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 [6]
- Solid particle erosion [17]

- Wind flow simulation [3]
- Combustion chamber simulations [18]
- Environmental and weather prediction [19]
- Automotive and motor sports [20, 7]

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 [8]. 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 [15]. 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 [15]. 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 [9]. It is used to describe a fluid based on probabilities using the Maxwell-Boltzmann equation in the fluid's equilibrium state [15].

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 [15].

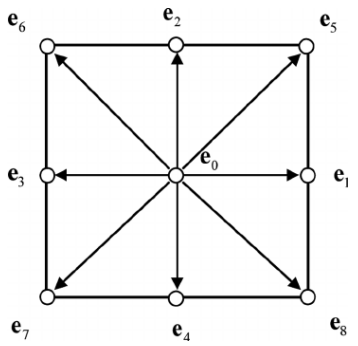


Figure 2.1: 2D LBM model using 9 particles. (D2Q9)

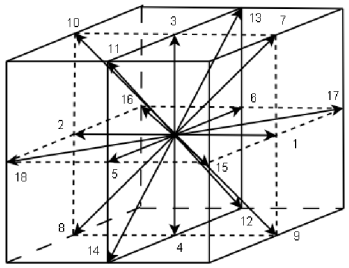


Figure 2.2: 3D LBM model using 19 particles. (D3Q19)

This way we can reap the advantages of both the macro and microscale approaches without the need of high end computers [15]. 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 [4].

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 [5].

These flows have several applications in the scientific and industrial community, ranging from pourous media fluid interactions [14] to flows containing gas bubbles dispersed in liquids [21]. Furthermore, almost every processing technology must take multiphase flows into account, including cavitating pumps, papermaking and many others [4].

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 [10] 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.

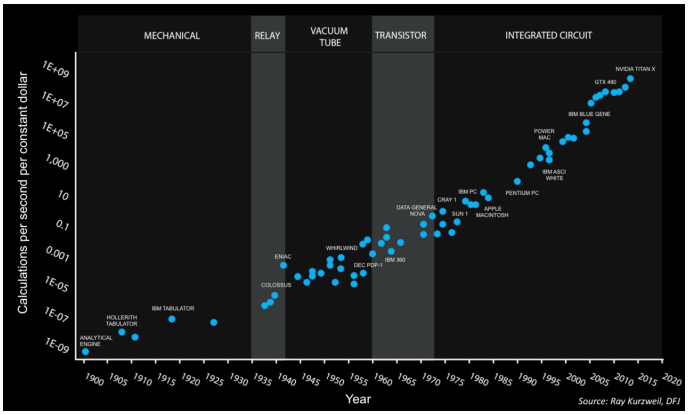


Figure 2.3: Moore's law over 120 years

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 [16]. 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.

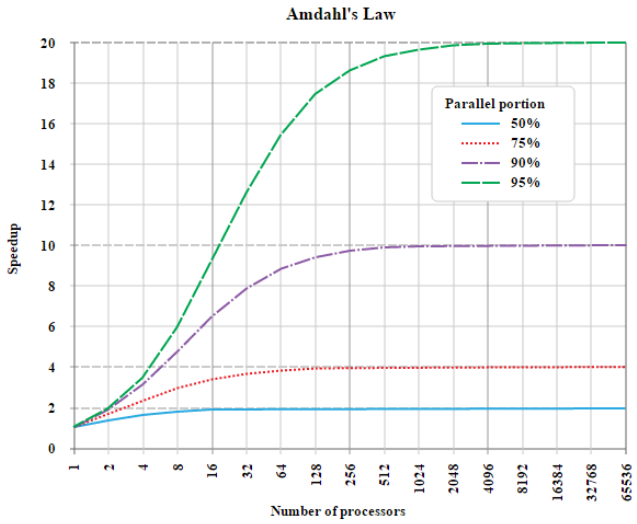


Figure 2.4: A graphical representation of Amdahl's law

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 [11]. 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 [12, 22]. 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 [12].

Szőke proposes a CPU parallelisation approach using Unified Parallel C, which is a Partitioned Global Address Space language[22]. 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) [22].

In 2015, Ádám Koleszár continued the work and further optimised the parallel version of the LBM method using CUDA [23]. 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 [13]. 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.

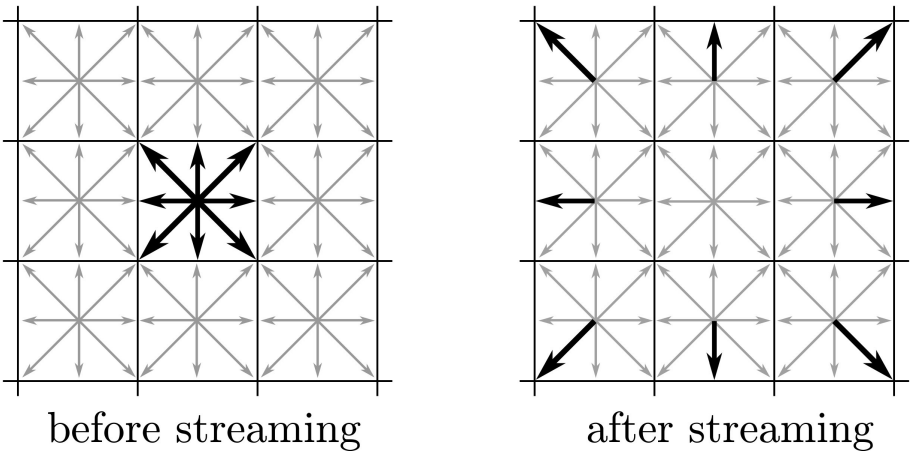


Figure 3.1: Streaming step following a 2DQ9 model.

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:

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

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

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

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.

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

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.

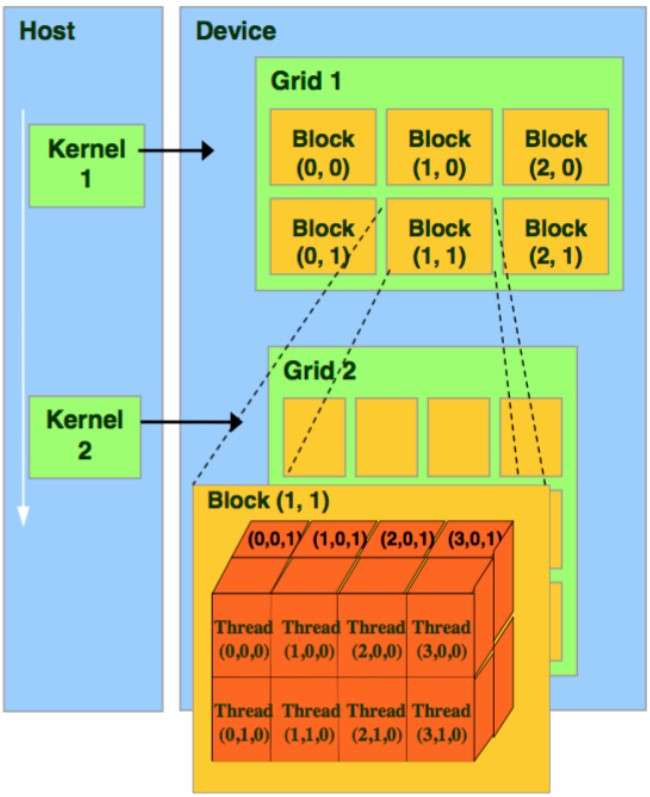


Figure 3.2: Thread arrangement in CUDA platform.

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.

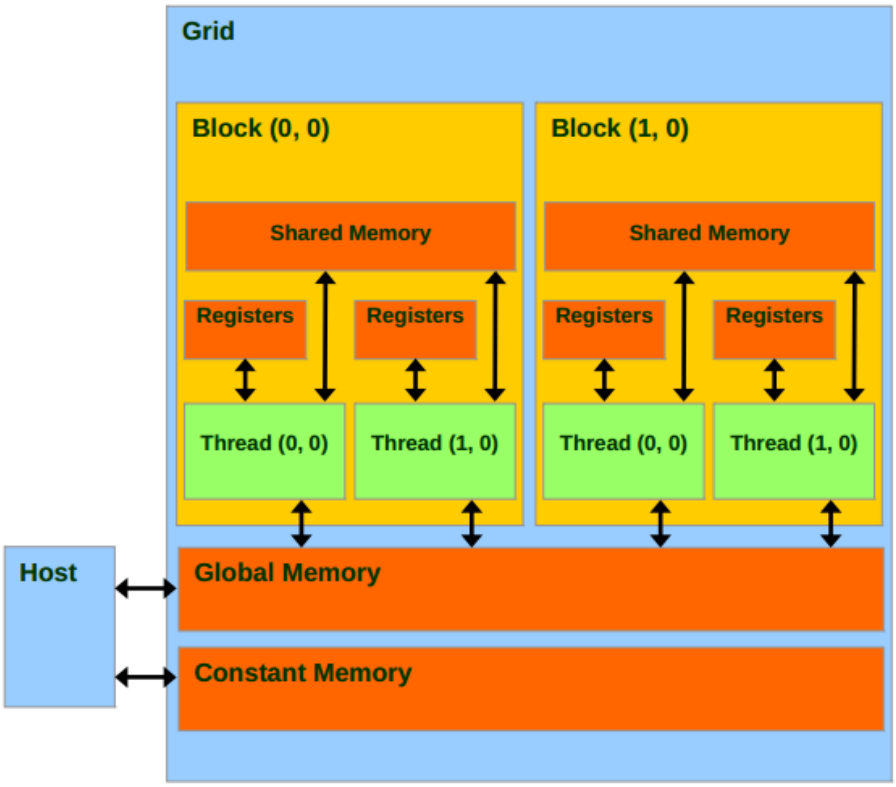


Figure 3.3: Memory arrangement in CUDA compliant GPU.

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 es-

sential 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.

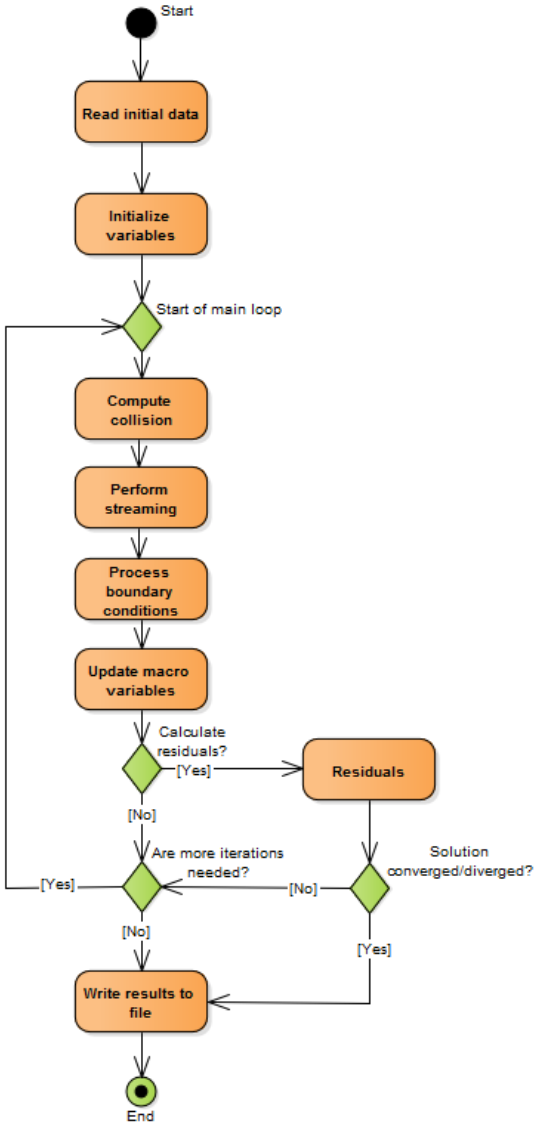


Figure 4.1: A simplified flowchart of the solver's activity

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 [13]. 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.

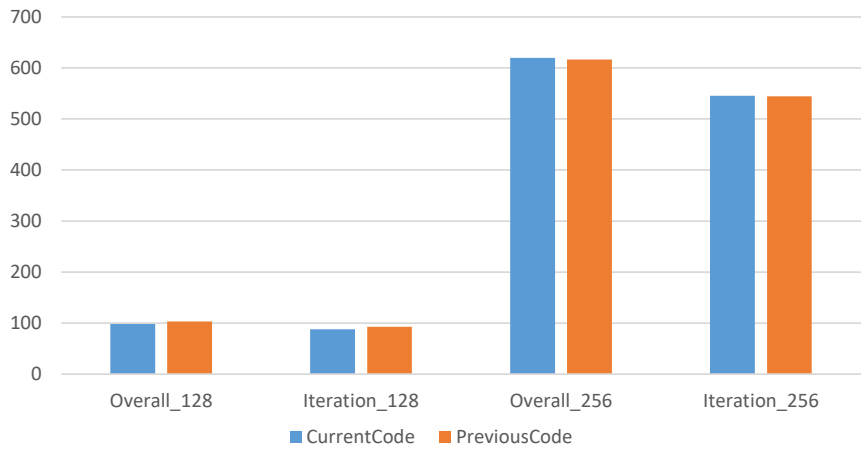


Figure 4.2: Runtime for Cavity_128 and Cavity_256

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 so-

lution 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.

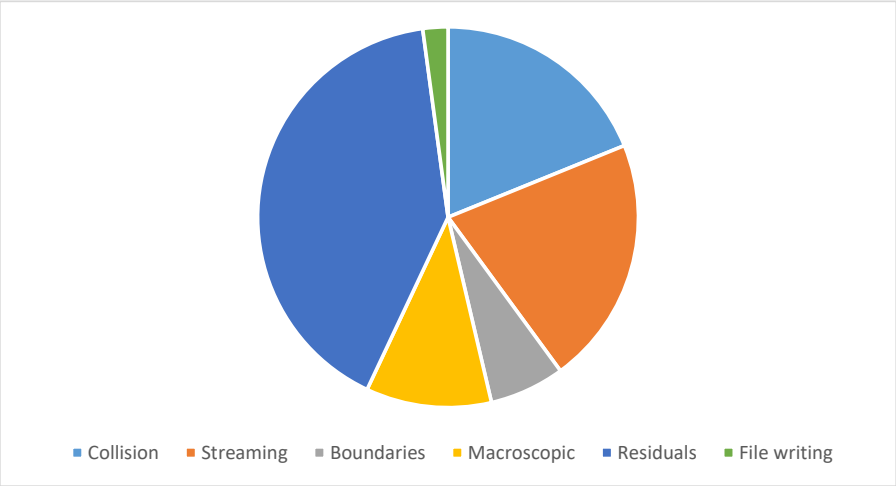
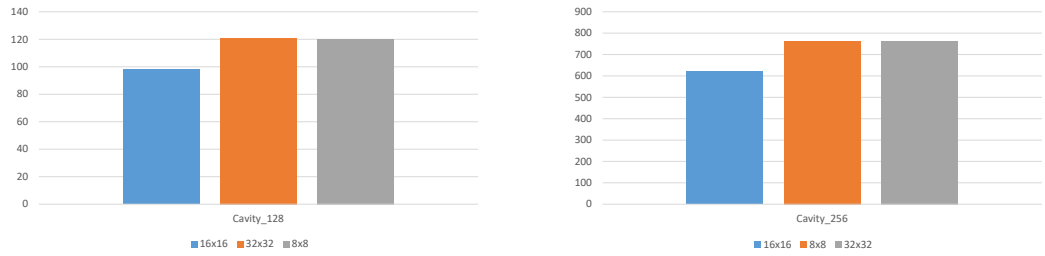


Figure 4.3: Code profiling for the run-times of Cavity_128 with MacroDiff residuals

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.

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] 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.

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

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

[6] 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.

[7] 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.

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

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

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

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

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

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

[14] 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.

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

[16] Irene Moulitsas. "high performance computing" lecture slides, 2016. Cranfield University.

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

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

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