copy for BC bitmask.
- __host__ FLOAT_TYPE gpu_sum_h (FLOAT_TYPE *C, FLOAT_TYPE *D, int size)

  Host function for GPU vector sum (calls gpu_sum256)

11.15.1 Detailed Description

Functions for sum on GPU.

Author

Adam Koleszar (adam.koleszar@gmail.com)

11.15.2 Function Documentation

11.15.2.1 __global__ void gpu_cond_copy (FLOAT_TYPE * A, FLOAT_TYPE * B, int * cond, int value, int size)

Conditional vector copy.

Deprecated use gpu_cond_copy_mask

Parameters

in	*A*	input vector
out	*B*	output vector
in	*cond*	array with the conditional values
in	*value*	value to compare the conditionals
in	*size*	vector size

11.15.2.2 __global__ void gpu_cond_copy_mask (FLOAT_TYPE * A, FLOAT_TYPE * B, int * mask, int value, int size)

Conditional vector copy for BC bitmask.

Parameters

in	*A*	input vector
out	*B*	output vector
in	*mask*	array of conditional values
in	*value*	value to compare to
in	*size*	vector size

11.15.2.3 __global__ void gpu_sqsub (FLOAT_TYPE * A, FLOAT_TYPE * B, FLOAT_TYPE * C, int size)

Compute square of the difference of two vectors: $(A-B)^2$.

Parameters

in	*A,B*	input vector
out	*C*	result vector
in	*size*	vector size

11.15.2.4 __global__ void gpu_sqsubi (int * A, int * B, FLOAT_TYPE * C, int size)

Compute square of the difference of two vectors: $(A-B)^2$.

Parameters

in	*A,B*	input vector
out	*C*	result vector
in	*size*	vector size

11.15.2.5 __global__ void gpu_sum (FLOAT_TYPE * A, FLOAT_TYPE * B, int size)

Sum of a vector.

Parameters

in	*A*	input vector
out	*B*	sum of vector (in the first element)
in	*size*	vector size

11.15.2.6 __global__ void gpu_sum256 (FLOAT_TYPE * *A*, FLOAT_TYPE * *B*, int *size*)

Sum of a vector (for 256 threads only)

Note

faster than gpu_sum but works only with 256 threads

Parameters

in	*A*	input vector
out	*B*	sum of vector (in the first element)
in	*size*	vector size

11.15.2.7 __host__ FLOAT_TYPE gpu_sum_h (FLOAT_TYPE * *C*, FLOAT_TYPE * *D*, int *size*)

Host function for GPU vector sum (calls gpu_sum256)

Parameters

C	input vector
D	temporary vector
size	vector size

Returns

sum of the vector

11.16 include/Iterate.h File Reference

The solver itself.

```
#include "Arguments.h"
```

Functions

- int Iteration (InputFilenames *inFn, Arguments *args)
  LBM solver.

11.16.1 Detailed Description

The solver itself.

Author

Adam Koleszar (adam.koleszar@gmail.com)

Most of the time the same variable name was used for the same data. Here is a list for most of them. The suffix _d is used to indicate GPU array. The old convention is the following:

name	size	description	available values	used in step
corner	MxN	corner boundary condition	0: none, 1: corner	boundaries
rho	MxN	density	any (float)	collision, macro
q	9xMxN	lattice length / grid spacing	any (float)	boundaries
stream	9xMxN	streaming indicator	0: no, 1: stream	streaming
boundary	MxN	node boundary type	0: none, 1: wall, 2: inlet, 3: outlet	boundaries
bc_id	9xMxN	lattice boundary type	0: none, 1: wall, 2: inlet, 3: outlet	boundaries, macro
u,v	MxN	velocity	any (float)	collision, macro
u0,v0	MxN	input velocity	any (float)	boundaries
coord x,y	MxN	node coordinates	any (float)	macro
fluid	MxN	node fluid condition	0: solid, 1: fluid	collision, stream, boundary, macro
f	9xMxN	distribution function	any (float)	collision, stream, boundary, macro
meta-f	9xMxN	temporary distribution func	any (float)	collision, stream, boundary, macro

Later the arrays containing boundary conditions are combined into a bitmask. Also the q and stream arrays were reduced in size beacuse there is no point storing zero information for the node itself (0) only the lattices (1..8). Changes in new convention:

name	size	description	available values	used in step
bc-mask	Nbc	BC bitmask (BcMacros.h)	bitmask	boundaries, macro
bc-idx	Nbc	BC node ID	0..MxN	boundaries
q	8xMxN	lattice length / grid spacing	any (float)	boundaries
stream	8xMxN	streaming indicator	0: no, 1: stream	streaming

The iteration takes the following steps: 0. initialisation (run once)

1. collision model
2. streaming
3. boundary conditions
4. macroscopic values

11.16.2 Function Documentation

11.16.2.1 int Iteration (InputFilenames * *inFn*, Arguments * *args*)

LBM solver.

Parameters

in	*inFn*	filenames
in	*args*	input parameters

Returns

0 if everything went okay, 1 if iteration diverged, 2 if file read error

11.17 include/LogWriter.h File Reference

Log writing function.

```
#include "FloatType.h"
#include "Arguments.h"
```

Enumerations

- enum taskTime {
  T_INIT =0, T_ITER, T_COLL, T_STRM,
  T_BNDC, T_MACR, T_RESI, T_WRIT,
  T_OALL }

  Enumeration of the tasks.

Functions

- void writeInitLog (const char *filename, Arguments *args, FLOAT_TYPE delta, int m, int n, int numInletNodes, FLOAT_TYPE maxInletCoordY, FLOAT_TYPE minInletCoordY)

  Print initialisation log.
- void writeNodeNumbers (const char *filename, int numNodes, int numConns, int bcCount)

  Print node numbers log.
- void writeEndLog (const char *filename, FLOAT_TYPE *taskTimes)

  Print time results into the log.
- void writeTimerLog (const char *filename, FLOAT_TYPE *taskTimes)

  Print time results into the time file.
- void writeResiduals (const char *filename, FLOAT_TYPE *norm, FLOAT_TYPE *drag, FLOAT_TYPE *lift, int size, int n)

  Print residual data.

11.17.1 Detailed Description

Log writing function.

Author

Adam Koleszar (adam.koleszar@gmail.com)

11.17.2 Enumeration Type Documentation

11.17.2.1 enum taskTime

Enumeration of the tasks.

Enumerator

T_INIT initialisation time

T_ITER iteration time

T_COLL collision time

T_STRM streaming time

T_BNDC bc time

T_MACR macroscopic time

T_RESI residuals time

T_WRIT file write time

T_OALL overall time

11.17.3 Function Documentation

11.17.3.1 void writeEndLog (const char * *filename*, FLOAT_TYPE * *taskTimes*)

Print time results into the log.

Parameters

filename	output filename
taskTimes	time values

11.17.3.2 void writeInitLog (const char * *filename*, Arguments * *args*, FLOAT_TYPE *delta*, int *m*, int *n*, int *numInletNodes*, FLOAT_TYPE *maxInletCoordY*, FLOAT_TYPE *minInletCoordY*)

Print initialisation log.

Parameters

in	*filename*	output filename
in	*args*	input parameters
in	*delta*	grid spacing
in	*m*	number of columns
in	*n*	number of rows
in	*numInletNodes*	number of inlet nodes
in	*maxInletCoordY*	maximum inlet coordinate y
in	*minInletCoordY*	minimum inlet coordinate y

11.17.3.3 void writeNodeNumbers (const char * *filename*, int *numNodes*, int *numConns*, int *bcCount*)

Print node numbers log.

Parameters

in	*filename*	output filename
in	*numNodes*	number of nodes
in	*numConns*	number of conditions
in	*bcCount*	number of boundary nodes

11.17.3.4 void writeResiduals (const char * *filename*, FLOAT_TYPE * *norm*, FLOAT_TYPE * *drag*, FLOAT_TYPE * *lift*, int *size*, int *n*)

Print residual data.

Parameters

filename	output filename
norm	norm of the distribution function change $f - f_{coll}$
drag,lift	drag and lift
size	mesh size (MxN)
n	number of iterations

11.17.3.5 void writeTimerLog (const char * *filename,* FLOAT_TYPE * *taskTimes*)

Print time results into the time file.

Parameters

filename	output filename
taskTimes	time values

11.18 include/ShellFunctions.h File Reference

Allocator and shell functions.

```
#include <string.h>
#include "FloatType.h"
```

Macros

- #define max(a, b)

  Get maximum of two values.
- #define min(a, b)

  Get minimum of two values.

Functions

- void CreateDirectory (const char *MainWorkDir)

  Make directory.
- void StringAddition (char *first, char *second, char *result)

  Concatenate strings.

11.18.1 Detailed Description

Allocator and shell functions.

Author

Istvan Tamas Jozsa (jozsait@gmail.com)

11.18.2 Macro Definition Documentation

11.18.2.1 #define max(*a, b*)

Value:

```
({ __typeof__ (a) _a = (a); \
       __typeof__ (b) _b = (b); \
     _a > _b ? _a : _b; })
```

Get maximum of two values.

Parameters

in	*a,b*	values to compare

Returns

maximum of (a,b)

11.18.2.2 #define min(*a, b*)

Value:

```
({ __typeof__ (a) _a = (a); \
       __typeof__ (b) _b = (b); \
     _a < _b ? _a : _b; })
```

Get minimum of two values.

Parameters

in	*a,b*	values to compare

Returns

minimum of (a,b)

11.18.3 Function Documentation

11.18.3.1 void CreateDirectory (const char * *MainWorkDir*)

Make directory.

Parameters

in	*MainWorkDir*	directory to be created

11.18.3.2 void StringAddition (char * *first,* char * *second,* char * *result*)

Concatenate strings.

Note

not used

Parameters

in	*first*	first string
in	*second*	second string

out	*result*	concatenated string

11.19 main.cu File Reference

Main file.

```
#include <string.h>
#include <stdio.h>
#include "Iterate.h"
#include "ShellFunctions.h"
#include "FilesReading.h"
#include "FilesWriting.h"
#include "CellFunctions.h"
#include "ComputeResiduals.h"
#include "Arguments.h"
#include "AllTests.h"
```

Functions

- int runAllTests (void)

  Run all unittests.
- int main (int argc, char ∗argv[])

  Main function.

11.19.1 Detailed Description

Main file.

Author

Adam Koleszar (adam.koleszar@gmail.com)

11.19.2 Function Documentation

11.19.2.1 int main (int *argc*, char ∗ *argv[]*)

Main function.

- handles input parameters
- creates directory for results
- runs the solver

Parameters

argc	number of command line parameters
argv	command line parameters

Returns

0: success n: error

11.19.2.2 int runAllTests (void)

Run all unittests.

Returns

0: success, n: number of failures

11.20 test/AllTests.h File Reference

All unittests need to be declared in this header.

```
#include "CuTest.h"
```

Functions

- CuSuite * computeResidualsGetSuite ()
  Unittests for residual computations.
- CuSuite * gpuSumGetSuite ()
  Unittests for sum on GPU.
- CuSuite * gpuBoundariesGetSuite ()
  Unittests for boundary conditions.
- CuSuite * gpuInitGetSuite ()
  Unittests for initialisation.
- CuSuite * gpuMacroGetSuite ()
  Unittests for macroscopic values.
- CuSuite * gpuCollisionGetSuite ()
  Unittests for collision models.
- CuSuite * gpuStreamGetSuite ()
  Unittests for streaming.
- CuSuite * iterateGetSuite ()
  Unittests for the solver.

11.20.1 Detailed Description

All unittests need to be declared in this header.

Author

Adam Koleszar (adam.koleszar@gmail.com)

11.21 test/TestComputeResiduals.cu File Reference

Unit tests for the residual computations.

```
#include "CuTest.h"
#include "GpuFunctions.h"
#include "ComputeResiduals.h"
#include "FloatType.h"
#include "ArrayUtils.h"
#include "TestUtils.h"
#include "GpuSum.h"
```

Functions

- void testGpuComputeResiduals (CuTest ∗tc)
  Unittest for GpuComputeResiduals.
- void cleanupTestGpuComputeResiduals ()
  Clean up after test case.
- void testGpuComputeResidMask (CuTest ∗tc)
  Unittest for GpuComputeResidMask.
- void cleanupTestGpuComputeResidMask ()
  Clean up after test case.
- void testComputeResidual (CuTest ∗tc)
  Unittest for computeResidual.
- void cleanupTestComputeResidual ()
  Clean up after test case.
- void testComputeDragLift (CuTest ∗tc)
  Unittest for computeDragLift.
- void cleanupTestComputeDragLift ()
  Clean up after test case.
- CuSuite ∗ computeResidualsGetSuite ()
  Unittests for residual computations.

11.21.1 Detailed Description

Unit tests for the residual computations.

Author

Adam Koleszar (adam.koleszar@gmail.com)

11.21.2 Function Documentation

11.21.2.1 void testComputeDragLift (CuTest ∗ *tc*)

Unittest for computeDragLift.

Parameters

tc	test case

Test

- Prepare arrays
- call function
- check the sum of the arrays against predefined values

11.21.2.2 void testComputeResidual (CuTest ∗ *tc*)

Unittest for computeResidual.

Parameters

tc	test case

Test
- Prepare arrays
- call function
- check the sum of the arrays against predefined values

11.21.2.3 void testGpuComputeResidMask (CuTest * *tc*)

Unittest for GpuComputeResidMask.

Parameters

tc	test case

Test
- Prepare arrays
- call function
- check the sum of the arrays against predefined values

11.21.2.4 void testGpuComputeResiduals (CuTest * *tc*)

Unittest for GpuComputeResiduals.

Parameters

tc	test case

Test
- Prepare arrays
- call function
- check the sum of the arrays against predefined values

11.22 test/TestGpuBoundaries.cu File Reference

Unittests for the boundary conditions.

```
#include "CuTest.h"
#include "GpuFunctions.h"
#include "ShellFunctions.h"
#include "ComputeResiduals.h"
#include "FloatType.h"
#include "TestUtils.h"
#include "ArrayUtils.h"
#include "GpuConstants.h"
```

Functions

- void BoundaryTestInit (FLOAT_TYPE af, FLOAT_TYPE bf)

  Initialise conditions for BC testcases.
- void testGpuBoundaries1 (CuTest *tc)

  Test gpu_boundaries1.
- void runTestGpuBoundaries2 (CuTest *tc, BoundaryType boundaryType)

Test gpu_boundaries2.

- void cleanupTestGpuBoundaries2 ()

  Clean up after test case.
- void testGpuBoundaries2_wcb (CuTest *tc)

  Test case for curved wall.
- void testGpuBoundaries2_wocb (CuTest *tc)

  Test case for straight wall.
- void testGpuBoundaries3 (CuTest *tc)

  Test gpu_boundaries3.
- void cleanupTestGpuBoundaries ()

  Clean up after test case.
- void testGpuBcInlet (CuTest *tc)

  Test gpuBcInlet.
- void cleanupTestGpuBcInlet ()

  Clean up after test case.
- void runTestGpuBcWall (CuTest *tc, BoundaryType boundaryType)

  Test gpuBcWall.
- void cleanupTestGpuBcWall ()

  Clean up after test case.
- void testGpuBcWall_wcb (CuTest *tc)

  Test case for curved wall.
- void testGpuBcWall_wocb (CuTest *tc)

  Test case for straight wall.
- void testGpuBcOutlet (CuTest *tc)

  Test gpuBcOutlet.
- void cleanupTestGpuBcOutlet ()

  Clean up after test case.
- void testCompareWall (CuTest *tc)

  Test to compare results from gpuBcWall and gpu_boundaries2.
- void runTestCompareOutlet (CuTest *tc, OutletProfile op)

  Test to compare results from gpuBcOutlet and gpu_boundaries3.
- void testCompareOutletProfile1 (CuTest *tc)

  compare outlet conditions with Zhu-He profile
- void testCompareOutletProfile2 (CuTest *tc)

  compare outlet conditions with 2nd order open boundary
- void testCompareOutletProfile3 (CuTest *tc)

  compare outlet conditions with 1st order open boundary
- void cleanupTestCompareOutlet ()

  Clean up after test case.
- void testCompareInlet (CuTest *tc)

  Test to compare results from gpuBcInlet and gpu_boundaries1.
- void cleanupTestCompareInlet ()

  Clean up after test case.
- CuSuite * gpuBoundariesGetSuite ()

  Unittests for boundary conditions.

11.22.1 Detailed Description

Unittests for the boundary conditions.

Author

Adam Koleszar (adam.koleszar@gmail.com)

11.22.2 Function Documentation

11.22.2.1 void BoundaryTestInit (FLOAT_TYPE *af*, FLOAT_TYPE *bf*)

Initialise conditions for BC testcases.

Parameters

af	value to fill the distribution function
bf	value to fill the u0 array

11.22.2.2 void runTestCompareOutlet (CuTest * *tc*, OutletProfile *op*)

Test to compare results from gpuBcOutlet and gpu_boundaries3.

Parameters

tc	test case
op	outlet profile

Test
- Prepare boundary conditions for the lid driven cavity
- Run the function
- Check the sum of all the arrays between the two algorithms

11.22.2.3 void runTestGpuBcWall (CuTest * *tc*, BoundaryType *boundaryType*)

Test gpuBcWall.

Parameters

tc	test case
boundaryType	straight or curved

Test
- Prepare boundary conditions for the lid driven cavity
- Run the function
- Check the sum of all the arrays against predefined values

11.22.2.4 void runTestGpuBoundaries2 (CuTest * *tc*, BoundaryType *boundaryType*)

Test gpu_boundaries2.

Parameters

tc	test case
boundaryType	straight or curved

Test
- Prepare boundary conditions for the lid driven cavity
- Run the function
- Check the sum of all the arrays against predefined values

11.22.2.5 void testCompareInlet (CuTest * *tc*)

Test to compare results from gpuBcInlet and gpu_boundaries1.

Parameters

tc	test case

Test
- Prepare boundary conditions for the lid driven cavity
- Run the function
- Check the sum of all the arrays between the two algorithms

11.22.2.6 void testCompareOutletProfile1 (CuTest * *tc*)

compare outlet conditions with Zhu-He profile

Parameters

tc	test case

11.22.2.7 void testCompareOutletProfile2 (CuTest * *tc*)

compare outlet conditions with 2nd order open boundary

Parameters

tc	test case

11.22.2.8 void testCompareOutletProfile3 (CuTest * *tc*)

compare outlet conditions with 1st order open boundary

Parameters

tc	test case

11.22.2.9 void testCompareWall (CuTest * *tc*)

Test to compare results from gpuBcWall and gpu_boundaries2.

Parameters

tc	test case

Test
- Prepare boundary conditions for the lid driven cavity
- Run the function
- Check the sum of all the arrays against predefined values

11.22.2.10 void testGpuBcInlet (CuTest * *tc*)

Test gpuBcInlet.

Parameters

tc	test case

Test
- Prepare boundary conditions for the lid driven cavity
- Run the function
- Check the sum of all the arrays against predefined values

11.22.2.11 void testGpuBcOutlet (CuTest * *tc*)

Test gpuBcOutlet.

Parameters

tc	test case

Test
- Prepare boundary conditions for the lid driven cavity
- Run the function
- Check the sum of all the arrays against predefined values

11.22.2.12 void testGpuBcWall_wcb (CuTest * *tc*)

Test case for curved wall.

Parameters

tc	test case

11.22.2.13 void testGpuBcWall_wocb (CuTest * *tc*)

Test case for straight wall.

Parameters

tc	test case

11.22.2.14 void testGpuBoundaries1 (CuTest * *tc*)

Test gpu_boundaries1.

Parameters

tc	test case

Test
- Prepare boundary conditions for the lid driven cavity
- Run the function
- Check the sum of all the arrays against predefined values

11.22.2.15 void testGpuBoundaries2_wcb (CuTest * *tc*)

Test case for curved wall.

Parameters

tc	test case

11.22.2.16 void testGpuBoundaries2_wocb (CuTest * *tc*)

Test case for straight wall.

Parameters

tc	test case

11.22.2.17 void testGpuBoundaries3 (CuTest * *tc*)

Test gpu_boundaries3.

Parameters

tc	test case

Test
- Prepare boundary conditions for the lid driven cavity
- Run the function
- Check the sum of all the arrays against predefined values

11.23 test/TestGpuCollision.cu File Reference

Unittests for the boundary conditions.

```
#include "CuTest.h"
#include "GpuFunctions.h"
#include "ShellFunctions.h"
#include "ComputeResiduals.h"
#include "FloatType.h"
#include "TestUtils.h"
#include "ArrayUtils.h"
#include "GpuConstants.h"
#include "CellFunctions.h"
```

Functions

- void testCompareMrt (CuTest *tc)

  Test to compare results from gpuCollMrt and gpu_mrt1 gpu_mrt2.
- void cleanupTestCompareMrt ()

  Clean up after test case.
- void testCompareTrt (CuTest *tc)

  Test to compare results from gpuCollTrt and gpu_trt1 gpu_trt2.
- void cleanupTestCompareTrt ()

  Clean up after test case.
- CuSuite * gpuCollisionGetSuite ()

  Unittests for collision models.

11.23.1 Detailed Description

Unittests for the boundary conditions.

Author

Adam Koleszar (adam.koleszar@gmail.com)

11.23.2 Function Documentation

11.23.2.1 void testCompareMrt (CuTest * *tc*)

Test to compare results from gpuCollMrt and gpu_mrt1 gpu_mrt2.

Parameters

tc	test case

Test
- Prepare boundary conditions for the lid driven cavity
- Run the function
- Check the sum of all the arrays between the two algorithms

11.23.2.2 void testCompareTrt (CuTest * *tc*)

Test to compare results from gpuCollTrt and gpu_trt1 gpu_trt2.

Parameters

tc	test case

Test
- Prepare boundary conditions for the lid driven cavity
- Run the function
- Check the sum of all the arrays between the two algorithms

11.24 test/TestGpuInit.cu File Reference

Unit tests for the init functions.

```
#include "CuTest.h"
#include "GpuFunctions.h"
#include "ArrayUtils.h"
#include "GpuConstants.h"
#include "TestUtils.h"
#include "Arguments.h"
```

Functions

- void runTestGpuInit1 (InletProfile inlt, FLOAT_TYPE u0, FLOAT_TYPE v0)

  Unittest for gpu_init_1.
- void testGpuInit1_inlet (CuTest *tc)

  Test case for inlet.
- void testGpuInit1_noInlet (CuTest *tc)

Test case for no inlet.

- void testGpuInit1_pulsatile (CuTest *tc)

  Test case for pulsatile.

- void cleanupTestGpuInit1 ()

  Clean up after test case.

- void testGpuInitInletProfile (CuTest *tc)

  Unittest for gpuInitInletProfile.

- void cleanupTestGpuInitInletProfile ()

  Clean up after test case.

- CuSuite * gpuInitGetSuite ()

  Unittests for initialisation.

11.24.1 Detailed Description

Unit tests for the init functions.

Author

Adam Koleszar (adam.koleszar@gmail.com)

11.24.2 Function Documentation

11.24.2.1 void runTestGpuInit1 (InletProfile *inlt*, FLOAT_TYPE *u0*, FLOAT_TYPE *v0*)

Unittest for gpu_init_1.

Parameters

inlt	inlet profile
u0,v0	input velocity

Test Unittest for gpu_init_1

- Prepare arrays
- call function
- check the sum of the arrays against predefined values
- test for all inlet profiles

11.24.2.2 void testGpuInit1_inlet (CuTest * *tc*)

Test case for inlet.

Parameters

tc	test case

11.24.2.3 void testGpuInit1_noInlet (CuTest * *tc*)

Test case for no inlet.

Parameters

tc	test case

11.24.2.4 void testGpuInit1_pulsatile (CuTest * *tc*)

Test case for pulsatile.

Parameters

tc	test case

11.24.2.5 void testGpuInitInletProfile (CuTest * *tc*)

Unittest for gpuInitInletProfile.

Parameters

tc	test case

Test
- Prepare arrays for inlet profile
- call function
- check the sum of the arrays against predefined values

11.25 test/TestGpuStream.cu File Reference

Unittests for the boundary conditions.

```
#include "CuTest.h"
#include "GpuFunctions.h"
#include "ShellFunctions.h"
#include "ComputeResiduals.h"
#include "FloatType.h"
#include "TestUtils.h"
#include "ArrayUtils.h"
#include "GpuConstants.h"
#include "CellFunctions.h"
```

Functions

- void testCompareGpuStream (CuTest *tc)

  Test to compare results from gpuStreaming and gpu_streaming.
- void cleanupTestCompareGpuStream ()

  Clean up after test case.
- CuSuite * gpuStreamGetSuite ()

  Unittests for streaming.

11.25.1 Detailed Description

Unittests for the boundary conditions.

Author

Adam Koleszar (adam.koleszar@gmail.com)

11.25.2 Function Documentation

11.25.2.1 void testCompareGpuStream (CuTest * *tc*)

Test to compare results from gpuStreaming and gpu_streaming.

Parameters

tc	test case

Test

- Prepare boundary conditions for the lid driven cavity
- Run the function
- Check the sum of all the arrays between the two algorithms

11.26 test/TestGpuSum.cu File Reference

Unittests for GPU sum functions.

```
#include "FloatType.h"
#include "CuTest.h"
#include "GpuSum.h"
#include "ArrayUtils.h"
#include "TestUtils.h"
```

Functions

- void testGpuCondCopy (CuTest *tc)

  Unittest for gpu_cond_copy.
- void cleanupTestGpuCondCopy ()

  Clean up after test case.
- void testGpuSquareSubstract (CuTest *tc)

  Unittest for gpu_sqsub.
- void cleanupTestGpuSquareSubsctract ()

  Clean up after test case.
- void testGpuSum (CuTest *tc)

  Unittest for gpu_sum.
- void testGpuSum256 (CuTest *tc)

  Unittest for gpu_sum256.
- void testGpusumHost (CuTest *tc)

  Unittest for gpu_sum_h.
- void cleanTestGpuSum ()

  Clean up after test case.
- CuSuite * gpuSumGetSuite ()

  Unittests for sum on GPU.

11.26.1 Detailed Description

Unittests for GPU sum functions.

Author

Adam Koleszar (adam.koleszar@gmail.com)

11.26.2 Function Documentation

11.26.2.1 void testGpuCondCopy (CuTest * *tc*)

Unittest for gpu_cond_copy.

Parameters

tc	test case

Test
- Allocate arrays and fill them with random values
- call function
- check the sum of the arrays against predefined values

11.26.2.2 void testGpuSquareSubstract (CuTest * *tc*)

Unittest for gpu_sqsub.

Parameters

tc	test case

Test
- Allocate arrays and fill them with fixed values
- call function
- check the sum of the arrays against predefined values

11.26.2.3 void testGpuSum (CuTest * *tc*)

Unittest for gpu_sum.

Parameters

tc	test case

Test
- Allocate arrays and fill them with fixed values
- call function
- check the sum of the arrays against predefined values

11.26.2.4 void testGpuSum256 (CuTest * *tc*)

Unittest for gpu_sum256.

Parameters

tc	test case

Test
- Allocate arrays and fill them with fixed values
- call function
- check the sum of the arrays against predefined values

11.26.2.5 void testGpusumHost (CuTest * *tc*)

Unittest for gpu_sum_h.

Parameters

tc	test case

Test
- Allocate arrays and fill them with fixed values
- call function
- check the sum of the arrays against predefined values

11.27 test/TestGpuUpdateMacro.cu File Reference

Unittests for macroscopic value update.

```
#include "CuTest.h"
#include "GpuFunctions.h"
#include "ShellFunctions.h"
#include "TestUtils.h"
#include "ArrayUtils.h"
#include "GpuConstants.h"
```

Functions

- void testGpuUpdateMacro_ (CuTest *tc)
  Unittest for gpu_update_macro.
- void cleanupTestGpuUpdateMacro ()
  Clean up after test case.
- void runTestGpuUpdatePart1 (FLOAT_TYPE hf)
  Prepare host array for lid driven cavity.
- void runTestGpuUpdatePart2 (int *IntSums, FLOAT_TYPE *FltSums)
  Compute sums and copy host arrays to GPU arrays.
- void runTestGpuUpdatePart3 (int *IntSums, FLOAT_TYPE *FltSums)
  Copy GPU array to host arrays and compute their sums.
- void testGpuUpdateNew (CuTest *tc)
  Unittest for gpu_update_new.
- void testGpuUpdateMacro (CuTest *tc)
  Unittest for gpuUpdateMacro.
- void cleanupTestGpuUpdate ()
  Clean up after test case.
- void testCompareUpdateMacro (CuTest *tc)
  Unittest to compare results of gpu_update_macro, gpu_update_new, gpuUpdateMacro.
- CuSuite * gpuMacroGetSuite ()
  Unittests for macroscopic values.

11.27.1 Detailed Description

Unittests for macroscopic value update.

Author

Adam Koleszar (adam.koleszar@gmail.com)

11.27.2 Function Documentation

11.27.2.1 void runTestGpuUpdatePart1 (FLOAT_TYPE *hf*)

Prepare host array for lid driven cavity.

Parameters

hf	value to fill the distribution function

11.27.2.2 void runTestGpuUpdatePart2 (int * *IntSums*, FLOAT_TYPE * *FltSums*)

Compute sums and copy host arrays to GPU arrays.

Parameters

IntSums	integer arrays sums
FltSums	float arrays sums

11.27.2.3 void runTestGpuUpdatePart3 (int * *IntSums*, FLOAT_TYPE * *FltSums*)

Copy GPU array to host arrays and compute their sums.

Parameters

IntSums	integer arrays sums
FltSums	float arrays sums

11.27.2.4 void testCompareUpdateMacro (CuTest * *tc*)

Unittest to compare results of gpu_update_macro, gpu_update_new, gpuUpdateMacro.

Parameters

tc	test case

Test • compare results from previous runs

11.27.2.5 void testGpuUpdateMacro (CuTest * *tc*)

Unittest for gpuUpdateMacro.

Parameters

tc	test case

Test
- Prepare arrays for lid driven cavity
- call function
- check the sum of the arrays against predefined values

11.27.2.6 void testGpuUpdateMacro_ (CuTest * *tc*)

Unittest for gpu_update_macro.

Parameters

tc	test case

Test
- Prepare arrays for lid driven cavity
- call function
- check the sum of the arrays against predefined values

11.27.2.7 void testGpuUpdateNew (CuTest * *tc*)

Unittest for gpu_update_new.

Parameters

tc	test case

Test
- Prepare arrays for lid driven cavity
- call function
- check the sum of the arrays against predefined values

11.28 test/TestIterate.cu File Reference

Unittests for the solver.

```
#include "GpuFunctions.h"
#include "ShellFunctions.h"
#include "FilesReading.h"
#include "CellFunctions.h"
#include "ComputeResiduals.h"
#include "LogWriter.h"
#include "Iterate.h"
#include "ArrayUtils.h"
#include "Check.h"
#include "TestUtils.h"
#include "CuTest.h"
```

Functions

- void runTestIteration (Arguments *args, InputFilenames *inFn, CuTest *tc)

  Test the solver by having all of its steps done the old and the new way.
- void testIterationLidMrt (CuTest *tc)

test for lid-driven cavity, using MRT model

- void testIterationExpBgkw (CuTest *tc)

  test for expansion, using BGKW model

- void testIterationExpMrt (CuTest *tc)

  test for expansion, using MRT model

- void cleanupTestIteration ()

  Clean up after test case.

- CuSuite * iterateGetSuite ()

  Unittests for the solver.

11.28.1 Detailed Description

Unittests for the solver.

Author

Adam Koleszar (adam.koleszar@gmail.com)

Warning

This test needs actual meshes, so put them into the test directory or give their path in the code.

Todo create a command-line argument for test meshes

11.28.2 Function Documentation

11.28.2.1 void runTestIteration (Arguments * *args*, InputFilenames * *inFn*, CuTest * *tc*)

Test the solver by having all of its steps done the old and the new way.

Parameters

args	input arguments
inFn	input files
tc	test case

11.28.2.2 void testIterationExpBgkw (CuTest * *tc*)

test for expansion, using BGKW model

Parameters

tc	test case

11.28.2.3 void testIterationExpMrt (CuTest * *tc*)

test for expansion, using MRT model

Parameters

tc	test case

11.28.2.4 void testIterationLidMrt (CuTest * *tc*)

test for lid-driven cavity, using MRT model

Parameters

tc	test case

11.29 test/TestUtils.h File Reference

Utility functions for the unittests.

```
#include "FloatType.h"
```

Macros

- #define MAX_LEN 80
  max length for banner

Functions

- __global__ void gpuArrayAddInt (int *array1_d, int *array2_d, int size)
  Add two integer vectors on the GPU.
- __global__ void gpuArrayAddFlt (FLOAT_TYPE *array1_d, FLOAT_TYPE *array2_d, int size)
  Add two floating point vectors on the GPU.
- void hostArrayAddInt (int *array1_h, int *array2_h, int size)
  Add two integer vectors on the host.
- void hostArrayAddFlt (FLOAT_TYPE *array1_h, FLOAT_TYPE *array2_h, int size)
  Add two floating point vectors on the host.
- int sumHostInt (int *array_h, int size)
  Summaries integer vector on the host.
- FLOAT_TYPE sumHostFlt (FLOAT_TYPE *array_h, int size)
  Summaries floating point vector on the host.
- void createLidBcFluid (int *fluid, int m, int n)
  Create fluid conditions for the lid driven cavity.
- void createLidBcBoundary (int *boundary, int m, int n)
  Create node boundary conditions for the lid driven cavity.
- void createLidBcBcId (int *bcid, int m, int n)
  Create lattice boundary conditions for the lid driven cavity.
- void createLidBcCorner (int *corner, int m, int n)
  Create corner conditions for the lid driven cavity.
- int getLidBcMaskSize (int m, int n)
  Boundary condition bitmask size for the lid driven cavity.
- void createLidBcIdx (int *index, int m, int n)
  Create boundary condition indices for the lid driven cavity.
- void createLidBcMask (int *mask, int m, int n)

Create boundary conditions bitmask for the lid driven cavity (collapsed)

- void createLidBcMaskFull (int *mask, int m, int n)

  Create boundary conditions bitmask for the lid driven cavity (full size)
- void createLidBoundaryId (int *bid, int m, int n)

  Create boundary IDs for the lid driven cavity.
- void fillLidCoordinates (FLOAT_TYPE *x, FLOAT_TYPE *y, int m, int n)

  Create coordinates for the lid driven cavity.
- void fillFDir (FLOAT_TYPE *f, FLOAT_TYPE fu, int dir, int size)

  Fill distribution function in only one direction.
- void createChannelBcMask (int *mask, int m, int n)

  Create boundary conditions bitmask for the channel.
- void createChannelBcBcId (int *bcid, int m, int n)

  Create lattice boundary conditions for the channel.
- void createChannelBcCorner (int *corner, int m, int n)

  Create corner conditions for the channel.
- int compareArraysInt (int *array1_h, int *array2_h, int size)

  Compare integer arrays by element.
- int compareArraysFlt (FLOAT_TYPE *array1_h, FLOAT_TYPE *array2_h, int size)

  Compare floating point arrays by element.
- __host__ int compareGpuArrayInt (int *a_d, int *b_d, int size)

  Compare GPU int array on CPU.
- __host__ int compareGpuArrayFlt (FLOAT_TYPE *a_d, FLOAT_TYPE *b_d, int size)

  Compare GPU float array on CPU.
- __host__ FLOAT_TYPE compareGpuArraySumFlt (FLOAT_TYPE *F1_d, FLOAT_TYPE *F2_d, FLOAT_TYPE *temp9a_d, FLOAT_TYPE *temp9b_d, int size)

  Compare two vectors on the GPU.
- __host__ FLOAT_TYPE compareGpuArraySumInt (int *F1_d, int *F2_d, FLOAT_TYPE *temp9a_d, FLOAT_TYPE *temp9b_d, int size)

  Compare two vectors on the GPU.
- void printBanner (const char *testname)

  Print a formated banner for the test case.

11.29.1 Detailed Description

Utility functions for the unittests.

Author

Adam Koleszar (adam.koleszar@gmail.com)

11.29.2 Function Documentation

11.29.2.1 int compareArraysFlt (FLOAT_TYPE * *array1_h*, FLOAT_TYPE * *array2_h*, int *size*)

Compare floating point arrays by element.

Parameters

array1_h,array2_h	arrays to compare

size	array size

Returns

number of different elements

11.29.2.2 int compareArraysInt (int * *array1_h,* int * *array2_h,* int *size*)

Compare integer arrays by element.

Parameters

array1_h,array2_h	arrays to compare
size	array size

Returns

number of different elements

11.29.2.3 __host__ int compareGpuArrayFlt (FLOAT_TYPE * *a_d,* FLOAT_TYPE * *b_d,* int *size*)

Compare GPU float array on CPU.

Compares two array value by value

Parameters

a_d,b_d	arrays to compare
size	array size

Returns

number of differences

11.29.2.4 __host__ int compareGpuArrayInt (int * *a_d,* int * *b_d,* int *size*)

Compare GPU int array on CPU.

Compares two array value by value

Parameters

a_d,b_d	arrays to compare
size	array size

Returns

number of differences

11.29.2.5 __host__ FLOAT_TYPE compareGpuArraySumFlt (FLOAT_TYPE * *F1_d,* FLOAT_TYPE * *F2_d,* FLOAT_TYPE * *temp9a_d,* FLOAT_TYPE * *temp9b_d,* int *size*)

Compare two vectors on the GPU.

Compare two vectors on the GPU by summing up the squared differences (Frobenius norm: $\sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}-b_{ij}|^2}$)

Parameters

F1_d,F2_d	vectors to compare
temp9a_d,temp9b_d	temporary vectors
size	vector size

Returns

Frobenius norm of the difference of the vectors

11.29.2.6 __host__ FLOAT_TYPE compareGpuArraySumInt (int * *F1_d,* int * *F2_d,* FLOAT_TYPE * *temp9a_d,* FLOAT_TYPE * *temp9b_d,* int *size*)

Compare two vectors on the GPU.

Compare two vectors on the GPU by summing up the squared differences (Frobenius norm: $\sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}-b_{ij}|^2}$)

Parameters

F1_d,F2_d	vectors to compare
temp9a_d,temp9b_d	temporary vectors
size	vector size

Returns

Frobenius norm of the difference of the vectors

11.29.2.7 void createChannelBcBcId (int * *bcid,* int *m,* int *n*)

Create lattice boundary conditions for the channel.

Parameters

out	*bcid*	lattice boundary array
in	*m*	number of columns
in	*n*	number of rows

11.29.2.8 void createChannelBcCorner (int * *corner,* int *m,* int *n*)

Create corner conditions for the channel.

Parameters

out	*corner*	corner boundary array
in	*m*	number of columns
in	*n*	number of rows

11.29.2.9 void createChannelBcMask (int * *mask,* int *m,* int *n*)

Create boundary conditions bitmask for the channel.

Parameters

out	*mask*	boundary condition bitmask array
in	*m*	number of columns
in	*n*	number of rows

11.29.2.10 void createLidBcBcId (int * *bcid*, int *m*, int *n*)

Create lattice boundary conditions for the lid driven cavity.

Parameters

out	*bcid*	lattice boundary array
in	*m*	number of columns
in	*n*	number of rows

11.29.2.11 void createLidBcBoundary (int * *boundary*, int *m*, int *n*)

Create node boundary conditions for the lid driven cavity.

Parameters

out	*boundary*	node boundary array
in	*m*	number of columns
in	*n*	number of rows

11.29.2.12 void createLidBcCorner (int * *corner*, int *m*, int *n*)

Create corner conditions for the lid driven cavity.

Parameters

out	*corner*	corner boundary array
in	*m*	number of columns
in	*n*	number of rows

11.29.2.13 void createLidBcFluid (int * *fluid*, int *m*, int *n*)

Create fluid conditions for the lid driven cavity.

Parameters

out	*fluid*	fluid array
in	*m*	number of columns
in	*n*	number of rows

11.29.2.14 void createLidBcIdx (int * *index*, int *m*, int *n*)

Create boundary condition indices for the lid driven cavity.

Parameters

out	*index*	boundary condition indices array
in	*m*	number of columns
in	*n*	number of rows

11.29.2.15 void createLidBcMask (int * *mask*, int *m*, int *n*)

Create boundary conditions bitmask for the lid driven cavity (collapsed)

Parameters

out	*mask*	boundary condition bitmask array
in	*m*	number of columns
in	*n*	number of rows

11.29.2.16 void createLidBcMaskFull (int * *mask*, int *m*, int *n*)

Create boundary conditions bitmask for the lid driven cavity (full size)

Parameters

out	*mask*	boundary condition bitmask array
in	*m*	number of columns
in	*n*	number of rows

11.29.2.17 void createLidBoundaryId (int * *bid*, int *m*, int *n*)

Create boundary IDs for the lid driven cavity.

Parameters

	bid	boundary id array
in	*m*	number of columns
in	*n*	number of rows

11.29.2.18 void fillFDir (FLOAT_TYPE * *f*, FLOAT_TYPE *fu*, int *dir*, int *size*)

Fill distribution function in only one direction.

Parameters

out	*f*	distribution fucntion
in	*fu*	value to fill
in	*dir*	direction
in	*size*	array size

11.29.2.19 void fillLidCoordinates (FLOAT_TYPE * *x*, FLOAT_TYPE * *y*, int *m*, int *n*)

Create coordinates for the lid driven cavity.

Parameters

out	*x,y*	coordinates
in	*m*	number of columns
in	*n*	number of rows

11.29.2.20 int getLidBcMaskSize (int *m*, int *n*)

Boundary condition bitmask size for the lid driven cavity.

Parameters

in	*m*	number of columns
in	*n*	number of rows

Returns

number of boundary conditions

11.29.2.21 __global__ void gpuArrayAddFlt (FLOAT_TYPE * *array1_d*, FLOAT_TYPE * *array2_d*, int *size*)

Add two floating point vectors on the GPU.

Parameters

in,out	*array1_d*	vector (result stored here)
in	*array2_d*	vector
	size	vector size

11.29.2.22 __global__ void gpuArrayAddInt (int * *array1_d*, int * *array2_d*, int *size*)

Add two integer vectors on the GPU.

Parameters

in,out	*array1_d*	vector (result stored here)
in	*array2_d*	vector
	size	vector size

11.29.2.23 void hostArrayAddFlt (FLOAT_TYPE * *array1_h*, FLOAT_TYPE * *array2_h*, int *size*)

Add two floating point vectors on the host.

Parameters

in,out	*array1_h*	vector (result stored here)
in	*array2_h*	vector
	size	vector size

11.29.2.24 void hostArrayAddInt (int * *array1_h*, int * *array2_h*, int *size*)

Add two integer vectors on the host.

Parameters

in,out	*array1_h*	vector (result stored here)
in	*array2_h*	vector
	size	vector size

11.29.2.25 void printBanner (const char * *testname*)

Print a formated banner for the test case.

Parameters

testname	test name

11.29.2.26 FLOAT_TYPE sumHostFlt (FLOAT_TYPE * *array_h,* int *size*)

Summaries floating point vector on the host.

Parameters

array_h	vector
size	vector size

Returns

sum of vector

11.29.2.27 int sumHostInt (int * *array_h,* int *size*)

Summaries integer vector on the host.

Parameters

array_h	vector
size	vector size

Returns

sum of vector

Index