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 [18]

- Wind flow simulation [3]
- Combustion chamber simulations [19]
- Environmental and weather prediction [20]
- Automotive and motor sports [21, 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 [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].

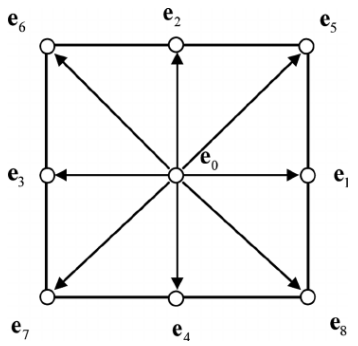


Figure 2.1: 2D LBM model using 9 particles. (D2Q9)

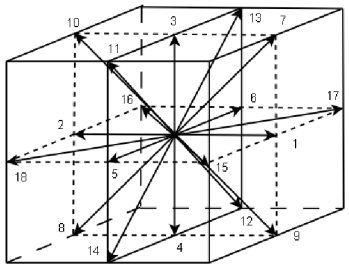


Figure 2.2: 3D LBM model using 19 particles. (D3Q19)

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 [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 [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 [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 [11].

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.

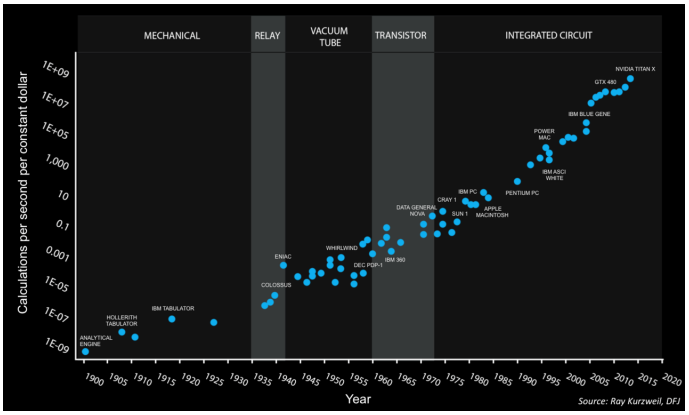


Figure 2.3: Moore's law over 120 years

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 [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.

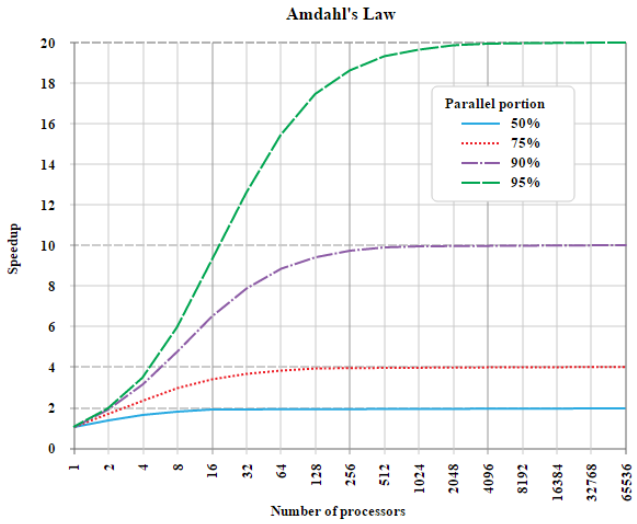


Figure 2.4: A graphical representation of Amdahl's law

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 [8]. 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 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.

3.1.1 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 3.1, was able to be made in a relatively small amount of time, which helps to gain a general understanding of the solver.

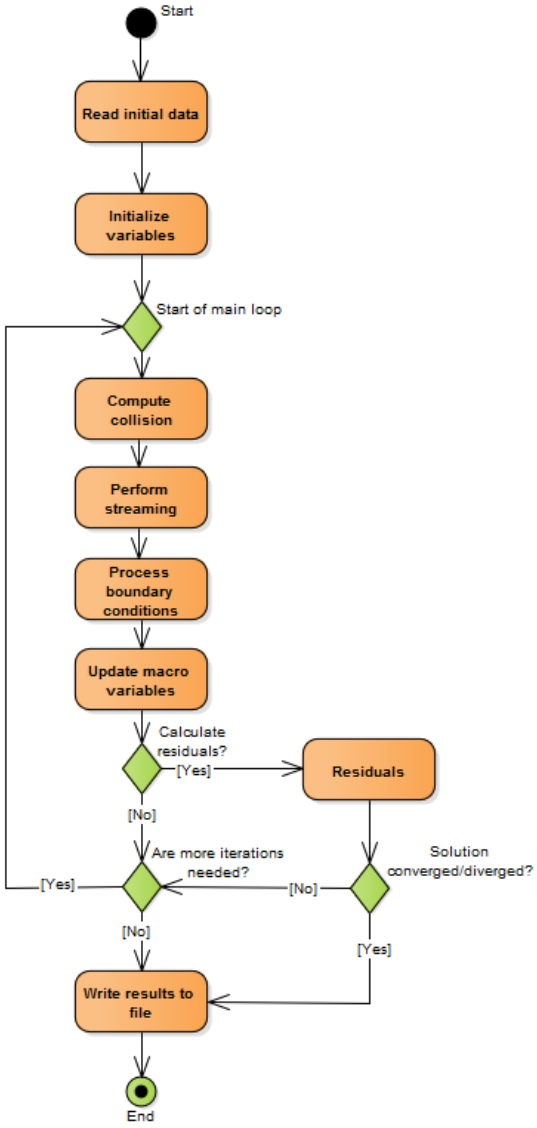


Figure 3.1: A simplified flowchart of the solver's activity

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).

3.1.2 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 3.2. 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.

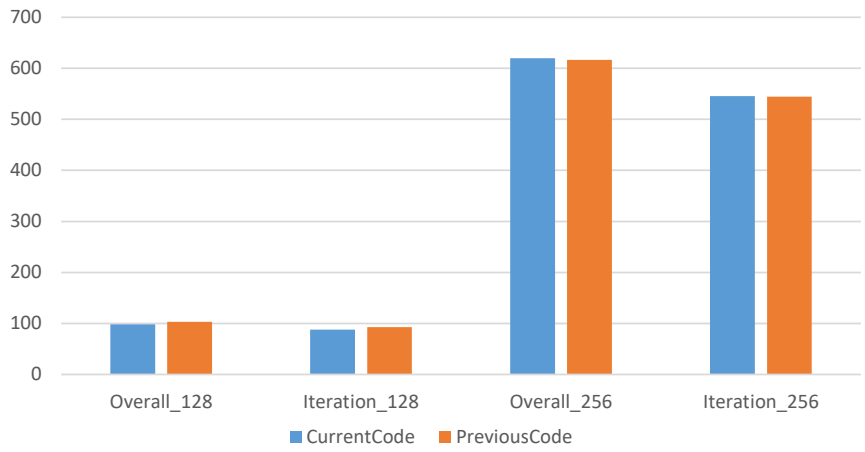


Figure 3.2: Runtime for Cavity_128 and Cavity_256

To gain a better understanding of the code, the solver's run-times were profiled according to the main phases of the method. Figure 3.3 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.

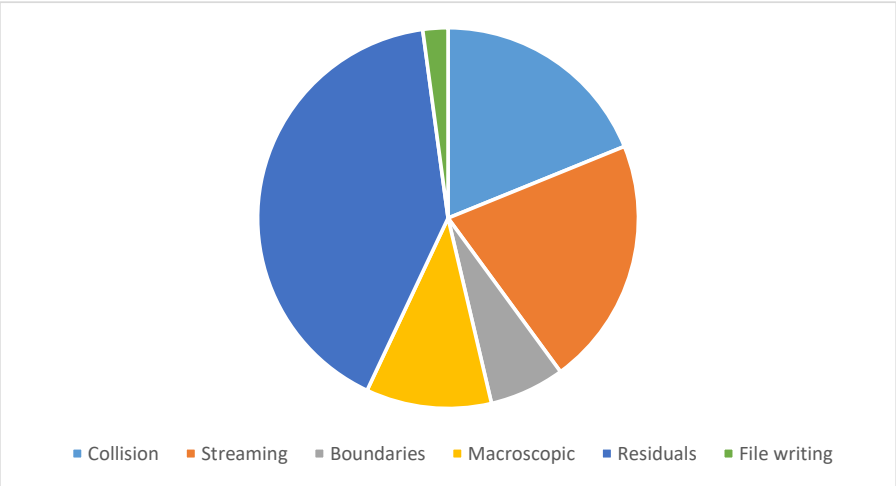
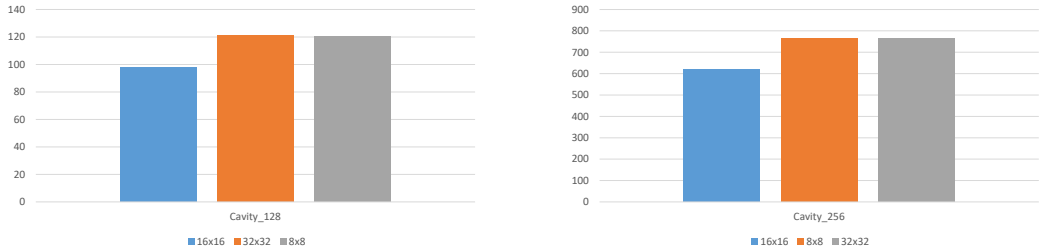


Figure 3.3: Code profiling for the run-times of Cavity_128 with MacroDiff residuals

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 3.4: Overall time comparison using three different threads per block configuration

Figure 3.4 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.

3.1.3 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 4

Results and Discussion

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

[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.