CRANFIELD UNIVERSITY

JOSÉ OLIVEIRA

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

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

MSc
Academic Year: 2016–2017

Supervisor: Dr Irene Moulitsas
August 2017

CRANFIELD UNIVERSITY

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

MSc

Academic Year: 2016–2017

JOSÉ OLIVEIRA

Development of Physics or HPC Optimisation of a parallel 2D lattice Boltzmann solver using GPUs/CUDA

Supervisor: Dr Irene Moulitsas
August 2017

This thesis is submitted in partial fulfilment of the requirements for the degree of MSc.

Abstract

Type your abstract here.

Keywords

Keyword 1; keyword 2; keyword 3.

Contents

List of Figures

List of Tables

List of Abbreviations

CUDA	Compute Unified Device Architecture
CFD	Computational Fluid Dynamics
GPGPU	General-purpose computing on graphics processing units
GPU	Graphics Processing Unit
LBM	Lattice Boltzmann Method

Acknowledgements

The author would like to thank . . .

Chapter 1

Introduction

Chapter 2

Literature review

In this chapter, we will be taken into the broad field of the Lattice Boltzmann method using the CUDA platform. The topic of this thesis is associated with a number of different study areas, such as Computational Fluid Dynamics, Lattice Boltzmann method, High performance computing and GPGPU. We will then present an extensive literature review on these themes in the remaining of this chapter. Firstly, we will look into Computational Fluid Dynamics and the most commonly used approaches to it. Then we will be giving an overview on the Lattice Boltzmann method. Afterwards we will discuss how it is possible to employ parallelisation techniques in scientific computing. Finally we will review some previous works in this area.

2.1 Computational Fluid Dynamics

Computational Fluid Dynamics (CFD) comes from the need to model fluid flows and associated processes. A wide range of applications come from studying CFD, notably:

- Aircraft design [7]
- Solid particle erosion [18]

- Wind flow simulation [4]
- Combustion chamber simulations [19]
- Environmental and weather prediction [20]
- Automotive and motor sports [21, 8]

CFD can be used as a design and troubleshooting tool, as well as making the process dynamics easier to understand. It is used extensively by scientists and researchers, but it also has innumerable applications in the industry. CFD simulations is a viable tool for manufacturing because it eliminates expensive simulations.

As such, it is the science of determining a solution to fluid flow through space and time [9]. The models needed to calculate the fluid computations include:

- Flow geometry
- Differential (Governing) equations – These describe the physics and chemistry of the flow
- Boundary and initial conditions
- Discretization of the domain

2.1.1 Macroscopic scale

In this approach, the fluid can be seen as a collection of a huge number of particles. To solve these governing equations, one needs to apply conservation of energy, mass and momentum [16]. But since these equations are difficult, or even impossible to solve analytically, discrete schemes, boundary and initial conditions are used to convert these equations into a system of algebraic equations. These equations can then be solved until an appropriate solution is produced.

These problems are usually solved using Navier-Stokes equations that describe the fluid being solved as a continuum, which apply Newton's second law to fluid motion.

2.1.2 Microscopic scale

If we consider the fluid to be represented by individual particles then we will fall under the microscopic approach. In this approach, there is no definition of temperature or viscosity and collision between particles needs to be considered. Thus one needs to solve the differential equation of Newton's second law [16]. Hence, the location and velocity of each particle needs to be taken into account.

We can easily see that this approach becomes unfeasible for normal fluid sizes as the number of equations needed to be solved grows to the order of billions (consider that one mole of water contains more than 6×10^{23} molecules.

2.2 Lattice Boltzmann Method

The Lattice Boltzmann method (LBM) is a mesoscopic scale approach to CFD and was first introduced as a Lattice-Gas Automata for the Navier-Stokes Equation [10]. It is used to describe a fluid based on probabilities using the Maxwell-Boltzmann equation in the fluid's equilibrium state [16].

In this method, we do not consider the individual characteristics of each particle. By grouping particles together in a D2Q9 (nodes containing 9 particles for 2D problems - Fig 2.1) or in a D3Q19 (nodes containing 19 particles for 3D problems - Fig 2.2) model, we can analyse the behaviour of the particles collectively [16].

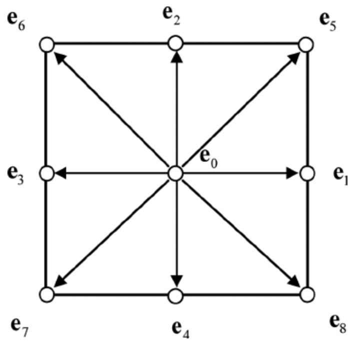


Figure 2.1: 2D LBM model using 9 particles. (D2Q9)

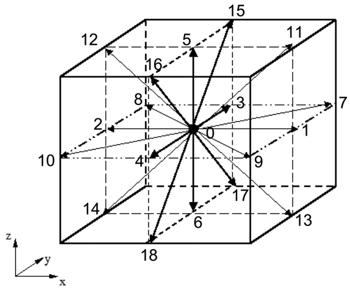


Figure 2.2: 3D LBM model using 19 particles. (D3Q19)

This way we can reap the advantages of both the macro and microscale approaches without the need of high end computers [16]. Since communications between nodes are very limited, LBM also offers the possibility of employing parallel computing to achieve the solution in even faster times.

2.2.1 Multiphase flow

Multiphase flows represent the simultaneous flow of materials in different states (phases). As such, multiphase flows have an enormous spectrum of representation and stand for interactions between gas/solid flows, liquid/solid flows or even liquid/liquid flows with different chemical properties [5].

These simulations present challenging problems because of inherent difficulties during modelling and the importance of engineering applications. However, the Lattice

Boltzmann method provides an alternative for these simulations due to it's relative simplicity when compared with traditional Navier-Stokes equation [6].

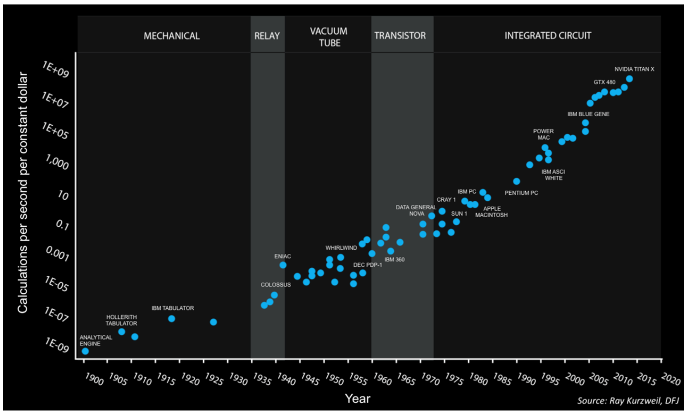
These flows have several applications in the scientific and industrial community, ranging from pourous media fluid interactions [15] to flows containing gas bubbles dispersed in liquids [22]. Furthermore, almost every processing technology must take multiphase flows into account, including cavitating pumps, papermaking and many others [5].

Several implementation strategies exist for Multiphase flows, however, for the purpose of this thesis, we will first follow the simple ”two-color” approach first proposed by Gunstensen et al [11] This is wrong, refer to RK. Also, explain the history and how to method was developed.

2.3 High Performance Computing

As humanity evolves, so too does our desire for expanding previous unobtainable goals. As computer technology kept progressing further and further, we soon realised that some problems simply took too many resources to be completed.

Figure 2.3 shows the evolution of computing power over the past 120 years. Moore's Law states that ”processor speeds, or overall processing power for computers will double every two years” [2]. Note that the last 7 most recent data points are NVIDIA GPUs.



Source: Ray Kurzweil, DFJ

Figure 2.3: Moore's law over 120 years

However, this evolution in computational power could only be achieved by combining CPU cores together. So what if we focused our efforts in splitting the workload, effectively using the multiple cores available to produce a solution?

High performance computing (HPC) comes from the harnessing of computer power to deliver a much higher performance that one could not obtain from a typical computer. To this end, we can talk of HPC as being a collection of computer resources, all of them working simultaneously to achieve a solution of the same problem. Problems that could otherwise take weeks, months or even years can now be solved in minutes, hours or days under these powerful devices. However, different parallelisation strategies for splitting the workload emerge, depending on the underlying hardware.

It is also important to understand that parallel computing is achieved with the help of processors that will execute different calculations or processes simultaneously. To measure the parallelisation's efficiency, it is worth introducing the term Speed-up. The Speed-up is the ratio of the execution time of the parallel algorithm on a single processor with the execution time of the parallel algorithm on P processors [17]. However, Amdahl's law states that "in parallelization, if P is the proportion of a system or program that can be made parallel (...), then the maximum speed-up that can be achieved using N number of processors is $\frac{1}{(1-P)+\frac{P}{N}}$" [1]. This means that our parallelisation efforts are limited by the amount of work that can be parallelised and, theoretically, should follow the distribution represented in Figure 2.4.

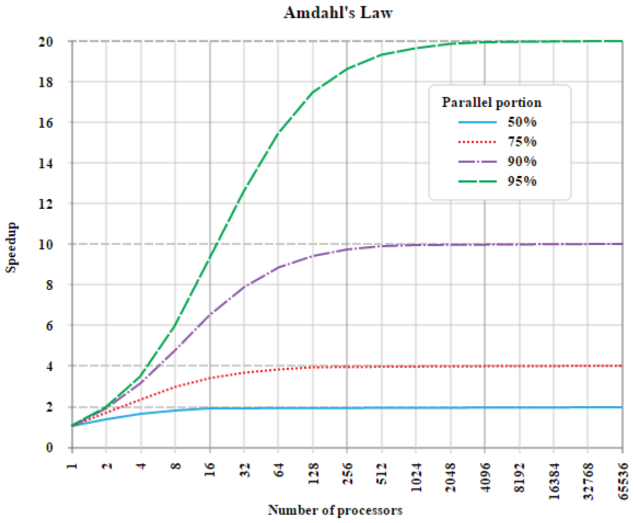


Figure 2.4: A graphical representation of Amdahl's law

2.3.1 Distributed memory

Following a distributed memory architecture for parallel computing means that each processor will have it's own independent local memory. In these systems, all the work that one process executes remains local to it, without interfering with the address space of all other processors. Hence, to solve meaningful computations, a communication network needs to be established in order for processes to share data with each other. Such is the case of the Message Passing Interface (MPI).

This means that this architecture is very scalable: with each processor being added to the system the size of the total memory increases. Also, each processor will be able to access it's own memory rapidly and without interference, reducing the usual constraint of memory access penalties.

However, this type of system requires a higher degree of skill from the programmer. The programmer will be the one responsible for most of the details of memory passing between processors and will need to ensure that no race conditions or deadlocks arise from the data communication. Also, whenever data from another processor is needed, the latency of the bandwidth in the network will introduce a heavy time penalty to the computations, as this data will need to be communicated before computations can be performed over it.

2.3.2 Shared memory

In a shared memory architecture, each processor is able to access all memory as global address space. This means that data can be handled seamlessly between processors, making programs easy to read and easy to write. An example of this model can be the Open Multi-Processing (OpenMP) API, which supports shared memory multiprocessing programming in C, C++ and Fortran. In this architecture, processors can operate independently while having access to the same memory resource pool as all other processors. Therefore, changes in the data handled by one process is visible and updated for all others.

This means that the programmer can have a user-friendly environment while working on his/her algorithm. All details concerning data flow between processes are abstracted, making this model very beneficial for users with little to no background in parallel programming. Also, the data being shared between processors is fast and uniform, making the penalties of memory access of this shared data less penalizing than in the distributed memory model.

However, this architecture presents little memory scalability, since adding more CPUs will increase the traffic of the shared memory path. Also, the programmer will have to pay close attention to memory access so as to prevent memory violation and ensure a correct synchronization of the access to the global adress space.

2.3.3 GPGPU

General Purpose computing on Graphical Processing Units stands for the use of Graphics Processing unit (GPU) to perform computations on applications normally performed by the CPU. One of the main advantages of using this approach is the amount of cores that a single GPU has. While a typical desktop CPU has up to 4 cores, a GPU can have thousands of cores, allowing users to take advantage of its massively parallel architecture. GPUs can now solve problems that were traditionally solved by the CPU. This is a big improvement, since GPUs are cheaper to acquire and more powerful than CPUs.

NVIDIA then developed CUDA. CUDA code allows programmers to take advantage of GPUs by employing a unified shader pipeline under the familiar C language [12]. Users were no longer required to have specific knowledge of OpenGL or DirectX and could now perform general computations (rather than graphic-specific computations) whilst benefiting from the massive computational powered offered by GPUs.

Programmers could now use the macros defined by CUDA to harness the full power of the GPU with a relatively small learning curve. If the user is already familiar with C language, then he can pick up on the details of the framework quickly. While this may seem like a big improvement, almost nothing comes without some disadvantages. CUDA can be excessively complicated to those unfamiliar with parallel programming. Although it offers the possibility of competing with several CPUs linked together, to optimise the kernel calls (device specific functions) takes a big attention to details and some knowledge on how the underlying hardware works. Users should not take this approach light-heartedly as they can easily become encumbered with work when compared to a simpler to use framework (such as OpenMP).

2.4 Previous parallelisation works

This thesis is a continuation of work that started some years ago in Cranfield University. As such, the work from the previous students needs to be analysed.

In 2014, Tamás Józsa and Máté Szőke, adapted two different in-house C and C++ codes into one single C code unifying the advantages of each one of the two original codes [13, 23]. Józsa then parallelised the critical parts of the C code using CUDA and ran tests on the Fermi GPU Cluster from Cranfield, achieving a three times speed-up in general, with a peak of 15 times speed-up [13].

Szőke proposes a CPU parallelisation approach using Unified Parallel C, which is a Partitioned Global Address Space language[23]. This means that it is possible to use shared memory to compute the solution. However, the author verified that a local memory-based approach (like MPI) provided the best results. He also compared the results obtained with the ones obtained by Józsa on the CUDA approach. They found that to achieve the same speed-up as that of a single GPU card, one needs an entire workstation (16 threads in the case of Astral) [23].

In 2015, Ádám Koleszár continued the work and further optimised the parallel version of the LBM method using CUDA [24]. He did an excellent job, resulting in a 10 times faster execution than the previous 2D parallel solver, which means that his new optimised code was 30 times faster than the original, in-house, serial solver.

Finally, in 2016, Maciej Kubat proposed a new version of the LBM solver.Firstly he converts the 2D parallel solver to a 3D parallel solver which entailed a major re-engineering of the code, from data containers to logic cycles [14]. After first trying for a direct adaptation, he found that his code was too slow to produce meaningful solutions. After optimising his own code he was able to reach an almost one hundred times speed-up. However, Kubat states that a lot can be done to improve his code, from boundary conditions to code readability and maintainability.

Chapter 3

Methodology

3.1 Lattice Boltzmann Method

The Lattice Boltzmann method is a mesoscopic approach to solve several fluid dynamics problems, as stated in Section. It relies on a statistical description of the system via a distribution function f. This distribution function is responsible for predicting the number of molecules at a certain time, positioned between 2 points and with velocities between 2 values. This function is used in combination with a collision operator Ω and, with no external force being applied, represent the Boltzmann equation as

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

giving us an advection equation. However, Eq. 3.1 is difficult to solve by itself. To offset this difficulty we need to approximate the collision operator with a simpler operator that will not introduce a significant error in the final solution. After solving this collision step, we then need to propagate the changes in each cell onto to their neighbours (streaming). We then solve boundary interactions within the nodes affected by the boundary spaces of the problem and finally we update the macroscopic values. These values will be fed into the collision step again where the process will repeat itself until the solution converges/diverges or the final number of iterations is reached.

3.1.1 Discretization

Before starting with the collision step, we must first discretize the domain into a mesh composed of various cells. Each of these cells will in turn affect and be affected by other cells, depending on the chosen speed model, as stated in Section 2.2. As such, we will use a 2DQ9 arrangement for the 2D problems and a 3DQ19, which roughly translates into each cell interacting with another 8 or 18 cells (neighbours), depending on the problem's dimension.

3.1.2 Collision

There are a few collision models capable of approximating the collision operator Ω, such as the BGKW cite, the TRT cite and the MRT cite. However, for the purpose of this thesis, we will focus solely on the BGKW model as this will be the approximation used by the Multiphase model.

In the BGKW model, we can use the following equation to approximate Ω

$$\Omega = \omega\left(f^{eq} - f\right) = \frac{1}{\tau}\left(f^{eq} - f\right) \tag{3.2}$$

where ω represents the collision frequency and τ stands for the relaxation factor. f^{eq} represents the local equilibrium of the distribution function, which is known as the Maxwell-Boltzmann distribution function. Now, we can use Eq. 3.2 in the discretized Boltzmann equation to obtain

$$f_i = \omega f_i^{eq} + (1 - \omega) f_i \tag{3.3}$$

which will be valid when following specific directions.

3.1.3 Streaming

After each node finishes calculating the changes brought on by the collision with other nodes, it is necessary to pass this information onto each neighbour. This streaming will take into account the directions imposed by the speed model. Figure 3.1 demonstrates how this step of the method works for the two dimension problem type.

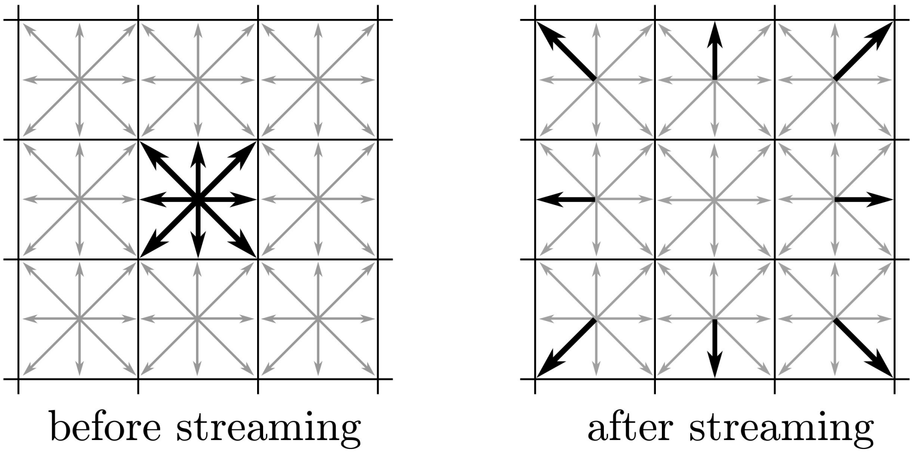


Figure 3.1: Streaming step following a 2DQ9 model.

For 3D problems the streaming step follows the same strategy.

3.1.4 Boundary update

After each node updates their neighbour, the nodes located on the boundaries of the problem need to be updated depending on the boundary condition applied to them. Again, there are quite a few boundary conditions implemented in the in-house solver. However, the author will only go into detail in the boundary conditions used by the new Multiphase model.

Wall

These boundary conditions are implemented with a bounce-back effect, where particles that are incoming into the solid wall bounce back into the domain. Applying the laws of conservation of mass and momentum, a particle incoming into the southern direction of the domain leads to

$$f_2 = f_4$$

$$f_5 = f_7$$

$$f_6 = f_8$$

Periodic boundary conditions

With these boundaries, particles that exit the domain on one side will enter the domain on the opposite, creating an infinite corridor between opposite sides. As such, if a particle reaches the north boundary, their distribution function will be updated as follows

$$f_4^N = f_4^S$$

$$f_7^N = f_7^S$$

$$f_8^N = f_8^S$$

Inlet

Zou and He cite proposed a method for solving these boundary conditions. In this case, the boundary simulates a fluid flow entering the domain with a certain velocity. Applying the proposed equations to the BGKW model on a cell located in the north boundary yields:

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

3.1.5 Macro-variables update

Finally, after updating the distribution function of every cell in the domain, we can update the macroscopic variables ρ, u and v. From Eq. 3.1, we can define the macroscopic values for density and velocity as

$$\rho(r,t) = \int mf(r,c,t)dc \tag{3.4}$$

$$\rho(r,t)u(r,t) = \int mcf(r,c,t)dc \tag{3.5}$$

where m represents the molecular mass.

3.2 Color Gradient Model

Models implementing multiphase flows for the Lattice Boltzmann method can generally be classified into several different categories. For the purpose of this thesis, we will focus on the Rothman-Keller methods, specifically on the model proposed by Reis and Phillips cite, modified using Latva-Kokko's recoloring operator, as proposed by Leclaire et al. cite leclaire.

In this model, two immiscible fluids are simulated under the LBM, a red fluid and a blue fluid, that only interact with each other in the interface between them. These fluids are associated to their own distribution functions, which implies that the total memory used by the in-house solver is likely to increase. Furthermore, the collision step will need to be created from scratch since this step introduces two new operators, while the standard collision step will also suffer some changes to account for the two fluids.

The fluid's distribution function now becomes

$$f_i^k\left(x+c_i,t+1\right)=f_i^k\left(x,t\right)+\Omega_i^k \tag{3.6}$$

where k stands for the fluid, either red or blue.Ω_i^k is the result of the combination of the 3 sub-steps present in the new collision step and is given by

$$\Omega_i^k=\left(\Omega_i^k\right)_{(3)}\left(\left(\Omega_i^k\right)_{(1)}+\left(\Omega_i^k\right)_{(2)}\right) \tag{3.7}$$

3.2.1 Single-phase collision operator

The first sub-step is similar to the collision operator used in the standard BGKW model, introduced in Eq. 3.2. However, a new operator, ϕ is introduced in the calculation of the local equilibrium distribution function.

$$f_i^{k(eq)}=\rho_k\left(\phi_i^k+W_i\left(3c_i\cdot u+\frac{9}{2}\left(c_i\cdot u\right)^2-\frac{3}{2}u^2\right)\right) \tag{3.8}$$

where ϕ_i^k is given by

- α_k for $i=0$
- $\frac{(1-\alpha_k)}{5}$ for $i=1,2,3,4$
- $\frac{(1-\alpha_k)}{20}$ for $i=5,6,7,8$

3.2.2 Perturbation operator

After computing the single-phase collision operator we need to add the result of the perturbation operator before passing it on to the recoloring sub-step. This is where we simulate the surface tension between the fluids and ensure that pressure difference is in equilibrium.

First, we must calculate the color gradient term, which is defined by

$$F = \sum_i c_i \left(\rho_r \left(x + c_i\right) - \rho_B \left(x + c_i\right)\right) \tag{3.9}$$

which is 4th order accurate. Then, we include F in the calculation of the perturbation term, defined by

$$\left(\Omega_i^k\right)_{(2)} \left(F_i^k\right) = F_i^k + \frac{A_k}{2} \|F\| \left(W_i \frac{\left(F \cdot c_i\right)^2}{\|F\|^2} - B_i\right) \tag{3.10}$$

3.2.3 Recoloring operator

The last step in the collision step is calculating the recoloring operator. This operator guarantees that the fluids remain immiscible and also controls the amount of a fluid sent to it's corresponding region. The operator can be defined as

$$\left(\Omega_i^k\right)_{(3)} \left(F_i^k\right) = \frac{\rho_k}{\rho} F_i + \beta \frac{\rho_r \rho_b}{\rho^2} \cos\left(\phi_i\right) \sum_k F_i^{k(eq)} \left(\rho_k, 0, \alpha_k\right) \tag{3.11}$$

where β is a free parameter between 0 and 1 that influences the thickness of the interface and $\cos\left(\phi_i\right)$ is the cosine of the angle between the color gradient F and the direction c.

3.2.4 Streaming and Boundary conditions

The streaming and boundary conditions in this model are analogous to the ones used in the LBM method, as defined in Sections 3.1.3 and 3.1.4. To compute them, we simply need to repeat these steps for the distribution function of each fluid.

3.2.5 Macro-variables update

This step is also similar to the one defined in Section 3.1.5. The main difference is that ρ is the result of the sum of the densities of each fluid and F_i is also the sum of the distribution functions of each fluid.

3.3 Meshes

For the LBM in-house solver we need to prepare structured meshes capable of representing the domain. These meshes allow us to discretize the functions and solve them according to the representation of space. Each mesh is composed of equidistant small spaces that represent the cells used in the equations. To this end, software capable of generating meshes is needed, that can both create the mesh according to the number of nodes required and specify the type of boundary conditions to be used. G. Abbruzzese developed software capable of generating meshes with these requirements cite which has been used in the previous thesis over the LBM in-house solver.

However, only the 3D lattice generator still exists, which means that we are restricted to the 2D meshes that already exist. Luckily, these meshes are sufficient to simulate the requirements of all of the proposed test cases. The boundaries however will need to be hard-coded to match the initial requirements and to adequately simulate the 2D test cases.

3.3.1 Used meshes

The test cases for the Color Gradient model are based on a simple square for 2D or a cube for 3D problems. However, the boundaries are specific to each case. All of them use periodic boundaries on the East and West directions and most of them also use them for the North and South directions, the exception being for the Couette flow where the North boundary is an Inlet and the South boundary acts a solid wall.

3.4 CUDA programming

As stated before, CUDA is a parallel programming platform introduced by NVIDIA. One of it's most appealing features is the fact that it is integrated into the well known programming languages C/C++, making it easy for programmers to start developing parallel code in a familiar environment. However, there are some details that one must know before being able to take benefit of the computational capabilities of GPUs. Note that the host is referring to the CPU and the device is referring to the GPU.

3.4.1 Thread arrangement

As is the case with several other parallel programming platforms, CUDA's base parallel agents are threads. CUDA allows for a massive number of threads running concurrently. However, accessing each thread is not as straightforward as one would hope. Figure 3.2 show how threads are organized inside CUDA's memory model.

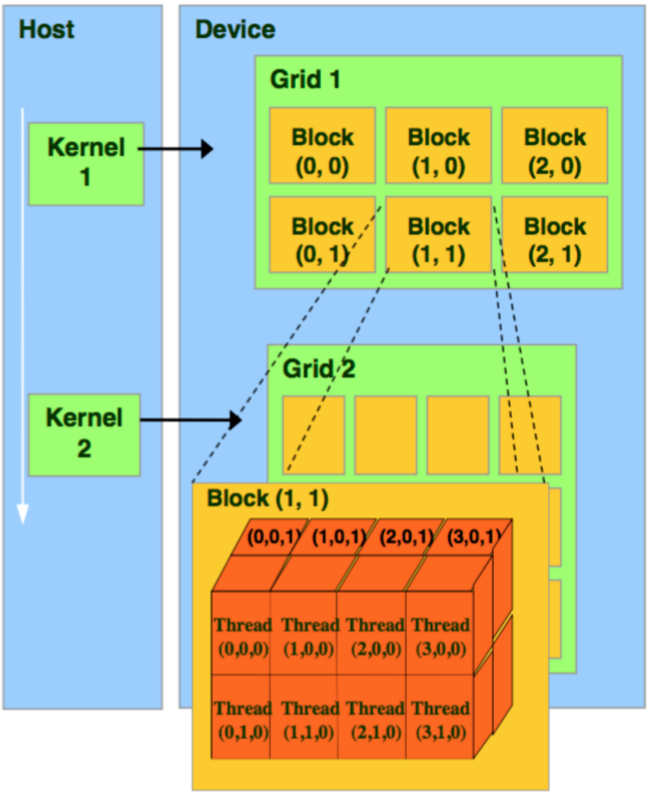


Figure 3.2: Thread arrangement in CUDA platform.

Threads are organized into blocks and blocks are organized in the grid. Both the grid of blocks and the blocks themselves can be of 1, 2 or 3 dimensions. CUDA provides a very useful way of obtaining the index of each thread. We can easily obtain the thread's index inside the block as well as the block's index in the grid. However, some math is needed to obtain the thread's global index, which is very useful for memory management, such as access to an array's element. Fortunately, this is an easy task since we can access all the needed information with threadIdx, blockDim, blockIdx and gridDim.

3.4.2 Memory structures

Since CUDA code runs on GPUs, it is worth showing what the internal architecture of a NVIDIA GPU looks like.

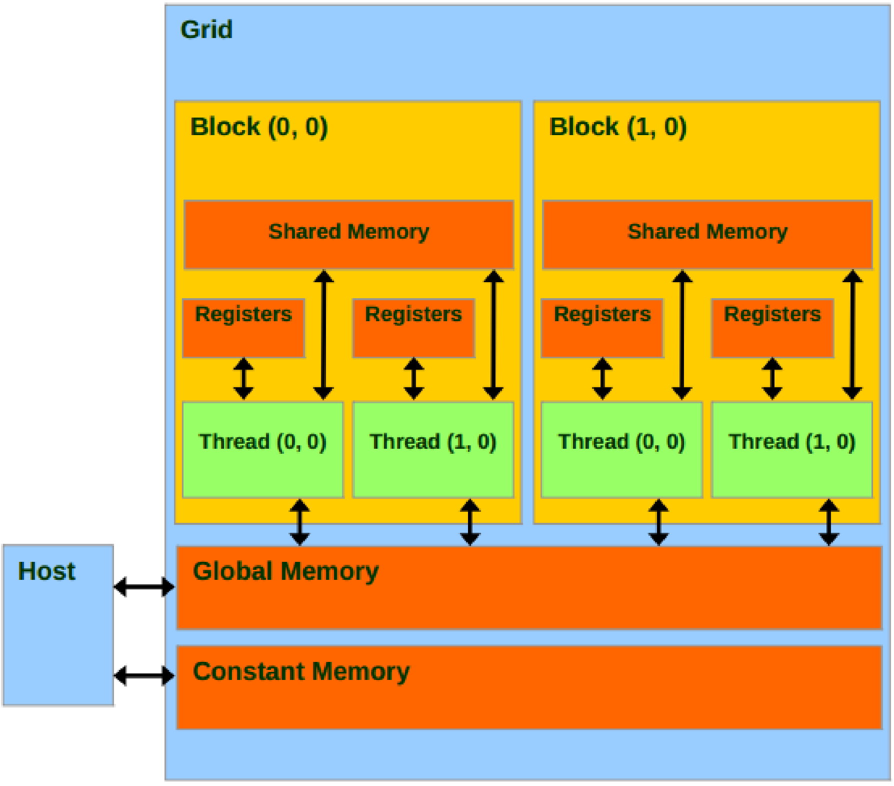


Figure 3.3: Memory arrangement in CUDA compliant GPU.

Figure 3.3 shows 4 different types of memory available to each thread

- Registers
- Shared memory
- Global memory
- Constant memory

There is also one more memory type, texture memory. However, this memory type will not be used in the scope of this thesis, so the author will not go into detail about them.

Registers

Registers are the fastest type of memory available to a thread. Each thread's register is only visible to itself and so every variable using this type of memory is visible only to the thread. This means that it is not possible to use data directly from the host on these structures. Also, this data is non-transferable between threads.

Registers work just like local memory for each thread, the difference is that local memory is comparable to global memory in terms of speed since it is not allocated directly on the GPU chip but is instead an abstraction of global memory. Local memory will only be used if the compiler determines that the thread's register size is not large enough to hold the thread's local memory.

Shared memory

Shared memory is the second fastest type of memory available to a thread. As opposed to registers, shared memory is, as the name suggests, shared between threads residing in the same block. This means that we can have threads cooperating with each other when using this type of memory. However, memory access needs to be properly managed, otherwise degrading memory access speed when bank conflicts occur. Bank conflicts means that two or more threads are trying to access the same memory address, and when this happens the memory access is serialized between the conflicting threads.

The nature of this memory makes it an excellent candidate for cases where threads require information from other threads to continue with their work, for example in a matrix vector multiplication. In these cases, a proper management of shared memory within the block can greatly boost the algorithms performance. As such, this a memory structure that should be used whenever threads require cooperation since with no bank conflicts shared memory can reach the speed of register memory.

Constant memory

This a read-only type of memory that is stored in the cache, making access to it faster than global memory. This is the first memory type that can transfer data between the host and the device since before using in the device data needs to be properly initialized within the host code. Cached memory has the benefit that a single read can be broadcast to other 15 threads, and also that consecutive reads from the same address space will not incur any additional memory traffic cite cuda. However, we can only benefit from a performance gain when using constant memory if we pay attention to the warps inside a block, since the memory read is broadcast to a half-warp (more on this later).

Global memory

Finally, we reach the slowest yet most versatile memory in a CUDA compliant GPU. Global memory has, as the name suggests, a global scope, meaning that every thread inside the grid can access it, as is the case with constant memory. However, it is not limited to read-only access as all threads can also write data onto it. This is the only other data that can be transferred between host and device (apart from texture memory) but it has the disadvantage of being the slowest memory type. To boost performance, data in global memory will usually be read into a faster memory type like registers or shared memory before having computations performed over it.

3.4.3 Warps

A warp is a group of 32 threads inside a block. Each block, when it is created, is assigned to a Streaming Multiprocessor that contain 32 processing cores cite webinar. The threads inside a warp are truly concurrent, meaning that at a given time each thread will execute the same instruction concurrently (lock-step fashion). Therefore, warp-awareness is essential to write highly-optimised code cite salvatore. One must guarantee that each thread inside the warp will follow the same control path, otherwise risk wasting the full potential of the warps concurrency by achieving warp divergence.

Chapter 4

Results and Discussion

4.1 In-house LBM solver

The implementation of a 3D multiphase flow using the Lattice Botzmann method will be based on an existing in-house code. As such, the first step in developing the new solver is to acquire a good understanding of how the previous solver works, specifically, it's structure and organisation, it's input and output data, it's performance and it's depending software (if any). Furthermore, the previous code needs to be validated so as to provide and accurate foundation for the project being developed. Without this validation, it would prove to be an arduous work to discover whether the new solver is behaving correctly.

4.1.1 Code organisation

After spending some time analysing the LBM solver and debugging some of the core features, I was able to obtain a generous understanding of how the existing code is organised and where most of the necessary algorithms are located. The solver itself was very well documented and most of the solver's features were written in a very user-friendly way.

Because of this good work from the authors of this solver, the following flow chart, Figure 4.1, was able to be made in a relatively small amount of time, which helps to gain a general understanding of the solver.

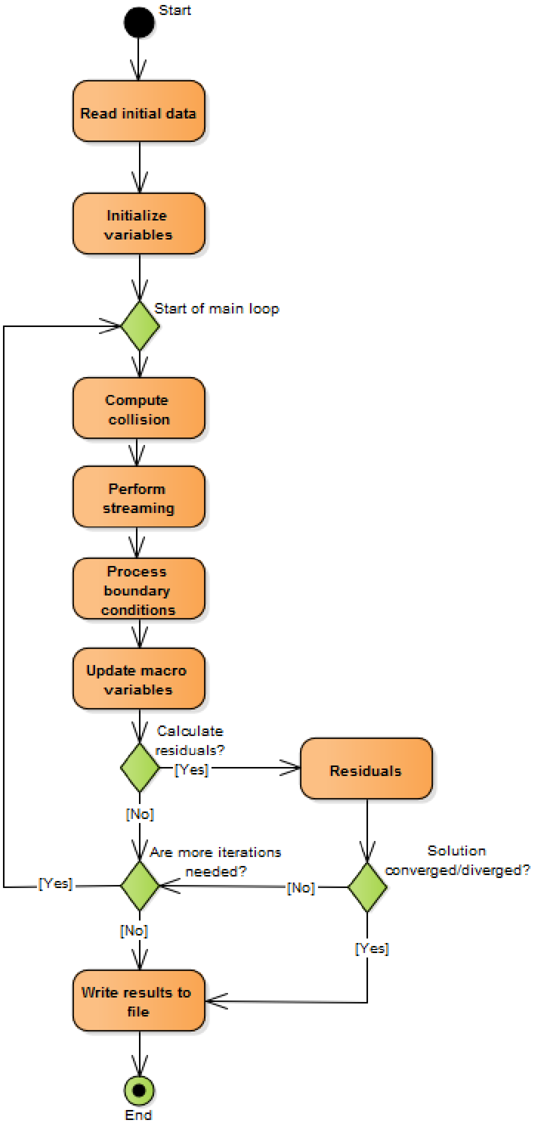


Figure 4.1: A simplified flowchart of the solver's activity

The previous authors strived for a optimized and easy to use code. As such, the SetUpData.ini file contains all the information needed to build the problem to be solved.

This file, combined with the mesh files that the user wishes to simulate, provide the initial step in running the solver. Note that external calculations need to be used to guarantee some aspects of the initial conditions, such as the Reynolds number.

Similarly, when the solver finishes computations, a Results folder is created containing information pertinent to the solution of the solver. These files include the final solution, the residuals, the run time, etc. The user also has the option of selecting which output format to produce the solution in (.vti, .dat or .csv).

4.1.2 Data representation

The original code mostly handles multidimensional arrays, following the same logic from the 2D solver onto the 3D version. As such, the arrays either store the macroscopic value of each node (h * m * n in 3D) or the coefficient for the microscopic values of each lattice (19 * h * m * n in 3D) . However, when using CUDA kernels, we will often have the need to copy data from the host (CPU) to the device (GPU), which can prove to be a tiring and inefficient task when dealing with 2, 3 or even 4 dimension arrays. To solve this problem, the previous authors flattened every array, which means that all arrays are represented as one dimensional arrays. Because of this, some calculations are required to access the desired index, as demonstrated by Fig.

These arrays are stored in a row-major fashion and the lattices are grouped together in 9 or 19 arrays, depending on the dimension of the problem, meaning that neighbours are separated from each other by m * n or h * m * n elements.

4.1.3 Performance

This thesis will be based on the work of previous Cranfield MSc students. As such, the received solver needs to be validated regarding the established performance in Kubat's thesis [14]. To this end, I generated the same meshes as the ones used in his thesis, specifically the lid driven cavity 128 and 256.

The lid driven cavity is a commonly used benchmark test for CFD solvers. It consists of a cube with a moving lid on the top which acts as the inlet for the flows. The numbers refer to the number of nodes in each direction, i.e. cavity 128 represents a 128x128x128 cube, resulting in 2097152 nodes.

The final run-times of both meshes were very similar to the ones previously obtained, as shown in Figure 4.2. Because of this, we can be sure that our version of the solver is the same as the final one used by the previous authors during their theses.

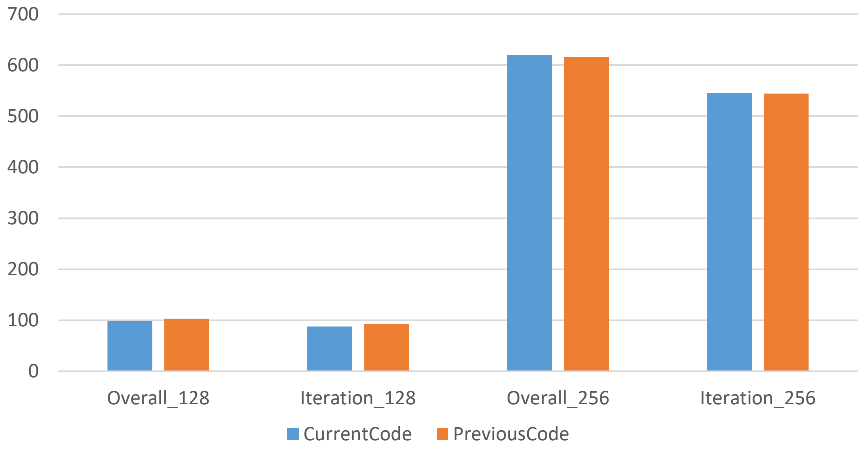


Figure 4.2: Runtime for Cavity_128 and Cavity_256

To gain a better understanding of the code, the solver's run-times were profiled according to the main phases of the method. Figure 4.3 shows that the solver spends about 40% of it's time calculating residuals. The residuals are used to check whether the solution has converged or diverged, making them a valuable method for potentially saving computation time (which in HPC centres means saving money). However, if the user is sure that the problem being solved needs to run for the specified iterations, he might be able to speed up the solver by not calculating the residuals, or even by specifying an interval of iterations before calculating residuals again.

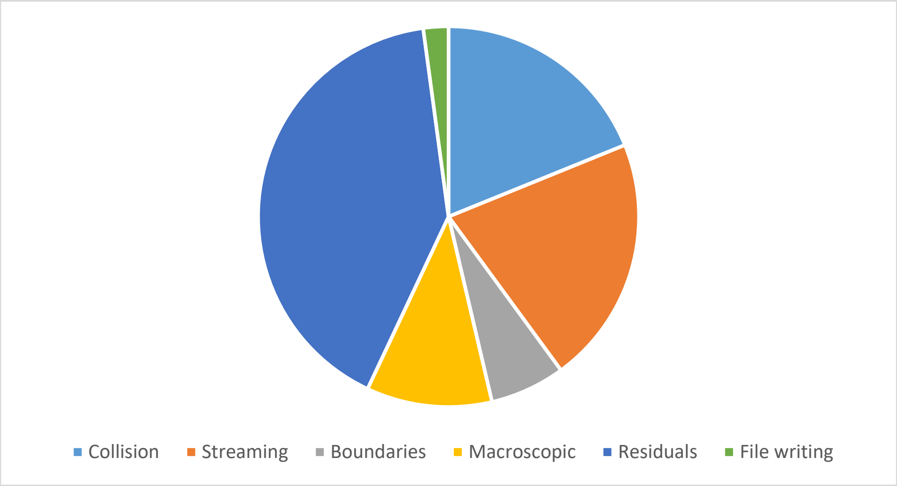
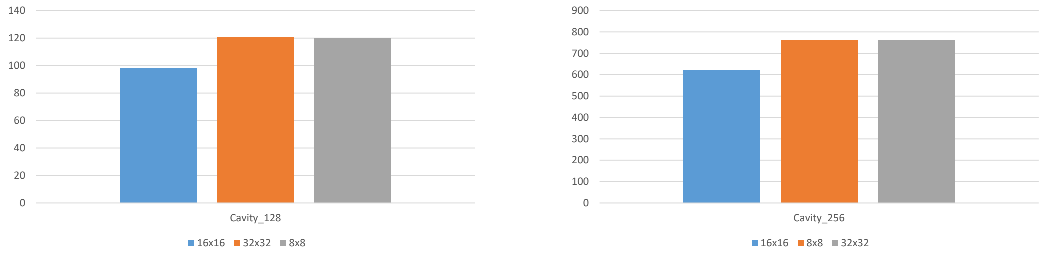


Figure 4.3: Code profiling for the run-times of Cavity_128 with MacroDiff residuals

The remaining time is split between the collision, streaming and macroscopic values calculation, with boundary condition calculations and file writing occupying a less important slice of execution time. With this profiling, we now know which are the most critical parts of the software (residuals, collision and streaming) and can focus our efforts in optimising those areas when developing the method for multiphase flows.

Finally, the received solver was programmed to run with a 2D configuration in the kernel calls. Both the blocks per grid and threads per block are set up to run with an equal number of elements in the two directions. The number of blocks is calculated dynamically with problem size in mind and sets 1 block per node. However, the number of threads per block is declared statically with a value of 16x16. This means that the solver will use 16 threads in the x direction and another 16 in the y direction per block, meaning that each block uses 256 threads. To understand how this value affects the solver, I have benchmarked the solver using two more configurations for the Cavity_128 and Cavity_256 meshes.



(a) Cavity_128

(b) Cavity_256

Figure 4.4: Overall time comparison using three different threads per block configuration

Figure 4.4 shows that the initial configuration (16x16) provides the best results. Because of this, the parallelisation strategy for the multiphase algorithm development will also start with this number of threads per block.

4.1.4 Validation

Now that the received code has been validated to perform under the same conditions as the final version used by the previous authors, we now have to verify whether the results that the received version is producing are the same as the final one. The validation of the results obtained throughout this project will be under the responsibility of Antonio

González. As such, the details for this initial validation can be found in his thesis [?].

This initial validation is essential for the development of a multiphase flow using the in-house solver. Without it, there would be no way of telling whether the solver would be functioning incorrectly due to a faulty algorithm implemented by us or if the solver was already producing wrong results. By doing so we can be sure that our code behaves correctly and that a proper result is achieved.

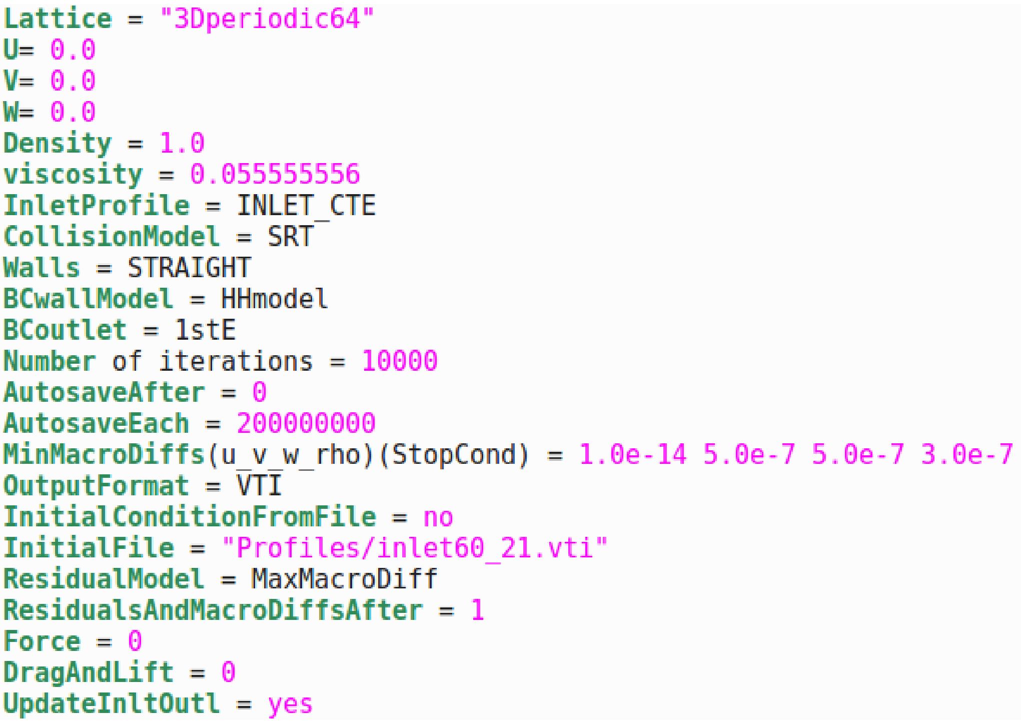
4.2 Developing the Color Gradient model

After making sure that the code we received was producing correct results, we could start implementing the Color Gradient (CG) model in the in-house LBM solver. Since there are two people working on this project, a proper separation of responsibilities was vital for a successful adaption of the solver. To this end, Antonio González [?] was in charge of the physical aspects of the model whilst I was in charge of the software side.

We first decided to implement the CG model in a serial fashion so as to easily and quickly find potential errors in the code. Antonio would then develop a working script of the model in Matlab and, after validating it, I would take that script and patch it into the LBM solver. After making sure that the model was working in full integration with the LBM solver, I could then start parallelising the algorithms and finally validate the final, parallel results. We felt that this process, while extensive and time consuming, presented the best way of correctly implementing the model, since it was easier to track bugs, and so was repeated for the 3D model.

4.2.1 Initial arguments

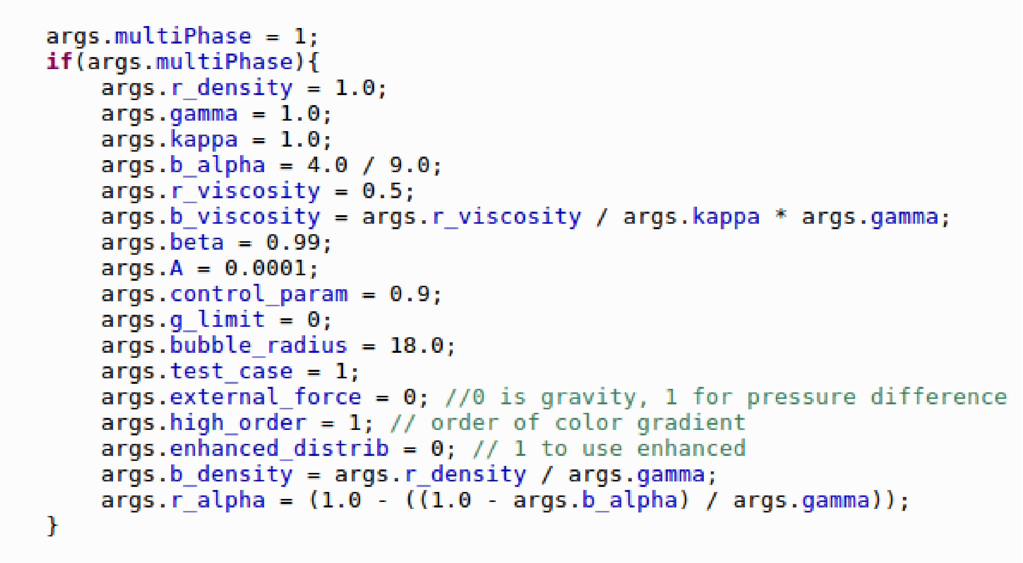
The LBM solver depends on some initial parameters to initialize the required variables correctly. To do so, a SetUpData.ini file needs to be filled with the desired values. Figure 4.5 shows the initial arguments on which the LBM solver depends upon.



```
Lattice = "3Dperiodic64"
U= 0.0
V= 0.0
W= 0.0
Density = 1.0
viscosity = 0.055555556
InletProfile = INLET_CTE
CollisionModel = SRT
Walls = STRAIGHT
BCwallModel = HHmodel
BCoutlet = 1stE
Number of iterations = 10000
AutosaveAfter = 0
AutosaveEach = 200000000
MinMacroDiffs(u_v_w_rho)(StopCond) = 1.0e-14 5.0e-7 5.0e-7 3.0e-7
OutputFormat = VTI
InitialConditionFromFile = no
InitialFile = "Profiles/inlet60_21.vti"
ResidualModel = MaxMacroDiff
ResidualsAndMacroDiffsAfter = 1
Force = 0
DragAndLift = 0
UpdateInltOutl = yes
```

Figure 4.5: Setup file for the LBM solver

However, the CG model requires several additional parameters. These parameters had to be added to the solver and can be viewed in Fig. 4.6.



```
args.multiPhase = 1;
if(args.multiPhase){
    args.r_density = 1.0;
    args.gamma = 1.0;
    args.kappa = 1.0;
    args.b_alpha = 4.0 / 9.0;
    args.r_viscosity = 0.5;
    args.b_viscosity = args.r_viscosity / args.kappa * args.gamma;
    args.beta = 0.99;
    args.A = 0.0001;
    args.control_param = 0.9;
    args.g_limit = 0;
    args.bubble_radius = 18.0;
    args.test_case = 1;
    args.external_force = 0; //0 is gravity, 1 for pressure difference
    args.high_order = 1; // order of color gradient
    args.enhanced_distrib = 0; // 1 to use enhanced
    args.b_density = args.r_density / args.gamma;
    args.r_alpha = (1.0 - ((1.0 - args.b_alpha) / args.gamma));
}
```

Figure 4.6: Initial arguments for the Color Gradient model

This model is supposed to be an addition to the original solver. As such, we need to keep the non-multiphase approach interchangeable with the new model. This was taken into account throughout the development of the code and so the user is provided with a way to easily switch between approaches.

The number of new parameters further shows how much the complexity of the solver will increase due to the new model. A higher complexity translates into more computations being performed, which in turn means that the CG model will hinder the original solver's performance. I will try to mitigate these effects during the parallelization of the solver, which I describe in more detail in Section 4.6. Also, two new distribution functions need to be added to the solver (one per fluid), as well as two new densities. All of this needs to be taken into account when developing the new solver.

4.2.2 2D model

After initializing all the parameters, we need to initialize the new distribution functions and the new densities. This is done depending on the test case to be used, where each distribution function and density array are initialized according to the initial parameters and to the region of the fluid. The test cases are explained in more detail in Section 4.4.

As mentioned in Section 3.2, the Color Gradient model imposes some changes on the sub-steps of the LBM solver. The most noticeable of theses changes happen in the collision sub-step. First, we need to calculate the color gradient of each node. Since this color gradient is the result of a sum of interactions from the 9 directions of the speed model, we need a separate loop before being able to use this array in the computation of the perturbation operator. However, the color gradient calculation depends on the orientation of the node, that is whether the node is in the north, south, east, west or middle of the mesh. We can then calculate the appropriate collision frequency and afterwards compute the perturbation, collision and recoloring operators by looping between the 9 directions. For the most part of this sub-step, parallelization is implemented by following the example of the solver's previous collision sub-step. Each node will have it's own thread to perform calculations, and since each node does not depend on it's neighbors we do not have to worry about conflicted access to memory. However, each thread needs to know it's orientation for the color gradient calculation. Instead of having each thread reverse calculate it's index in an x,y format, a new array was created and initialized in the initialization section of the solver. This array has an element per node and it's value represents the orientation of the node in the following manner:

- 1 - North
- 2 - South

- 3 - East
- 4 - West
- 0 - Middle
- -1 - Corner

This way each thread simply needs to access this array with it's index to discover which orientation it needs to take into account.

The streaming step behaves like the original solver's streaming. We now have to stream not just one distribution function but both of them, as each fluid's changes are specific to it's own function. However, the 2D model was not behaving correctly when nodes near the boundary needed to be updated. This could be fixed by analyzing the mesh interpolation and how the stream array is built, which specifies whether a direction of the speed model should be updated or not. Since there was not enough time to perform this analysis, I used the same array as the one used in the color gradient to find out the orientation of node. Then I simply needed to specify how the streaming should work for each orientation.

The boundary step also behaves like the original solver's boundary update. However, the tests used to validate the Color Gradient model mostly use periodic boundaries, which did not exist in the 2D version of the previous solver. As such, these boundaries needed to be implemented and added to the 2D version of the solver. Furthermore, the boundaries also need an interpolation of the mesh in order for each thread to know what type of boundary that node should be updated with. Since the lattice generator software was out of scope for this thesis, I again used the same array built with the orientations for the color gradient model. This way, each thread could identify which boundary condition should be applied depending on the position of the node.

The macro variable updates need to be replicated for both distribution functions and densities. Also, the calculation of the velocity needs to take both fluids into account. Other than that, it behaves in the same manner as the original solver.

Finally, the residuals calculation also needed to be updated since the previously implemented residuals based on the sum of absolute differences between each iteration's distribution function was not properly obeying the convergence criteria for our multiphase approach. Now, the residuals are based on the maximum difference of the distribution function and the user can choose which should the convergence criteria be in the setup file.

4.2.3 3D model

The 3D model adaptation was much easier than the 2D implementation since for the most part it was just a matter of extending the computations on the distribution functions, from 9 to 19 directions. The core logic behind every operation was the same. However, the collision model needed to be updated since the collision frequencies, perturbation operator and equilibrium distribution function calculation were based on a enhanced version of the 2D model's version. Eventually we also modified the 2D version to use this enhanced collision step since it was meant to provide more accurate results.

One of the most significant improvements from implementing the 3D model was that the previous solver had more enhancements in 3D than in 2D. We could now use the proper streaming and boundary steps since there were no complications when generating 3D meshes and the software could adequately interpolate these meshes and initialize control arrays correctly.

4.3 Validation

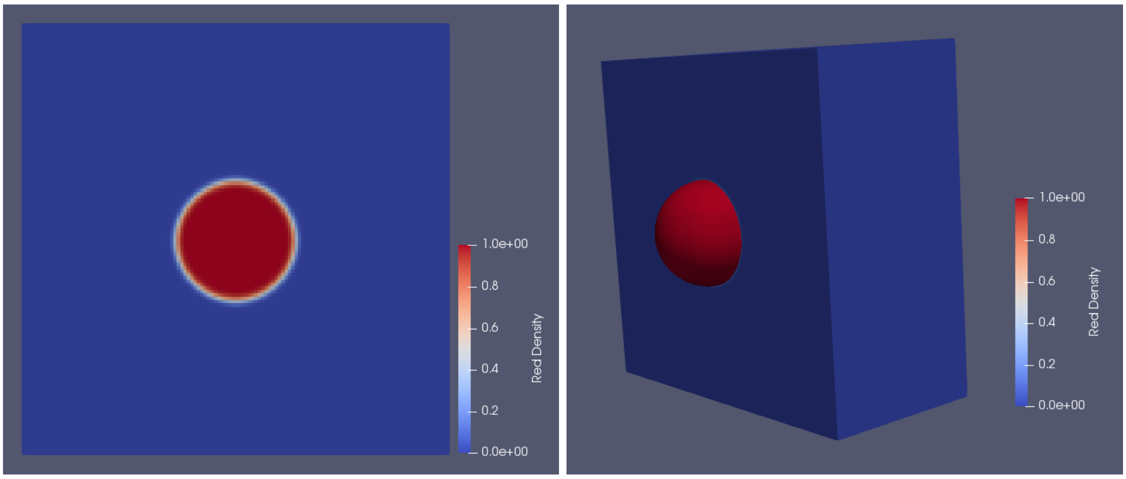
The validation of the results was performed by Antonio González [?]. To do so, every test case was implemented according to their specific needs. Almost every case was based on the same mesh, a simple square or cube depending on the dimension. The only exception was the Rayleigh-Taylor instability where a rectangular mesh was used. Appendix Ref appendix validation shows that the model produces correct results for every test case.

The solver has the option of producing results in different formats, .dat, .vti and .csv. .VTI files were the addition of last year's authors and can be read using Paraview [3]. The main advantage of these files is that they require significantly less memory than the other formats. Because of this, we are able to save time by transferring the results from Cranfield's HPC center faster and also by making post-processing analysis less time consuming. For this reason, we extended the creation of these files to use the new multiphase model arrays.

4.4 Test cases

The validation of the CG model depends not only on the analysis of the actual result of the simulation, but also in the type of test case used. Each test case had different requirements in the software since they depended on different arrays and each used their own type of verification. Because of this, I added the option to change between test cases as an argument in the setup file. During the execution of the solver, this options allows for a correct initialization of the densities and distribution functions, as well as adequately storing information when needed. A total of 6 test cases were implemented for the 2D model and 4 for the 3D model.

4.4.1 Steady bubble



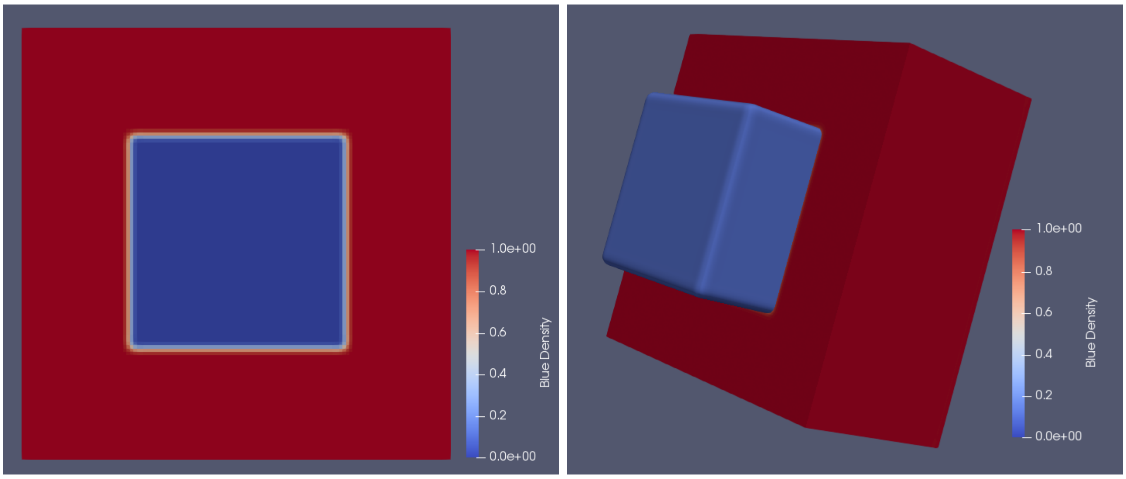
(a) 2D model

(b) 3D model

Figure 4.7: Red fluid density for the Steady bubble case

This test aims to simulate a stationary bubble of an immiscible fluid inside another immiscible fluid. Although this test is the simplest physically, it is one the most computationally expensive tests to perform. To validate this test, we need to store the evolution of the surface tension over time. This entails calculating the pressure difference inside and outside the bubble, resulting in 4 additional reduce operations in every iteration. Reduce operations are very expensive in GPUs since these are operations that are dominated by memory access, requesting a heavy amount of memory access inside the GPU as well as when transferring information between the CPU and GPU. We are likely to see an increase in performance when running simulations with other, less computationally demanding test cases.

4.4.2 Deforming bubble



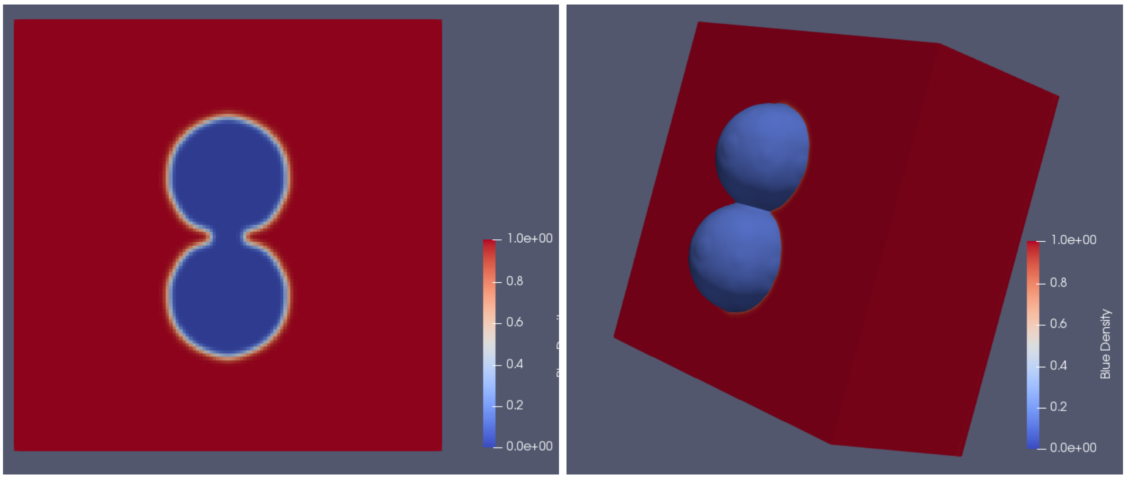
(a) 2D model

(b) 3D model

Figure 4.8: Blue fluid density for the deforming bubble case

This test simulates a fluid that starts as a deformed bubble, a square, and eventually converges into a steady, stable bubble. This test is much less computationally expensive. The only verification needed is the difference between the final bubble radius and the predicted radius, which is only done once, at the end of the simulation.

4.4.3 Coalescing bubbles



(a) 2D model

(b) 3D model

Figure 4.9: Blue fluid density for the coalescing bubbles case

This test simulates the coalescence of two bubbles of the same fluid inside another fluid. The bubbles start by having only a small point of contact between each other and then eventually coalesce into a bigger, steady bubble. As was the case with the deforming bubble test, in this scenario we only need to compare the final radius of the bubble with a predicted one, making this test also less computationally demanding.

4.4.4 Oscillating bubble

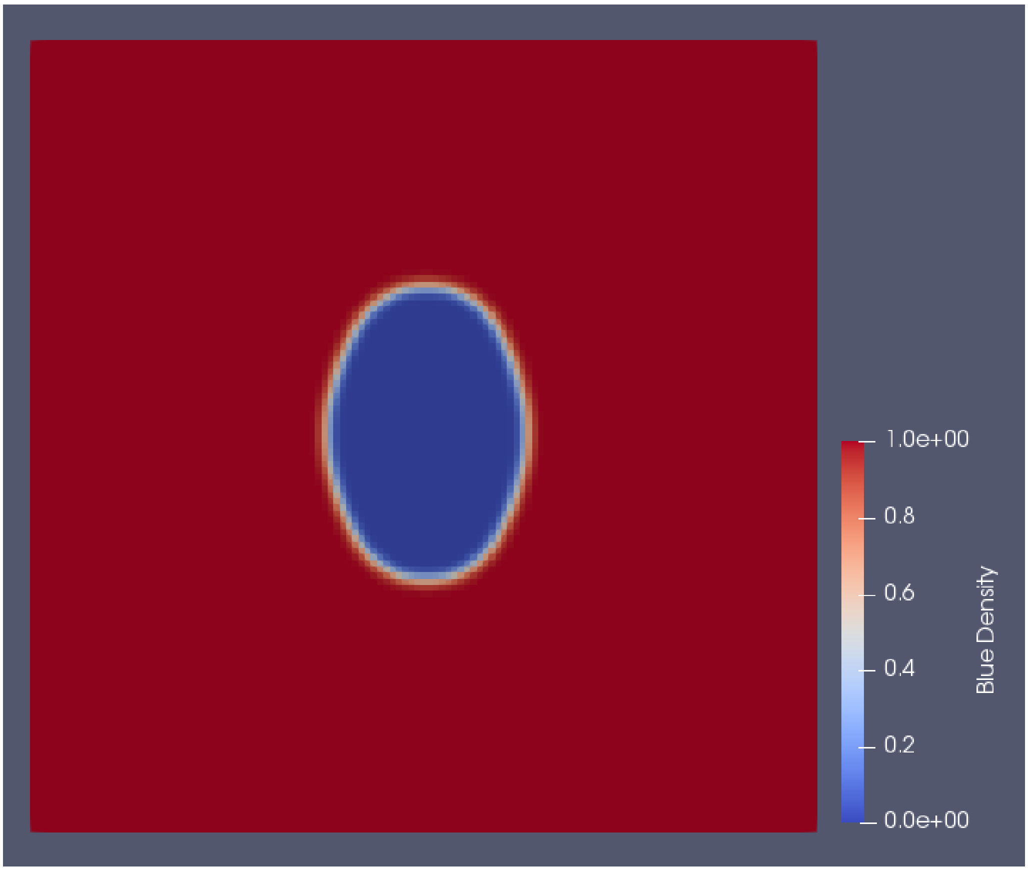
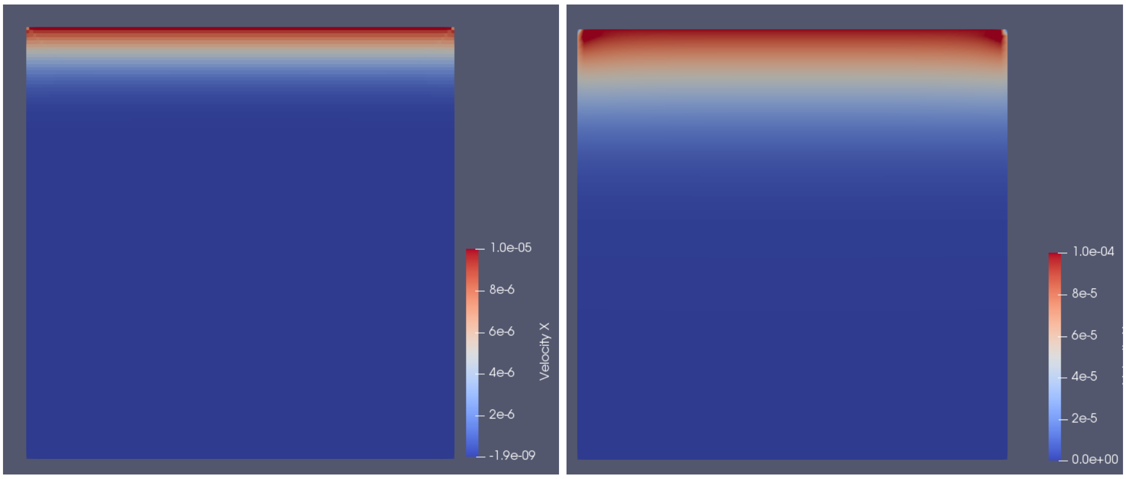


Figure 4.10: Blue fluid density for the 2D oscillating bubble

In this test, a bubble following an ellipsoid is created inside another fluid. The bubble then starts oscillating vertically and horizontally until it stabilizes into a steady bubble. For this test case, we need to find out the maximum height in y of the interface between the two fluids. This implies more computations and memory accesses per iteration which will hinder the solver's performance. This is needed so we can obtain a profile of the bubble's oscillation which will later be post-processed at the end of the simulation.

4.4.5 Couette flow



(a) 2D model

(b) 3D model

Figure 4.11: X velocity with a few iterations using the Couette flow case.

All of the previous tests were not fully using all the initial parameters, they were tests where the external conditions were of no great importance. To fully test our new model, we decided to test conditions where the boundaries of the mesh were not all periodic, and so we decided to test the Couette flow. In this case, the north boundary is an inlet with a constant velocity in x, while the south boundary is a wall where the fluid bounces back from. Once again, this test is also less computationally demanding since we only need to verify the profile of the velocity once at the end of the simulation.

4.4.6 Rayleigh-Taylor instability

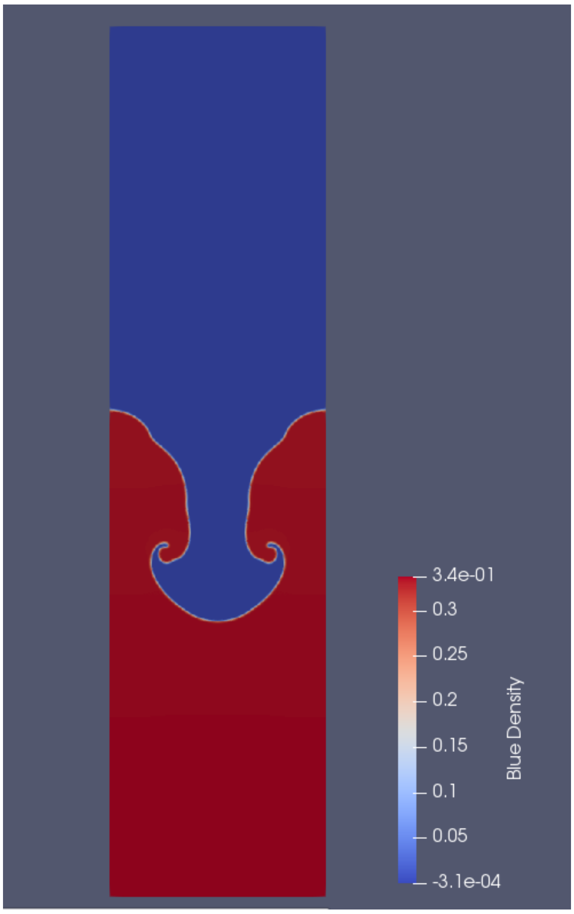


Figure 4.12: Blue fluid density for the 2D Rayleigh-Taylor instability

Finally, we chose to implement the Rayleigh-Taylor instability since in this test we can adequately test the functionalities of our CG model. Here, a heavy fluid is set on top of another, lighter fluid. Then the heavier fluid starts descending with the force of gravity and eventually starts creating vortices in the other fluid. The needs for this case are similar to the oscillating bubble case, but instead of calculating the maximum height of the interface we calculate the minimum. Because of the same memory access and number of computations, this is the third most computationally expensive test in the new LBM solver.

4.5 Memory usage

The amount of memory used is a crucial factor when performing large scale simulations. Even more so if we want the results to mimic real-word scenarios. For this reason, I have performed an analysis on the amount of memory that the new Color Gradient model requires on the LBM solver from the GPU. Tables 4.1 and 4.2 show the amount of memory used by the 2D and 3D mesh containing 128 nodes in each direction.

Variables	**Variable size**	**Memory (MB)**
Initialization	$3*m*n$	0.1875
BC initialization	$7*N_{Conn}$	0.0399475098
Macroscopic values	$9*m*n$	0.5625
Boundary conditions	$16*N_{BC}$	0.0302734375
Distribution functions	$6*9*m*n$	3.375
Residuals	$3*9*m*n$	1.6875
Temporary values	$5*m*n$	0.3125
Temporary values 2	$8*m*n$	0.5
Sum		6.7

Table 4.1: 2D memory usage for Cavity 128 with Single precision

For the 2D model, we see that the solver requires a small amount of memory, which is normal since a 128x128 is considered a coarse mesh for 2D problems. We could easily increase the number of nodes computed without affecting the required memory in a significant manner.

Variables	**Variable size**	**Memory (MB)**
Initialization	$4*m*n*h$	32
BC initialization	$8*N_{Conn}$	14.953125
Macroscopic values	$9*m*n*h$	72
Boundary conditions	$3*N_{BC}$	0.0115356445
Distribution functions	$6*19*m*n*h$	912
Residuals	$3*19*m*n*h$	456
Temporary values	$5*m*n*h$	40
Temporary values 2	$18*m*n*h$	144
Sum		1562.964

Table 4.2: 3D memory usage for Cavity 128 with Single precision

In the 3D model, the memory usage is on a completely different scale. We now see that for the same mesh as in Table 4.1, extended to another direction, the used memory increases approximately 233 times. This makes the memory used by the 2D model almost insignificant when compared with the 3D version.

To further analyze what this increase in memory means, I have prepared Table 4.3 showing the different memory requirements for both single and double precision and comparing between a 128 and 256 sized mesh.

Type	**Nodes**	**Single precision**	**Double precision**
2D_128	16384	6.32	11.7
2D_256	65536	24	46.57
3D_128	2097152	1563	3026
3D_256	16777216	12336	24054

Table 4.3: Memory usage for the Color Gradient model in MB

We can now clearly see how the memory used by the solver is one of the most important aspects to take into account. In fact, the memory requirements are so big in some cases (for the 3D, 256 sized mesh) that we cannot test the solver against this type of mesh since the maximum available global memory in a Cranfield's GPU is 11440 MBytes.

There were some ways to decrease this memory requirement. We could eliminate one of the distribution functions (and add more calculations to compensate) or change the residuals calculation to be based on the macroscopic variables instead of the distribution function. However, we found that these changes would imply a decrease in the solver's accuracy and stability, for which we chose to not implement these "improvements" in the final version of the solver.

4.6 Performance of the parallel code

In this section, we will analyze and discuss the performance obtained throughout the various stages of development of the code. We will first compare optimizations techniques employed, followed by the usage of different code and compiler features. We will then compare the different hardware used and finally the impact in performance between using double and single precision.

For the performance analysis, the tests will be performed using 10000 iterations. Running until convergence is achieved is not necessary since the times spent per iteration are very similar with each other. This means that we should run the solver for some iterations so as to disperse the initial warming up of the machine but that we do not have to run the simulation until the convergence criteria is met since 10000 iterations is enough to show the effects of the parallelization techniques.

4.6.1 GPU specification

To perform the tests, we first need the hardware on which to perform the tests on. Since we are using CUDA, we must obviously use machines with NVIDIA GPUs. Table 4.4 shows the specification of the characteristics of each machine's GPU. GRID and Delta are a part of Cranfield's HPC center, while the laptop is my own personal laptop with a less powerful GPU.

	GRID	**Delta**	**Personal laptop**
GPU	Tesla K40m	Tesla K80	GeForce GTX 850M
Memory	12288 MB	11440 MB	4044 MB
CUDA capability	3.5	3.7	5.0
CUDA cores	2880	2496	640
GPU clock rate	745 MHz	824 MHz	902 MHz
Mem. clock rate	3004 MHz	2505 MHz	1001 MHz

Table 4.4: Specification of the used GPUs

4.6.2 2D model

We will now present the results obtained in the major stages of the implementation. Each result was executed using a square mesh containing 128 nodes in both directions. We choose to display the results using this mesh since it was the mesh used for every test case and also the most commonly used in the 2D model.

Normal distribution function v1.0

This version of the code was the first parallel version using the Color Gradient model. In it, we specify the collision frequency to be used in the collision step in the initialization of the solver. Also, we calculate the perturbation operator for each distribution function, as well as the single phase collision step and recoloring operator.

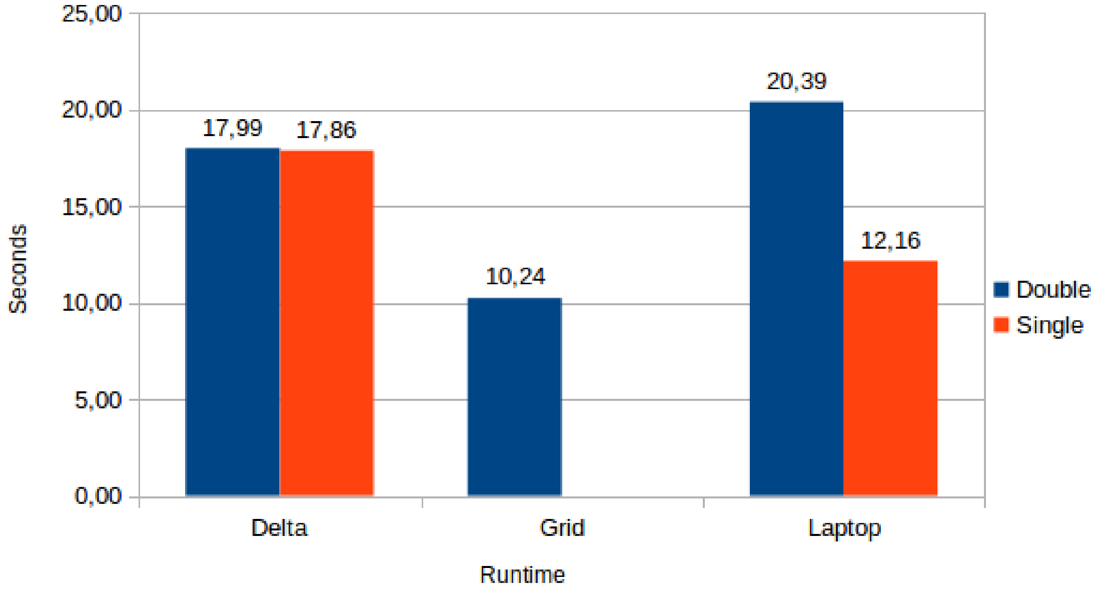


Figure 4.13: Runtime comparison for v1.0

In this version, the personal laptop displays the expected results. Double precision takes longer to compute than the single precision and both are acceptable times for 10000 iterations using a mesh containing $128x128$ nodes.

In the Delta machine results follow a different trend. Although it displays a better performance than the laptop for double precision, this does not happen for single precision. Furthermore, the difference in performance was very small, which should not happen with different precisions. However, this can be explained due to Delta's different GPU configuration. This machine is essentially two K40's joined together through shared memory and a common PCI bus, each card containing 2 GPUs. Also, Delta has extensive hardware power management, meaning that the GPU will take longer to start warming up. Small problems, like most using 2D models, might be too small too notice the better performance of this GPU. Another explanation can be the fact that Delta is a newly implemented machine in Cranfield's HPC center. Some optimizations techniques might not be lacking to fully exploit the machine's performance.

In the Grid, we encountered some portability issues, where the solver was diverging when using single precision, most likely due to a memory access that behaves differently on this machine and eventually produces NaN values. However, we can clearly see that this machine obtains better performance that the others using double precision, making it a potentially better candidate for running the CG simulations, provided that the single precision issue disappears.

Enhanced distribution function v2.0

After validating version 1.0, we decided that it would be best if we could increase the numerical accuracy of the solver, which would help in validating some test cases. We also wanted to increase the solver's performance, and so we decided to implement an enhanced version of the equilibrium distribution function, which should help achieve both goals. This version would replace the calculation of the collision frequency to make it being updated in every iteration and eliminate the need for calculating the perturbation and single phase collision operator for both distribution functions. We would only need to calculate them for another distribution function that would be the sum of both fluids.

Figure 4.14 shows some interesting results. Overall the new version performed in the same way for all of the machines. While the performance improved for single precision, this did not happen with the double precision. Our base assumption that by reducing the perturbation operator the performance would increase was correct. However, to do so we had to impose more calculations and memory accesses for the collision frequency computation, which had a bigger effect on double precision than on single precision.

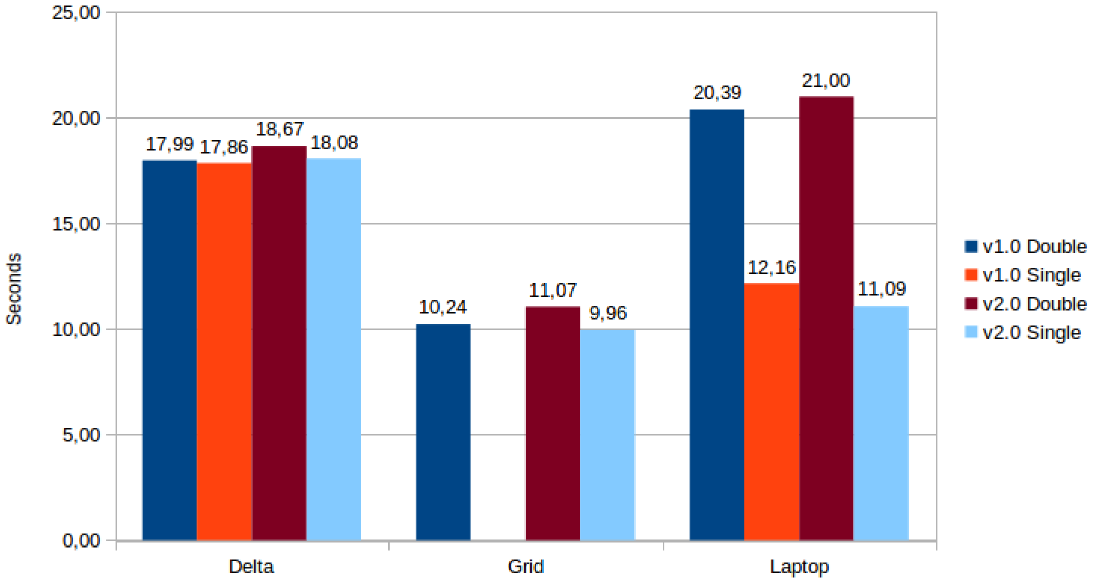


Figure 4.14: Runtime comparison for v2.0 with v1.0

In this version, we were also able to fix the Grid's portability issue in single precision. This bug was probably being caused by a memory access that was not behaving as expected in this machine which eventually was fixed by changing the calculation of the distribution function.

Optimizations v2.1

Hoping that we could still increase the performance obtained in version 2.0, we tried implementing some optimizations on the way that the collision step was being calculated. Loop unrolling is a commonly used technique to boost performance when the number of iterations in a loop inside a kernel is known beforehand. If we can tell the compiler that the loop will run for a certain number of times, then the compiler will not need to waste time in performing an if statement to check if the loop is at the end of it's iterations. To this end, we used CUDA's MACRO #pragma loop unroll n to perform the loop unrolling for us. Also, we noticed that some verifications for the collision frequency could be done beforehand, effectively removing an inefficient if statement in every iteration.

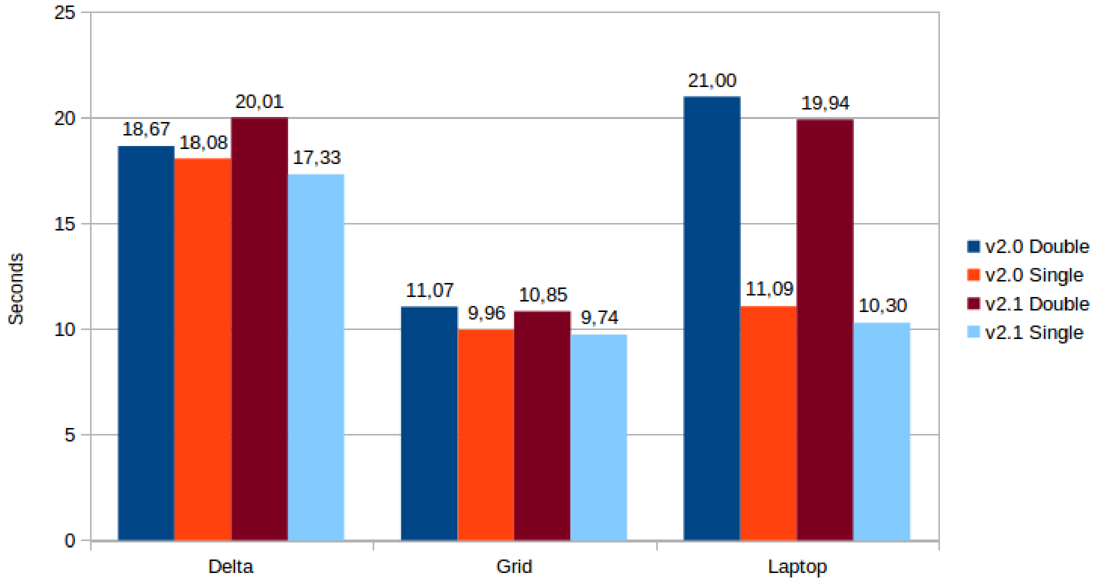


Figure 4.15: Runtime comparison for v2.1 with v2.0

We can see that overall the performance increased, the exception being for Delta under double precision. Although small, the difference indicates that we are on the right track since, hopefully, the results will show a larger difference for bigger problems, such as in the 3D case.

Loop unrolling v2.2

We decided to try manually unrolling every loop so as to not only eliminate the completeness verification in every loop, but also to eliminate some memory accesses inside the actual loop.

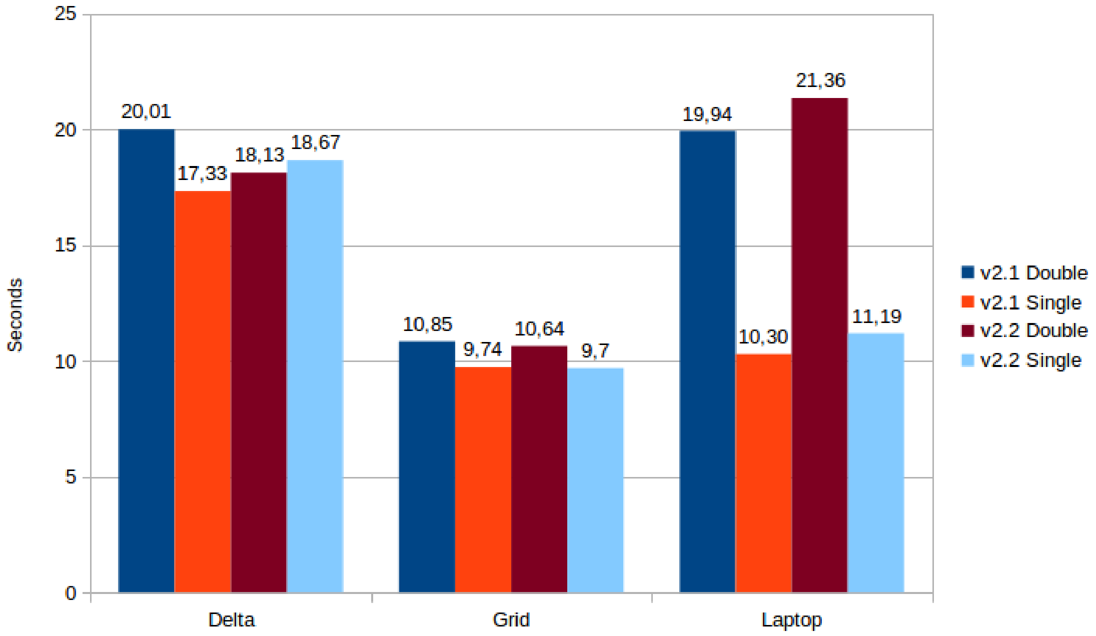
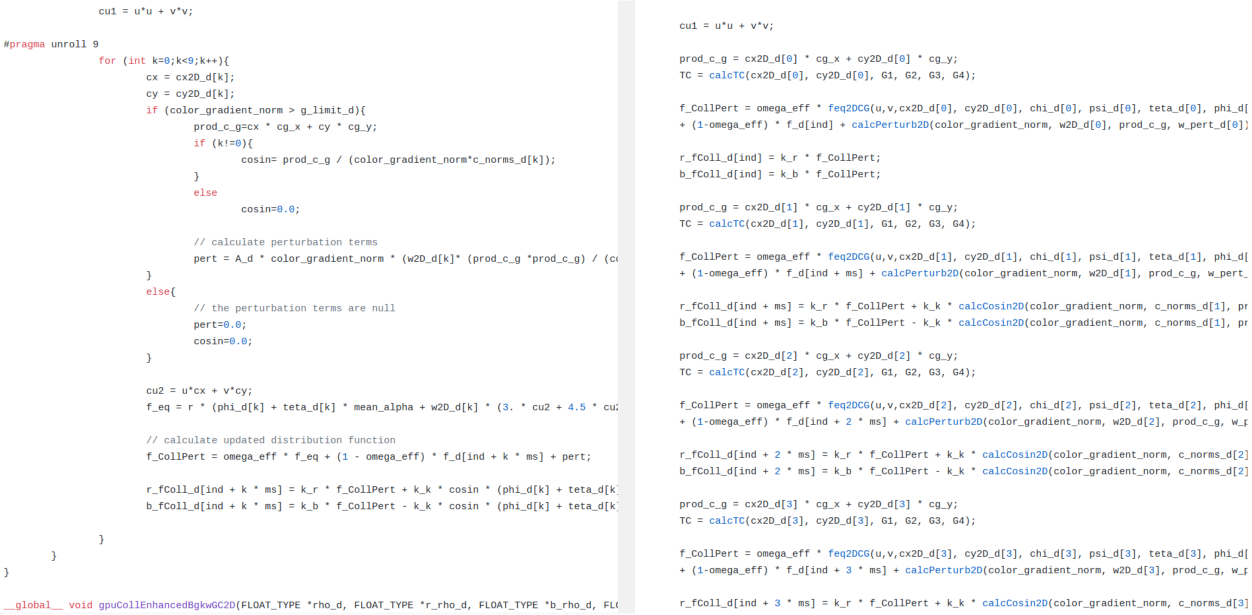


Figure 4.16: Runtime comparison for v2.2 with v2.1

In Figure 4.16 we get some unexpected results. Although the solver takes fewer seconds to execute in general, Delta's single precision now takes longer to compute than it's double precision. This can be explained by overhead times, since the machine might take longer to initialize in some instances. Furthermore, the laptop's performance got worse with version 2.2, possibly due to the compiler already loading the needed constant arrays into cache, making the manually unrolled loops introduce some unnecessary overhead into the solver.

Also, manually unrolling the loops means making the code less maintainable. We need a bigger amount of lines of code in order to perform this, which makes the code much less readable and hard to modify.



```
        cu1 = u*u + v*v;

#pragma unroll 9
        for (int k=0;k<9;k++){
            cx = cx2D_d[k];
            cy = cy2D_d[k];
            if (color_gradient_norm > g_limit_d){
                prod_c_g=cx * cg_x + cy * cg_y;
                if (k!=0){
                    cosin= prod_c_g / (color_gradient_norm*c_norms_d[k]);
                }
                else
                    cosin=0.0;

                // calculate perturbation terms
                pert = A_d * color_gradient_norm * (w2D_d[k]* (prod_c_g *prod_c_g) / (cc
            }
            else{
                // the perturbation terms are null
                pert=0.0;
                cosin=0.0;
            }

            cu2 = u*cx + v*cy;
            f_eq = r * (phi_d[k] + teta_d[k] * mean_alpha + w2D_d[k] * (3. * cu2 + 4.5 * cu

            // calculate updated distribution function
            f_CollPert = omega_eff * f_eq + (1 - omega_eff) * f_d[ind + k * ms] + pert;

            r_fColl_d[ind + k * ms] = k_r * f_CollPert + k_k * cosin * (phi_d[k] + teta_d[k
            b_fColl_d[ind + k * ms] = k_b * f_CollPert - k_k * cosin * (phi_d[k] + teta_d[k

        }
    }
}

__global__ void gpuCollEnhancedBgkwGC2D(FLOAT_TYPE *rho_d, FLOAT_TYPE *r_rho_d, FLOAT_TYPE *b_rho_d, FL(
```

```
cu1 = u*u + v*v;

prod_c_g = cx2D_d[0] * cg_x + cy2D_d[0] * cg_y;
TC = calcTC(cx2D_d[0], cy2D_d[0], G1, G2, G3, G4);

f_CollPert = omega_eff * feq2DCG(u,v,cx2D_d[0], cy2D_d[0], chi_d[0], psi_d[0], teta_d[0], phi_d[
+ (1-omega_eff) * f_d[ind] + calcPerturb2D(color_gradient_norm, w2D_d[0], prod_c_g, w_pert_d[0])

r_fColl_d[ind] = k_r * f_CollPert;
b_fColl_d[ind] = k_b * f_CollPert;

prod_c_g = cx2D_d[1] * cg_x + cy2D_d[1] * cg_y;
TC = calcTC(cx2D_d[1], cy2D_d[1], G1, G2, G3, G4);

f_CollPert = omega_eff * feq2DCG(u,v,cx2D_d[1], cy2D_d[1], chi_d[1], psi_d[1], teta_d[1], phi_d[
+ (1-omega_eff) * f_d[ind + ms] + calcPerturb2D(color_gradient_norm, w2D_d[1], prod_c_g, w_pert_

r_fColl_d[ind + ms] = k_r * f_CollPert + k_k * calcCosin2D(color_gradient_norm, c_norms_d[1], pr
b_fColl_d[ind + ms] = k_b * f_CollPert - k_k * calcCosin2D(color_gradient_norm, c_norms_d[1], pr

prod_c_g = cx2D_d[2] * cg_x + cy2D_d[2] * cg_y;
TC = calcTC(cx2D_d[2], cy2D_d[2], G1, G2, G3, G4);

f_CollPert = omega_eff * feq2DCG(u,v,cx2D_d[2], cy2D_d[2], chi_d[2], psi_d[2], teta_d[2], phi_d[
+ (1-omega_eff) * f_d[ind + 2 * ms] + calcPerturb2D(color_gradient_norm, w2D_d[2], prod_c_g, w_p

r_fColl_d[ind + 2 * ms] = k_r * f_CollPert + k_k * calcCosin2D(color_gradient_norm, c_norms_d[2]
b_fColl_d[ind + 2 * ms] = k_b * f_CollPert - k_k * calcCosin2D(color_gradient_norm, c_norms_d[2]

prod_c_g = cx2D_d[3] * cg_x + cy2D_d[3] * cg_y;
TC = calcTC(cx2D_d[3], cy2D_d[3], G1, G2, G3, G4);

f_CollPert = omega_eff * feq2DCG(u,v,cx2D_d[3], cy2D_d[3], chi_d[3], psi_d[3], teta_d[3], phi_d[
+ (1-omega_eff) * f_d[ind + 3 * ms] + calcPerturb2D(color_gradient_norm, w2D_d[3], prod_c_g, w_p

r_fColl_d[ind + 3 * ms] = k_r * f_CollPert + k_k * calcCosin2D(color_gradient_norm, c_norms_d[3]
```

Figure 4.17: Code comparison between CUDA unroll and manual unroll

Figure 4.17 shows how much the code is affected with both unrolls, with the code on the left belonging to CUDA's macro unroll and the one on the right to the manual unroll. Note that the image for the code with the manual unroll is only showing 4 out of 9 iterations. For these reasons, we decided to discard v2.2 and continue upgrading the code from version 2.1.

Final version v2.3

Finally, we decided to make one last effort to increase the code's performance and usability. We added in an option to allow the user to choose between the normal and enhanced distribution functions, as well as using a different equilibrium function, based on the enhanced one but mimicking the effects of the normal distribution function.

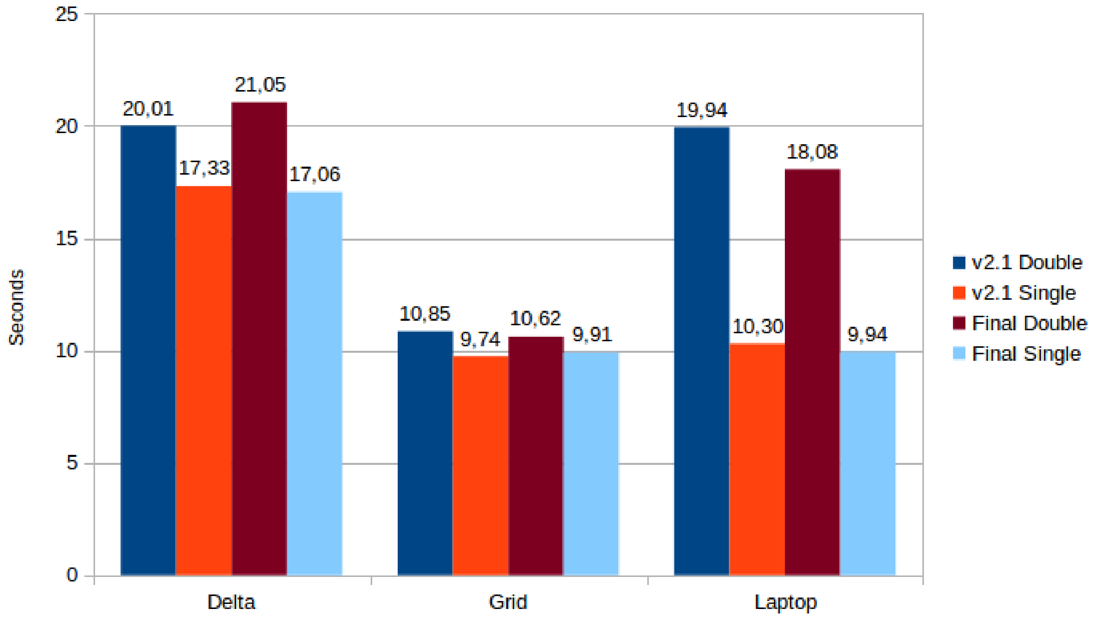


Figure 4.18: Runtime comparison for the final version with v2.1

Again we can see a slight improvement in the solver's performance, except for Delta's double precision, but again, this can be explained to a higher overhead time. Although small, these changes will be propagated to the 3D model where we will hopefully see a bigger difference.

High order color gradient

To further analyze the influence of the calculation of the color gradient in the solver's performance, we have implemented a higher order version of this phase of the collision step. A higher order color gradient means more accuracy in the final solution, at the cost of degrading the system's performance.

It is important to see how the collision step affects the total runtime of the iteration, since this is where the order of the color gradient will have most impact. To avoid analyzing a complicated chart, the chart containing the run-times for all machines can be found in the Appendix, Section .1.

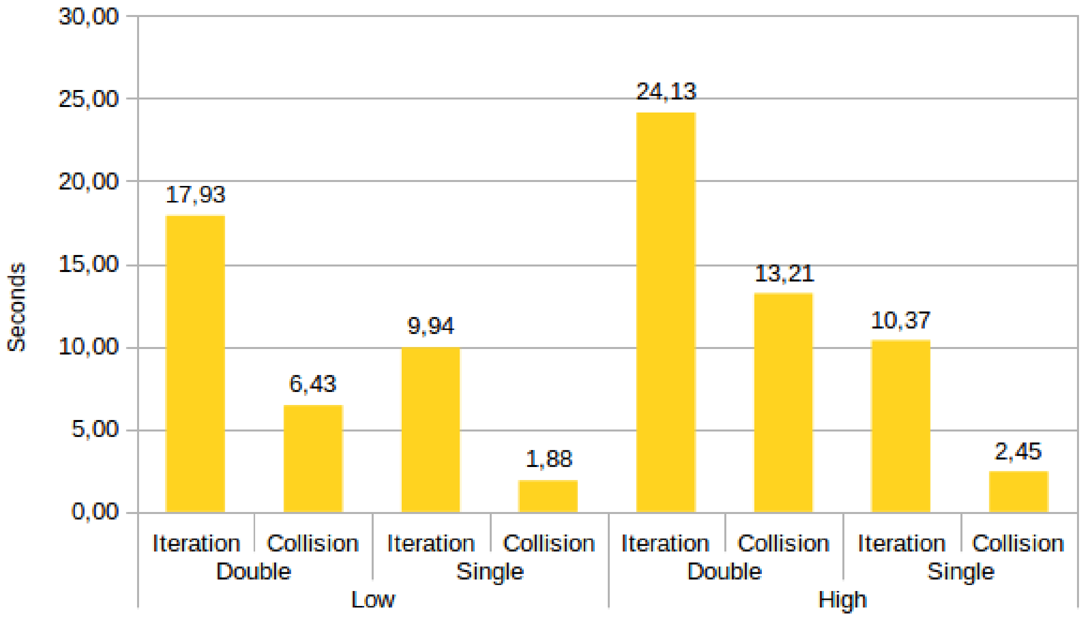


Figure 4.19: Runtime comparison between low and high order color gradient in the laptop

In Figure 4.19 we can see the degradation in performance introduced by the higher order color gradient. The time taken for the collision step doubles, severely impacting the performance of the solver. We decided that the trade-off is not advantageous in this case, but kept the option to compute the higher order version, if the user desires to do so.

CUDA compute capability

All of these tests were performed using the same CUDA compute capability for every machine, so as to obtain a fair comparison between them. However, every GPU is optimized for it's own CUDA capability, making it very important to also show the final results using each machine's adequate compute capability.

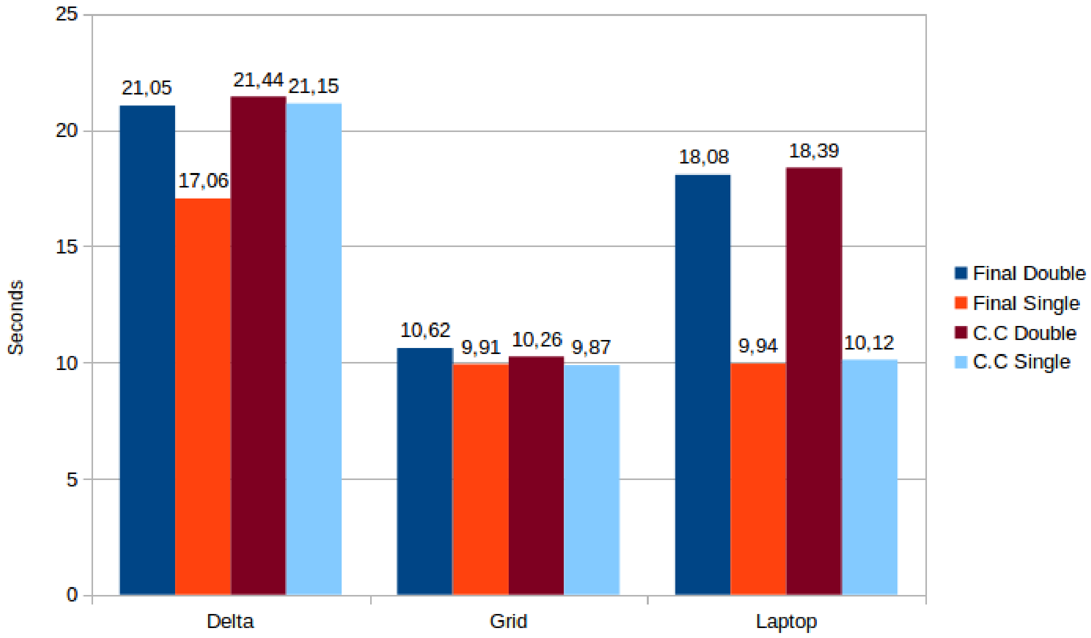


Figure 4.20: Runtime comparison for the C. capability version with the final version

However, the run-times remain almost the same between both versions, although Delta's single precision performs worse when computed with it's 3.5 compute capability. This is, nonetheless, the proper way to run the solver in each GPU, for which the user must take the machine's capability into account when compiling the program.

4.6.3 3D model

Similarly to the previous Section, we will now present the results obtained using the 3D model. Each result was obtained using a Cube-like mesh containing 128 nodes in all directions. Although this was not the mesh used when developing the solver, it is the one where we can best see the effects of our parallelization. However, using this mesh takes long amounts of time to compute under 10000 iterations. For this reason, the following tests were obtained using only Delta and Grid.

Enhanced distribution function v1.0

Unlike in the 2D model, in the 3D version we decided to start the implementation using the enhanced distribution function. This was mainly influenced by the fact that most relevant papers regarding this topic also used this enhanced distribution function. Since they presented the best way to validate our results, we decided to also use this version directly.

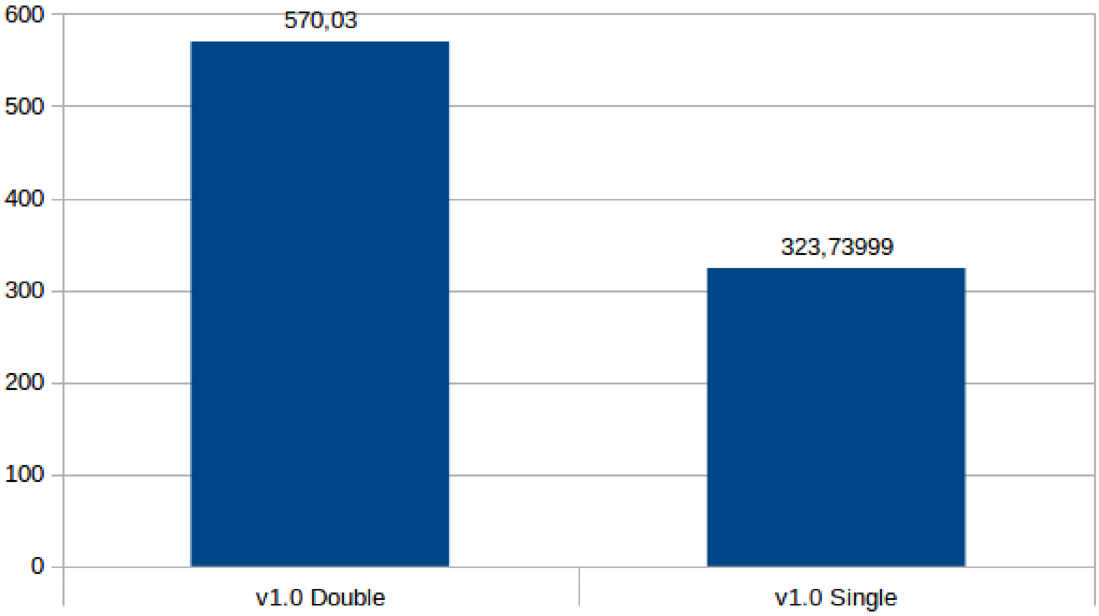


Figure 4.21: Runtime comparison for version 1.0 on the Grid

However, this version suffered from portability issues in Delta, most likely due to faulty memory accesses. Figure 4.21 shows the run-times for version 1.0 in the Grid. We now have much bigger run-times since the scale of the program has also increased. Now, the difference between double and single precision is much more noticeable, and so will, hopefully, the changes between optimizations.

Final version v1.1

We then applied the same changes that were applied in the 2D model. We unrolled the loops inside the kernel calls and improved the way that the distribution function was being calculated. All of these changes were done together since we knew from the 2D model that they had a positive effect on the final result. In this version, we did not test the manually unrolled loops. No matter how good the performance could be (which should be a couple of seconds more), Figure 4.17 shows how much worse the code gets when this technique is used for this model, even more so if we take into account that from 2D to 3D the loops increase from 9 iterations to 19, making it even harder to manually unroll them.

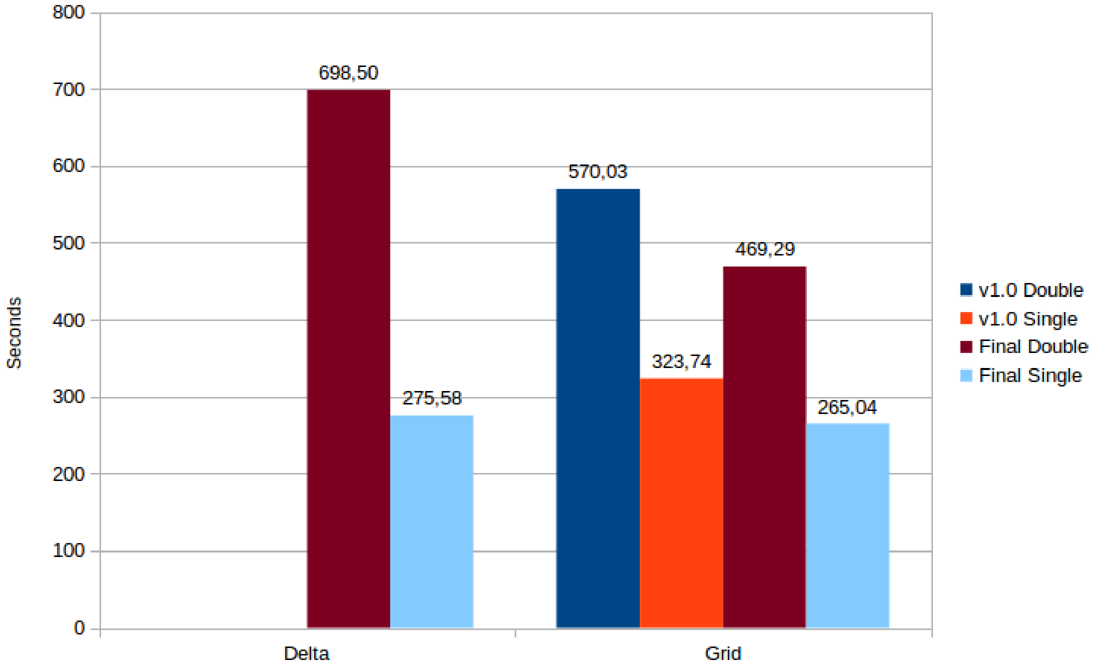


Figure 4.22: Runtime comparison between the final version and version 1.0

High order color gradient

CUDA compute capability

4.6.4 Hardware

4.6.5 Double vs Single

Chapter 5

Conclusions

References

[1] Amdahl's law. https://www.techopedia.com/definition/17035/amdahls-law. Accessed: 16-05-2017.

[2] Moore's law. http://www.mooreslaw.org. Accessed: 28-04-2017.

[3] Paraview software. https://www.paraview.org/. Accessed: 09-08-2017.

[4] B. Blocken, A. van der Hout, J. Dekker, and O. Weiler. Cfd simulation of wind flow over natural complex terrain: Case study with validation by field measurements for ria de ferrol, galicia, spain. *Journal of Wind Engineering and Industrial Aerodynamics*, 147:43–57, 2015.

[5] Christopher E. Brennen. *Fundamentals of Multiphase Flows*. Cambridge University Press, 2005.

[6] Shiyi Chen and Gary D Doolen. Lattice boltzmann method for fluid flows. *Annual review of fluid mechanics*, 30(1):329–364, 1998.

[7] R. Czyba, M. Hecel, K. Jablonski, M. Lemanowicz, and K. Platek. Application of computer aided tools and methods for unmanned cargo aircraft design. pages 1068–1073, 2015.

[8] S. Desai, E. Leylek, C.-M.B. Lo, P. Doddegowda, A. Bychkovsky, and A.R. George. Experimental and cfd comparative case studies of aerodynamics of race car wings, underbodies with wheels, and motorcycle flows. *SAE Technical Papers*, 2008.

[9] Joel Ducoste. An overview of computational fluid dynamics. Ghent University, 2008.

[10] U. Frisch, B. Hasslacher, and Y. Pomeau. Lattice-gas automata for the navier-stokes equation. *Phys. Rev. Lett.*, 56:1505–1508, Apr 1986.

[11] Andrew K Gunstensen, Daniel H Rothman, Stéphane Zaleski, and Gianluigi Zanetti. Lattice boltzmann model of immiscible fluids. *Physical Review A*, 43(8):4320, 1991.

[12] E. Kandrot J. Sanders. *CUDA by Example: An Introduction to Generalpurpose GPU Programming*. Addison-Wesley Professional, 2010.

[13] Tamás I. Józsa. Parallelization of lattice boltzmann method using cuda platform, 2014.

[14] Maciej Kubat. Development of physics or hpc optimisation of a parallel 2d lattice boltzmann solver using gpus cuda, 2016.

[15] N.S. Martys and H. Chen. Simulation of multicomponent fluids in complex three-dimensional geometries by the lattice boltzmann method. *Physical Review E - Statistical Physics, Plasmas, Fluids, and Related Interdisciplinary Topics*, 53(1 SUPPL. B):743–750, 1996.

[16] A. A. Mohamad. *Lattice Boltzmann Method*. 2011.

[17] Irene Moulitsas. “high performance computing” lecture slides, 2016. Cranfield University.

[18] D.A. Pandya, B.H. Dennis, and R.D. Russell. A computational fluid dynamics based artificial neural network model to predict solid particle erosion. *Wear*, 378-379:198–210, 2017.

[19] A. Penkner and P. Jeschke. Analytic rayleigh pressure loss model for high-swirl combustion in a rotating combustion chamber. *CEAS Aeronautical Journal*, 6(4):613–625, 2015.

[20] S. Reichrath and T.W. Davies. Using cfd to model the internal climate of greenhouses: Past, present and future. *Agronomie*, 22(1):3–19, 2002.

[21] B.S. Rosen, J.P. Laiosa, and W.H. Davis Jr. Cfd design studies for america's cup 2000. 2000.

[22] K. Sankaranarayanan, X. Shan, I.G. Kevrekidis, and S. Sundaresan. Analysis of drag and virtual mass forces in bubbly suspensions using an implicit formulation of the lattice boltzmann method. *Journal of Fluid Mechanics*, 452:61–96, 2002.

[23] Máté Tibor Szőke. Efficient implementation of a 2d lattice boltzmann solver using modern parallelisation techniques, 2014.

[24] Ádám Koleszár. Optimisation of 2d lattice boltzmann method using cuda, 2015.

Appendix

.1 Performance

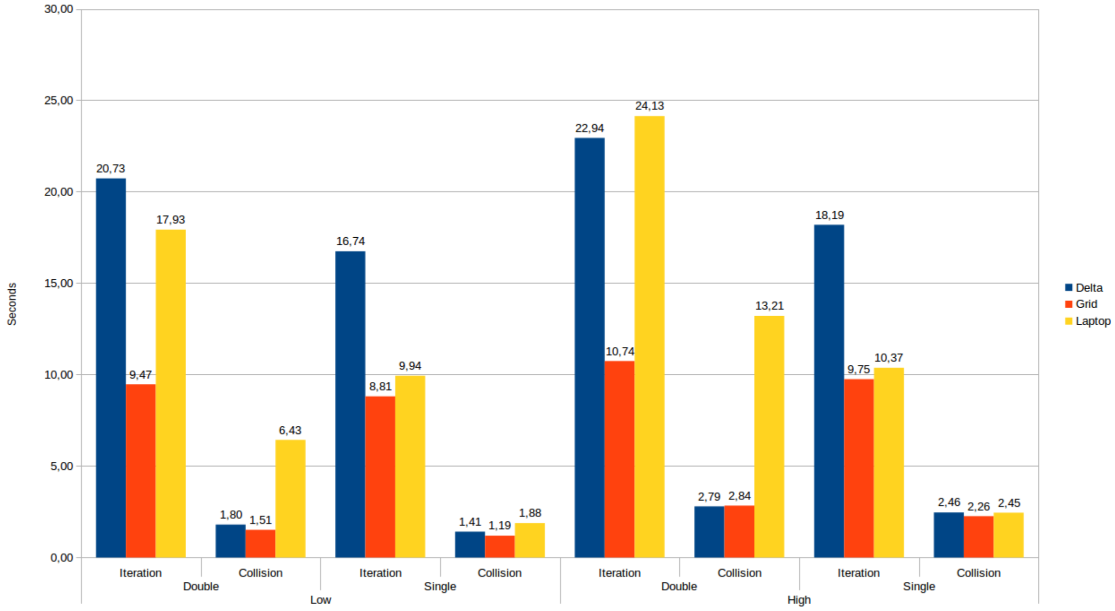


Figure 1: Runtime comparison between low and high order color gradient

.2 Validation

.3 Code