Fabíola Martins Campos de Oliveira

GPU implementation of a fluid dynamics interactive simulator based on the Lattice Boltzmann method

Implementação em GPU de um simulador interativo de fluidodinâmica com o método das Redes de Boltzmann

31/2015

CAMPINAS
2015

UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA MECÂNICA

Fabíola Martins Campos de Oliveira

GPU implementation of a fluid dynamics interactive simulator based on the Lattice Boltzmann method

Implementação em GPU de um simulador interativo de fluidodinâmica com o método das Redes de Boltzmann

Master Thesis presented to the School of Mechanical Engineering of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Mechanical Engineering, in the Area of Solid Mechanics and Mechanical Design.

Dissertação de Mestrado apresentada à Faculdade de Engenharia Mecânica da Universidade Estadual de Campinas como parte dos requisitos exigidos para obtenção do título de Mestra em Engenharia Mecânica, na Área de Mecânica dos Sólidos e Projeto Mecânico.

Orientador: Prof. Dr. Luiz Otávio Saraiva Ferreira

ESTE EXEMPLAR CORRESPONDE À VERSÃO FINAL DA DISSERTAÇÃO DEFENDIDA PELA ALUNA FABÍOLA MARTINS CAMPOS DE OLIVEIRA, E ORIENTADA PELO PROF. DR. LUIZ OTÁVIO SARAIVA FERREIRA.

.....
ASSINATURA DO ORIENTADOR

CAMPINAS
2015

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca da Área de Engenharia e Arquitetura
Luciana Pietrosanto Milla - CRB 8/8129

OL4g

Oliveira, Fabíola Martins Campos de, 1988-
GPU implementation of a fluid dynamics interactive simulator based on the Lattice Boltzmann method / Fabíola Martins Campos de Oliveira. – Campinas, SP : [s.n.], 2015.

Orientador: Luiz Otávio Saraiva Ferreira.
Dissertação (mestrado) – Universidade Estadual de Campinas, Faculdade de Engenharia Mecânica.

1. Fluidodinâmica computacional (CFD). 2. Modelagem e simulação. 3. Programação paralela (Computação). 4. Visualização. 5. Computação gráfica. I. Ferreira, Luiz Otávio Saraiva,1956-. II. Universidade Estadual de Campinas. Faculdade de Engenharia Mecânica. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Implementação em GPU de um simulador interativo de fluidodinâmica com o método das Redes de Boltzmann
Palavras-chave em inglês:
Computational fluid dynamics (CFD)
Simulation and modelling
Parallel programming (Computer Science)
Visualization
Computer graphics
Área de concentração: Mecânica dos Sólidos e Projeto Mecânico
Titulação: Mestra em Engenharia Mecânica
Banca examinadora:
Luiz Otávio Saraiva Ferreira [Orientador]
Marco Lúcio Bittencourt
Wu Shin-Ting
Data de defesa: 20-02-2015
Programa de Pós-Graduação: Engenharia Mecânica

UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA MECÂNICA

COMISSÃO DE PÓS-GRADUAÇÃO EM ENGENHARIA MECÂNICA

DEPARTAMENTO DE MECÂNICA COMPUTACIONAL

DISSERTAÇÃO DE MESTRADO

GPU implementation of a fluid dynamics interactive simulator based on the Lattice Boltzmann method

Implementação em GPU de um simulador interativo de fluidodinâmica com o método das Redes de Boltzmann

Autora: Fabíola Martins Campos de Oliveira
Orientador: Prof. Dr. Luiz Otávio Saraiva Ferreira

A Banca Examinadora composta pelos membros abaixo aprovou esta Dissertação:

Prof. Dr. Luiz Otávio Saraiva Ferreira, Presidente
DMC/FEM/UNICAMP

Prof. Dr. Marco Lúcio Bittencourt
DSI/FEM/UNICAMP

Profª. Drª. Wu Shin-Ting
DCA/FEEC/UNICAMP

Campinas, 20 de fevereiro de 2015.

Dedicatória

Dedico este trabalho aos meus queridos avós, Celso, Miriam, Laura e Francisco; este último me acompanhou somente até metade do meu mestrado em vida, mas estará sempre em meu coração. Também dedico este trabalho à minha tia Cláudia, que pôde estar presente quase até o fim, e sempre me ajudou, motivou e deu bons conselhos, e à minha família e ao meu namorado.

Agradecimentos

Ao meu orientador, Prof. Dr. Luiz Otávio Saraiva Ferreira, pela dedicação, apoio e confiança na orientação deste trabalho.

À minha família, pela fé e apoio incondicional na realização deste trabalho e pela educação que recebi.

Ao meu namorado, Helói, pela paciência e dedicação na revisão deste trabalho e pelo apoio em todos os momentos.

Aos meus colegas de laboratório e de graduação, pelos momentos de descontração e apoio.

Aos professores, funcionários e institutos que me acolheram ao longo da graduação e do mestrado.

Aos membros da banca de Defesa de Mestrado, Prof. Dr. Marco Lúcio Bittencourt e Profª. Drª. Wu Shin-Ting.

Aos membros da banca do Exame de Qualificação, Prof. Dr. William Roberto Wolf e Profª. Drª. Wu Shin-Ting, pelo acompanhamento e sugestões durante o mestrado.

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), pelo financiamento de minha bolsa de mestrado.

À Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), pelo financiamento do projeto de pesquisa do Prof. Dr. Luiz Otávio Saraiva Ferreira, com o qual foi possível a criação do Laboratório de Supercomputação do Departamento de Mecânica Computacional (processo número $2009/09717-7$).

À Universidade Estadual de Campinas (UNICAMP), pela infraestrutura disponibilizada para minha formação.

E à população brasileira, pelo financiamento das instituições UNICAMP, CAPES e FAPESP.

O maior pecado contra a mente humana é acreditar em coisas sem evidências. A ciência é somente o supra-sumo do bom-senso – isto é, rigidamente precisa em sua observação e inimiga da lógica falaciosa.

Thomas Henry Huxley

Resumo

Recentes avanços na tecnologia de processadores multinúcleos e vários-núcleos popularizaram a computação paralela, acelerando a execução de programas e possibilitando a simulação de domínios maiores. Dentre os problemas complexos que requerem alta velocidade de processamento, os problemas de fluidodinâmica computacional se destacam, já que suas simulações tendem a ter um alto custo computacional e exigem grandes domínios de simulação. O método baseado nas Redes de Boltzmann é um método de fluidodinâmica computacional apropriado para o uso de paralelismo que vem ganhando destaque na comunidade científica. Embora haja trabalhos que explorem o paralelismo em GPU nesse método, um simulador eficiente na execução e visualização interativa ainda não foi explorado adequadamente. Assim, a proposta deste trabalho é implementar em GPU um simulador interativo de fluidodinâmica com o método das Redes de Boltzmann. Inicialmente, o simulador foi desenvolvido em linguagem C e foi paralelizado em CPU usando MPI. Em seguida, foi paralelizado em GPU usando CUDA e convertido para linguagem orientada a objetos em C++. Depois, a visualização interativa foi acrescentada utilizando técnicas como interoperabilidade entre CUDA e OpenGL, texturização 3D, fluxo programável da GPU, além de funções de interação com o usuário. O simulador foi validado para casos 2D e 3D em fluxos monocomponentes monofásicos. Além disso, para demonstrar o ganho de desempenho em velocidade de processamento de problemas paralelizados em relação a execuções sequenciais, um conjunto de testes com tamanhos crescentes de domínio foi desenvolvido. O resultado dessa comparação indicou que o simulador implementado em GPU com visualização interativa teve desempenho 71.3 vezes maior em relação à versão sequencial em CPU sem visualização interativa. Dessa forma, a abordagem de paralelização em GPU com visualização interativa mostrou-se muito adequada à execução de simulações fluidodinâmicas, sendo uma ferramenta útil no estudo de escoamentos fluídicos, capaz de executar inúmeros cálculos e lidar com a grande quantidade de memória exigida por simulações fluídicas.

Palavras-chave: fluidodinâmica computacional (CFD), modelagem e simulação, programação paralela (computação), visualização, computação gráfica.

Abstract

Recent advances on multicore and many-core processor technology have popularized the parallel computing, accelerating program execution and enabling the simulation of larger domains. Within the complex problems that require a high processing speed, the computational fluid dynamics problems stand out, since their simulations tend to have high computational cost and demand large simulation domains. The method based on the Lattice Boltzmann is an appropriate computational fluid dynamics algorithm to explore parallelism that has been noteworthy in scientific community. Although there are several works that approach GPU parallelism in this method, an efficient simulator implementation and interactive visualization have not been explored adequately. Thus, the purpose of this work is to implement in GPU an interactive fluid dynamics simulator based on the Lattice Boltzmann method. First, the simulator was developed in C language and was parallelized in CPU using MPI. Next, it was parallelized in GPU using CUDA and converted into C++ object-oriented language. Then, the interactive visualization was added using techniques such as CUDA-OpenGL interoperability, 3D texturing, GPU programmable pipeline, and interaction features. The simulator was validated for 2D and 3D cases in single component, single phase flows. Besides that, to show the performance gain in processing velocity of parallelized problems in relation to sequential executions, a test set with increasing domain sizes was developed. This comparison result indicated the GPU-implemented interactive simulator was 71.3 times faster in relation to sequential CPU version without interactive visualization. Thereby, the GPU parallelization approach with interactive visualization showed to be very adequate to fluid dynamics simulations, being a useful tool in fluid flow study, capable of simulating numerous calculations and dealing with the large amount of memory required in fluidic simulations.

Keywords: computational fluid dynamics (CFD), simulation and modelling, parallel programming (computer science), visualization, computer graphics.

List of Figures

List of Tables

List of Abbreviations and Acronyms

Latin Letters

a	-	Half slit width	$[lu]$
c	-	Lattice sound speed	$[lu/lt]$
$d\mathbf{x}$	-	Differential of x	
$E(n,p)$	-	Code efficiency	
e	-	Microscopic velocity	$[lu/lt]$
$\mathbf{F}$	-	External force	$[mu\,lu/lt^2]$
$f^{(1)}$	-	Single particle distribution function	
G	-	Interaction strength	
L	-	Characteristic length	$[lu]$
m	-	Mass	$[mu]$
N	-	Number of lattice nodes	
n	-	Number of elements of domain	
P	-	Pressure	$[mu\,lu/lt^2]$
$\mathbf{p}$	-	Lattice momentum	$[lu/lt]$
p	-	Number of processes	
R	-	Ideal gas constant	$[J/mol\,K]$
$rgba$	-	Texture array	
$S(n,p)$	-	Speedup	
s	-	Map of solids	
T	-	Temperature	$[K]$
$T(n,p)$	-	Run-time	$[s]$
t	-	Simulation time	$[lt]$
$\mathbf{u}$	-	Macroscopic velocity	$[lu/lt]$
$\mathbf{x}$	-	Lattice position	$[lu]$
w	-	Weight	

Greek Letters

$\Gamma^{(+)}d\mathbf{x}d\mathbf{p}dt$	-	Number of molecules that arrive in $(\mathbf{x}+d\mathbf{x}, \mathbf{p}+d\mathbf{p})$ but did not start at $(\mathbf{x}, \mathbf{p})$ during time dt	
$\Gamma^{(-)}d\mathbf{x}d\mathbf{p}dt$	-	Number of molecules that do not arrive in $(\mathbf{x}+d\mathbf{x}, \mathbf{p}+d\mathbf{p})$ but started at $(\mathbf{x}, \mathbf{p})$ during time dt	
$\Delta\mathbf{u}$	-	Increment in macroscopic velocity	$[lu/lt]$
Δt	-	Simulation time step	$[lt]$
Δx	-	Lattice spacing	$[m/lu]$
μ	-	Dynamic viscosity	
ν	-	Kinematic viscosity	$[lu^2/lt]$
ρ	-	Macroscopic density	$[mu/lu^2]$ or $[mu/lu^3]$
τ	-	System relaxation parameter	
ψ	-	Interaction potential	$[mu/lu^2]$ or $[mu/lu^3]$

Superscripts

$*$	-	Linear gradient
eq	-	Equilibrium

Subscripts

a	-	Direction a
ads	-	Adsorption
$parallel$	-	Parallel implementation
$serial$	-	Serial implementation
lb	-	lattice
$phys$	-	physical
max	-	maximum

Acronyms

API	-	Application Programming Interface
BGK	-	Bhatnagar, Gross and Krook

CAD - Computer Aided Design
CVMLCPP - Common Versatile Multi-purpose Library for C++
CPU - Central Processing Unit
CUDA - Compute Unified Device Architecture
DdQq - Model with d dimensions and q directions
DRAM - Dynamic Random Access Memory
EOS - Equation of State
FAPESP - Fundação de Amparo à Pesquisa do Estado de São Paulo
FLOP - Floating-point operations per second
fps - Frames per second
GPGPU - General-purpose Graphics Processing Unit
GPU - Graphics Processing Unit
GDDR - Graphics Double Data Rate
GLEW - OpenGL Extension Wrangler Library
GLSL - OpenGL Shading Language
IEEE - Institute of Electrical and Electronics Engineers
LOC - Laboratory on a Chip
LBM - Lattice Boltzmann method
lt - Lattice time
LTS - Long-term support
lu - Lattice unit
MLUPS - Millions of Lattice Updates per Second
MPI - Message-passing Interface
μTAS - Micro Total Analysis System
mu - Mass unit
OpenGL - Open Graphics Library
PBO - Pixel Buffer Object
PCIe - Peripheral Component Interconnect express
RAM - Random Access Memory
Re - Reynolds number
SCMP - Single component, multiphase
SCSP - Single component, single phase
SM - Streaming multiprocessor
SP - Streaming processor
STL - Stereolithography

VTK - Visualization Toolkit

Other Notations

1D - One-dimensional
2D - Two-dimensional
3D - Three-dimensional
$\nabla_x f^{(1)}$ - Gradient of $f^{(1)}$ along x
$\partial f^{(1)}/\partial t$ - Partial derivative of $f^{(1)}$ with respect to t

SUMMARY

1 Introduction

The chemical laboratory on a chip (Lab-On-a-Chip, or LOC) is a microfluidic device that has been increasingly used in chemical and biochemical analyses (van den Berg and Lammerink, 1998; Ehrfeld, 2003). LOCs tend to replace current analysis equipments, because of its high economic impact. The LOCs' main advantage is the higher performance chemical analyses, but other fundamental advantages over traditional analyses are their size reduction, the volume reduction of needed reagents and samples, the manufacturing low cost and the shorter time needed for analyses (Giannitsis, 2011; Cheng *et al.*, 1998). Also called Micro Total Analysis Systems (μTAS), these devices perform several different analytical steps, like sample introduction, sample pretreatment, chemical reactions, analytical separation, and detection (Qin *et al.*, 1998; Manz *et al.*, 1990). The μTASes convert chemical information in electrical or optical signals, enabling a higher level of automation in systems (Mairhofer *et al.*, 2009). The need for LOCs has arisen in the pharmaceutical industry as a requirement for faster development of new drugs at a low cost (Jackson, 2006). The simultaneous synthesis of thousands of different chemical compounds, known as combichem (combinatorial chemistry analysis), combined with the ability to perform data acquisition and analysis all in an only device, has boosted the use and development of LOCs (Giannitsis, 2011).

LOCs require an accurate and uniform transport of a small amount of fluid (Elwenspoek *et al.*, 1994; Woias, 2004). In order to do it, some phenomena that may be neglected on analyses of macroscopic systems have to be included on microfluidic systems. These phenomena, like interfacial slip, wetting, surface tension and Brownian motion, are difficult to be incorporated in traditional Computational Fluid Dynamics methods, but are properly treated in simulation programs based on the Lattice Boltzmann method (LBM) (Zhang, 2011). Furthermore, porous media flows, like petroleum extraction, and surface flows, are also simulated with this method (Valderhaug, 2011).

The LBM was introduced by McNamara and Zanetti (1988) and originated from the combination of the Lattice Gas Cellular Automata for fluid flow simulation with the Boltzmann kinetic theory of gases (Wolf-Gladrow, 2005). Aidun and Clausen (2010) considered it an efficient and reliable method to simulate complex fluid flows. Besides that, the LBM has some advantages over traditional methods: it is highly data-parallelizable (Obrecht *et al.*, 2011b), which may lead to high performance applications on the Graphics Processing Unit (GPU), and it easily deals with some characteristics that common methods can not deal with or are very slow, like complex boundaries and multicomponent, multiphase flows (Succi, 2001). Another advantage of the method is that it

may be considered as a general-purpose solver method to be applied in other fields, like heat transfer, electrical and magnetic fields, and diffusion processes (Mohamad, 2011; Zhang, 2011). Thus, a general-purpose simulator can be implemented to run simulations of all these fields using LBM.

In 2007, NVIDIA® presented its new generation of general-purpose GPUs based on the Compute Unified Device Architecture (CUDA), a proprietary technology for massively parallel processing on GPU (Kirk and Hwu, 2010; NVIDIA, 2014b,a,c). Since then, several works about the LBM on GPUs have been published (Volkov and Krafczyk, 2008; Tölke and Krafczyk, 2008; Micikevicius, 2009; Ma *et al.*, 2009; Oliveira and Ferreira, 2012, 2013; Oliveira *et al.*, 2013). Xian and Takayuki (2011) demonstrated that LBM is appropriate for execution on GPUs, since the LBM is an efficiently data-parallelizable problem. Therefore, when there is data parallelism, parallel programming on GPUs is advantageous over parallel programming on Central Processing Units (CPUs), due to the greater amount of processing units of GPUs (Sanders and Kandrot, 2011).

In this work, the objective was to implement in GPU an interactive fluid dynamics simulator using the LBM, meeting some requirements of the Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) project, called *Fluid simulator implementation in microsystems based on particle methods for massively parallel processing in GPU cluster* (grant number $2009/09717-7$), in which this master thesis is included. The implementation has evolved from versions running on a single CPU, passing through on parallel CPU and then on GPU, until versions with and without interactive visualization. The source code of the last version, parallel GPU code with interactive visualization, is available at Oliveira (2015). Classical literature cases were run, validating the simulator for two-dimensional (2D) and three-dimensional (3D) cases, for single component, single phase (SCSP) and for some single component, multiphase (SCMP) cases. The current version was compared to serial and parallel CPU code, showing that the GPU implementation with interactive visualization is the most useful, because it combines the faster execution on GPU with interactive features and online visualization, which allow an early intervention on simulation by changing inappropriate simulation conditions interactively, and avoid computing time waste.

1.1 Literature Review

The Lattice Boltzmann equation has strong difficulties in simulating fluid flows with high Reynolds number (Re). Li and Kwok (2004) proposed a new model with high Re and external

forces to simulate electrokinetic phenomena in microfluidics, by considering pressure an external force, which may represent the Lorentz force associated with external electric and magnetic fields and other related forces. They obtained excellent agreement with experimental data in pressure-driven microchannel flow and showed that this model is an effective computational tool to simulate complex microfluidic systems that can not be described by electrokinetic theory. Kang *et al.* (2008) investigated the electrokinetic effect on the fluid flow and mixing in a rectangular microchannel. Yong *et al.* (2011) simulated multicomponent, multiphase flows in microfluidic devices successfully, and Zhang (2011) made a review of LBM applications in microfluidic systems for single phase and multiphase flows, also including heat, electric and magnetic fields, and diffusion processes. Moreover, Agarwal and Chusak (2010) simulated non-newtonian fluids with the LBM.

Several innovations were performed considering LBM on GPU. Palmer and Nieplocha (2002) described several methods for ghost cell update of large, distributed arrays. The Global Array toolkit was used to implement one version of the update algorithm and another three versions of the shift algorithm that were applied to a LBM solver. They showed that no algorithm presented optimal performance for all architectures and that the most efficient update depends on the system architecture and on the problem size and dimensionality. Fan (2008) presented efficient ways to use GPUs and GPU clusters for general-purpose GPU (GPGPU) in flow simulation and visualization. He implemented a novel GPU-based adapted unstructured LBM algorithm for simulating flow on arbitrary 3D triangular surfaces, and a LBM implementation of irregular-shaped simulation domain on a GPU cluster. These contributions were applied for thermal fluid dynamics in a pressurized water reactor of a nuclear power plant, and, finally, Fan (2008) developed Zippy, a general framework for GPU cluster programming.

Chopard (2008) improved the accuracy of Lattice Boltzmann calculations, which were used by Aksnes (2009) to simulate porous rocks. Tölke and Krafczyk (2008) implemented an efficient 2D LBM solver, introducing new schemes to avoid memory misalignments. Stürmer *et al.* (2009) presented a fast and optimised LBM solver that saves memory on domains that have a large percentage of solid nodes. Bailey *et al.* (2009) improved GPU LBM results for the D3Q19 model by increasing GPU multiprocessor occupancy, resulting in a maximum performance increase of 20%, and by introducing a space-efficient storage method that reduces GPU RAM requirements by 50% at a slight detriment to performance, called swap algorithm. Schreiber (2010) implemented a LBM free surface fluid simulation and compared different memory layouts for this implementation, showing its realism in computer games with fixed fluid simulation domains, the importance of advanced visualization methods for scientific simulations and to accelerate photo realistic ren-

derings. Aksnes and Elster (2009) investigated the complex geometry of porous rocks to efficiently simulate on GPU and techniques for reducing round-off errors. They also used the swap algorithm in LBM implementations. Obrecht *et al.* (2011a) studied the global memory access mechanism of GPUs to optimise regular data-parallel applications, and analysed the impact of memory misalignments on GPUs for the LBM. Xian and Takayuki (2011) simulated 3D LBM on a multi-node GPU cluster with CUDA and Message-passing Interface (MPI). They analysed performance of different domain partitioning in order to optimise communication, and introduced an overlapping technique between computation and communication, which improved the simulator performance. Habich *et al.* (2011) used the optimisations proposed by Tölke and Krafczyk (2008) to implement a LBM solver focusing on memory alignment and register shortage, and also analysed data transfer rates for the Peripheral Component Interconnect express (PCIe) to explore multi-GPU parallelism. Obrecht *et al.* (2011b) introduced new data transfer schemes in global memory to avoid misaligned memory accesses, leading to 86% of the global memory maximum throughput for the GT200 graphics card and efficient implementation even for complex LBM models. Obrecht *et al.* (2011c) also implemented a thermal flow LBM solver on GPUs, proposing an efficient thermal handling and achieving good agreement with literature and better performance, when compared to CPU solvers. Valderhaug (2011) increased the maximum data set possible to simulate on GPUs by using techniques to decrease memory requirements while maintaining numerical precision, and overlapped communication and computations on a GPU cluster.

1.2 Contributions

There are promising results for the LBM, although it would be interesting to offer the user an appropriate system with online visualization and interaction features, from which one may form hypotheses about the simulation parameters, refine them and reevaluate their effects without restarting the simulation. Such an implementation would benefit LOCs and μTAS design costs, and also other fluidic analyses. However, to our best knowledge, there is not such a system that integrates both LBM simulations, online visualization, and satisfactory interaction.

This work resulted in some versions of the LBM simulator: C, C++, MPI, and CUDA without and with interactive visualization, as can be seen in Figure 1.1. The evolution of these versions aimed performance gain, combined with the use of the swap algorithm optimisation, the bitmap and the STL files import for data input to ease the simulation process, and the VTK files output

to ease analyses. Furthermore, the algorithm was validated for SCSP flows in the literature. Those were the contributions in the LBM field.

Besides those contributions, some work in LBM visualization was done considering the following visualization problems: (1) how to share GPU resources between simulation and visualization algorithms; (2) how to share simulation data between CUDA C GPGPU language and the Open Graphics Library (OpenGL) graphics API; (3) how to make visualized objects capable of promptly reacting to user actions; and (4) how to change visualization to see different data and different color maps. The first two problems were solved by exploiting the CUDA-OpenGL interoperability resource. The third problem was solved by using the 3D texturing visualization method and by using the Qt library (Qt, 2014) for the graphical user interface and the interaction tools. Finally, the last problem was solved by using the OpenGL Shading Language (GLSL) to change visualized data or color maps. All the contributions including a general view of the work are presented in Figure 1.2. Furthermore, the simulation speed with online visualization happening at interactive rate can be considered a contribution, as these functionalities were added without great loss in performance. The source code of the interactive GPU simulator is available at Oliveira (2015).

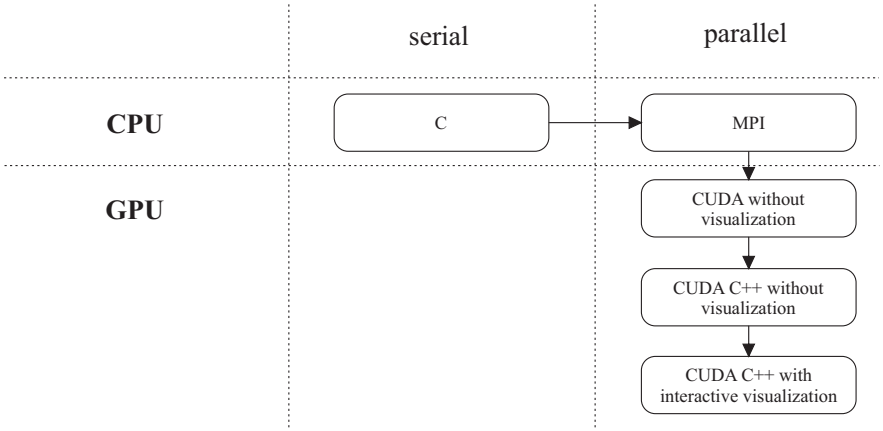


Figure 1.1: Simulator versions.

1.3 Work organisation

Each chapter of this work is briefly described next. Figure 1.3 presents the relation between the chapters.

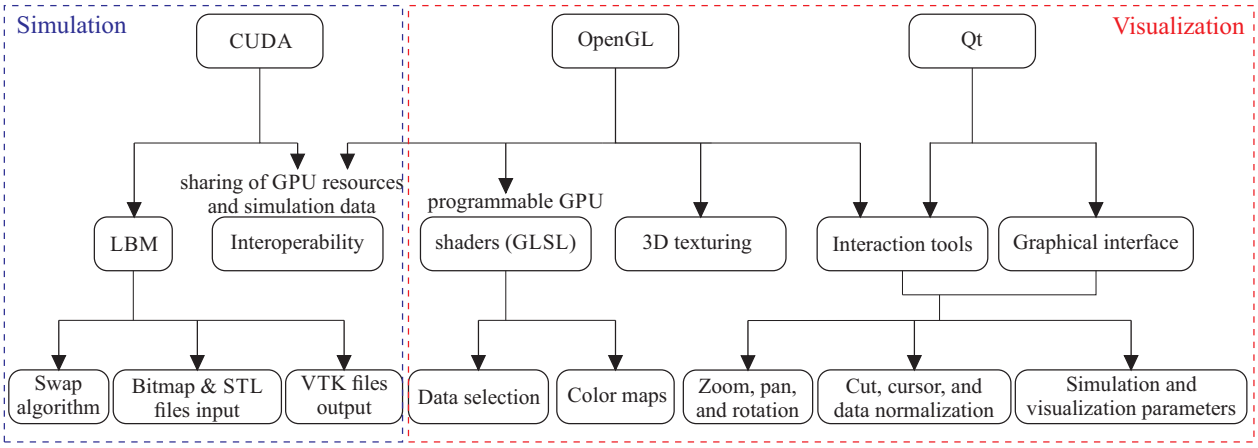


Figure 1.2: Work contributions and general view.

In chapter 2, the theory concerning the LBM is presented. A simplified form of the Boltzmann equation is used to derive the Lattice Boltzmann equation. 2D and 3D models are discussed, and the basic LBM is presented with the most common boundary conditions and application of external forces. The SCMP theory is derived, approaching interparticles forces, equation of state and solid walls. Finally, the chapter ends with important fluid concepts in the LBM context.

Chapter 3 presents the computational tools used in this work, with CUDA being the most important language. Furthermore, a brief history of CPU parallelism and GPU evolution are given, as well as MPI notions that were important to understand the simulator versions. Next, the GPU visualization is discussed, including the OpenGL-based 3D texturing technique used in this work, and a key concept to achieve performance, the CUDA-OpenGL interoperability. To finish the chapter, performance measurements are shown.

Chapter 4 gives details about the simulator implementation. This work met some FAPESP project requirements and they were discussed in this chapter. The used optimisation techniques are presented with figures to clarify the implementation. The visualization and graphical interface are presented, exploring the use of shaders and Qt interaction features. Finally, the previous versions of the simulator are discussed.

In chapter 5, the results are presented. First, SCSP flow validation tests, such as: Poiseuille flow driven by gravity, Poiseuille flow driven by velocity/pressure boundaries (the entry length effect), and flows past a cylinder under different Re regimes, are shown. These cases were compared to literature experiments or simulations and had good agreement. Then, the SCMP cases that were

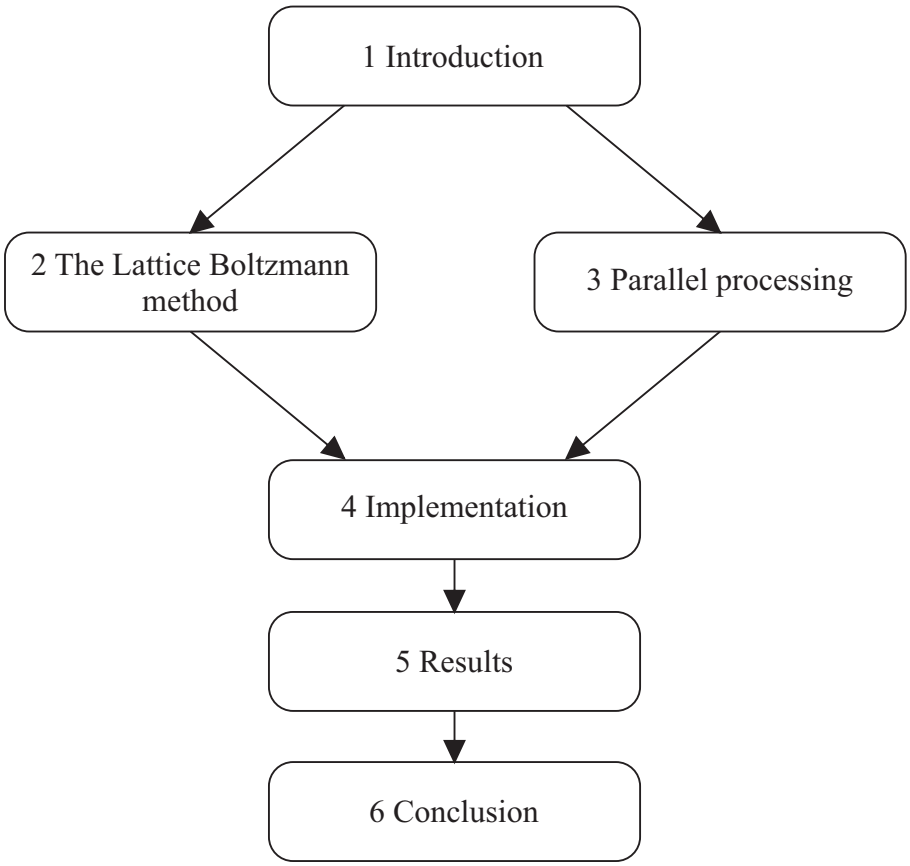


Figure 1.3: A global view of this work chapters and their relations.

simulated successfully are presented, and they were: phase separation and interface minimization, superficial tension estimation, flat interfaces, heterogeneous cavitation, and contact angles. After that, a performance comparison was done, considering a 2D SCSP flow simulation for increasing domain sizes. The GPU with interactive visualization code was compared to CPU serial and parallel codes, indicating that the GPU interactive simulator was superior in relation to the other versions, both by the simulation speed and by the use of interactive tools in simulation analyses.

Chapter 6 concludes this dissertation and presents some ideas for future work, and the annex shows the two papers published during this work.

2 The Lattice Boltzmann method

The LBM follows the idea that a gas is composed of interacting particles that can be described by classical mechanics, but should be statistically treated (Sukop and Thorne Jr., 2005). The mechanics comprehends streaming in space and perfectly elastic collisions that, despite of being simple, reproduce the behaviour of real fluids.

In this chapter, the LBM equation and models are presented. Then, the boundary conditions and the external forces are applied to the model. After that, the SCMP flow theory and the calculation of viscosity are described, the Reynolds number is reviewed, and the unit conversion from real experiments into LBM simulations is explained. Finally, the Poiseuille flow and the Laplace law are presented.

2.1 A simplified form of the Boltzmann equation

Statistical mechanics can describe a system using probability distribution functions. The distribution $f^{(1)}(\mathbf{x}, \mathbf{p}, t)$ provides the probability of finding a molecule at position $\mathbf{x}$, with momentum $\mathbf{p}$, and at time t. This is the single particle distribution function, which is appropriate to describe gas properties that do not depend on relative positions of molecules (Sukop and Thorne Jr., 2005).

The streaming process can be described with the term $f^{(1)}(\mathbf{x}, \mathbf{p}, t)d\mathbf{x}d\mathbf{p}$, which gives the probability of finding molecules with position coordinates in the range $\mathbf{x} \pm d\mathbf{x}$ and momentum coordinates $\mathbf{p} \pm d\mathbf{p}$. Considering an external force $\mathbf{F}$ small in relation to the intermolecular forces, if there are no collisions, at time $t + dt$, the new positions of molecules of mass m starting at $\mathbf{x}$ are $\mathbf{x}+(\mathbf{p}/m)dt = \mathbf{x}+(d\mathbf{x}/dt)dt = \mathbf{x}+d\mathbf{x}$ and the new momenta are $\mathbf{p} = \mathbf{p}+\mathbf{F}dt = \mathbf{p}+(d\mathbf{p}/dt)dt = \mathbf{p} + d\mathbf{p}$. Then, the streaming equation, which determines $f^{(1)}$ at time $t + dt$, is:

$$f^{(1)}(\mathbf{x} + d\mathbf{x}, \mathbf{p} + d\mathbf{p}, t + dt)d\mathbf{x}d\mathbf{p} = f^{(1)}(\mathbf{x}, \mathbf{p}, t)d\mathbf{x}d\mathbf{p}. \tag{2.1}$$

However, when there are collisions between molecules, some of them go to unexpected places and do not follow the streaming process. To determine the collision equation, $\Gamma^{(+)}d\mathbf{x}d\mathbf{p}dt$ is defined as the number of molecules that arrive in $(\mathbf{x} + d\mathbf{x}, \mathbf{p} + d\mathbf{p})$, but did not start at $(\mathbf{x}, \mathbf{p})$ during time

dt, and $\Gamma^{(-)}d\mathbf{x}d\mathbf{p}dt$ as the number of molecules that do not arrive in $(\mathbf{x}+d\mathbf{x}, \mathbf{p}+d\mathbf{p})$, but started at $(\mathbf{x}, \mathbf{p})$. Adding to the streaming equation:

$$f^{(1)}(\mathbf{x}+d\mathbf{x}, \mathbf{p}+d\mathbf{p}, t+dt)d\mathbf{x}d\mathbf{p} = f^{(1)}(\mathbf{x}, \mathbf{p}, t)d\mathbf{x}d\mathbf{p} + [\Gamma^{(+)} - \Gamma^{(-)}]d\mathbf{x}d\mathbf{p}dt. \tag{2.2}$$

Then, the Taylor series expansion of the left-hand side of Equation 2.2 with the first-order terms:

$$f^{(1)}(\mathbf{x}+d\mathbf{x}, \mathbf{p}+d\mathbf{p}, t+dt) = f^{(1)}(\mathbf{x}, \mathbf{p}, t) + d\mathbf{x}.\nabla_x f^{(1)} + d\mathbf{p}.\nabla_p f^{(1)} + \left(\frac{\partial f^{(1)}}{\partial t}\right) dt + ..., \tag{2.3}$$

gives the Boltzmann equation:

$$\begin{aligned}\left[f^{(1)}(\mathbf{x}, \mathbf{p}, t) + d\mathbf{x}.\nabla_x f^{(1)} + d\mathbf{p}.\nabla_p f^{(1)} + \left(\frac{\partial f^{(1)}}{\partial t}\right) dt + ...\right] d\mathbf{x}d\mathbf{p} = \\ f^{(1)}(\mathbf{x}, \mathbf{p}, t)d\mathbf{x}d\mathbf{p} + [\Gamma^{(+)} - \Gamma^{(-)}]d\mathbf{x}d\mathbf{p}dt\end{aligned} \tag{2.4}$$

or

$$\mathbf{v}.\nabla_x f^{(1)} + \mathbf{F}.\nabla_p f^{(1)} + \left(\frac{\partial f^{(1)}}{\partial t}\right) = \Gamma^{(+)} - \Gamma^{(-)}. \tag{2.5}$$

The Boltzmann equation can be derived for more than one chemical component. With a collision operator written more explicitly, the Boltzmann equation is a nonlinear integral differential equation, which took 50 years to have an approximate solution (Harris, 1971). Nevertheless, with Lattice Boltzmann methods, the equation can have an approximate solution from the particle perspective using explicit collision and streaming terms.

2.2 The Lattice Boltzmann models

The discretization of the Boltzmann equation in space reduces the number of possible positions and microscopic momenta that a particle can assume. The length unit of LBM is the lattice unit (lu), the time unit is the lattice time (lt) and the mass unit is called just mass unit (mu).

For the 2D case, considering a square lattice, the continuum space and momenta are reduced to eight directions, three velocity magnitudes and a single particle mass (Qian *et al.*, 1992). This particle mass is uniform and equals to $1\ mu$, which makes the microscopic velocities and momenta equivalent. This is the most common 2D model, called D2Q9 in literature, which consists of nine

velocities as shown in Figure 2.1, with e_0 representing particles at rest. The microscopic velocity magnitude is $1\ lu/lt$ for e_1 to e_4, and $\sqrt{2}\ lu/lt$ for e_5 to e_8, which makes all x- and y-components equal to 0 or ± 1, and one only needs to care about the microscopic velocity direction.

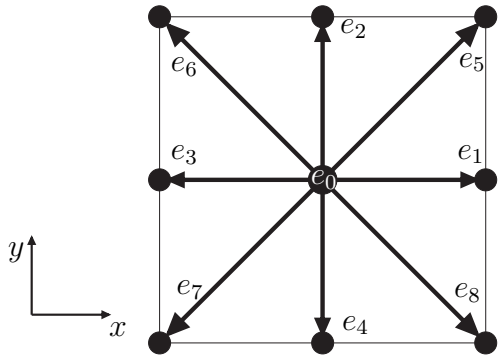


Figure 2.1: D2Q9 Lattice Boltzmann model.

For the 3D case and a cubic lattice, the most popular models are D3Q15, D3Q19, and D3Q27 (Aidun and Clausen, 2010), with the D3Q27 being equivalent to the D2Q9 in terms of accuracy. While the D3Q15 is less precise, the D3Q27 is very computationally expensive, which makes D3Q19 the model that better balances accuracy and efficiency (Mei *et al.*, 2000). The D3Q19 model consists of nineteen directions for the microscopic velocities and momenta (Mohamad, 2011), as shown in Figure 2.2, and the particle mass is also uniform and equals to $1\ mu$. Analogously to the D2Q9 model, the microscopic velocity magnitude is $1\ lu/lt$ for e_1 to e_6, and $\sqrt{2}\ lu/lt$ for e_7 to e_{18}, which makes all x-, y- and z-components equal to 0 or ± 1.

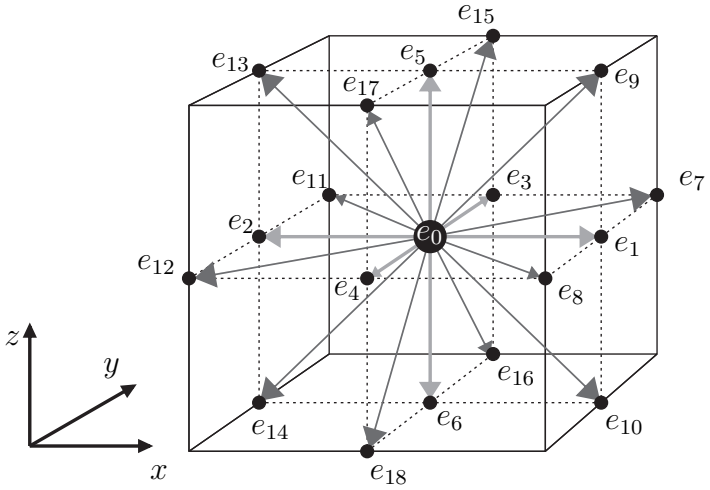


Figure 2.2: D3Q19 Lattice Boltzmann model.

2.3 The basic Lattice Boltzmann equation

The LBM is a discretization on time, space and momentum of the Boltzmann kinetic theory (Aidun and Clausen, 2010; Zhang, 2011). It recovers the Navier-Stokes equation in the continuum limit, through the Chapman-Enskog expansion (Succi, 2001; Wolf-Gladrow, 2005). Simulations with this method may be divided into two steps: streaming and collision, which are performed iteratively (Wolf-Gladrow, 2005). The boundary conditions may be included in those steps or may be performed in separate functions (Sukop and Thorne Jr., 2005). The streaming step means the propagation of the fluid along the lattice, according to its velocity, while the collision step represents the collisions between the fluid particles and between fluid particles and solid walls (Chen and Doolen, 1998). The basic equation of the method is:

$$f_a(\mathbf{x} + \mathbf{e_a}\Delta t, t + \Delta t) = f_a(\mathbf{x},t) - \frac{[f_a(\mathbf{x},t) - f_a^{eq}(\mathbf{x},t)]}{\tau}, \tag{2.6}$$

in which $\mathbf{x}$ is a lattice position, $\mathbf{e_a}$ is its microscopic velocity on direction a, t is the simulation time, Δt is the simulation time step, f_a is the single particle distribution function on direction a, f_a^{eq} is the equilibrium distribution function on direction a, and τ is the system relaxation parameter (Sukop and Thorne Jr., 2005). The first two terms $f_a(\mathbf{x} + \mathbf{e}_a\Delta t, t + \Delta t) = f_a(\mathbf{x},t)$ correspond to the streaming step, and $\frac{[f_a(\mathbf{x},t) - f_a^{eq}(\mathbf{x},t)]}{\tau}$ is the collision step. This model for the collision step is a simplification developed by Bhatnagar *et al.* (1954) and is known as BGK (Bhatnagar, Gross and Krook) approximation.

To calculate the collision term, one must first calculate the equilibrium distribution function:

$$f_a^{eq}(\mathbf{x}) = w_a\rho(\mathbf{x})\left[1 + 3\frac{\mathbf{e_a}.\mathbf{u}}{c^2} + \frac{9}{2}\frac{(\mathbf{e_a}.\mathbf{u})^2}{c^4} - \frac{3}{2}\frac{\mathbf{u}^2}{c^2}\right], \tag{2.7}$$

in which w_a is the weight for each particle. For instance, for the D2Q9 model, it is $4/9$ for the rest particles ($a = 0$), $1/9$ for $a = 1, 2, 3, 4$, and $1/36$ for $a = 5, 6, 7, 8$, and for the D3Q19 model, w_a is $4/9$ for $a = 0$, $1/9$ for $a = 1, 2, ..., 6$ and $1/36$ for $a = 7, 8, ..., 18$. Finally, c is the lattice sound speed, usually $1\ lu/lt$ (Sukop and Thorne Jr., 2005). It is also necessary to calculate the macroscopic velocity $\mathbf{u}$ and the macroscopic density ρ for each lattice node:

$$\rho = \sum_{a=0}^{8} f_a \tag{2.8}$$

$$\mathbf{u} = \frac{1}{\rho} \sum_{a=0}^{8} f_a \mathbf{e_a}. \tag{2.9}$$

These equations are also the method output values, which enable the conversion between LBM microscopic velocities and continuum macroscopic velocities.

2.4 Boundary conditions

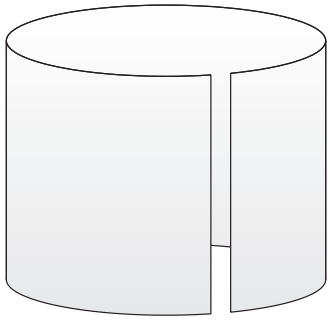
In this section, the periodic, bounceback, constant velocity, and constant pressure boundary conditions are shown. The periodic boundaries and the constant velocity and pressure conditions are usually applied on the domain edges, while the bounceback is applied to solid nodes. The simplicity of the boundary conditions is one of the most advantageous characteristics of the LBM, because it allows the simulation of complex geometries.

2.4.1 Periodic boundaries

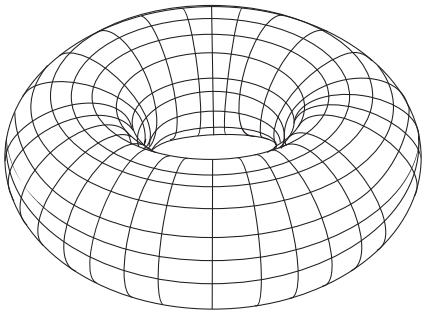
The periodic boundaries are the simplest conditions, in which the system is closed by their edges (Sukop and Thorne Jr., 2005). The open ends of a slit can be connected to form an infinite slit (the domain can be represented by a paper sheet that has its laterals connected, as in Figure 2.3(a)), or it is possible to have fully periodic boundaries, which form an infinite domain, as in the toroid of Figure 2.3(b). These figures represent 2D domains, but the same concept can be applied to a 3D domain, having one, two, or three periodic boundaries. This condition may be implemented in the streaming step by feeding the inlet nodes (only the distribution functions that point inside the domain) with the outlet nodes (distribution functions that point outside the domain), and vice versa.

2.4.2 Bounceback boundaries

The bounceback boundary condition treats the solid nodes. Its advantage is the need to only identify the solid nodes, which enables the simulation of complex geometries (Sukop and Thorne Jr., 2005). The solid nodes may be divided into two types: solid nodes that make an interface with



(a) 2D periodic boundaries applied on the left and right edges of the domain.



(b) 2D periodic boundaries applied on all the four edges of the domain (left, right, top and bottom.

Figure 2.3: Periodic boundaries.

fluid nodes, and solid nodes that are surrounded by other solid nodes. Thus, solid nodes surrounded by other solid nodes are not considered during the LBM iterations. This may save lots of unnecessary computations, specially if there are a few fluid nodes in the domain.

The bounceback imposes a no-slip condition, or zero velocity condition, for the solid nodes, and mass conservation (Chen *et al.*, 1996). When the distribution functions are propagated towards a solid node, they go back to the fluid node that they were before the streaming step, but with inverted directions. This condition works fine for $\tau \approx 1$, which makes the non-equilibrium terms in Equation 2.6 be cancelled. The boundary called half-way wall bounceback leads to a second-order error (Chen *et al.*, 1996). In this boundary condition, fluid nodes adjacent to solid nodes have all their opposite distribution functions inverted, such as f_1 with f_3, f_2 with f_4, f_5 with f_7, and f_6 with f_8 for the 2D case (directions are shown in Fig. 2.4). The 3D case is analogous.

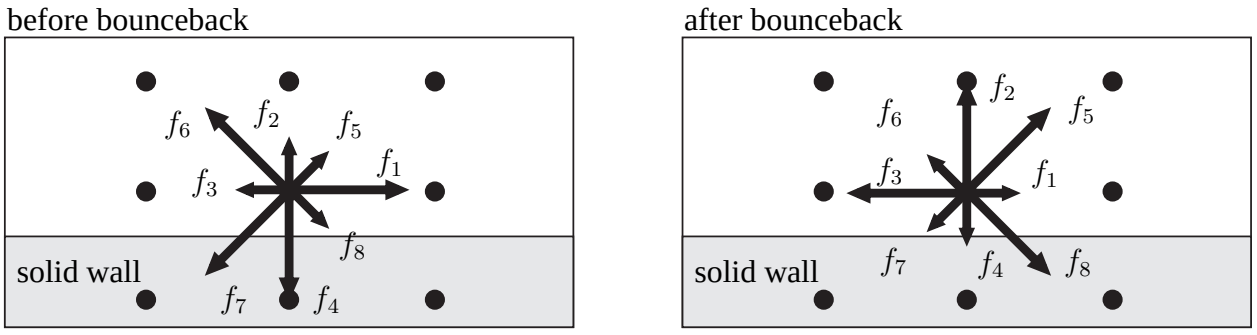


Figure 2.4: Half-way wall bounceback scheme.

2.4.3 Constant velocity boundaries

The constant velocity boundary imposes a constant value for velocity in every axis of a domain edge. From this value, the macroscopic density can be calculated, and considering that, after the streaming step of a 2D simulation, three microscopic densities (or distribution functions) are unknown (Figure 2.5), these values need to be calculated to complete the boundary specification (Sukop and Thorne Jr., 2005).

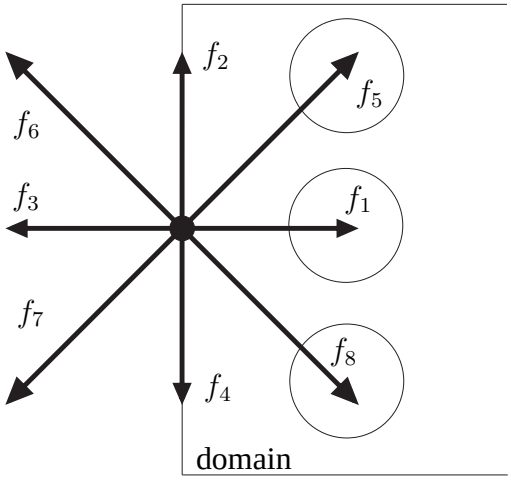


Figure 2.5: The three unknown microscopic densities (circled) of the left edge after the streaming step.

The derivation follows Zou and He (1997) for the left edge of a 2D domain. If the desired constant velocity is $\mathbf{u_0} = \begin{bmatrix} u_0 \\ 0 \end{bmatrix}$, four equations are needed to solve for ρ, f_1, f_5, and f_8. With Equation 2.9, it is possible to write two equations for each axis:

$$\rho u_0 = f_1 - f_3 + f_5 - f_6 - f_7 + f_8, \tag{2.10}$$

and

$$0 = f_2 - f_4 + f_5 + f_6 - f_7 - f_8. \tag{2.11}$$

The other two equations are the macroscopic density (Equation 2.8), and the bounceback rule for the non-equilibrium part of the particle distribution normal to the edge (Zou and He, 1997):

$$f_1 - f_1^{eq} = f_3 - f_3^{eq}. \tag{2.12}$$

Equations 2.8 and 2.10 can be equaled by isolating the microscopic densities f_1, f_5, and f_8, and then the result is solved to find ρ:

$$\rho = \frac{f_0 + f_2 + f_4 + 2(f_3 + f_6 + f_7)}{1 - u_0}. \tag{2.13}$$

One can solve for f_1 using Equation 2.12:

$$f_1 = f_3 + \frac{2}{3}\rho u_0. \tag{2.14}$$

Now, Equation 2.11 is used to replace f_5, and Equation 2.14 is used to replace f_1 in Equation 2.10 to find f_8:

$$f_8 = f_6 + \frac{1}{6}\rho u_0 + \frac{1}{2}(f_2 - f_4). \tag{2.15}$$

The same can be done to find f_5, replacing f_8 from Equation 2.11:

$$f_5 = f_7 + \frac{1}{6}\rho u_0 + \frac{1}{2}(f_4 - f_2). \tag{2.16}$$

2.4.4 Constant pressure boundaries

The constant pressure boundary imposes a constant value for pressure in every axis of a domain edge. Pressure is equivalent to density through the equation of state (EOS) that relates them directly:

$$P = c\rho = \frac{\rho}{3}, \tag{2.17}$$

in which a density ρ_0 is specified and then the velocity is calculated using it. The maximum velocity should be small relative to $1\ lu/lt$.

The solution is analogous to the constant velocity boundaries, but here the condition is derived for the right edge of a 2D domain. In this case, the macroscopic velocity and three microscopic densities are unknown after the streaming step, according to Figure 2.6.

If ρ_0 is the desired density at the edge, four equations are needed to find u, f_3, f_6, and f_7. With Equation 2.9, two equations are written for each axis:

$$\rho_0 u_0 = f_1 - f_3 + f_5 - f_6 - f_7 + f_8, \tag{2.18}$$

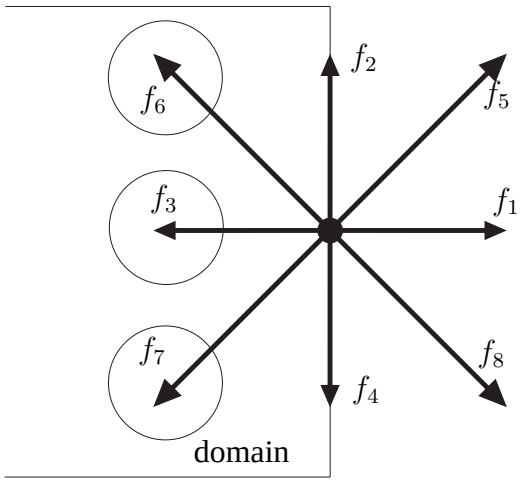


Figure 2.6: The three unknown microscopic densities (circled) of the right edge after the streaming step.

and

$$0 = f_2 - f_4 + f_5 + f_6 - f_7 - f_8. \tag{2.19}$$

Again, the other two equations are the macroscopic density (Equation 2.8) and the bounceback rule for the non-equilibrium part of the particle distribution normal to the edge (Zou and He, 1997) (Equation 2.12). Equations 2.8 and 2.18 can be equaled by isolating the microscopic densities f_3, f_6, and f_7, and then the result is solved to find u:

$$u = -1 + \frac{f_0 + f_2 + f_4 + 2(f_1 + f_5 + f_8)}{\rho_0}. \tag{2.20}$$

Solution of f_3 is found using Equation 2.12:

$$f_3 = f_1 - \frac{2}{3}\rho_0 u. \tag{2.21}$$

Now Equation 2.11 is used to replace f_6, and Equation 2.21 is used to replace f_3 in Equation 2.18 to find f_7:

$$f_7 = f_5 - \frac{1}{6}\rho_0 u + \frac{1}{2}(f_2 - f_4). \tag{2.22}$$

The same can be done to find f_6, replacing f_7 from Equation 2.19:

$$f_6 = f_8 - \frac{1}{6}\rho_0 u + \frac{1}{2}(f_4 - f_2). \tag{2.23}$$

2.5 External forces

External forces, such as gravitational acceleration, can be added to the simulation through a term in the equilibrium velocity (Sukop and Thorne Jr., 2005). Using the second law of Newton, and considering that density is proportional to mass, and that the relaxation time means the time of collisions:

$$\Delta \mathbf{u} = \frac{\tau \mathbf{F}}{\rho}, \tag{2.24}$$

in which $\Delta \mathbf{u}$ is a change in velocity, that must be added to the equilibrium distribution function velocity (Equation 2.7):

$$\mathbf{u^{eq}} = \mathbf{u} + \frac{\tau \mathbf{F}}{\rho}. \tag{2.25}$$

2.6 Single component, multiphase (SCMP) flow

One of the most important characteristics of LBM is the ability to efficiently simulate SCMP flows. The SCMP flow simulation comprehends only one chemical element divided into two phases: liquid and vapor. As examples, with this system one can study surface tension, evaporation, condensation, cavitation, contact angles, capillary rise, adsorption, capillary condensation and flows in porous media. To simulate phases, a long-range attractive force has to be added into the simulation. This force represents an interaction strength between particle distribution functions, and is part of the fundamentals of the van der Waals EOS. As a result of these interactions, the level of parallelism in LBM decreases (Sukop and Thorne Jr., 2005). The model described in this section was developed by Shan and Chen (1993), who first worked with multiphase LBM models.

2.6.1 LBM interparticle forces

The long-range attractive forces are calculated using the nearest neighbour particle distribution functions:

$$\mathbf{F}(\mathbf{x},t) = -G\psi(\mathbf{x},t) \sum_{a=1}^{8} w_a \psi(\mathbf{x} + \mathbf{e_a}\Delta t, t)\mathbf{e_a}, \tag{2.26}$$

in which G is the interaction strength, and w_a is the weight for each particle. For example, in the D2Q9 model, it is $1/9$ for $a = 1, 2, 3, 4$, and $1/36$ for $a = 5, 6, 7, 8$, and ψ is the interaction potential (Shan and Chen, 1993):

$$\psi(\rho) = \psi_0 e^{\frac{-\rho_0}{\rho}}, \tag{2.27}$$

in which ψ_0 and ρ_0 are arbitrary constants. This interaction potential represents an isothermal process, and must be monotonically increasing and bounded (Shan and Chen, 1993). Other equations of the interaction potential are also used, such as $\psi(\rho) = \rho_0 \left[1 - e^{\frac{-\rho}{\rho_0}}\right]$ (Shan and Chen, 1993), and $\psi(\rho) = \rho$ (Martys and Chen, 1996). Figure 2.7 shows the interaction potential of Equation 2.27 for $\psi_0 = 4$ and $\rho_0 = 200$.

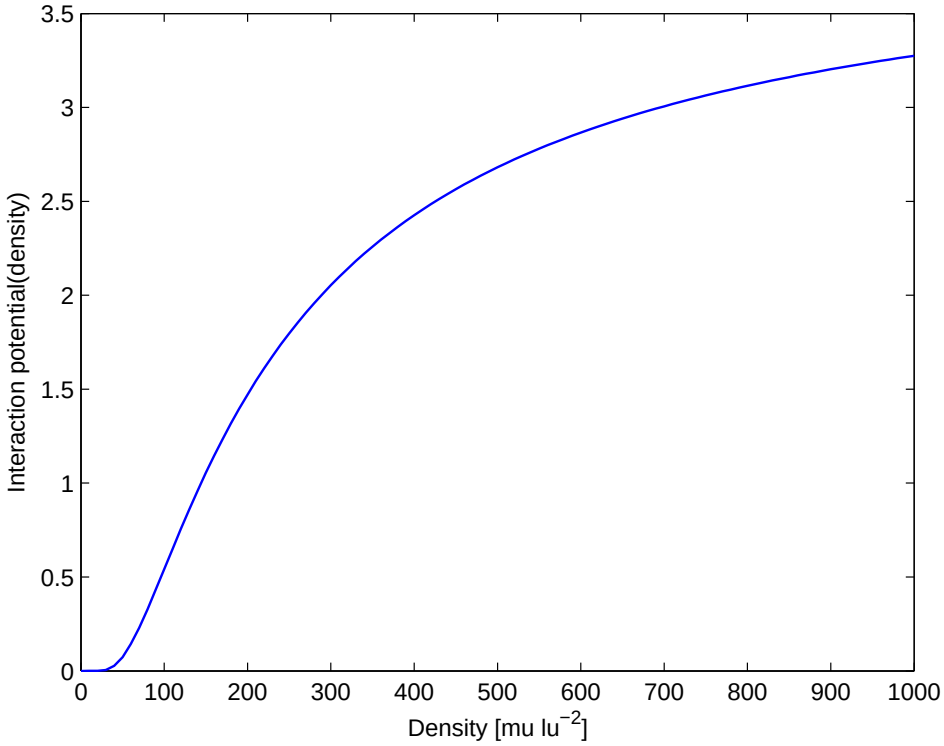


Figure 2.7: Isothermal interaction potential function for $\psi_0 = 4$ and $\rho_0 = 200$.

The interaction strength G is negative for attraction between particles and, as the interparticle force becomes stronger as density increases, the surface tension phenomenon can be simulated. Thereby, liquid regions undergo a stronger cohesive force than vapor regions. Furthermore, the attractive interparticle force is included in the model as an external force using Section 2.5.

2.6.2 The SCMP equation of state of the Lattice Boltzmann method

The non-ideal EOS must be used to simulate fluids using Equations 2.26 and 2.27:

$$P = \rho RT + \frac{GRT}{2}[\psi(\rho)]^2. \tag{2.28}$$

The ideal gas law is ρRT, which is used for the SCSP model. The non-ideal part is $^1/_2\, GRT[\psi(\rho)]^2$, which represents the attractive force between molecules and reduces pressure when G is negative. Phase separation may take place for negative G and nonmonotonic EOS. Applying $RT = 1/3$ for both SCSP and SCMP models (Sukop and Thorne Jr., 2005) in Equation 2.28, the non-ideal EOS becomes:

$$P = \frac{\rho}{3} + \frac{G}{6}\psi^2(\rho). \tag{2.29}$$

Figure 2.8 shows EOS plotted for some G values, with $G = -92.4$ being the critical value. This model does not include a repulsive force, which makes the liquid phase be more compressible than the vapor phase (Sukop and Thorne Jr., 2005). The equilibrium liquid-vapor configurations remain the same, but some simulations are more difficult to be executed.

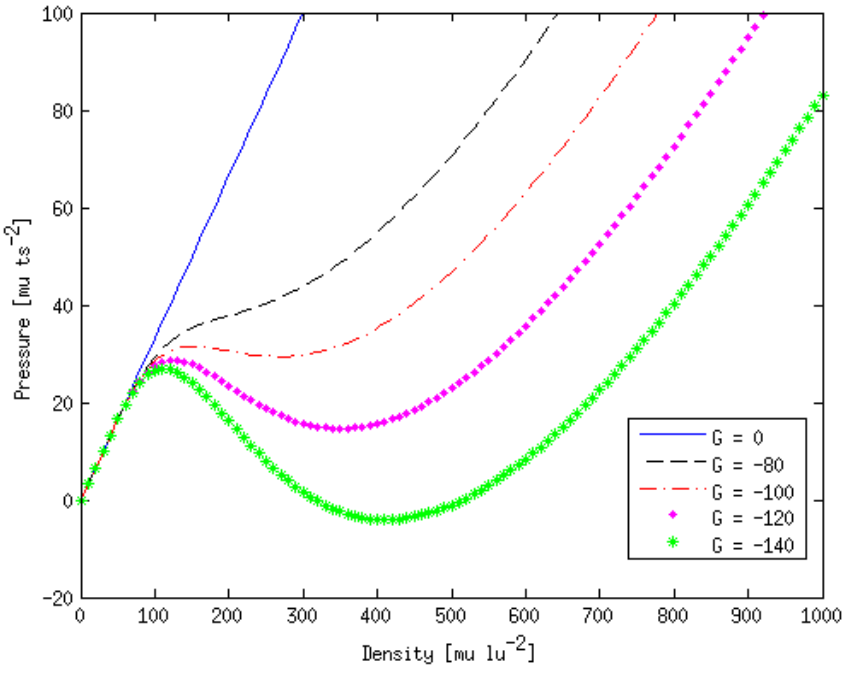


Figure 2.8: SCMP EOS for $\psi_0 = 4$, $\rho_0 = 200$, and $G = 0, -80, -100, -120$, and -140.

2.6.3 SCMP flow with solid walls

There is an adhesive interaction force between fluid particles and solid surfaces. Martys and Chen (1996) developed an elegant and simple model that calculates an analogous particle-particle interaction force, with the difference that the solids around a node are summed up, instead of ψ values:

$$\mathbf{F_{ads}}(\mathbf{x},t) = -G_{ads}\psi(\mathbf{x},t)\sum_{a=1}^{8} w_a s(\mathbf{x}+\mathbf{e_a}\Delta t,t)\mathbf{e_a}, \tag{2.30}$$

in which s values 1 for solid nodes and 0 for fluid nodes. Equation 2.27 is also used here to determine the interaction potential ψ.

2.7 Viscosity

The fluid kinematic viscosity in lu^2/lt is:

$$\nu = \frac{1}{3}(\tau - \frac{1}{2}). \tag{2.31}$$

The relaxation parameter τ should be greater than $1/2$ for positive viscosity, and thus physical viscosity. Instabilities start to appear when $\tau \approx 1/2$, and the most stable value for τ is 1 ($\nu = {}^1\!/_6$ lu^2/lt) (Sukop and Thorne Jr., 2005).

2.8 Reynolds number

The Reynolds number (Re) is a non-dimensional number given by:

$$Re = \frac{uL}{\nu} = \frac{\rho uL}{\mu}, \tag{2.32}$$

in which u is the fluid velocity, L is the characteristic length and μ is the dynamic viscosity. The Re shows a balance between viscous and inertial forces (Sukop and Thorne Jr., 2005), and real and simulated flows should have the same Re value. Low Re means the flow is laminar, with predominance of viscous forces, usually for internal flows with low velocity and high viscosity. When $Re << 1$, the flow is called Stokes or creeping flow, and often happens for liquids in porous

media because of the small pore sizes. When the Re gets higher, the Stokes flow starts to separate from the obstacle and then eddies start to form. On the other hand, high Re leads to transition or to turbulent flows, when inertial forces are predominant. High velocities, large characteristic lengths or low viscosity can make the flow unstable.

2.9 Parameter conversion

To run simulations and interpret their results correctly, the LBM parameters should be chosen properly, such as lattice size, numerical viscosity and maximum velocity (Januszewski, 2015). Given the fluid velocity, the Re and the measurements of the channel whereby a real fluid flows, it is possible to calculate the simulation parameters.

Using the lattice units given in Section 2.2, a simulation time step is by definition $1\ lt$, and the distance between two adjacent lattice nodes is $1\ lu$. To find the lattice spacing Δx, one should use:

$$\Delta x = \frac{L}{N-1}, \tag{2.33}$$

in which L is the channel height in meters and N is the number of lattice nodes. The lattice spacing unit is $[m/lu]$. Equation 2.34 calculates the lattice flow velocity:

$$u_{lb} = \frac{\Delta t}{\Delta x} u_{phys}, \tag{2.34}$$

in which u_{phys} is the experiment flow velocity. The lattice kinematic viscosity is:

$$\nu_{lb} = \frac{\Delta t}{\Delta x^2} \frac{u_{phys} L}{Re} = \frac{u_{phys}(N-1)}{Re}. \tag{2.35}$$

By choosing a number of lattice nodes in one direction, the lattice spacing is calculated with Equation 2.33 and, by choosing the lattice flow maximum velocity u_{max}, it is possible to compute the lattice time step size:

$$u_{phys} = u_{max} \frac{\Delta x}{\Delta t}. \tag{2.36}$$

From this result, the lattice viscosity is computed. The difficulty of this method is the appropriate choice of the lattice flow maximum velocity, because, despite the safe value of $0.1\ lu/lt$, some flows may present instabilities at this value.

There is another way to perform these calculations, by starting with a numerical viscosity number. As the range of values that do not create instabilities in the simulation is small ($0 < \nu_{lb} < 0.17$), this approach is appropriate to avoid instabilities. After picking up a value in this range and a lattice flow maximum velocity, the same Equations 2.33 to 2.36 can be used to calculate the other parameters.

2.10 Poiseuille flow

The Poiseuille flow occurs in a pipe or in a slit between two parallel surfaces (Sukop and Thorne Jr., 2005). The velocities at the walls are zero, satisfying the no-slip condition, and the maximum velocity is in the center of the flow. The velocity profile in a slit of width $2a$ is parabolic:

$$\mathbf{u}(x) = \frac{G^*}{2\mu}(a^2 - x^2), \tag{2.37}$$

in which G^* is the linear pressure gradient $\frac{(P_{in}-P_{out})}{L}$ or a gravitational pressure gradient (ρg). The average velocity in a slit is $2/3$ of the maximum velocity in the slit or pipe:

$$\mathbf{u}_{average} = \frac{2}{3}\frac{G^*}{2\mu}a^2. \tag{2.38}$$

2.11 Laplace law

The Laplace law states that there is a pressure difference between the inside and the outside of a bubble or a drop, and the pressure is always higher inside the bubble or drop. For 2D drops or bubbles, there is only one possible radius of curvature, and the law is (Sukop and Thorne Jr., 2005):

$$\Delta P = \frac{\sigma}{r}. \tag{2.39}$$

This law applies to interfaces between liquid and vapor phases of the same component, when σ is called surface tension, and between different components, when σ is called interfacial tension.

3 Parallel processing

From 1986 until 2002, software developers and final users could rely on technical advances of microprocessors to increase performance of their programs, with an average of 50% more speed per year (Pacheco, 2011). According to Sanders and Kandrot (2011), the first personal computers ran with internal clocks operating at 1 MHz until 1 GHz and 4 GHz after 30 years, a gain of more than 1000 times in the clock speed. However, after 2002, the performance gain decreased to 20% per year due to technical issues, like the difficulty of dissipating heat of microprocessors with high density of transistors, and also the reach of the physical limit to the transistor size.

Supercomputers also followed this way of achieving high performance gains, but combined with the increasing number of processors, of usually tens or hundreds of thousands of processor cores (Sanders and Kandrot, 2011). On the trail of supercomputers, aiming to overcome the smaller amount of performance gain in the clock speed, the next step for industries was adding more than one processor on a single chip. The *multicore revolution*, as it is sometimes referred, brought two-, three-, four-, six-, eight-, twelve-, and even sixteen-core CPUs (Sanders and Kandrot, 2011).

The main consequence of this decision for programmers was that their programs based on single core CPU would no longer benefit from new technologies as before, since programs written for single-core processors do not recognise multiple cores. Since 2005, when most companies started offering multicore processors, serial codes needed to be rewritten in order to fairly increase performance using these new processors. There is a trend for developing translation programs, which would automatically convert a serial code into parallel code, but since this approach has only been good for specific cases, it is needed to write more efficient parallel algorithms for each case separately (Pacheco, 2011).

Other trend in parallel processing is the recent general-purpose computation on GPUs. In the early 1990s, the popularity of operating systems with graphical interface increased, creating a demand for 2D display accelerators (Sanders and Kandrot, 2011). At the same time, a company called Silicon Graphics already offered 3D graphics for professional computing, such as government, and defense applications and scientific and technical visualization. They released the Open Graphics Library (OpenGL) specification, the programming interface they used in their hardware. After that, applications requiring 3D graphics had scaled incredibly, usually in realistic environments for games. Other companies also developed their graphics accelerators, for example, NVIDIA®,

ATI, and 3dfx. GPUs performed more and more functions of the OpenGL graphics pipeline, until the NVIDIA®'s release of the GeForce® 3 series, which was the first chip to contain both programmable vertex and pixel shading stages and to enable more control over the GPU calculations. Then, researchers began to use GPUs for general-purpose applications, but were harnessed to the graphics application programming interface (API), being necessary to convert their problems into rendering problems and deal with resource and programming restrictions from GPU, besides the need of learning the OpenGL API. Five years after the release of the GeForce® 3 series, NVIDIA® launched the GeForce® 8800 GTX, the first GPU with CUDA, that included new components specially designed for general-purpose applications, and aimed at easing GPU programming for these users. As with multicore CPUs, programming GPUs requires rewriting serial code to take advantage of the many-core approach.

In this chapter, an API for parallel programming on CPUs is presented, called Message-passing Interface (MPI). Then, parallel programming on GPUs is discussed, starting with the GPU concept and afterward CUDA, the scientific visualization field based on OpenGL and an important tool called CUDA-OpenGL interoperability are explained.

3.1 Message-passing Interface

MPI is a library of functions for languagues such as C, C++, and Fortran, and uses the concept of message passing to perform communications in distributed-memory systems, which consist of computer nodes with their own memory, which are only accessible to its own node (Pacheco, 2011). Each instance of a program running on these nodes is called a process, which needs to communicate information with other processes from different nodes to solve a problem. MPI does this communication by using functions for point-to-point communications, which involve only two processes, and collective communications, for more than two processes. As supercomputers are usually distributed-memory systems, MPI is the adequate approach to use them, which makes MPI an important tool in parallel computing.

In this subsection, the basic details of MPI are given, such as communication pattern, which is the main characteristic of this API, and performance evaluation, which enables the measurement of how faster a parallel program is in relation to serial code.

3.1.1 Communication

All MPI code should be between functions *MPI_Init()* and *MPI_Finalize()*. The first function does all MPI setup, allocating memory for message buffers and identifying the process rank, while the last one frees all memory used. *MPI_Init()* also defines a communicator called *MPI_COMM_WORLD*, which consists of the collection of processes that are able to communicate to each other (Pacheco, 2011). The communicator can provide information such as the total number of processes by using the function *MPI_Comm_size()*, and the rank of the process that calls the function *MPI_Comm_rank()*.

Recalling that MPI follows the single program, multiple data (SPMD) approach, it means that only one program is written independently of the number of processes that will run, and this is done by making branches that specify different tasks for each process. As input can only be dealed by process 0, while all processes can deal with output, communications are necessary to exchange information among processes. Some point-to-point communications comprehend the functions *MPI_Send()* and *MPI_Recv()* for sending and receiving messages; *MPI_Ssend()*, which makes a synchronous send that always block until matching the right receive function; *MPI_Sendrecv()*, which sends and receives a message in the same call; and *MPI_Sendrecv_replace()*, which sends and receives a message when buffers are the same. On the other hand, some collective functions are *MPI_Reduce()* and *MPI_Allreduce()*, for reduction operations; *MPI_Bcast()* for broadcasting information; *MPI_Scatter()*, which divides an array equally to the processes and sends to each process only the information it needs; and *MPI_Gather()* and *MPI_Allgather()*, which does the opposite of *MPI_Scatter*, collecting all parts of an array from each process and sending it into a process or into all processes, respectively (Pacheco, 2011).

Another important detail on MPI communications is the data distribution. In block partition, data blocks are divided equally among the processes, and the first data block is assigned to the first process, the second data block is assigned to the second process, and so forth. Other type of data distribution is the cyclic partition, which assigns data like a round-robin scheduling, for example, the first position of data goes to the first process, the second position goes to the second process, and so on. These two types can be combined to create a third type of data distribution, called block-cyclic partitioning, when blocks of data are assigned to the processes in a round-robin manner. Most basic MPI functions works with block partition type, so, to have cyclic or block-cyclic partitioning, a MPI-derived data type can be created in order to communicate different

partitioning (Pacheco, 2011). The cost of sending, for example, ten memory positions in multiple messages with size of one memory position is higher than sending one message with all ten positions, because of the communication overhead and the message setup that follows the transmited data. MPI-derived data types are worthy because they can reduce the amount of messages between processes, in case of different data types that need to be sent. A new MPI type can be built with the function *MPI_Type_create_struct()*, which needs to be further committed with *MPI_Type_commit()* and later freed with *MPI_Type_free()*.

3.1.2 Performance assessment

To measure how faster a parallel code is than a serial code, MPI provides the function *MPI_Wtime()*, which returns the number of seconds that have passed. The subtraction of two calls for *MPI_Wtime()* gives time elapsed to run the code that is contained between these two calls. It is worth noting that the time counted in this function includes CPU idle time, such as time waiting for a call to *MPI_Recv()* returns (Pacheco, 2011). Before calling *MPI_Wtime()*, the collective communication function *MPI_Barrier()* must be called to guarantee that all processes start the next instruction at the same time. After measuring time, every process has its own value for the elapsed time; a reduction operation has to be done to choose the maximum value, i.e., the time taken for the slowest process to run.

When a program is assessed, it should be statistically treated, i.e., some measures with the same input and same number of processes should be made in the same equipment with the same configuration in order to get a more realistic value. The difference in measures is due to the interaction of the program with the operating system, which is unpredictable, for example, the operating system may be processing a data package from the Internet, or an input/output request from the printer, etc. From these measures, the minimum runtime should be picked up, since these interactions hardly ever make the program run faster. The number of MPI processes used should be the same number of physical cores that a node has, in order to reduce interconnect bottlenecks (Pacheco, 2011). This improves performance and reduces variability in run-times. Even microprocessors that can run more than one thread per core are still counted as only one process per core. Another thing to consider is that serial code is different from MPI parallel code running with only one process. This happens because of the MPI overhead, so MPI code running with only one process should not be used in performance evaluation.

3.2 Graphics Processing Unit

GPUs follow a many-core approach and have a large number of smaller cores, when compared to CPU. As other many-core processors, GPUs have the best floating-point performance since 2004, as shown in Figure 3.1. They also have higher memory bandwidth, as shown in Figure 3.2.

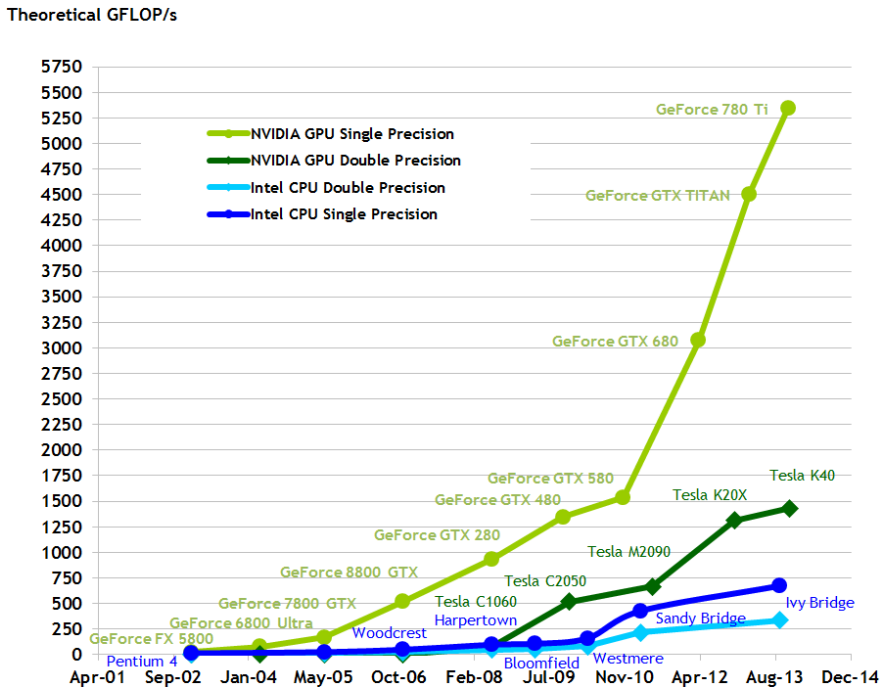


Figure 3.1: Temporal evolution of theoretical billion of floating-point operations per second (GFLOPS) for NVIDIA® GPUs and Intel CPUs, for single and double precision (NVIDIA, 2014b).

The performance difference between GPUs and CPUs is due to design characteristics of both types of processors. CPU is optimized for sequential code, so it has a sophisticated control logic and large cache memories. The control logic can execute instructions in parallel and change their order of execution, while cache memories reduce instruction and data access latencies (Sanders and Kandrot, 2011). In GPU, control logic is very simple and cache memory is smaller, which leaves room for lots of arithmetic units. These differences are shown in Figure 3.3, in which the number of transistors dedicated to calculations is much higher on GPU, because the number of transistors dedicated to cache memory and control logic is much smaller than on CPU (NVIDIA, 2014b). The difference in memory bandwidth values is also due to design characteristics. While CPUs have to satisfy requirements from the operating system, applications, and input/output devices, GPUs do

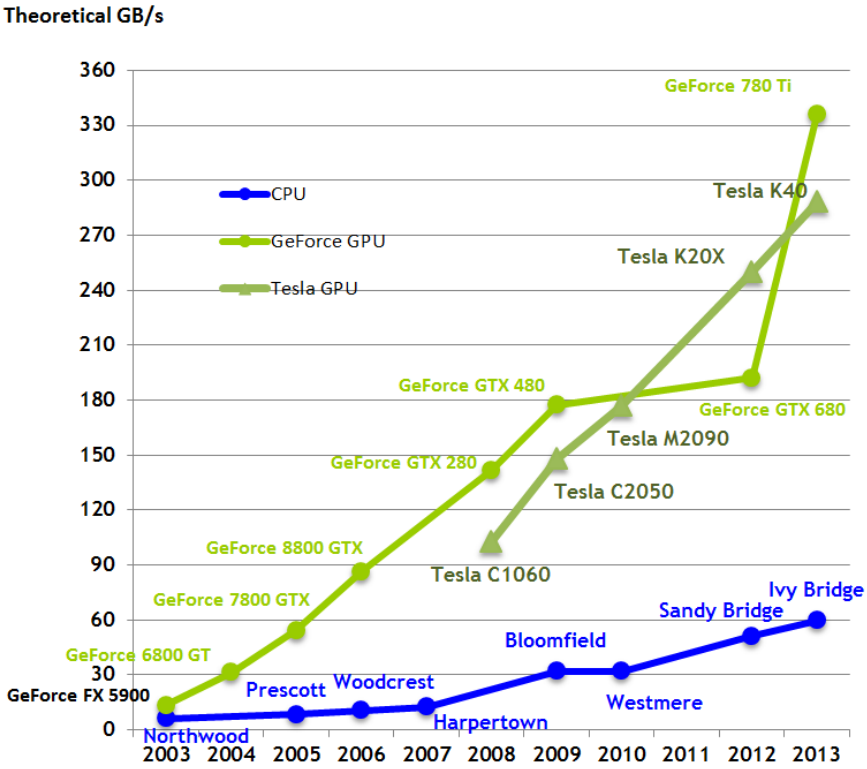


Figure 3.2: Temporal evolution of theoretical memory bandwidth for NVIDIA® GPUs and Intel CPUs (NVIDIA, 2014b).

not have these requirements, and with less constraints they are easily built with higher memory bandwidth. The low cost is another advantage of GPUs, which offer high performance and are not so expensive as clusters, so a great part of customers can have a computer with an off-board GPU.

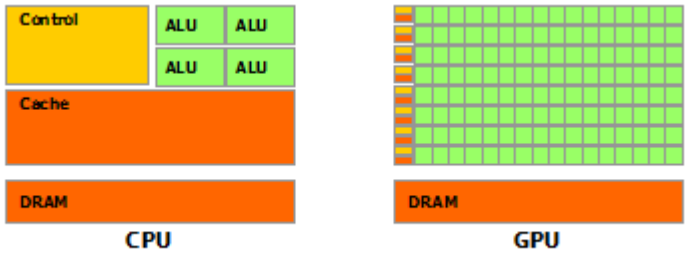


Figure 3.3: Differences between CPU and GPU architectures (NVIDIA, 2014b).

GPU has increased its advantage when it started to work with the Institute of Electrical and Electronics Engineers (IEEE) floating-point standard, which makes results more predictable, since the NVIDIA®'s 80 series launch. The support is comparable to that of CPU's, as well as precision.

Nowadays double-precision speed is half of the single precision speed, just like high-end CPUs.

A GPU is organized into 8 streaming multiprocessors (SMs) in the NVIDIA® GeForce® GTX 560 Ti graphics card, released in 2011. There, each SM has 48 streaming processors (SPs) that share control logic and instruction cache, so there are 384 SPs, reaching 1.5 TFLOP. The SM and SP are shown in Figure 3.4, in which multiprocessor 1, multiprocessor 2, ..., multiprocessor N are the SMs; processor 1, processor 2, ..., processor M are the SPs, and device is the GPU. With these values, thousands of threads can run, a number much higher than the number of threads in multicore CPUs. The device, constant and texture memories are the Graphics Double Data Rate (GDDR) Dynamic Random Access Memory (DRAM), which has more latency than CPU motherboard memory, but this is hidden by the GPUs higher bandwidth. Although constant and texture memories are together with device memory, each SM has its own constant and texture caches, as shown in Figure 3.4. GTX 560 Ti has 1 GB of GDDR DRAM, and its memory bandwidth from CPU to GPU is 5.7 GB/s, from GPU to CPU is 6 GB/s and internally it is 128 GB/s. The lower values of communication with CPU are due to the PCIe bus bandwidth, and are comparable with those from CPU Random Access Memory (RAM) memory. It is worth noting that the current compute capability of new NVIDIA® graphics cards is 5, though the graphics card that was used in this work, the NVIDIA® GeForce® GTX 560 Ti, have compute capability equal to 2.1 (Fermi architecture).

Although GPUs have advantage over CPUs in computations, there are tasks that are better performed on CPU, like input/output handling. Thus, to combine the best advantages of both architectures, most applications use hybrid systems, with CPU in sequential parts and GPU in intensive calculations. For hardware-accelerated 2D and 3D rendering, a typical multi-platform API used to interact with GPU is OpenGL. Its specification began as an initiative by SGI in 1991 and its new versions are regularly released by the Khronos Group. For general purpose processing, NVIDIA® created a parallel computing platform and programming model on top of GPUs. The CUDA programming model is designed to support this approach, aiming for performance improvement.

3.3 CUDA

The CUDA platform was launched in November, 2006 by NVIDIA® (NVIDIA, 2014b). Its current software version is the 6.5, even though this work was based on the version 5.5. CUDA

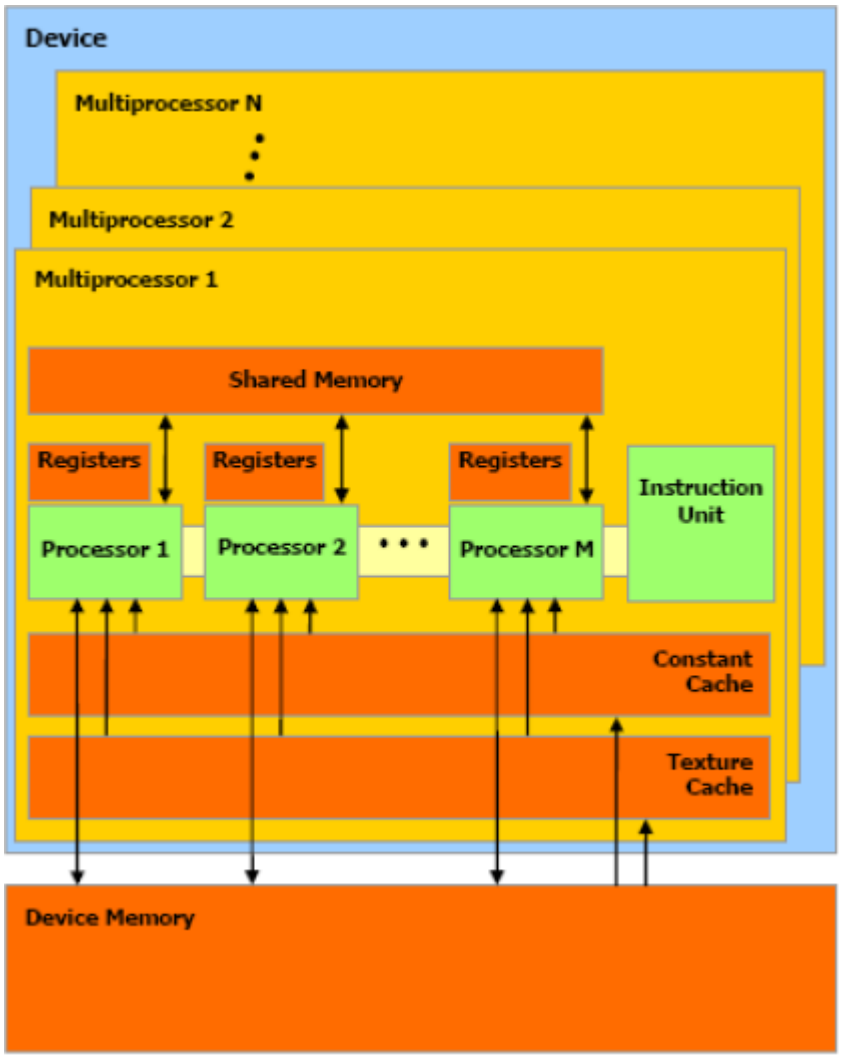


Figure 3.4: GPU architecture (NVIDIA, 2014b).

C is an extension to C that permits the programmer to define kernels, functions that are executed on GPU, in which the user can define the number of threads that will be run in parallel. Threads compose a block, and blocks compose a grid, so all kernels runs with a number of blocks per grid and threads per block. This CUDA configuration is exemplified in Figure 3.5. Blocks and grids may have up to three dimensions, and they have a maximum number of threads per block and blocks per grid, limited by the shared SP memory. On GTX 560 Ti, a block may have up to 1024 threads.

The decomposition in blocks and threads enables automatic scalability according to the number of SMs that a graphics card has. As thread blocks can be executed in any order, the GPU can schedule the blocks in any order, increasing scalability. The scalability increases with the number

of the GPU SMs: the more the number of SMs, the faster a program runs, as shown in Figure 3.6 (NVIDIA, 2014b).

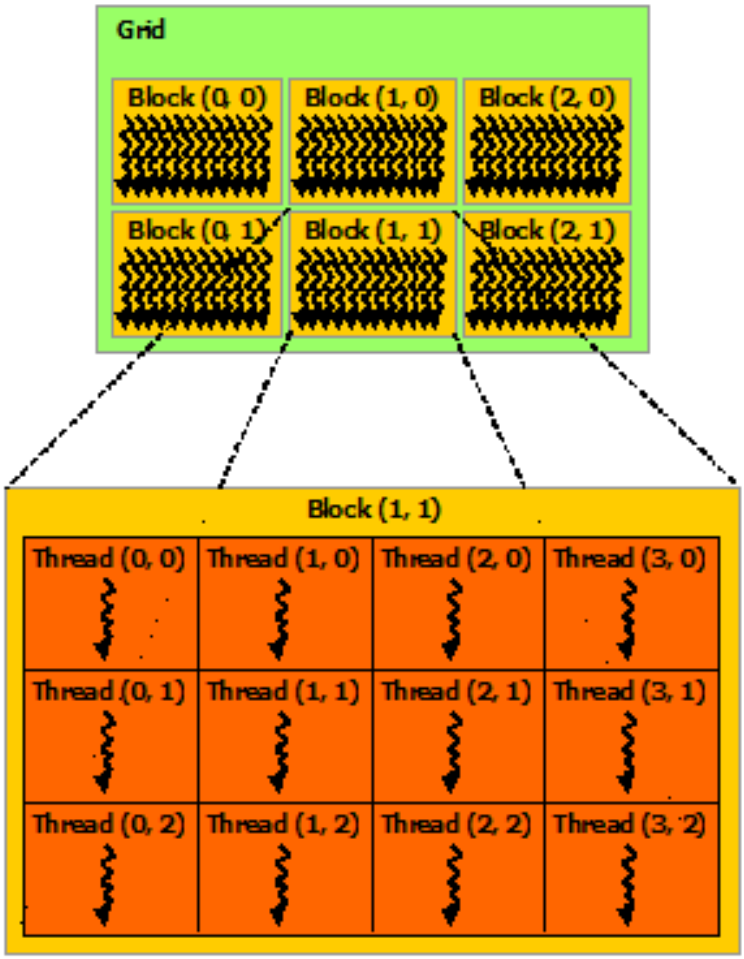


Figure 3.5: CUDA programming model (threads, blocks and grid) (NVIDIA, 2014b).

CUDA has a versatile memory hierarchy that should be used to achieve higher performance, since the private local memory from each thread and the shared memory for each block are faster than the global memory, which is available to all threads (NVIDIA, 2014b). Figure 3.7 shows this memory hierarchy available to the threads. Local memory available to each thread are the registers in Figure 3.4, and global memory is the device memory in the same figure. Threads also access two read-only memory spaces: the constant and the texture memories. The use of constant memory requires less memory bandwidth than the use of global memory, while texture memory also improves performance when reads have specific access patterns (Sanders and Kandrot, 2011).

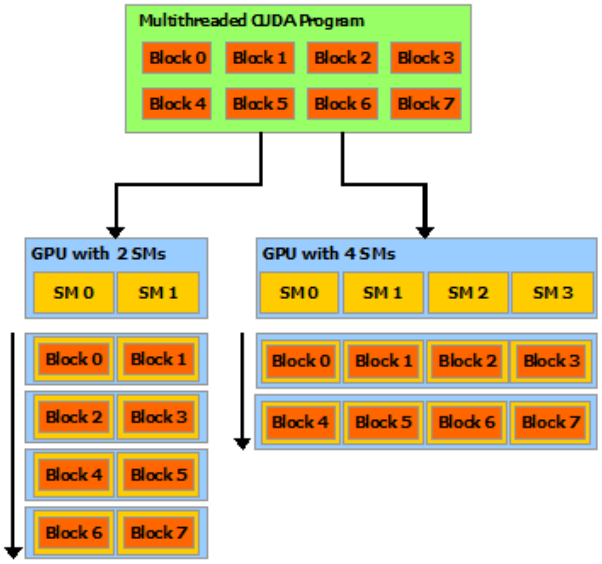


Figure 3.6: Automatic scalability of CUDA programs according to the number of SMs of a GPU (NVIDIA, 2014b).

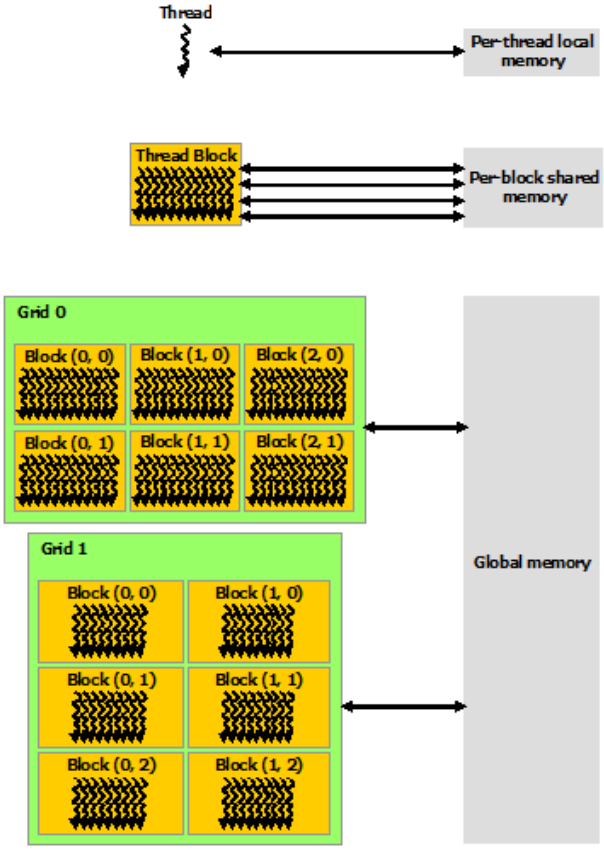


Figure 3.7: CUDA memory hierarchy (NVIDIA, 2014b).

3.4 Scientific visualization

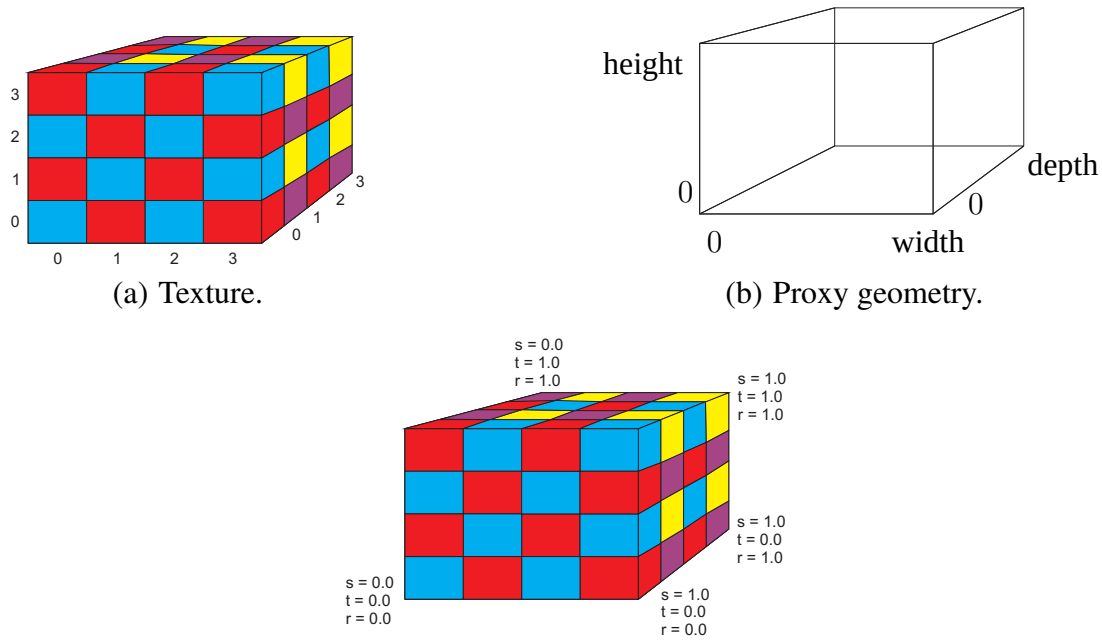
In this section, the 3D texturing technique for volumetric visualization and the use of shaders in the GPU programable pipeline are presented.

3.4.1 3D texturing

To solve the problem of simulating and visualizing online, a simple and efficient technique of visualization should be chosen. The 3D texturing has wide support in GPU, is simple to implement and offers high frame update rates (Engel *et al.*, 2006). Texture mapping is a technique that aims at improving the quality of rendered images (Blythe, 1999) whose domain is a regular grid (Engel *et al.*, 2006). Usually, textures are used to provide more details about color and lighting of a complex surface, through a fragment color modification, in order to give the scene more realism (Akenine-Möller *et al.*, 2008). In scientific visualization, texturing techniques can be applied with other objectives, like simulation data visualization and image filtering (Telea, 2008).

Often a texture is an array of color values. Each element of this array is called texel, which is an abbreviation for texture element (Akenine-Möller *et al.*, 2008). A texture can have one, two or three dimensions, and is attached to a model to create the desired effect (Shreiner, 2009). To be able to visualize data, the texture needs to be bound to a proxy geometry. In scientific visualization, a 3D texture may be data that one wants to visualize, like velocity or pressure in a fluid flow simulation.

The way of connecting the texture to the proxy geometry is through the texture coordinates (Shreiner, 2009). Each vertex of the proxy geometry must be associated to a texture coordinate. Then, the texture coordinates of the geometry vertices are interpolated, so that each pixel of the rasterized image has a correspondence in the texture (Akenine-Möller *et al.*, 2008). Figure 3.8 shows an example of the association between texture coordinates and geometry coordinates. In this example, the volume data with size 4 x 4 x 4 represents a texture (Figure 3.8(a)). This texture should be associated with a proxy geometry (Figure 3.8(b)) to result in Figure 3.8(c). In this figure, the texture coordinates are normalized in the range from 0 to 1 in each dimension.



(a) Texture.

(b) Proxy geometry.

(c) Final result of texture bind.

Figure 3.8: Example of 3D texture mapping.

3.4.2 Shaders

Modern GPUs have a programmable pipeline that aims for improvements on lighting and on texturing effects, and better image quality (Shreiner, 2009). A shader is a program to be run in GPU, which replaces predefined functions of the fixed pipeline. There are four types of shaders: vertex shaders, geometry shaders, fragment shaders, and tesselation shaders. Both three types of shaders have similar syntax, and are processed with the same unit of a modern graphics card.

3.5 CUDA-OpenGL interoperability

When CUDA 5.5 is used, lots of memory copies between CPU and GPU usually happen, which tends to be the bottleneck of this kind of implementation. With the simultaneous use of OpenGL for visualization, it is possible to avoid these data transfers and also data transfers inside the GPU at each change of contexts (CUDA and OpenGL). Furthermore, data are allocated on CPU only in the beginning, and on GPU they are allocated only once for both contexts. This resource

can be achieved because the CUDA programming interface allows the use of allocated resources on the OpenGL context in CUDA kernels (Alt, 2012), through mapping/unmapping an OpenGL buffer into CUDA's memory space. Its objective is performance gain and memory savings, as less memory transfers and less memory allocations are needed.

In our case, to use interoperability, a Pixel Buffer Object (PBO) is created in OpenGL, which allows the buffering of a rendering that will not be immediately shown on the screen. This buffer stores simulation results from CUDA calculations, which are shown after each CUDA-OpenGL iteration. The PBO should also be associated to a *cudaGraphicsResource* CUDA pointer, which contains simulation results in the CUDA context. In other words, as the PBO and the *cudaGraphicsResource* pointer are associated, they contain the same data. After that, for each iteration, the *cudaGraphicsResource* pointer is mapped and recovered, then it is passed as an argument to the CUDA kernels. Thereafter, CUDA calculations can be performed normally and, when they end, the resource must be unmapped to be used in the OpenGL context.

To visualize PBO data in OpenGL, the PBO is binded to the 3D texture that is being visualized, before associating its coordinates to a proxy geometry. Summing up the steps and functions to use CUDA and OpenGL in the same graphics card:

- create a PBO;
- create a *cudaGraphicsResource* pointer;
- associate both variables with *cudaGraphicsGLRegisterBuffer()*;
- for each iteration of CUDA and OpenGL:
    - map the *cudaGraphicsResource* pointer (*cudaGraphicsMapResources()*);
    - obtain the mapped pointer (*cudaGraphicsResourceGetMappedPointer()*);
    - perform CUDA calculations passing the mapped pointer as a kernel argument;
    - unmap CUDA pointer (*cudaGraphicsUnmapResources()*); and
    - visualize with OpenGL.

3.6 Performance evaluation

To evaluate performance of serial, CPU parallel and GPU parallel codes, and versions with and without interactive visualization, or some other modifications, run-times of each one of these versions are used to calculate a relation called speedup:

$$S(n,p) = \frac{T_{serial}(n)}{T_{parallel}(n,p)}, \tag{3.1}$$

in which $T_{serial}(n)$ is the run-time of the serial code, which depends on the number n of elements of the domain, and $T_{parallel}(n,p)$ is the run-time for the parallel code, which not only depends on the number of elements of the domain, but also on the number of processes p (or the number of cores in GPU). If the speedup is equal to p, then the code shows linear speedup, the ideal value that is hardly achieved. With this ideal value, it is possible to calculate the code efficiency, which measures the distance from the ideal value, and is the speedup per process:

$$E(n,p) = \frac{S(n,p)}{p} = \frac{T_{serial}(n)}{p \times T_{parallel}(n,p)}, \tag{3.2}$$

in which $E(n,p)$ is less than 1.

Another way of assess performance of parallel implementations is the scalability. A program is scalable when its efficiency does not decrease as the number of processes and the problem size increase. A program that shows constant efficiency regardless of the problem size is called strongly scalable. On the other hand, programs that shows constant efficiency only when the problem size increases at the same rate as the number of processes are called weakly scalable.

4 Implementation

This chapter describes the simulator implementation, the simulator engine, the visualization procedures, and the graphics user interface. Finally, the serial version and the MPI parallel CPU version, both without interactive visualization, are discussed. The source code of the GPU version with interactive visualization is available at Oliveira (2015).

4.1 FAPESP project

This work aims at meeting some design requirements of the FAPESP project *Fluid simulator implementation in microsystems based on particle methods for massively parallel processing in GPU cluster*. These requirements were: to import 2D files and 3D computer aided design (CAD) files to specify solid boundaries of the LBM domain, to import a previously defined experiment, to define boundary conditions, initial conditions and excitations, to solve the fluid-structure problem, to visualize simulation, to monitor the simulation with virtual sensors, and to save a simulation instant. Details about these requirements are given in the next subsections.

4.1.1 2D and 3D files import

To import files that define the solid nodes of the simulation domain, it is required a monochromatic bitmap file for the 2D case, and a stereolithography (STL) file for the 3D case. In the 2D case, black pixels indicate solid nodes in the simulation, while white pixels indicate fluid nodes, with one pixel per node. In the 3D case, a STL file can be exported from CAD programs. The 3D model has its solid parts as solid nodes, and the empty space is filled with fluid in the simulation. An input parameter in the input parameter file (Subsection 4.1.2) defines the simulation dimension: if it is equal to 1, it defines a 2D simulation, otherwise, if it is equal to any positive integer value, it defines the value of the largest dimension of the 3D file. STL 3D models can be built with any size, and the simulation size is defined at execution time.

The monochromatic bitmap file is currently read by the simulator with the help of the *QImage* class of the Qt library (Qt, 2014), although in the first versions of the simulator, it was read with

C support for file readings. Qt is a cross-platform application and a user interface framework for developers using C++. In this work, Qt 5.1 is used to implement the user interface, interaction features and some useful Qt classes that were used to improve the simulator.

The STL file is read with the help of the Common Versatile Multi-purpose Library for C++ (CVMLCPP). This library aims at offering high-quality implementation of commonly needed functions, eliminating redundancy in basic implementation that may even have bugs (University of Geneva, 2012). A regular 3D matrix, which is a basic structure of this library, and a geometry, for 3D surface models, are declared. Then the *readSTL* function is called to read the input STL file, and the *voxelize* function converts the read 3D surface model into voxel data (in the 3D matrix), with value 1 for solid voxels, and 0 for fluid voxels (a voxel is a volume element, just like pixel is a picture element for 2D images). After that, the volume can be read through the matrix and the solid matrix is filled with input values.

4.1.2 Previously-defined experiment import

To import previously-defined experiments, the simulator needs two files: a 2D or 3D model (Section 4.1.1) and a file of input parameters. The input parameters are contained in a *.dat*-extension file, and include:

- number of GPUs (*devices*);
- length of the maximum dimension for 3D domains (1 defines D2Q9 model) [lu];
- initial velocity of the fluid in x, y and z axes (one axis per line) [lu/ts];
- initial density of the fluid [mu/lu^2 for 2D and mu/lu^3 for 3D];
- fluid viscosity [lu^2/ts];
- number of steps between each screen update [ts];
- external forces applied on the fluid in x, y and z axes (one axis per line) [$mu\,lu/ts^2$];
- interaction strength [dimensionless]; and
- adsorption interaction strength [dimensionless].

4.1.3 Definition of boundary conditions

The standard boundary condition is set with the periodic condition, i.e., if no other condition is defined, the periodic condition is applied to every border of the domain. For solid nodes, the standard is the bounceback condition. The simulator can also be changed to apply constant velocity or pressure in the borders. All conditions were implemented according to Section 2.4.

4.1.4 Definition of initial conditions

Besides the previously-defined experiment import file, some other conditions can be initially defined, such as uniform density or density plus a random component, vertical or horizontal interface of liquid and gas, and seed bubbles or drops.

4.1.5 Definition of excitations

External forces can be applied to the whole domain, like the gravitational force, and were implemented according to the Section 2.5. The initial value in each axis is defined in the input parameters file, and this value is applied during the entire simulation.

4.1.6 Fluid-structure problem solution

The fluid-structure problem was solved through the bounceback boundary condition explained in Subsection 2.4.2. This condition is applied whenever there is an interface between solid and fluid. The simplicity of this condition allows the simulation of complex geometries.

4.1.7 Simulation visualization

The simulation visualization was implemented using the OpenGL (version 4.3) and Qt (version 5.1) libraries, and the OpenGL Shading Language (GLSL) (version $4.30.8$ with shader prepro-

cessor in version 430) for the shaders. The 3D texturing technique was applied to the simulation domain, and the shaders, user interface, and interaction features that compose the visualization are better explained further in this chapter.

4.1.8 Virtual sensors for simulation monitoring

The virtual sensor was implemented as a cursor in the user interface, which measures fluid properties of the point on which the cursor is placed. Its details are given further in this chapter.

4.1.9 Save of a simulation instant

A simulation instant can be saved through an icon in the user interface, which calls a function to save Visualization Toolkit (VTK) files. These files contain information about the simulation, such as pressure and velocity fields, the number of components for the velocity, the number of files in case of parallel save, and which data positions are contained in each file.

4.2 Simulation engine

Figure 4.1 shows a scheme of the performed LBM steps. First, memory is allocated on CPU for LBM initial data. After the filling of initial data on the allocated memory, memory on GPU is allocated, in order to copy CPU data into GPU global memory. Once done the setup, the LBM steps starts running iteratively.

Simulation data (ρ, $\mathbf{u}$, and each f) were stored in a structure-of-arrays, in which ρ, $\mathbf{u}$, and each direction of f are an array of size equal to the number of lattice nodes of the domain. A structure-of-arrays is useful to achieve coalescing in GPU, in contrast to the array-of-structures that is better for CPUs. Threads read the arrays in contiguous memory sections, which allows efficient data reading and writing (Aksnes and Elster, 2009). Results of each iteration were calculated in the collision step, and they are: the macroscopic density of each node (ρ), the macroscopic velocity of each node ($\mathbf{u}$), or the macroscopic velocity in each axis (u_x, u_y or u_z) of each node.

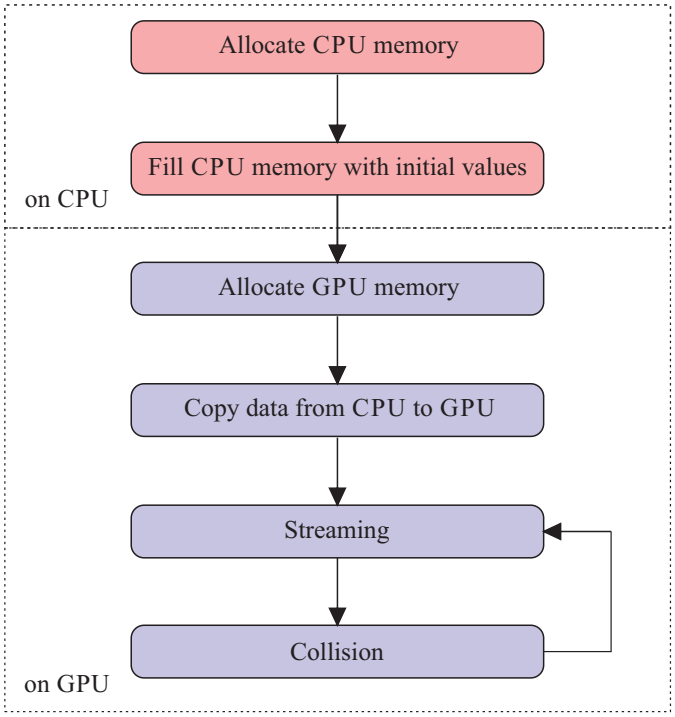
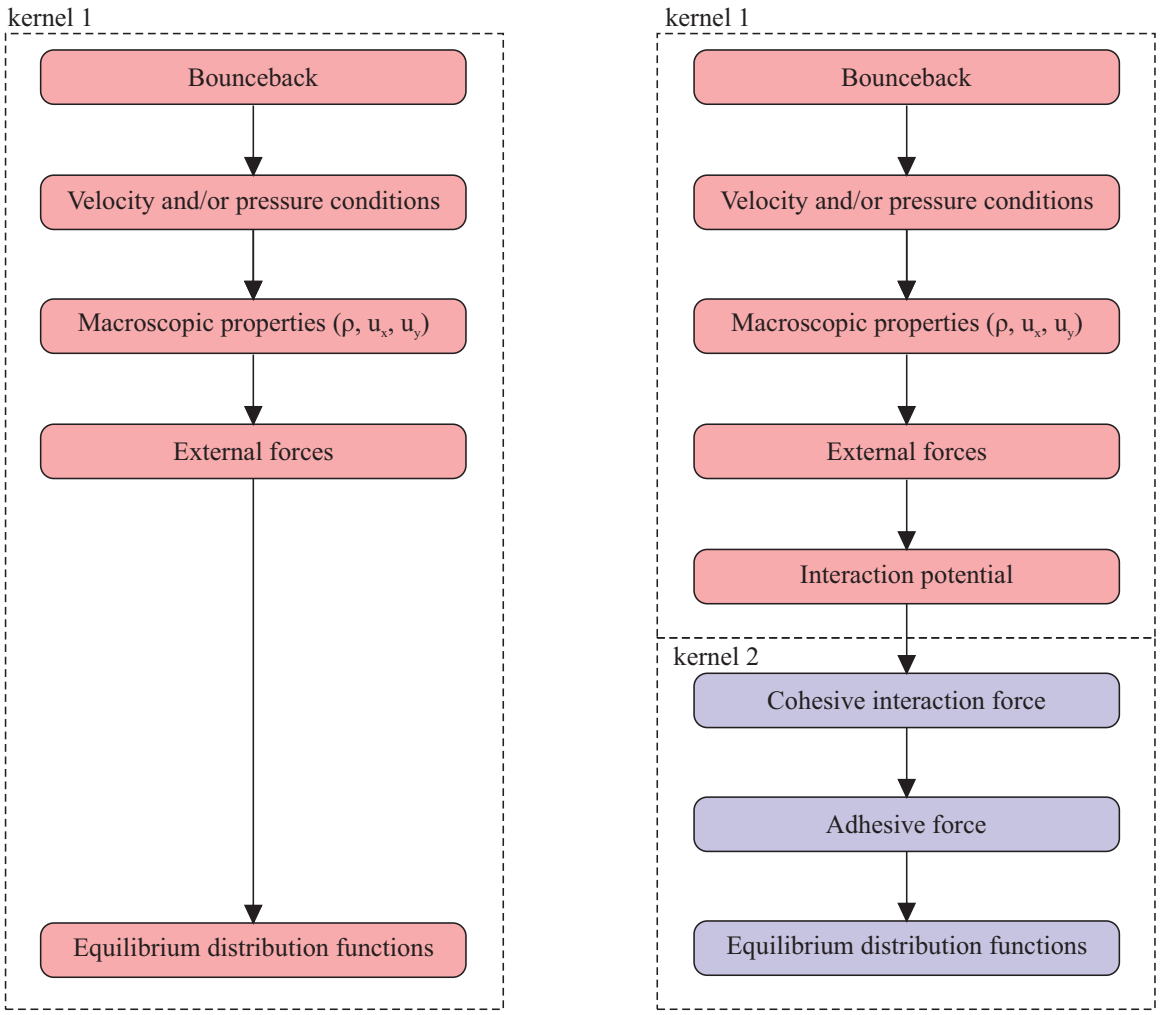


Figure 4.1: Simulation engine.

The LBM steps were performed in GPU using CUDA, by calling two kernels iteratively in case of single phase flows, and three kernels in case of multiphase flows. For single phase flows, one kernel performed the streaming step and periodic boundaries, and the other performed the collision step, bounceback, and inlet and outlet boundary conditions. For multiphase flows, the same kernel was called for the streaming step and periodic boundaries, the second kernel performed the other boundary conditions, the first part of collision, and the interaction potential calculation, and the last kernel calculated the cohesive and adhesive forces and ended the collision. Figure 4.2 shows the difference in kernels for single phase and multiphase flows, respectively. This difference is due to the interaction potential field, that should be calculated for all points of the domain before the simulation proceeds. It is worth noting that external forces and multiphase flow were implemented only for 2D domains.

In all kernels executions, each block had 32 threads in x-axis, 4 blocks in y-axis, and 1 block in z-axis, and the domain size was divided by these numbers in each axis to define the grid size ($width/32$ in x-axis, $height/4$ in y-axis, and $depth$ in z-axis). Each lattice node corresponds to one thread.



(a) Single phase kernel. (b) Multiphase kernels.

Figure 4.2: Collision kernels.

A sophisticated algorithm derived from the two-step algorithm, called swap algorithm, was also used (Mattila *et al.*, 2006; Bailey *et al.*, 2009). In the two-lattice or shift algorithms, memory needs to be allocated twice to represent two copies of the simulated domain. This is necessary because data may be lost during the streaming step, which exchanges distribution values with the neighbour nodes. In the two-step algorithm or the swap algorithm, this additional memory does not need to be allocated, because a specific pattern of exchange of distribution values plays the role of the additional memory (Latt, 2007; Aksnes and Elster, 2009). The swap algorithm works as follows: in the streaming step, for each node, according to the example of Figure 4.3, the distribution value

e_1 of the current node (lattice node 0 in this example) is exchanged with the value e_3 of the node to which e_1 points (lattice node 1 in this example), e_2 of the current node is exchanged with e_4 of the node to which e_2 points (lattice node 3), e_5 from node 0 with e_7 from node 2, and e_6 from node 0 with e_8 from node 4. After that, in the collision step, all distribution values within a fluid node (the current node or node 0 in the example of Figure 4.3) are inverted, e.g., e_1 with e_3 within node 0, e_2 with e_4 within node 0 and so forth. The same algorithm can be applied to the D3Q19 model analogously.

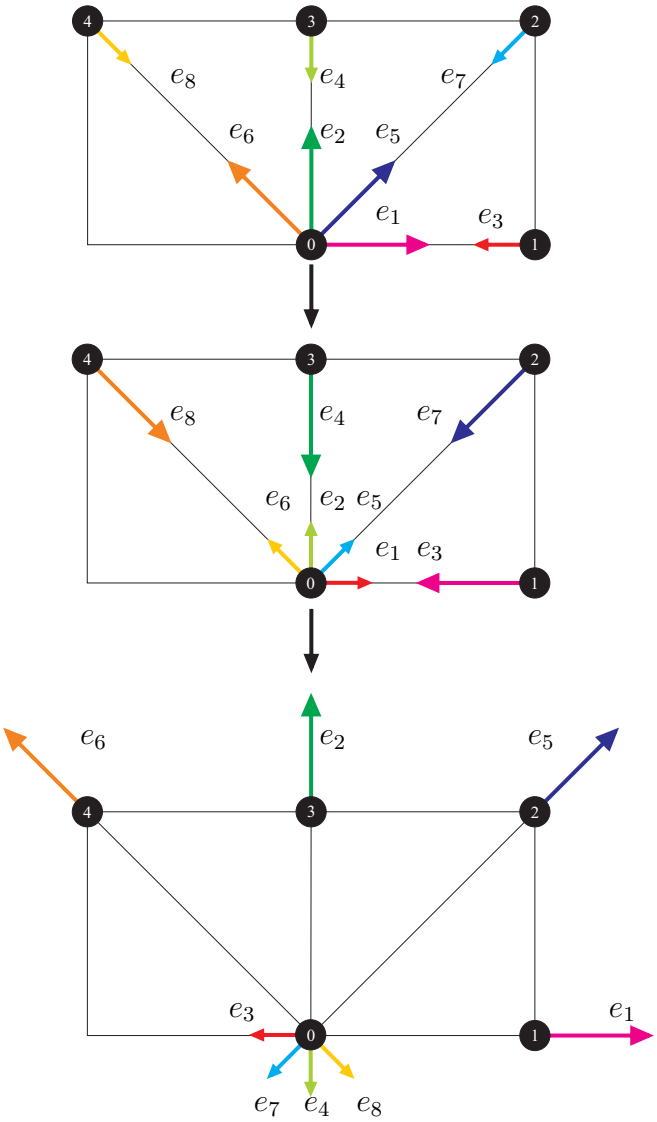


Figure 4.3: Swap algorithm. Top: initial f's configuration. Middle: Exchange of f's with its neighbours. Bottom: inversion of f's inside a node in the collision step.

The swap algorithm advantages are that the bounceback condition for the solid nodes are already treated in it, because in the collision step only the fluid nodes have their distribution values inverted. Furthermore, the swap algorithm performs up to 43% more lattice updates per second, when compared to the two-step algorithm, and requires almost half of the memory required, when compared to the two-lattice algorithm (Mattila *et al.*, 2006).

4.3 Visualization and graphical interface

As stated in Subsection 4.1.7, the visualization was implemented through the 3D texturing technique to visualize the simulation domain, and the shaders to select data to be visualized by the 3D texture. The 3D texturing followed the Subsection 3.4.1, while the shaders and the user interface are detailed on the next subsections. All code for visualization and graphical interface are available at Oliveira and Volpe (2013).

4.3.1 Shaders

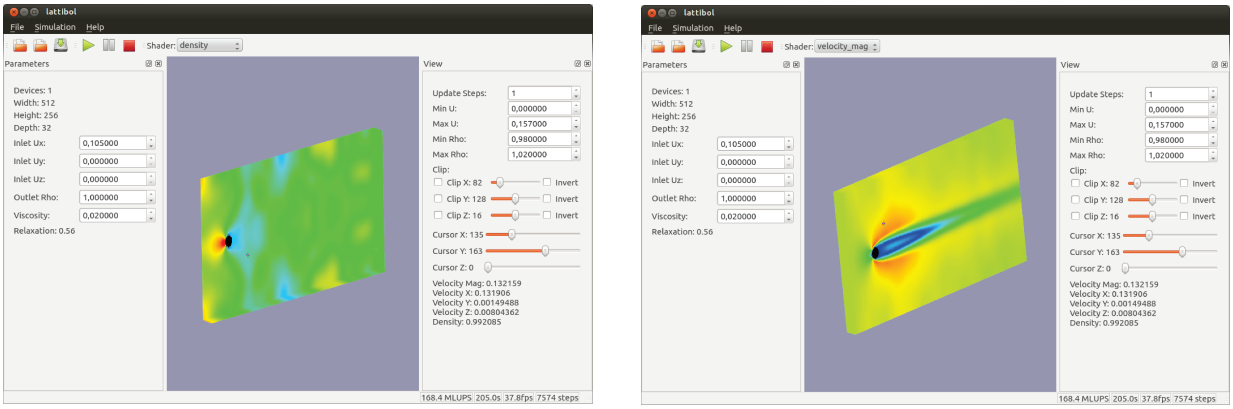
Shaders were used to allow the selection of data to be shown, like velocity magnitude or density, and to change between different color maps to help data visualization. Details of both implementations are given next.

Pixel buffers are used to map the simulation results stored in the CUDA's memory, so that each texel of coordinates (r, g, b, and a) contains velocity values from the three axes (x, y, and z) and density values. Shaders were selected on dependence of the data to be shown, and Qt offers a selection box that was placed on the toolbar icon to choose between these data. Visual effects of different shaders are shown in Figure 4.4, and the following shaders are available:

- *density*: colors are mapped according to density;
- *velocity_mag*: colors are mapped according to velocity magnitude;
- *velocity_x*: colors are mapped according to velocity in x axis;
- *velocity_y*: colors are mapped according to velocity in y axis; and

- *velocity_z*: colors are mapped according to velocity in z axis.

In the visualization part, steps that make interface with the shader programming languague were performed through the Qt support for shaders: the *QGLShaderProgram* and *QGLShader* classes. An object of these classes is capable of specifying the type of shader that is going to be loaded, compiling and linking it, sending the necessary data, initializing GLSL, and using the programs. Another way to perform these steps is through the OpenGL API directly (Shreiner, 2009).



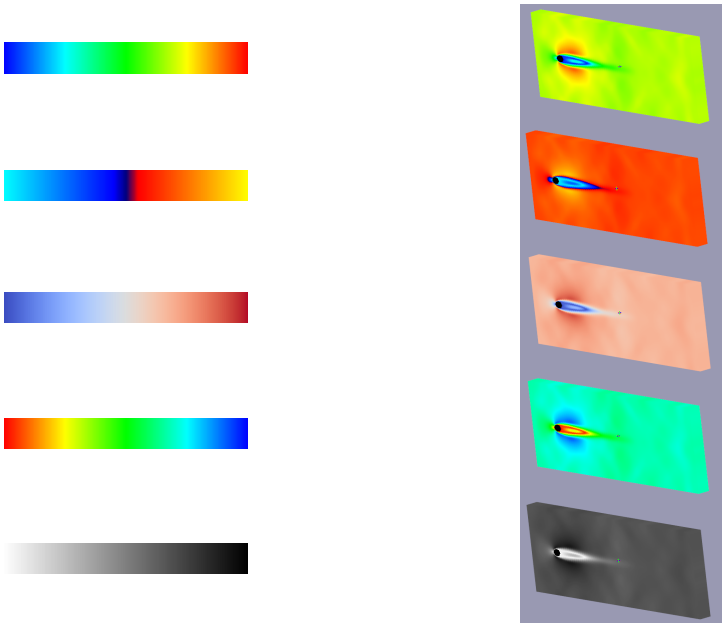
(a) Density shader. (b) Velocity magnitude shader.

Figure 4.4: Effect of shaders.

The vertex shader does the most of the job: basically it only reads the part of texture that is interesting (r for x-axis velocity, g for y-axis velocity, b for z-axis velocity, rgb for magnitude velocity and a for density), and converts every texture value into color values to be shown in screen. The shader accesses the texture data that has been previously created as texture object, the selected color map, and minimum and maximum values of the selected parameter.

Color maps modify the 1D texture to be sent to each shader. A color map is defined by creating a 256 x 256 pixels *png*-format bitmap with all desired colors disposed into columns along x axis, from the minimum value up to the maximum value, as shown in Figure 4.5(a). Five different color maps are available in this simulator: Jet, Cold and Hot, Cool, Gray, HSV, and X-ray. User-defined color maps can be included considering types of vision, such as normal, protanope, deuteranope or tritanope visions (Matlab, 2014). Different color maps and their effects can be seen

in Figure 4.5.



(a) Color maps.

(b) Effect of color maps in the simulation domain.

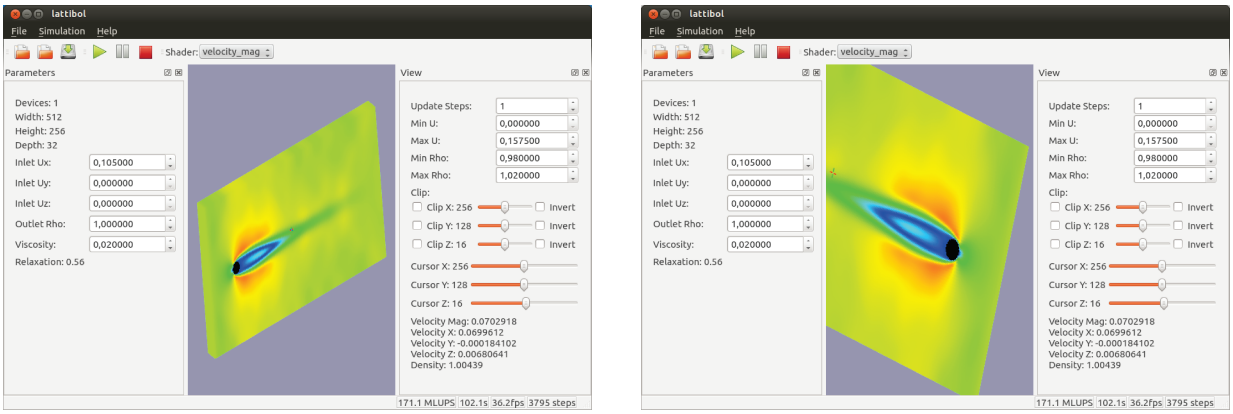
Figure 4.5: Use of shaders for color maps.

4.3.2 User Interface

The simulator interface was implemented using the Qt 5 programming framework, allowing the development of a graphical interface in C++ with OpenGL support, through the *QGLWidget* class. The visualization area contains the simulated domain and stays in the center of the program window. The content of this area was implemented with OpenGL, and the mouse interactions were implemented with Qt 5. The implemented interactions are shown in Figure 4.6 and consist of:

- rotation: left button of mouse rotates the domain (implemented using the Arcball technique (Wikibooks, 2014));
- pan: right button of mouse moves the domain along the screen; and

- zoom: mouse wheel can zoom in or out the domain.



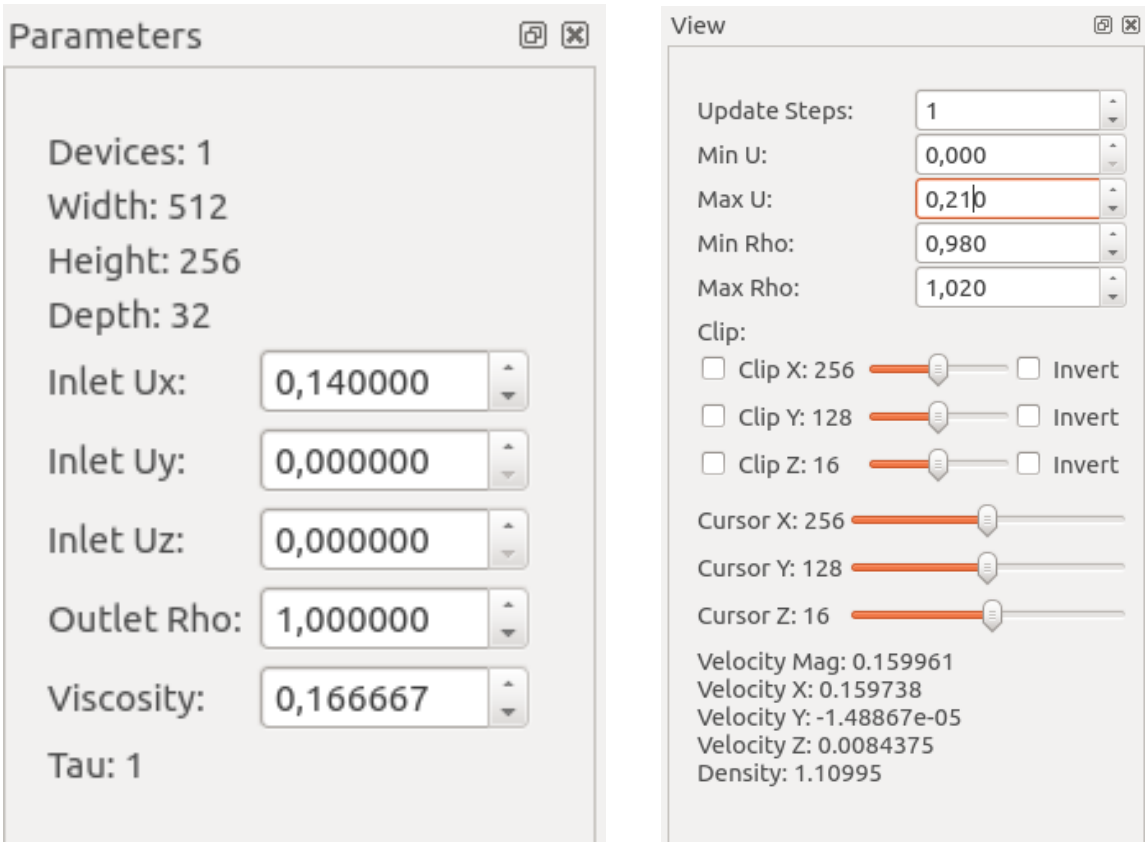
(a) Simulation domain before rotation, pan and zoom.

(b) Simulation domain after rotation, pan and zoom.

Figure 4.6: Interactions in visualization area.

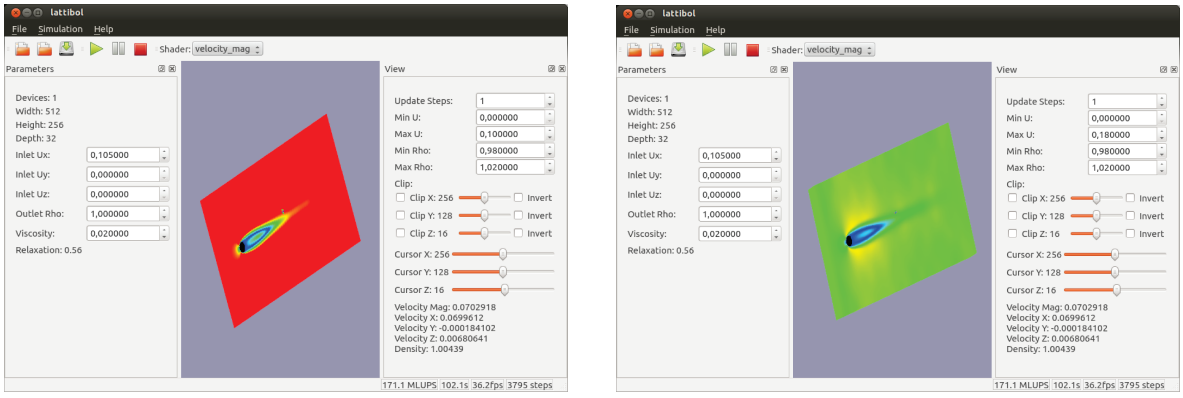
The simulation panel on the left side of Figure 4.6(a) was implemented as a *QDockWidget* dock in Qt 5 and contains information like: amount of devices (GPUs), width, height, and depth of the domain and relaxation time Tau (Figure 4.7(a)). There are also parameters that can be changed interactively: x-, y-, and z-axis inlet velocities, density, and viscosity. Furthermore, the relaxation time depends on viscosity; if viscosity changes, so does the relaxation time. This panel can be seen in Figure 4.7(a).

The visualization panel on the right side of Figure 4.6(a) was also implemented as a *QDockWidget* dock and contains features to change the visualization (Figure 4.7(b)). The first feature is the update steps, which define how many LBM iterations are performed before updating screen, helpful to modify the simulation speed or the frame update rate. The second feature deals with normalization: it defines minimum and maximum values of density and velocity to scale these parameters, which helps to visualize details of a domain region that otherwise would not be possible to see (Figure 4.8). The clipping feature cuts the domain in each axis separately or in combination, being possible to choose the cut position and to invert the cut (Figure 4.9). Finally, the cursor can show simulation data, such as velocity and density of a particular position of the domain, defined by a cursor in the visualization area or by the sliders in the visualization panel (Figure 4.10).



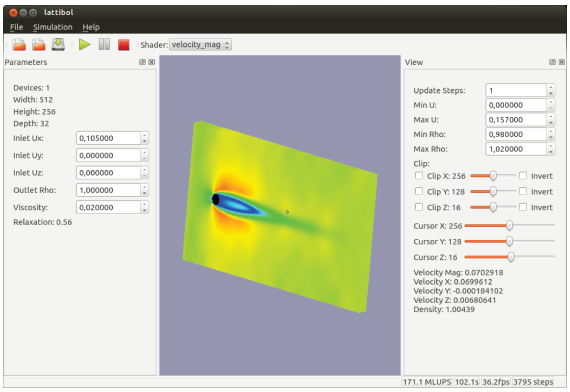
(a) Simulation panel. (b) Visualization panel.

Figure 4.7: Simulator panels.

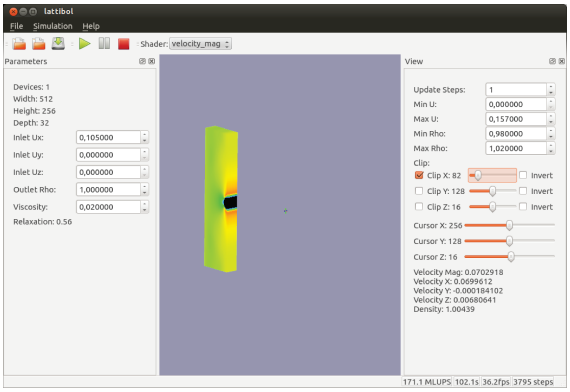


(a) Maximum velocity: 0.1. (b) Maximum velocity: 0.18.

Figure 4.8: Result of different normalization values for velocity.

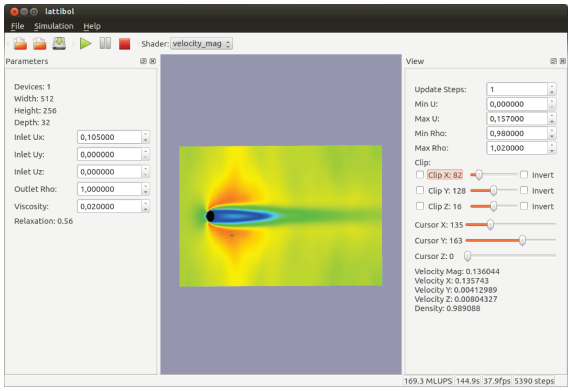


(a) Simulation domain before cut.

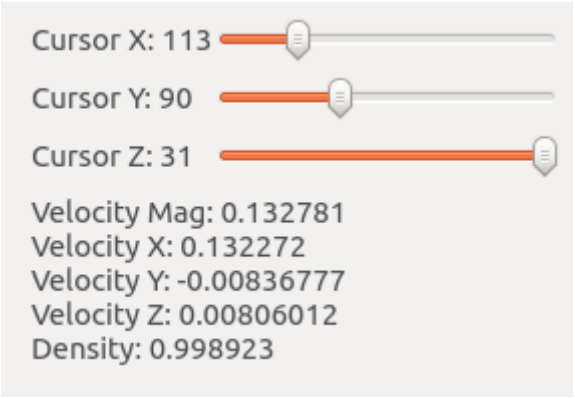


(b) Simulation domain after cut.

Figure 4.9: Result of clipping feature.



(a) Cursor placed in the orange region of the domain.



(b) Cursor position and its respective information.

Figure 4.10: Use of cursor to extract information from simulation.

There are three context menus, named *File*, *Simulation*, and *Help*. In the *File* menu, there are options to open a simulation configuration (input data), to open a color map, to save a VTK file (with the state of a simulation), and to exit. In the *Simulation* menu, the options are run, pause or stop the simulation, and the *Help* menu contains information about the simulator version. These menus can be seen in detail in Figure 4.11.

The icon toolbar contains nearly the same information of the context menus, with icons to open a simulation configuration, to open a color map, to save a VTK file, to run, pause, and stop

the simulation, and to choose between different visualization data (density, velocity magnitude, or velocity in any axis). Both the context menu and the icon toolbar can be seen in Figure 4.11.

Figure 4.11: Context menu and icon toolbar.

The status toolbar on the bottom of Figure 4.10(a) contains information about the simulation: status tips on the left side and performance parameters on the right side. The performance parameters are the amount of millions of lattice updates per second (MLUPS), simulation duration (s), the amount of frames per second (fps), and the amount of performed simulation steps. This information was added through the *QLabel* widget and is present in Figure 4.12.

Figure 4.12: Status bar.

4.4 Previous versions of the simulator

The final version of the simulator presented in this master thesis was developed step-by-step, in a way that the following versions were implemented: serial CPU code, parallel CPU code, and parallel GPU code. The interactive visualization was added later and only to the GPU version, so that features such as the CUDA-OpenGL interoperability could be adopted. Besides helping in the final version evolution, these versions could be used to assess performance of the interactive simulator, by comparing the MLUPS performance measure and run-times.

The parallel CPU code was implemented with MPI and is explained next. Data-parallelism was applied to serial code, dividing the domain into smaller column blocks, as in Figure 4.13. Block-cyclic partitioning was used with block size equal to the number of processes.

Steps of streaming, collision, and boundary conditions were performed just like the serial algorithm, with each process calculating its own subdomain, but after the streaming it is necessary

to send the left and right borders of each subdomain to the correct process in order to get the correct result, as Figure 4.14 shows. There, each subdomain left border should go to the subdomain left border on the right, while each subdomain right border should go to the subdomain right border on the left, except for the first subdomain right border, that should go to the last subdomain right border, and the last subdomain left border, that should go to the first subdomain left border (periodic correction). Collision and boundary conditions do not need any correction because lattice nodes do not access neighbour nodes, so calculations only depend on the node itself. This version used the swap algorithm too, and that is why after the streaming step the distribution values from the subdomain borders go to subdomains in the opposite direction in relation to their own direction.

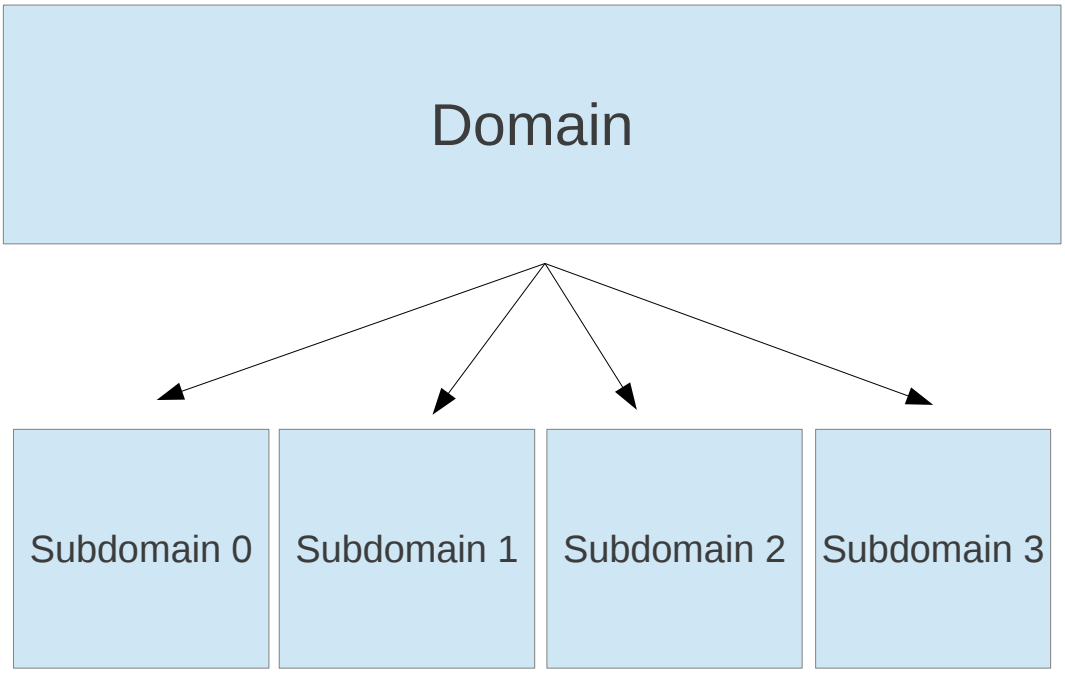


Figure 4.13: Domain division among processes.

MPI functions were used to broadcast initial data to all processes, divide work for them, measure time and, on each iteration, exchange borders to the correct places. Output data is a set of VTK extension files, saved by each process individually and so also parallelized. The simulation result can be watched in a VTK viewer program, such as ParaView.

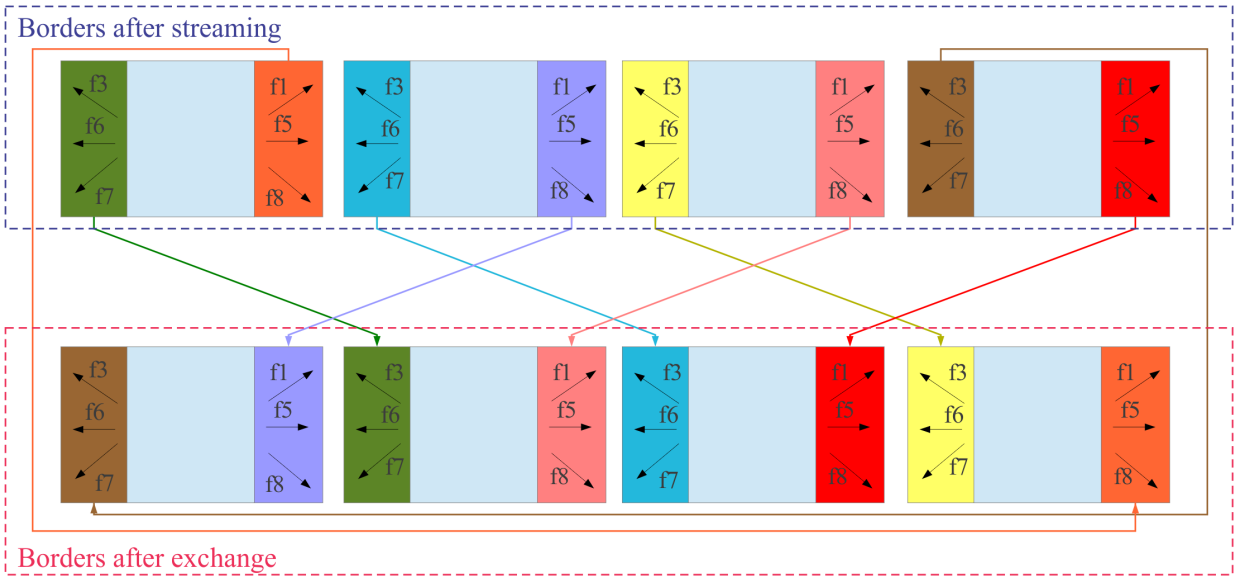


Figure 4.14: Exchange of borders process. Each subdomain left border goes to the subdomain left border on the right, and each subdomain right border goes to the subdomain right border on the left. The first subdomain right border goes to the last subdomain right border, and the last subdomain left border goes to the first subdomain left border (periodic correction). It is the swap algorithm that makes the distribution values from the subdomain borders go to subdomains in the opposite direction in relation to their own direction.

5 Results

This chapter presents results in terms of validation cases and performance assessment of the interactive simulator. The validation tests were divided into SCSP and SCMP cases, and the performance evaluation was performed based on a 2D simulation of a single phase flow for increasing domain sizes, comparing serial, MPI and CUDA versions. After the SCSP validation tests, an application of a microfluidic device called oscillator was shown. Visualization results, such as the graphical interface and the visualization of the domain, assisted the validation and performance analyses, making them easier and faster.

The software code is available at Oliveira (2015) and was executed in a computer equipped with the Intel® Core i7 CPU 950 at 3.07 GHz $\times$ 8 processors (4 physical cores and 8 threads), 6 GB of RAM memory, and NVIDIA® GeForce® GTX 560 Ti graphics card with 384 cores and 1 GB of GDDR5 RAM memory, on Ubuntu 12.04 long-term support (LTS) Linux-based operating system. The Intel processor theoretical throughput is 49 GFLOPS and NVIDIA® graphics card theoretical throughput is 1.5 TFLOPS. Results were measured in fps, in MLUPS, and in seconds. These values can be seen in the interactive simulator in the lower right corner, on the status bar (Figure 4.12).

5.1 Single component, single phase flow validation tests

SCSP cases were validated for 2D and 3D models for the following cases: Poiseuille flow driven by gravity (only 2D), Poiseuille flow driven by velocity/pressure boundaries (entry length effect), flows past a cylinder and unstable flow at high Reynolds numbers. SCSP models simulates the behaviour of a single gas phase or a single liquid phase. Simulator results for these cases were compared to analytical solutions, experimental results, or simulated results, showing the simulator is appropriate for fluid flow simulations.

5.1.1 Poiseuille flow driven by gravity

Poiseuille flow driven by gravity in a slit is one of the simplest LBM simulations, and requires bounceback condition for the walls, periodic boundaries for the open ends of the domain, and

gravitational force applied as external force in the entire domain. The slit is actually infinite and there is no entry length effect: it actually happens in the begin of simulation but at steady state the flow is completely developed. It is worth noting that channels should be at least 5 lattice units wide to show good results, and that lattice velocities should not be higher than aproximately $0.1\ lu/lt$ (Sukop and Thorne Jr., 2005). The velocity condition is due to the low Mach number approximation for the LBM, meaning the LBM only works for the low Mach number hydrodynamics, because there is a small velocity expansion implicitly used in the derivation of the Navier-Stokes equation when one starts with the Lattice Boltzmann equation (He and Luo, 1997).

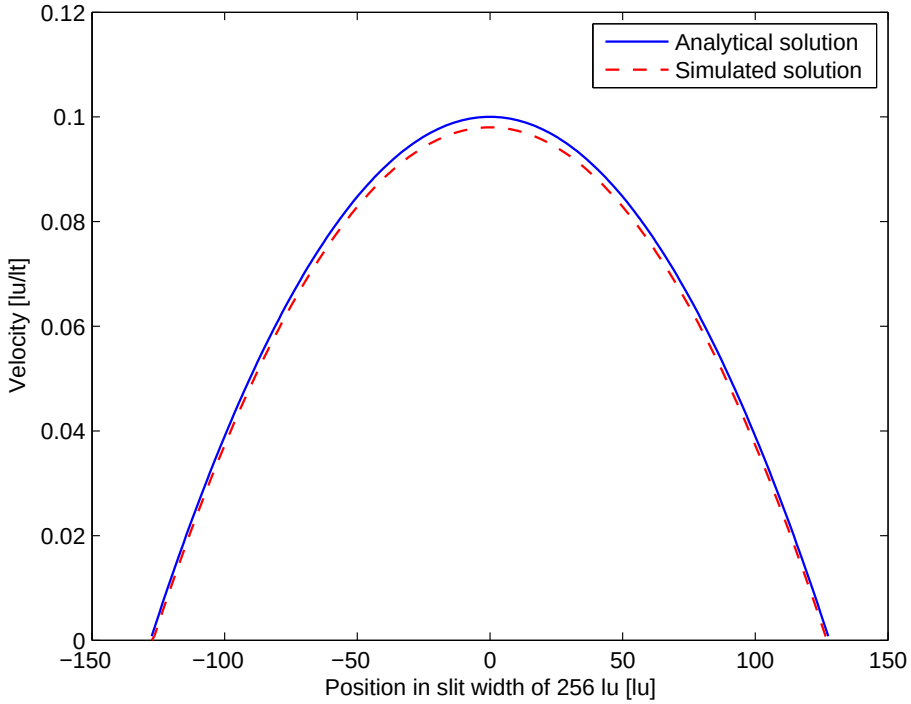


Figure 5.1: Analytical and simulation result comparison for the Poiseuille flow.

In this 2D simulation, a domain of 1024 x $256\ lu^2$ was chosen, and $\tau = 1$ to have stable results using the bounceback condition, leading to a kinematic viscosity of $1/6\ lu^2/lt$. To keep the maximum velocity approximately equal to $0.1\ lu/lt$, and using Equation 2.37, the gravitational acceleration needed was $2.035 \times 10^{-6}\ lu/lt^2$. The fluid density was chosen as $1\ mu/lu^2$, leading to a Reynolds number of 102.4 (using Equation 2.38). This flow is laminar, since turbulent flow happens at $Re > 1000$ in a slit and $Re > 2000$ for a pipe. Figure 5.1 was plotted from the output data generated by the simulator with help of the software MATLAB®, and shows analytical and

simulated results for this flow. The analytical result comes from the analytical solution shown in Section 2.8. From this figure, it is possible to see the good agreement between analytical and simulation results. The maximum error was approximately 2%.

5.1.2 Poiseuille flow driven by velocity/pressure boundaries

Velocity and pressure boundaries are useful to simulate entry length effects. When a fluid enters a pipe, it takes some distance from the beginning of the pipe until the flow is fully developed and shows the Poiseuille flow. In some cases, this distance can be long, which justifies the importance of this simulation.

In this simulation, constant velocity boundary condition was applied in the inlet, and constant pressure boundary was applied in the outlet of the domain. The inlet constant velocity was $0.01\ lu/lt$ and the outlet constant density (the relation between density and pressure is explained in Section 2.4.4) was $1\ mu/lu^2$ (2D) or $1\ mu/lu^3$ (3D). The domain was 1024 x $256\ lu^2$, and the viscosity was $1/6\ lu^2/lt$, giving a Re of approximately 10. Figure 5.3(a) shows the simulated entry length effect from Sukop and Thorne Jr. (2005), and Figure 5.3(b) shows the result from the interactive simulator. In Figure 5.3(a), a normalized width of $0.2\ lu$ was used, while in Figure 5.3(b) the width had $256\ lu$, and the velocity profiles of the interactive simulator were plotted at distances equivalent to the distances of Figure 5.3(a). Figure 5.3(b) and Figure 5.4(b) were plotted with help of the software ParaView, using the VTK files generated by the simulator. One can note the matching results between both figures, validating the simulator for LBM with 2D velocity and pressure boundaries.

The 3D case was flow in the annulus between two pipes. The same conditions for the slit were considered in this simulation. The domain can be seen in Figure 5.2 (plotted with ParaView) and measures 128 x 128 x $32\ lu^3$. In this figure, blue color represents solid nodes, and the flow is represented by a Jet color map. Figure 5.4(a) shows the analytical entry length effect from Nouar *et al.* (1995), and Figure 5.4(b) shows the result from the interactive simulator. In Figure 5.4(a), a normalized width of $0.2\ lu$ was used, while in Figure 5.4(b) the width had $24\ lu$, and the velocity profiles of the interactive simulator were plotted at distances equivalent to the distances of Figure 5.4(a). The results of both figures match, validating the simulator for LBM with 3D velocity and pressure boundaries.

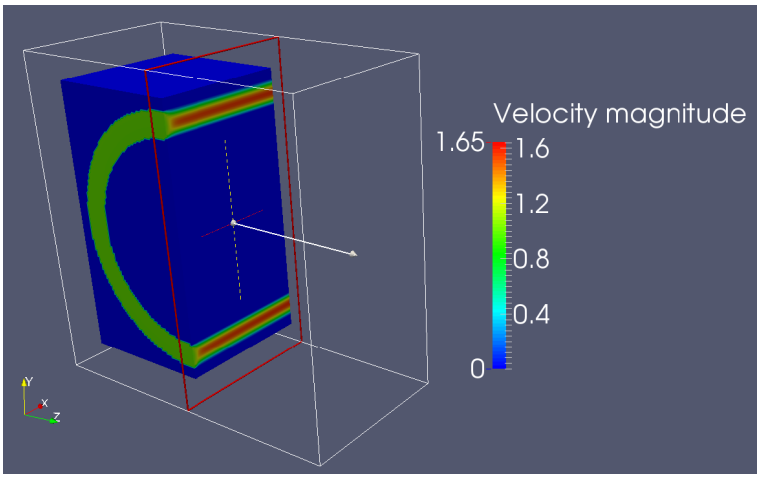
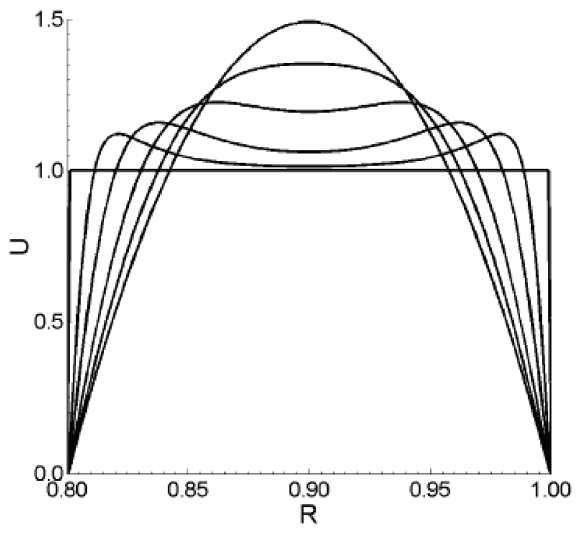
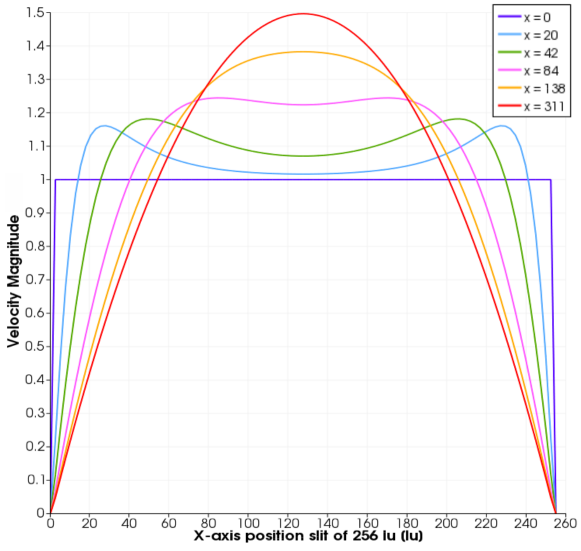


Figure 5.2: Flow in the annulus between two pipes. Blue color represents solid nodes, and the flow is represented by a Jet color map..

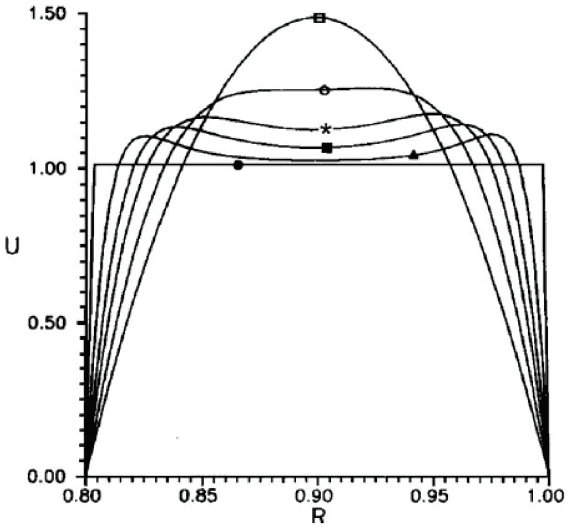


(a) Literature normalized velocity profiles at dimensionless distances from inlet: 0, 0.016, 0.033, 0.066, 0.1083, and 0.245 scaled relative to a slit width of 0.2 (Sukop and Thorne Jr., 2005).

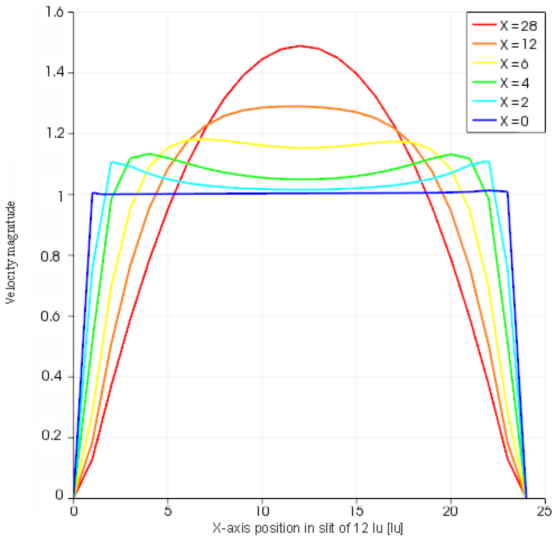


(b) Simulated velocity profiles related to an inlet velocity of $0.01\,lu/lt$ at dimensionless distances from inlet: 0, 20, 42, 84, 138, and 311 scaled relative to a slit width of $256\,lu$.

Figure 5.3: Simulated entry length effect in a slit.



(a) Analytical normalized velocity profiles at dimensionless distances from inlet: 0 (●), 0.016 (▲), 0.033 (■), 0.066 ($*$), 0.1083 (○), and 0.245 (□) scaled relative to a slit width of 0.2 (Nouar *et al.*, 1995).



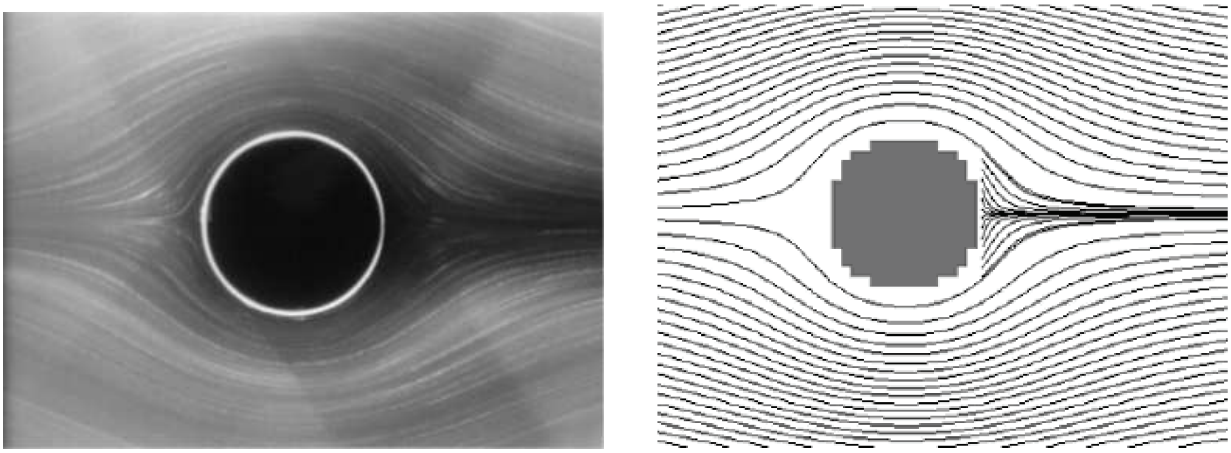
(b) Simulated velocity profiles related to an inlet velocity of $0.01\,lu/lt$ at dimensionless distances from inlet: 0, 2, 4, 8, 12, and 28 scaled relative to a slit width of $24\ lu$.

Figure 5.4: Entry length effect in annular space between two pipes.

5.1.3 Stokes flow past a cylinder

Flows past an obstacle have been a problem of particular interest to fluid dynamics, since it is possible to analyse flows past airfoils in aircraft research, and other important immersed bodies. Simple flows can be analysed with a cylinder or sphere as an obstacle.

This simulation had a Re of 0.16, resulting in a creeping or Stokes flow. Constant velocity boundary was applied in the inlet, and constant pressure boundary was applied in the outlet of the domain. The domain was 1024 x 256 lu^2 for the 2D case and 512 x 256 x 32 lu^3 for the 3D case, the inlet velocity was 0.001333 lu/lt, the outlet density was 1 mu/lu^2 for 2D and 1 mu/lu^3 for 3D, and the viscosity was $1/6$ lu^2/lt. Figure 5.5(a) shows the photograph of a creeping flow (Taneda, 1979), while Figure 5.5(b) shows current lines of a simulated creeping flow (Sukop and Thorne Jr., 2005).

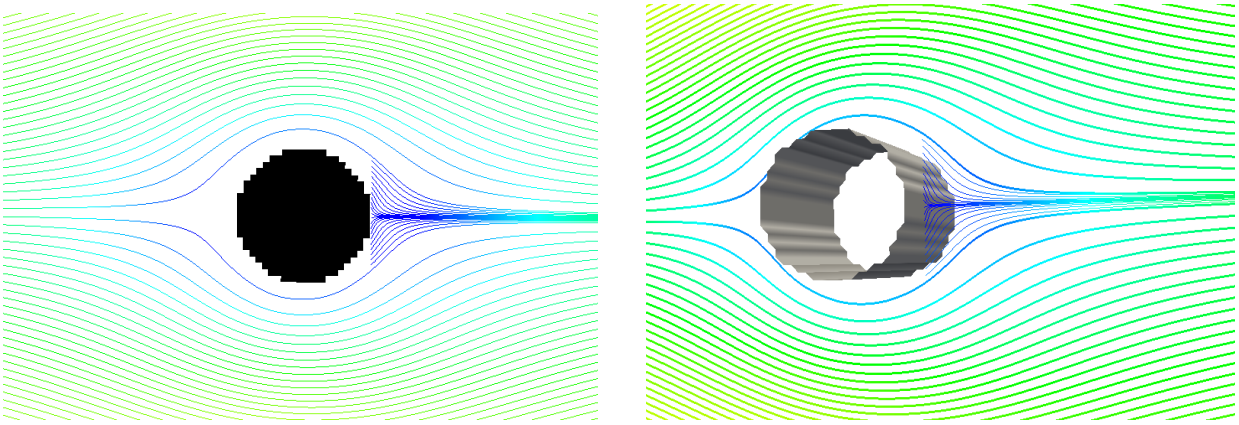


(a) Photograph of Stokes flow (Taneda, 1979).

(b) Current lines of simulated Stokes flow (Sukop and Thorne Jr., 2005).

Figure 5.5: Stokes flows past a cylinder ($Re = 0.16$).

Figure 5.6 shows the simulated Stokes flow from the interactive simulator for the 2D and 3D cases (plotted with ParaView). It is possible to note that both results are qualitatively similar to the photograph and to the simulated result from Figure 5.5. In both figures, colors indicate the velocity magnitude, according to the Jet color map.



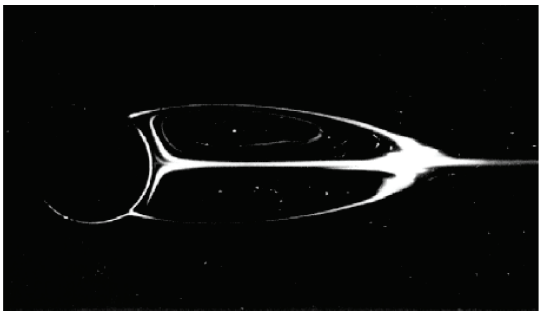
(a) 2D case.

(b) 3D case.

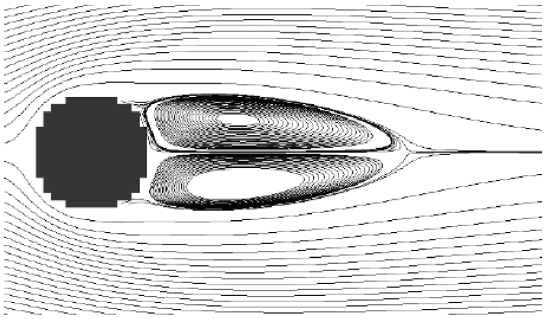
Figure 5.6: Current lines of simulated Stokes flow past a cylinder ($Re = 0.16$).

5.1.4 Flow separation past a cylinder

Separation occurs when Re is increased, and eddies are formed behind the cylinder. This simulation had a Re of 41, resulting in flow separation with eddy generation. Constant velocity boundary was applied in the inlet, and constant pressure boundary was applied in the outlet of the domain. The domain was 1024 x 256 lu^2 for the 2D case and 512 x 256 x 32 lu^3 for the 3D case, the inlet velocity was 0.041 lu/lt, the outlet density was 1 mu/lu^2 for 2D and 1 mu/lu^3 for 3D, and the viscosity was 0.02 lu^2/lt. Figure 5.7(a) shows the photograph of a flow separation (Taneda, 1956), while Figure 5.7(b) shows current lines of a simulated flow separation (Sukop and Thorne Jr., 2005). Figure 5.8 shows the simulated flow separation from the interactive simulator for the 2D and 3D cases (plotted with ParaView), with colors indicating the velocity magnitude, according to the Jet color map. It is possible to note that both results are qualitatively similar to the photograph and to the simulated results from Figure 5.7.



(a) Photograph of flow separation (Taneda, 1956).

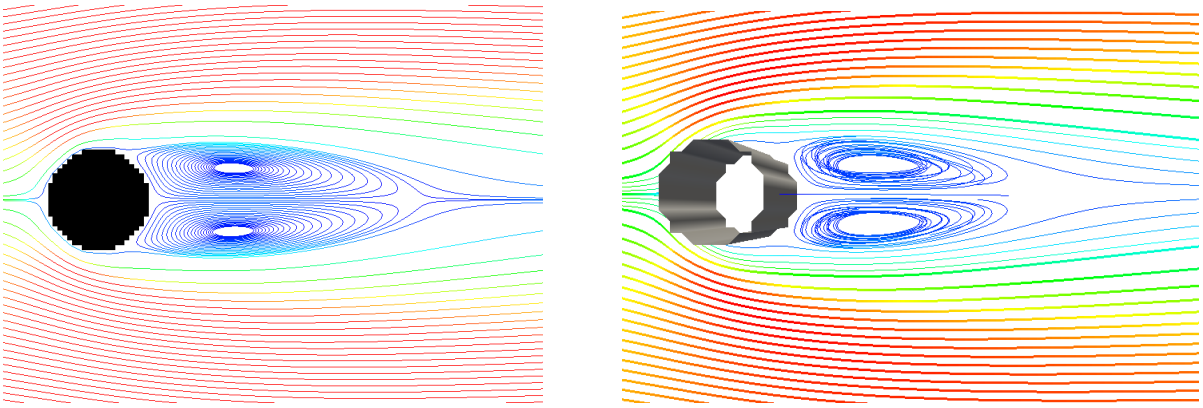


(b) Current lines of simulated flow separation (Sukop and Thorne Jr., 2005).

Figure 5.7: Flow separation past a cylinder ($Re = 41$).

5.1.5 Unsteady flow past a cylinder

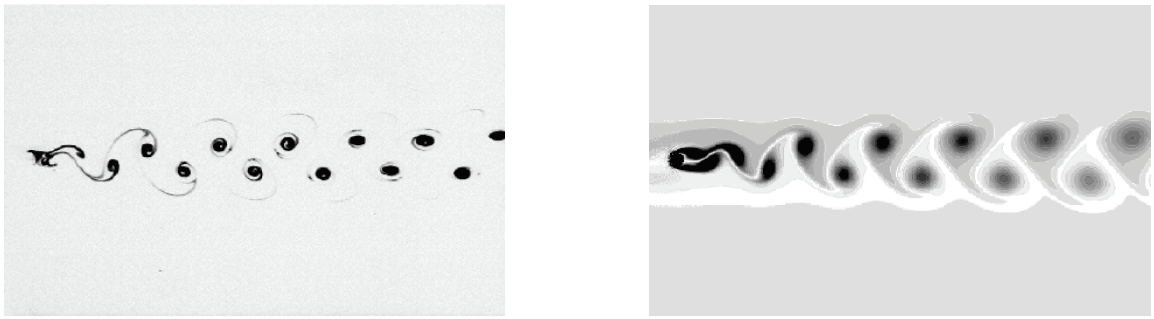
Higher Re leads to unsteady flows with vortex shedding that moves downstream. This flow is known as von Kármán street and is also of great interest in fluid dynamics and LBM research (Sukop and Thorne Jr., 2005; Succi, 2001). In this simulation, Re was 105, resulting in flow with vortex shedding. Constant velocity boundary was applied in the inlet, and constant pressure boundary was applied in the outlet of the domain. The domain was 1024 x 256 lu^2 for the 2D case and 512



(a) 2D case. (b) 3D case.

Figure 5.8: Current lines of simulated flow separation past a cylinder ($Re = 41$).

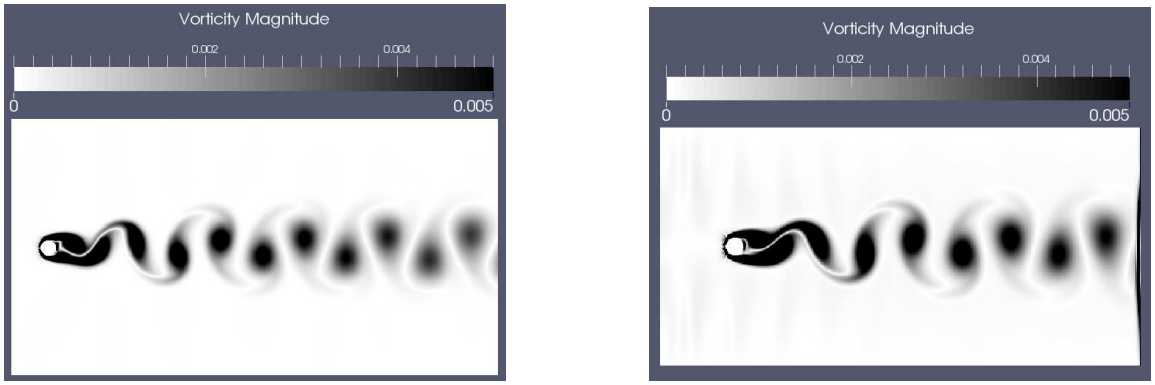
x 256 x 32 lu^3 for the 3D case, the inlet velocity was $0.105\ lu/lt$, the outlet density was $1\ mu/lu^2$ for 2D and $1\ mu/lu^3$ for 3D, and the viscosity was $0.02\ lu^2/lt$. Figure 5.9(a) shows the photograph of a flow with vortex shedding and von Kármán street at $Re = 105$ (Taneda, 1956), while Figure 5.9(b) shows the vorticity magnitude of a simulated flow at same Re (Sukop and Thorne Jr., 2005). Figure 5.10 shows the simulated vorticity magnitude from the interactive simulator for the 2D and 3D cases (plotted with ParaView), with colors indicating the velocity magnitude, according to the X-ray color map. Both results are qualitatively similar to the photograph and to the simulated results from Figure 5.9.



(a) Photograph of unstable flow (Taneda, 1956).

(b) Vorticity magnitude of simulated unstable flow (Sukop and Thorne Jr., 2005).

Figure 5.9: Vortex shedding and von Kármán street in unstable flow past a cylinder ($Re = 105$).



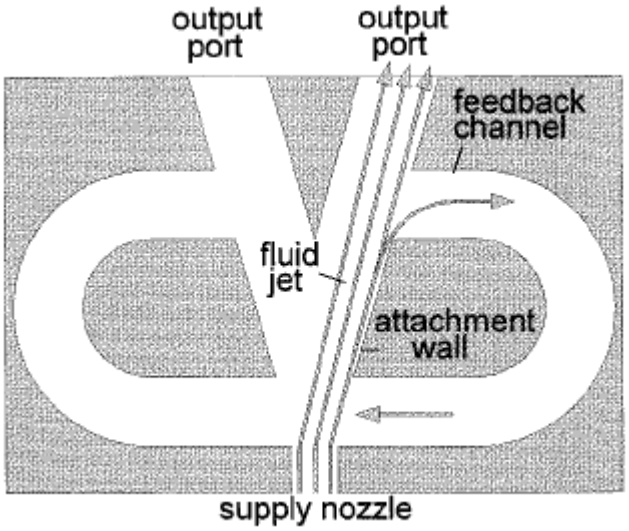
(a) 2D case. (b) 3D case.

Figure 5.10: Vorticity magnitude of simulated flow with vortex shedding and von Kármán street ($Re = 105$) at X-ray color map.

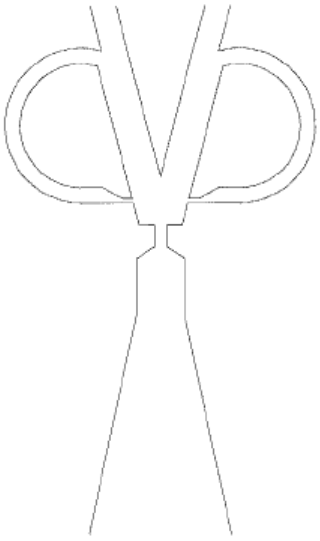
5.2 A Single component, single phase flow application: the fluidic oscillator

After SCSP flows validation, it is possible to show a LOC application called fluidic oscillator (Gebhard *et al.*, 1996). Figure 5.11(a) shows the operation of such a device. A fluid jet enters the oscillator and tilts towards the attachment wall due to small fluctuations, creating a low-pressure flow that attaches the fluid to the wall and makes it exit through one of the outlets. The feedback channel allows some portion of the jet to come back closer to the supply nozzle, which makes the jet to be switched to the other attachment wall. This way, a periodic output can be obtained, with the fluid exiting one output port at each time.

The fluidic oscillator was drawn according to Gebhard *et al.* (1996) work and can be seen in Figure 5.11(b). In this simulation, a 2D domain of 1664 x 1024 was used with an inlet velocity of $0.02\,lu/lt$, and density of $1\,mu/lu^2$. Figure 5.12 and Figure 5.13 show the result of the fluidic oscillator during its operation, recalling that the density field is at the top of the simulator interface, and the velocity field is at the bottom. In Figure 5.12, it is possible to see the pressure difference between each feedback channel, with the higher pressure in the channel beneath. The fluid in the attachment wall beneath also has the higher velocity, and exits the oscillator while the upper outlet has almost no fluid exiting it. The opposite happens in Figure 5.13, which was taken $14s$ after Figure 5.12: the upper attachment wall has higher pressure and the fluid exits the oscillator through it, creating an oscillatory device.



(a) Operation of a fluidic oscillator (Gebhard *et al.*, 1996).



(b) Fluidic oscillator design (Gebhard *et al.*, 1996).

Figure 5.11: Fluidic oscillator operation and design.

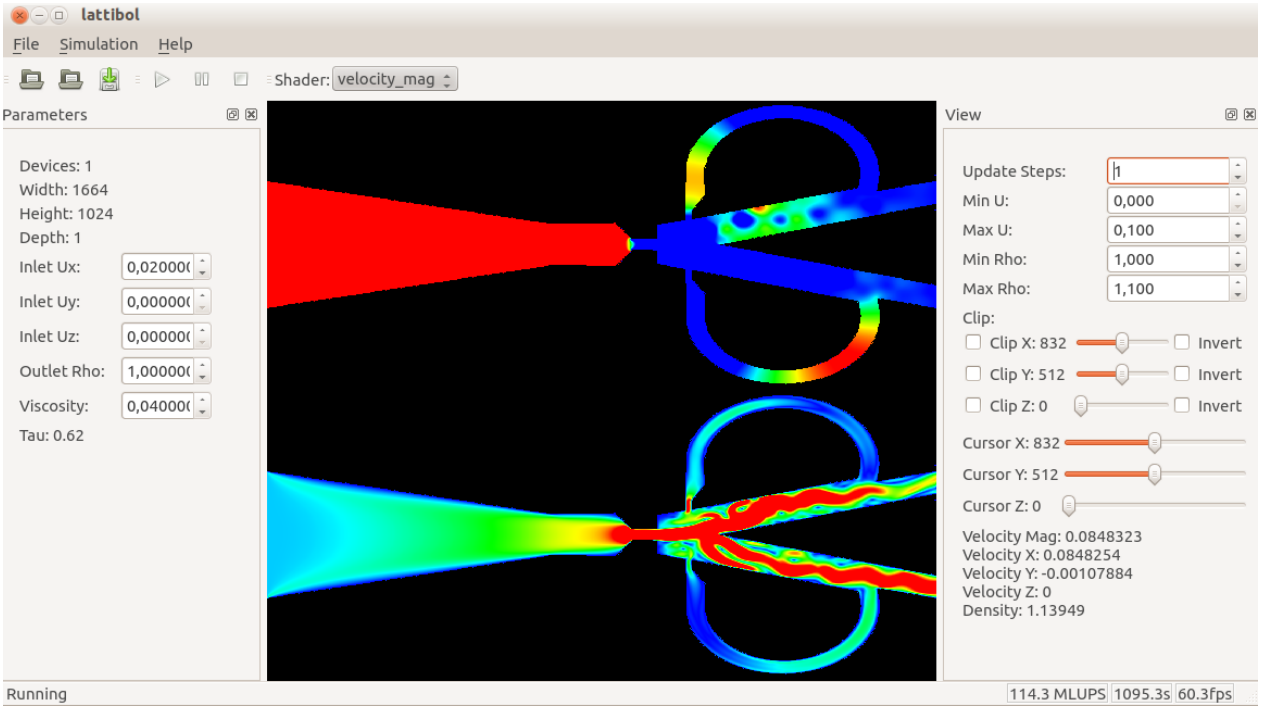


Figure 5.12: Result of the fluidic oscillator simulation in the interactive simulator: fluid exiting through the lower outlet (lower feedback channel with higher pressure).

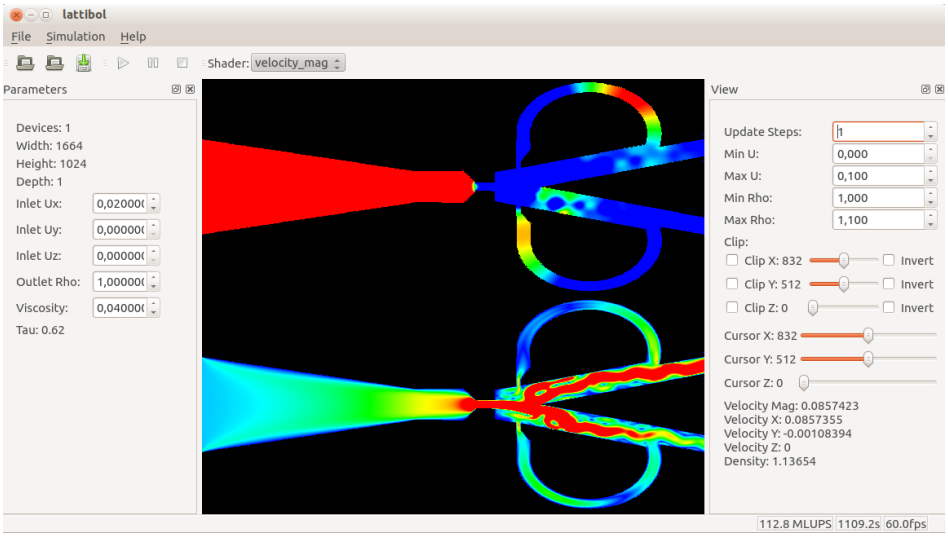


Figure 5.13: Result of the fluidic oscillator simulation in the interactive simulator: fluid exiting through the upper outlet (upper feedback channel with higher pressure).

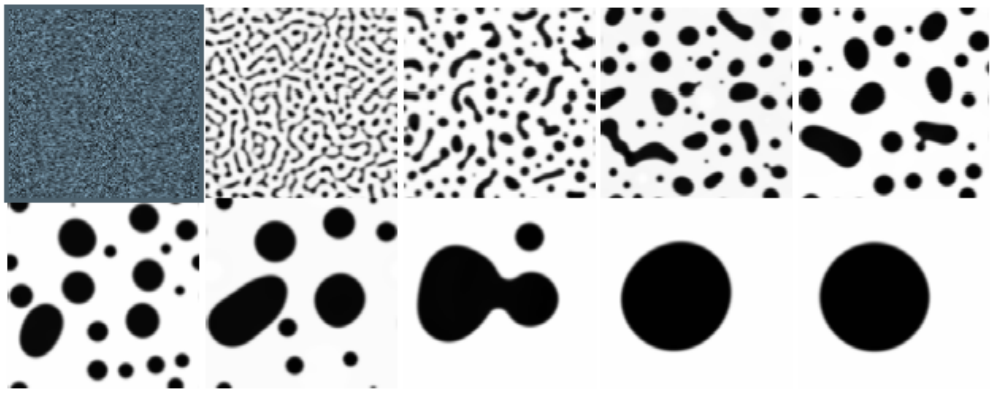
5.3 Single component, multiphase flow validation tests

SCMP flow cases were validated for 2D models for the following cases: phase separation and interface minimization, surface tension estimation, flat interfaces, heterogeneous cavitation, and contact angles with solid surfaces. SCMP models can simulate the behaviour of two phases of the same component, such as vapor and liquid phases. Simulator results for these cases were compared to literature simulated results, showing the simulator is appropriate for SCMP fluid flow simulations.

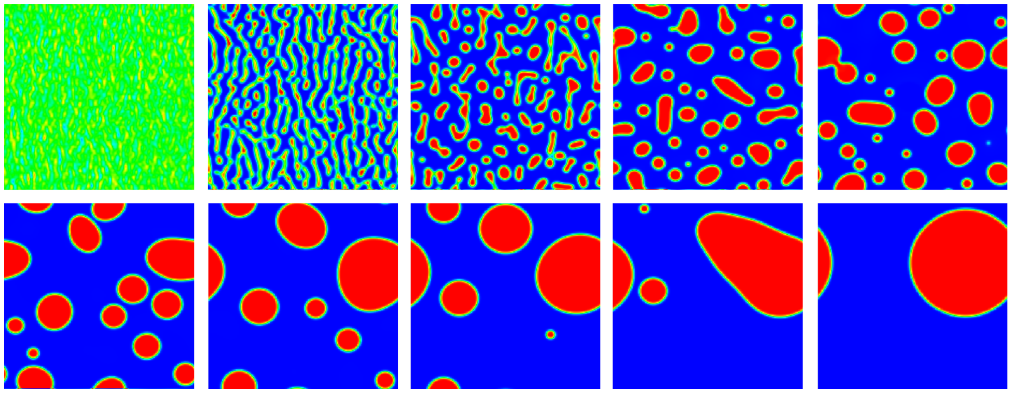
5.3.1 Phase separation and interface minimization

At steady state, phase separation becomes a single liquid droplet in vapor atmosphere, or a single vapor bubble in liquid medium, depending on the total mass of the domain, which in turn depends on the initial density. This happens because of the free energy minimization that rearranges the domain into the minimum surface area volume (a circle in 2D case). Condensation and evaporation are also simulated: bubbles or drops may grow, decrease, appear, or disappear (Sukop and Thorne Jr., 2005).

Phase separation of liquid and vapor can be simulated without solid wall implementation. This simulation had a density of 200 mu/lu^2 with a initial random variation, $\psi_0 = 4$ and $\rho_0 = 200$ for the interaction potential, $\tau = 1$, and $G = -120$ to calculate the interparticle force, because this value is higher than the critical value and leads to phase separation. Figure 5.14 shows time series of phase separation simulation with the same conditions simulated here for a 200 x 200 domain in literature and a 192 x 200 domain for the interactive simulator. In Figure 5.14(a), black regions indicate liquid phase and white regions are vapor phase, while in Figure 5.14(b) blue regions indicate vapor phase, red regions are liquid phase, and green regions are liquid and vapor mixed, according to the Jet color map explained in Section 4.3.1. In Figure 5.15, the complete interface of the simulator at one time step of simulation (after 800 iterations) is shown, with the velocity field under the density field in the visualization area.



(a) Literature result (Sukop and Thorne Jr., 2005).



(b) Interactive simulator result.

Figure 5.14: Time series of liquid-vapor phase separation after 0, 100, 200, 400, 800, 1600, 3200, 6400, 12800, and 25600 time steps.

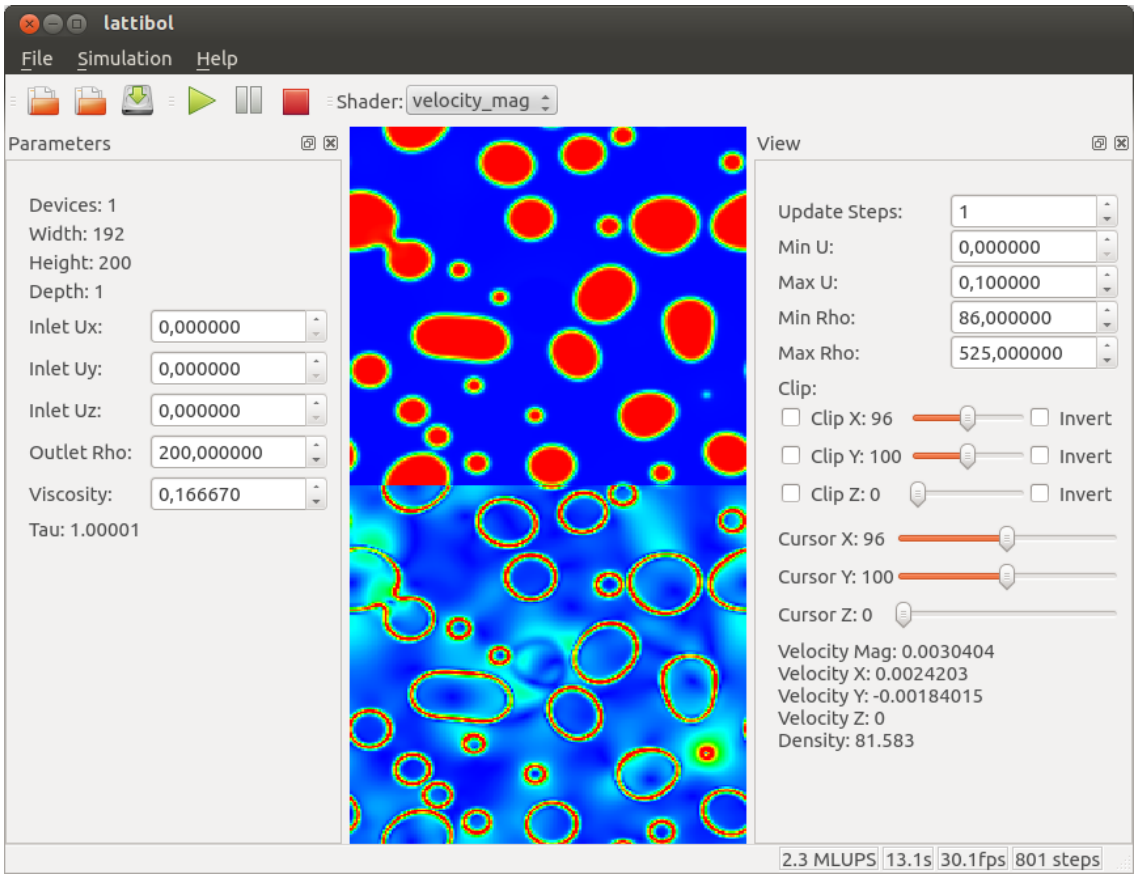


Figure 5.15: Interactive simulator after 800 time steps (the top area shows the density field and the bottom area shows the velocity field).

5.3.2 Surface tension estimation

Phase separation simulation allows the surface tension estimation by using some sizes of drops and bubbles, which means changing the initial density. The surface tension calculation is done with the drop or bubble radii and inside and outside densities, which are converted into pressures with the EOS, and becomes a pressure difference. Then it is possible to plot $1/radius$ versus ΔP and its slope is the surface tension, according to Section 2.11.

The same parameters of Section 5.3.1 were used with different initial densities in order to plot this graph. Figure 5.16 shows the slope of Sukop and Thorne Jr. (2005) with a surface tension of $14.332\,lu\,mu/lt^2$, while Figure 5.17 shows the slope of the interactive simulator with a surface tension of $13.178\,lu\,mu/lt^2$.

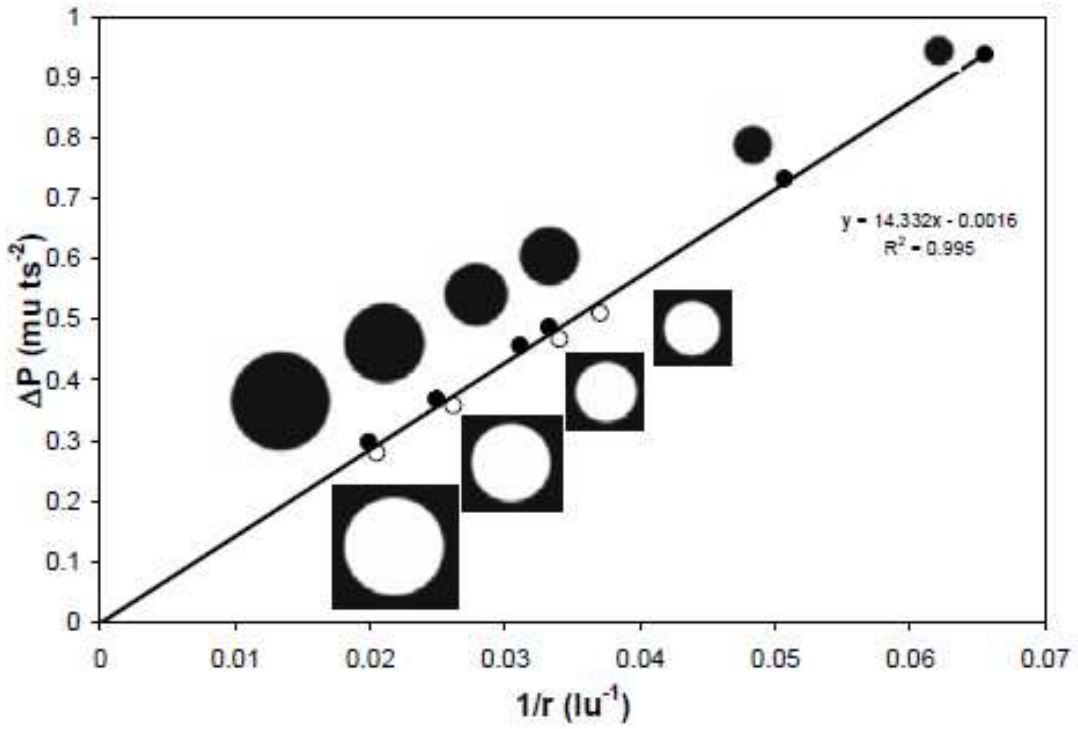


Figure 5.16: Literature surface tension estimation (Sukop and Thorne Jr., 2005).

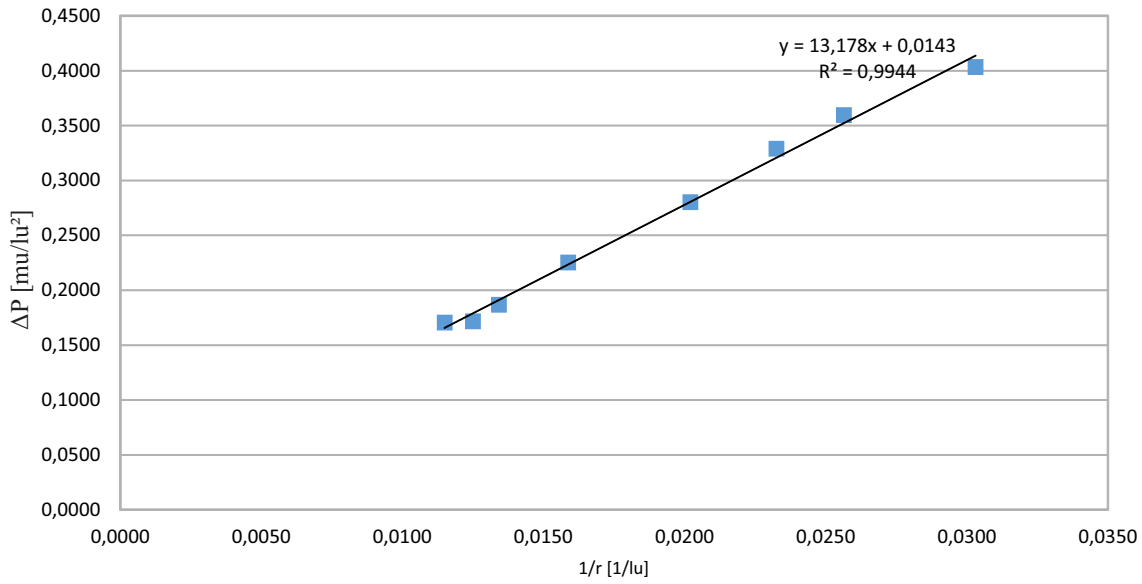


Figure 5.17: Surface tension estimation from the interactive simulator.

5.3.3 Flat interface

Flat interfaces simulation allows the measure of saturation vapor and liquid pressures. In this simulation, fully periodic domain and an initial condition of density $500\,mu/lu^2$ in half of the domain and density $80\,mu/lu^2$ in the other half were applied. The other parameters were the same from Section 5.3.1. Figure 5.18 shows the vapor phase density of $85.7\,mu/lu^2$ at saturation vapor pressure, while Figure 5.19 shows the liquid phase density of $524.39\,mu/lu^2$. These values completely agree with Sukop and Thorne Jr. (2005).

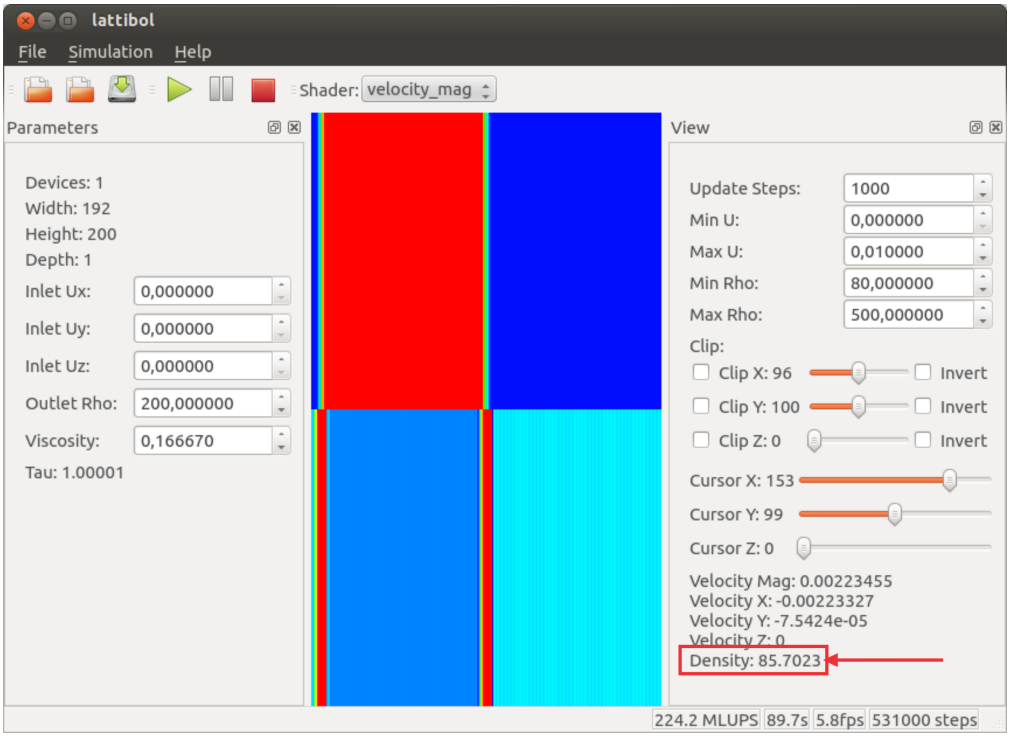


Figure 5.18: Vapor phase density in a flat interface simulation.

5.3.4 Heterogeneous cavitation

Cavitation is a sudden transition from liquid to vapor. Heterogeneous cavitation happens when there are preexisting bubbles or other disruptions in the liquid structure (Sukop and Thorne Jr., 2005).

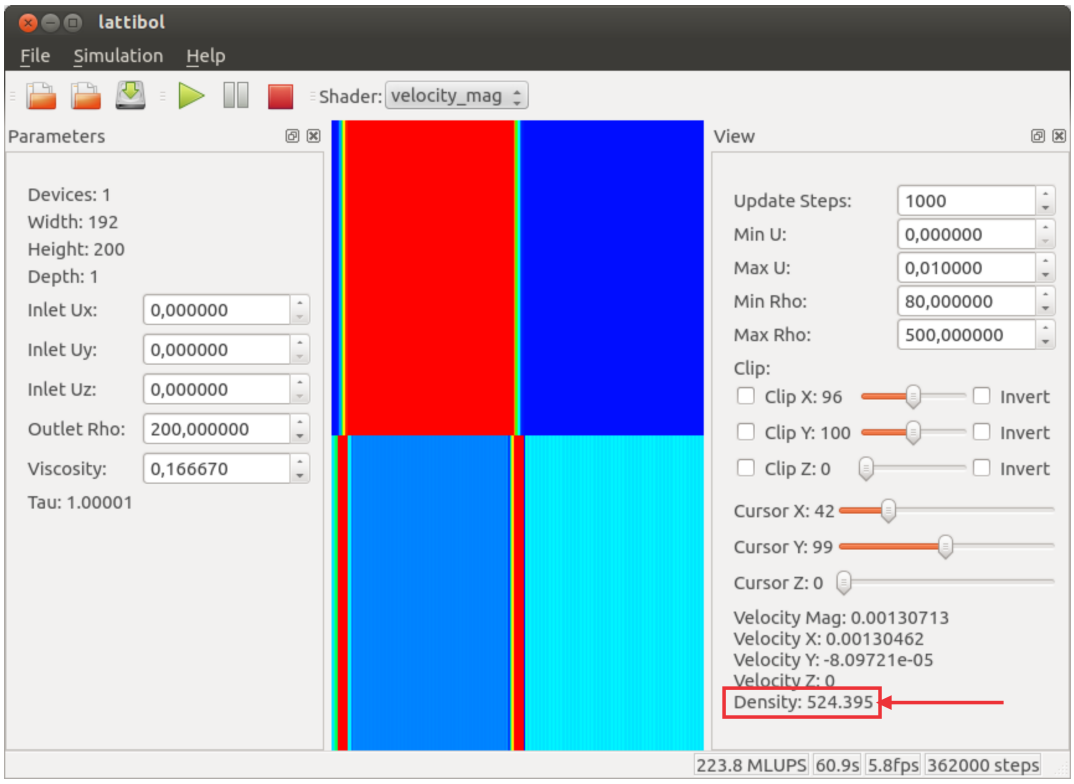
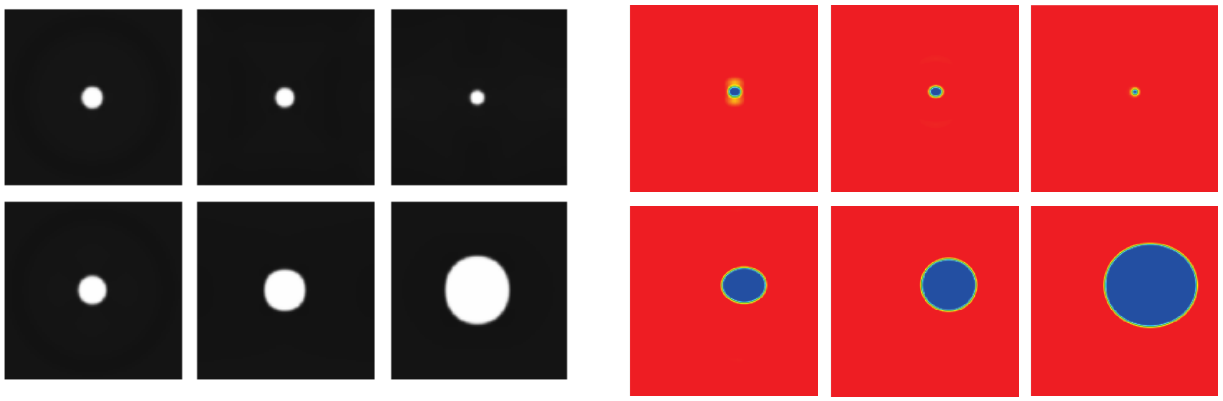


Figure 5.19: Liquid phase density in a flat interface simulation.

In this simulation, periodic boundaries were applied on the horizontal edges of a $200 \times 200\, lu^2$ domain while pressure boundaries were applied on the left and right sides. A density of $500\, mu/lu^2$ was used on the boundaries, which corresponds to a liquid pressure below that of the flat interface simulation ($524.39\, mu/lu^2$). This condition stretches the fluid, leading to cavitation.

Vapor bubbles with radius below the critical value condense, while bubbles with radius above the critical value grow and cavitation occurs. The critical value is defined by the Laplace law. Using the estimated surface tension from Subsection 5.3.2 and Equation 2.39 for the Laplace law from Section 2.11, with a pressure difference of $-1\, mu/lt$, the critical radius is $13.178\, lu$. First, a seed bubble of radius $12\, lu$ is used, and then a bubble of radius $15\, lu$. Figure 5.20(a) shows simulations from Sukop and Thorne Jr. (2005), with the upper images being the simulation with seed bubble of radius $12\, lu$, and the underneath images being the simulation with seed bubble of radius $15\, lu$. Figure 5.20(b) shows the same result for the interactive simulator, with red colour being liquid and blue colour being vapor phase, demonstrating that bubbles below the critical radius disappears, while bubbles above the critical radius grows.



(a) Literature result (Sukop and Thorne Jr., 2005).

(b) Interactive simulator result.

Figure 5.20: Heterogeneous cavitation.

5.3.5 Contact angles

The appropriate value for G_{ads} simulates all possible contact angles. To simulate contact angles of 0, 90, and 180 degrees, it is necessary to calculate G_{ads} by balancing cohesive and adhesive forces (Sukop and Thorne Jr., 2005). Assuming the lattice nodes are pure liquid or pure vapor, all neighbours of a node have the same density of this node, so its cohesive force is:

$$\mathbf{F}^{\text{phase}} = -G\psi_{\text{phase}}^2 \sum_{a=1}^{8} w_a \mathbf{e_a}, \tag{5.1}$$

in which all ψ are equal, and phase can be v for vapor or l for liquid.

A fluid node that is completely surrounded by solid walls undergoes the adhesive force:

$$\mathbf{F}_{\mathbf{ads}}^{\text{phase}} = -G_{ads}\psi_{\text{phase}} \sum_{a=1}^{8} w_a \mathbf{e_a}. \tag{5.2}$$

Fluid nodes that have the average ψ value represent an interface between liquid vapor, and

experience:

$$\bar{\mathbf{F}} = -G\bar{\psi}^2 \sum_{a=1}^{8} w_a \mathbf{e_a}, \tag{5.3}$$

$$\bar{\mathbf{F}}_{\mathbf{ads}} = -G_{ads}\bar{\psi} \sum_{a=1}^{8} w_a \mathbf{e_a}, \tag{5.4}$$

in which $\bar{\psi} = \frac{1}{2}(\psi_l + \psi_v)$.

To simulate a contact angle of $0°$, which means a surface completely wetted by liquid, the adhesive force between the solid and the liquid is equal to the liquid cohesive force:

$$\mathbf{F^l} = \mathbf{F^l_{ads}} \Leftrightarrow G_{ads} = G\psi_l. \tag{5.5}$$

Using the liquid density from Section 5.3.3 ($524.39\, mu/lu^2$), $\psi = 2.732$. The interaction strength of this simulation was -120, so $G_{ads} = -327.79$.

To simulate the contact angle of $90°$ on a surface wetted by liquid at a quantity exactly between completely wettable and completely non-wettable, the adhesive force of the solid on the interface between liquid and vapor should equal the cohesive force at the liquid-vapor interface:

$$\bar{\mathbf{F}} = \bar{\mathbf{F}}_{\mathbf{ads}} \Leftrightarrow G_{ads} = G\frac{(\psi_l + \psi_v)}{2}, \tag{5.6}$$

in which $G_{ads} = -187.16$ with vapor density from Section 5.3.3 of $85.7\, mu/lu^2$ and $\psi_v = 0.388$.

Finally, to simulate the contact angle of $180°$ on a surface completely non-wettable by the liquid, the adhesive force between the solid and the vapor must be equal to the cohesive force of the vapor:

$$\mathbf{F^v} = \mathbf{F^v_{ads}} \Leftrightarrow G_{ads} = G\psi_v, \tag{5.7}$$

with $G_{ads} = -46.534$ and $\psi_v = 0.388$.

Figure 5.21 shows good agreement between the results of Sukop and Thorne Jr. (2005) simulations and the interactive simulator for these three contact angles.

(a) Literature result (Sukop and Thorne Jr., 2005).

(b) Interactive simulator result.

Figure 5.21: Simulation of three different contact angles: $0°$, $90°$, and $180°$.

5.4 Performance tests

To test the performance of the simulator versions, a set of various domain sizes (Table 5.1) was created to run the Poiseuille flow during 500 iterations. The domain sizes varied from approximately 32 thousand nodes up to more than 2 millions nodes at GPU and more than 16 millions nodes at CPU. The amount of MLUPS was measured by carrying out 100 measures for serial and parallel CPU versions, while 10 measures were taken for the interactive GPU version. To measure run-time, the same 10 measures were carried out for the interactive GPU version, but for the serial and parallel CPU versions, 3 measures were carried out. The difference in the amount of measures was due to the possibility to choose a specified number of MLUPS measures in the serial and parallel CPU versions, while the other measurements had to be taken manually. Two types of measures were carried out in the interactive GPU algorithm: one was taken at maximum fps, here denoted GPU 1, while the other had a limited value of around 30 fps, here denoted GPU 2. Next subsections show results for each type of measurement.

Table 5.1: Domain sizes used in the performance test.

Domain number	Domain size	Number of lattice nodes
1	512×64	32,768
2	1024×128	131,072
3	1024×256	262,144
4	2048×256	524,288
5	2048×512	1,048,576
6	4096×512	2,097,152
7	8192×1024	8,388,608
8	8192×2048	16,777,216

5.4.1 MLUPS assessment

Figure 5.22 shows the amount of MLUPS measured for each domain size tested for the MPI version with varying number of processes (1, 2, 4, and 8). In this graph, the curves were generated using the smoothed lines graph type from Excel, using a logarithmic x-axis. It is possible to note that the serial code had a performance of 10.90 MLUPS on the average, while the parallel version with MPI running with 1 process had a performance of around 5.22 MLUPS on the average, and with 2 processes it was around 10.42 MLUPS on the average. These values indicate that it is not recommended to consider MPI code with 1 process as serial code, because due to the MPI overhead, its performance is worse than the pure serial code (of around half the performance). Although the MPI code with 2 processes had a performance similar to serial code, the MPI code with 4 processes had a performance of 19.90 MLUPS on the average, almost 83% more lattice updates than the serial code.

Considering the MPI code with 8 processes, the performance was 20.14 MLUPS on the average, with almost no gain when compared to the MPI code with 4 processes. This happens because the processor used in these tests had 4 physical cores and 8 threads, being possible to achieve a slightly better performance with 8 threads, but nothing substantial. With more than 8 processes, they would have to be divided among the processor threads, and the threads would have more than one process running sequentially, so there would be no gain in performance neither this test was not performed.

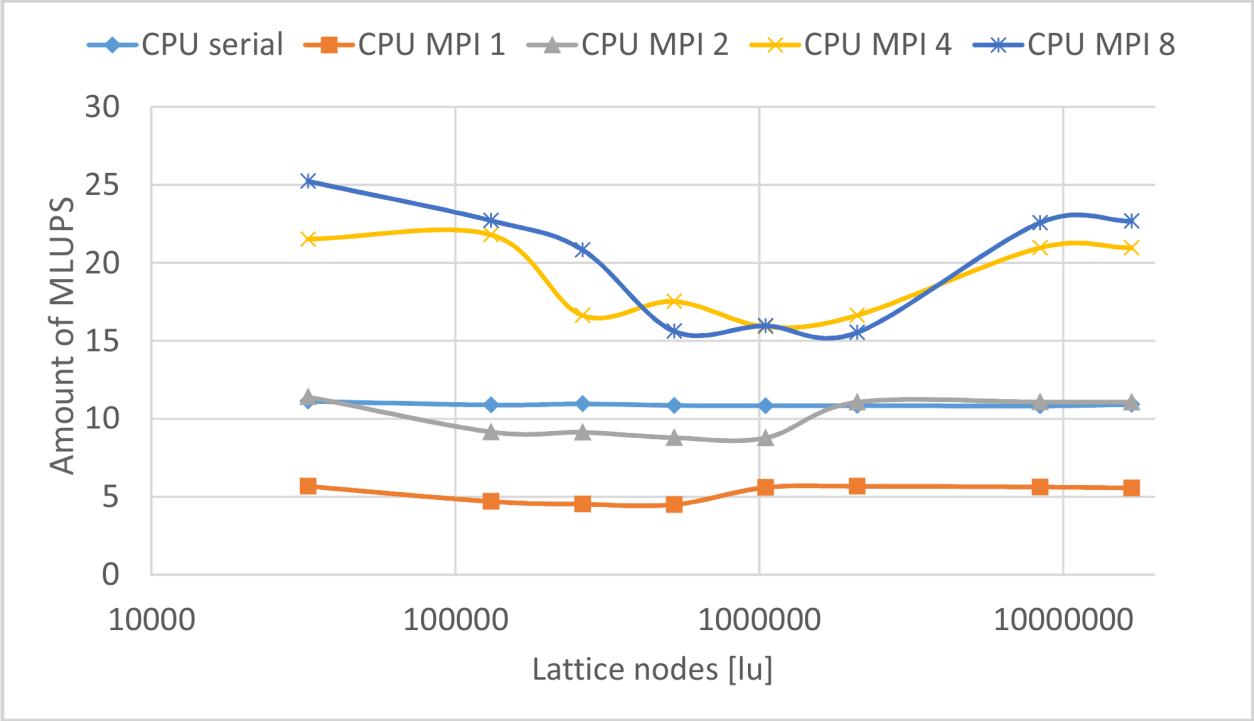


Figure 5.22: Amount of MLUPS measured for CPU serial and parallel codes.

Figure 5.23 shows the amount of MLUPS measured for each domain tested for the interactive GPU version, the MPI version with 8 processes to compare to GPU results, and also the amount of frames per second for GPU 1. In this graph, the curves were generated using the smoothed lines graph type from Excel, using a logarithmic x-axis. As general result, the interactive GPU version had a much greater performance than the MPI and serial CPU versions: when the frame rate was set to its maximum value, the GPU code had 293.67 MLUPS on the average, and when the frame rate was limited to around 30 fps, it had 403.17 MLUPS on the average. When the frame rate was not limited, the executions had 281.5 fps on the average, so the measurements were also done limiting the fps to 30 because this number is sufficient for human vision.

5.4.2 Run-times and speedup assessment

Figure 5.24 shows the run-times for the MPI code with 1, 2, 4, and 8 processes, and Figure 5.25 shows run times for the GPU code and MPI with 8 processes. In this graph, data points were interpolated by a linear fit, again using a logarithmic x-axis. As expected, run-times were consistent

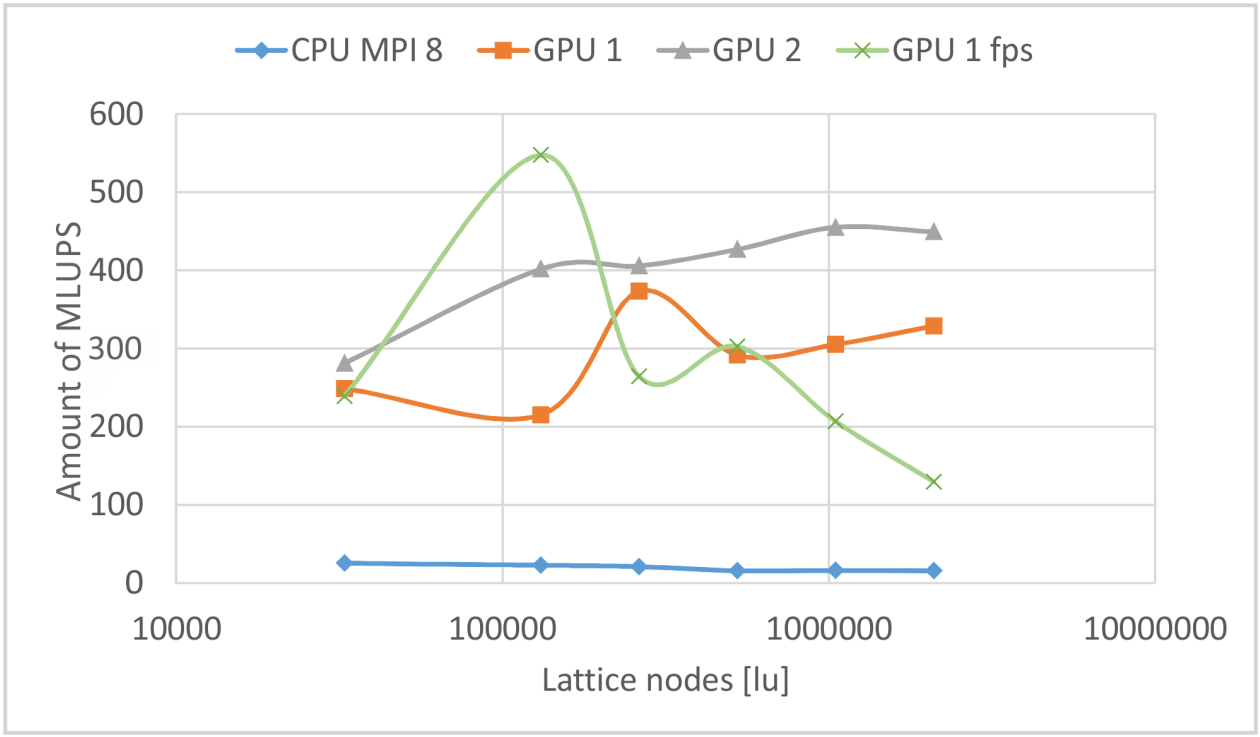


Figure 5.23: Amount of MLUPS measured for MPI with 8 processes and for interactive GPU without limiting fps (GPU 1) and with fps limited at 30 (GPU 2).

with the amount of MLUPS: MPI with 1 process is slower than serial code, which in turn is slower than MPI with 2 processes, and MPI with 4 and 8 processes are faster and differ slightly. GPU with frame rate limited to 30 fps was better than GPU without any limit, as shown in Figure 5.25 (with the same linear fit).

Figure 5.26 shows the speedup of the MPI code in relation to serial code, while Figure 5.27 shows the speedup of the GPU code compared to MPI code with 8 processes, both in relation to serial code. In these graphs, the curves were generated using the smoothed lines graph type from Excel, with a logarithmic x-axis. In Figure 5.26, the result is similar to the amount of MLUPS: MPI with 1 process harms the performance by 64% when compared to serial code, MPI with 2 processes is 11% better than serial code; with 4 processes the code is 86% better than serial code, and with 8 processes it is 98% better. All these values are average values. GPU results are also similar to the amount of MLUPS in Figure 5.27: without limiting the frame rate, the interactive simulator on GPU is 25 times faster than serial code in the average, and by limiting the frame rate to 30 fps, the performance gain reaches 71.3 times faster than serial code.

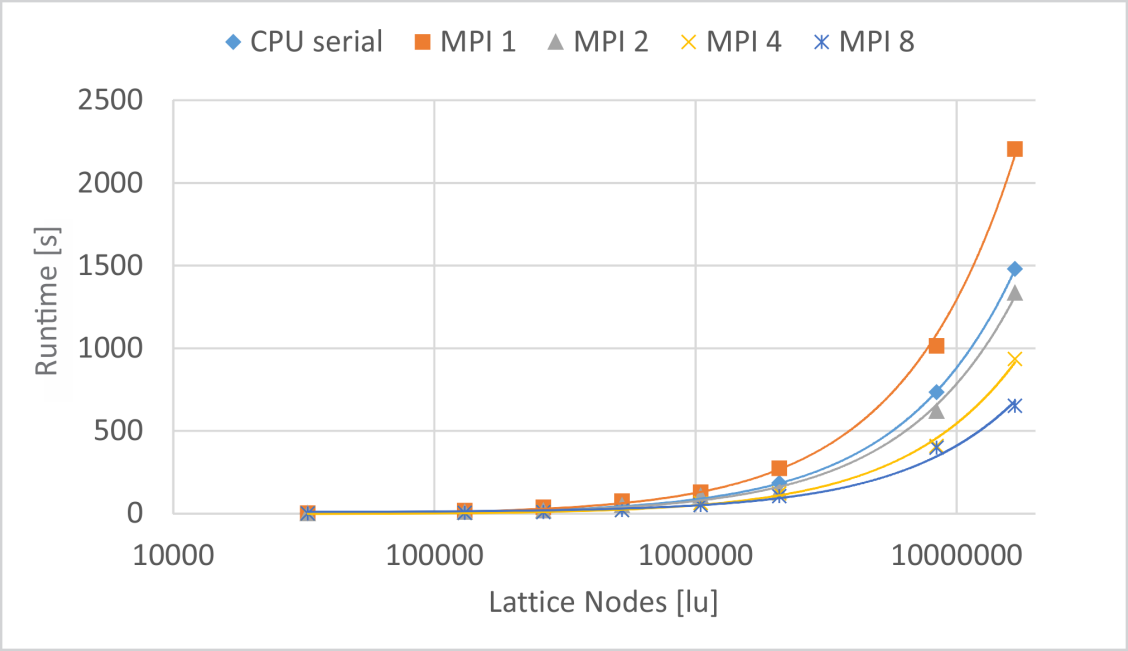


Figure 5.24: Run-times of CPU serial and parallel codes.

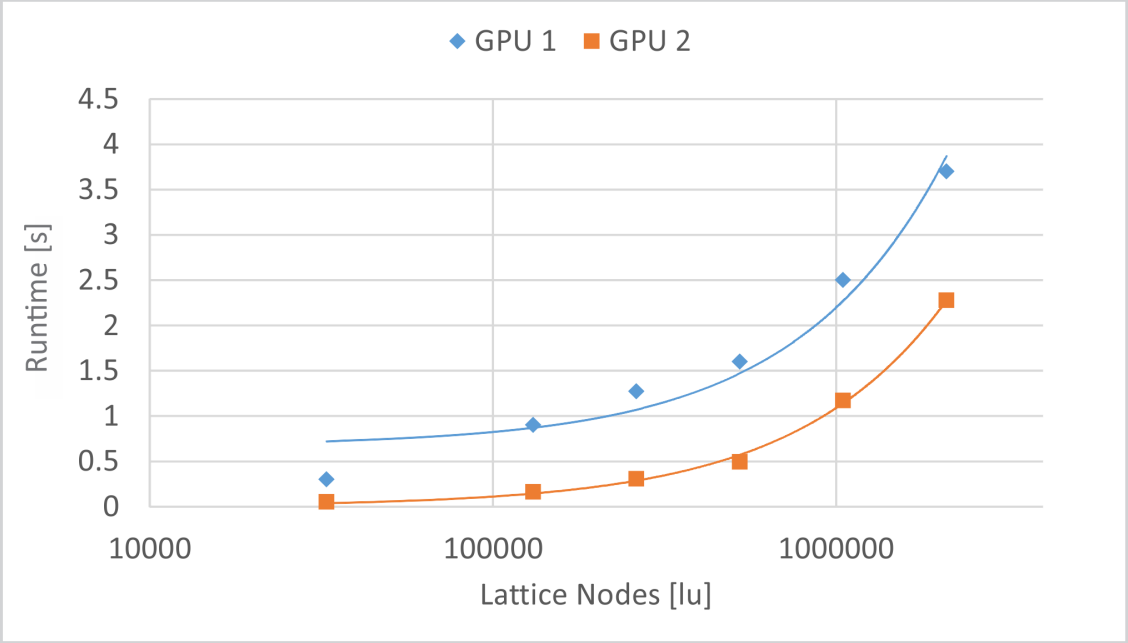


Figure 5.25: Run-times of parallel CPU with 8 processes and interactive GPU with maximum amount of fps (GPU 1) and with fps limited at 30 (GPU 2).

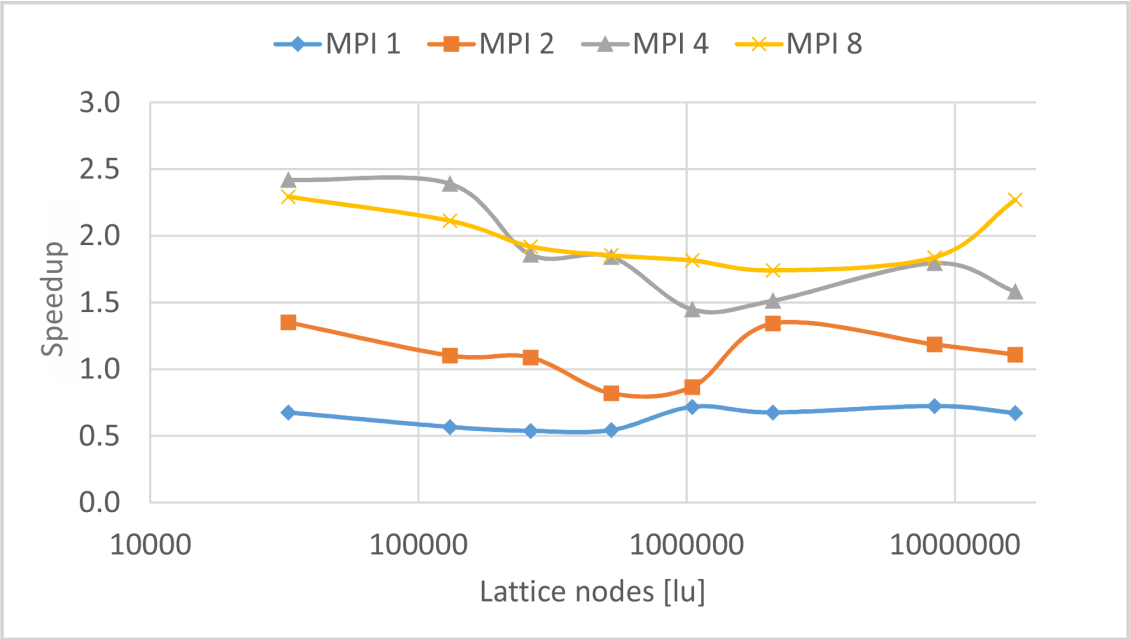


Figure 5.26: Speedup result of CPU serial and parallel codes related to sequential code.

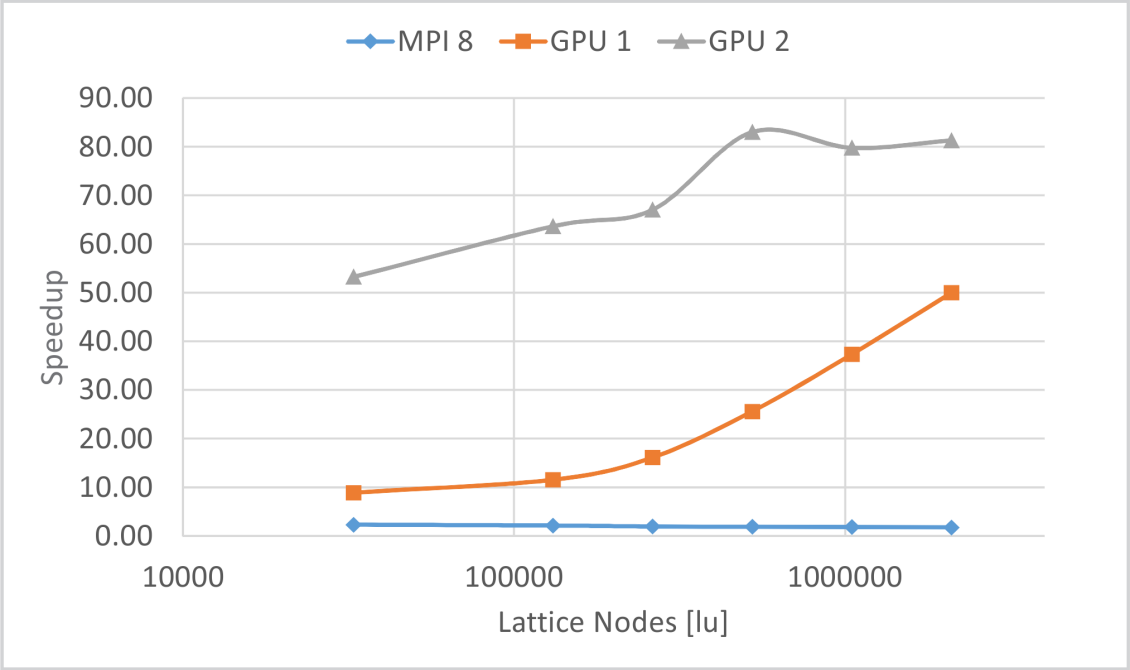


Figure 5.27: Speedup result of parallel CPU with 8 processes and interactive GPU with maximum amount of fps (GPU 1) and with fps limited at 30 (GPU 2), related to sequential code.

With these performance results, it is possible to affirm that the interactive simulator based on GPU is more advantageous than a CPU implementation without visualization. The gain in simulation speed when using GPU accelerates analyses and transforms the simulator in an interactive time simulator, while the interactive visualization brings useful tools that allow parameter modifications on the simulation at execution time, and the results visualization is eased by the interaction tools.

6 Conclusion

In this work, the LBM was implemented in order to be achieve high performance simulations. The equations of the SCSP and the SCMP flows were studied, as well as boundary conditions and external forces. To achieve high performance, the algorithm was parallelized in two architectures: CPU using the MPI extension, and GPU using the CUDA extension. The swap algorithm was also used to reduce memory requirements without compromising performance.

After that, an interactive 3D visualization procedure was implemented, aiming to have a unique tool to simulate, visualize, and interact with fluid flows. The 3D texturing technique was used in the simulation domain visualization, shaders were employed to switch between different simulation data to be visualized, Qt classes were utilised to implement the interactive features, and the interoperability between CUDA and OpenGL was used to gain performance by keeping all data inside GPU, which avoided data transfers.

The simulator accepts a bitmap file to identify solid nodes in the domain for 2D simulations, and a STL file for 3D simulations, which allowed to import domains drawn in CAD programs. A feature to save VTK files was also included, being possible to save simulation instants to analyse flow properties in other programs. The code was also made available at Oliveira (2015).

Simulation results of the literature classical flows showed good agreement for SCSP cases, validating the simulator for single phase flows. Some SCMP cases were successfully simulated as well, like the phase separation and interface minimization, flat interfaces, surface tension estimation, heterogeneous cavitation, and contact angles.

From the implemented versions of this work, a performance test could be done considering serial and parallel CPU codes, and interactive GPU code. The test considered 8 increasing-size domains for a SCSP flow, 500 iterations, and the performance measurements were the amount of MLUPS, run-time, and speedup. Results showed that the MPI code could almost double the performance of serial code using the Intel® Core i7 CPU 950 processor, and that the interactive GPU code achieved 71.3 times more speed than CPU serial code using the NVIDIA® GeForce® GTX 560 Ti graphics card. It is worth noting that serial and MPI codes did not have interactive visualization, and that the GPU code was limited to show around 30 fps. When the frame rate was not limited, the performance gain was 25 times faster than serial code. These results showed that the

GPU code is more appropriate for fluid flow simulations based on the LBM, because it can reach interactive simulation speed. Moreover, the interactive visualization demonstrated to be useful in LBM analyses, because it allows an early intervention at the simulation, change of simulation parameters and fast achievement of simulation information.

The next steps of this work may be divided into two parts: LBM and parallelism. LBM improvements, such as other algorithms and optimisations, might be added to the simulator in order to enlarge the flows that can be simulated with LBM. Some SCMP cases remain to be simulated; they were studied but were not included in this work because they still need adjustments. The cases were: homogeneous cavitation, capillary rise, and vapor invasion in porous media. Multicomponent multiphase flows, which were not discussed in this work, might be included in the simulator as well. The GPU performance gain might be increased by optimisations on the code, by using more than one GPU on the same node, and by using a cluster of GPUs that is presented at the Supercomputing Laboratory.

References

AGARWAL, R.K. and CHUSAK, L. Lattice boltzmann simulations of slip flow of non-newtonian fluids in microchannels. v. 74, 247–256, 2010.
URL: *http://dx.doi.org/10.1007/978-3-642-14438-7_26*

AIDUN, C.K. and CLAUSEN, J.R. Lattice-Boltzmann method for complex flows. **Annual review of fluid mechanics**, v. 42, 439–472, 2010.

AKENINE-MÖLLER, T.; HAINES, E. and HOFFMAN, N. **Real-Time Rendering**. A. K. Peters, Ltd., Natick, MA, USA, 3^{rd} ed., 2008. ISBN 987-1-56881-424-7.

AKSNES, E.O. Simulation of fluid flow through porous rocks on modern GPUs. 2009.

AKSNES, E.O. and ELSTER, A.C. Porous rock simulations and Lattice Boltzmann on GPUs. In **PARCO**, pp. 536–545. 2009.

ALT, A. Mixing graphics and compute with multiple GPUs. http://on-demand.gputechconf.com/gtc/2012/presentations/S0267A-Mixing-Graphics-and-Compute-with-Multiple-GPUs-Part-A.pdf, 2012.

BAILEY, P.; MYRE, J.; WALSH, S.D.C.; LILJA, D.J. and SAAR, M.O. Efficient algorithms for ghost cell updates on two classes of MPP architectures. 2009.

BHATNAGAR, P.L.; GROSS, E.P. and KROOK, M. A kinetic approach to collision processes in gases. I. Small amplitude processes in charged and neutral one component systems. Technical report, Massachusetts Institute of Technology, USA, 1954.

BLYTHE, D. **Lighting and shading techniques for interactive applications**. Silicon Graphics, 1999.

CHEN, S. and DOOLEN, G.D. Lattice Boltzmann method for fluid flows. **Annual Review of Fluid Mechanics**, v. 30, 329–364, 1998.

CHEN, S.; MARTÍNEZ, D. and MEI, R. On boundary conditions in lattice Boltzmann methods. **Physics of Fluids**, v. 8, 1996.

CHENG, J.; KRICKA, L.J.; SHELDON, E.L. and WILDING, P. Sample preparation in microstructured devices. **Microsystem Technology in Chemistry and Life Science**, v. 194, 215–231, 1998.

CHOPARD, B. How to improve the accuracy of Lattice Boltzmann calculations. http://wiki.palabos.org/_media/howtos:singleprecisionlb.pdf, 2008.

EHRFELD, W. Electrochemistry and microsystems. **Electrochimica Acta**, v. 48, 2857–2868, 2003.

ELWENSPOEK, M.; LAMMERINK, T.S.J.; MIYAKEI, R. and RUITMAN, J.H.J. Towards integrated microliquid handling systems. **Journal of Micromechanics and Microengineering**, v. 4, 227–245, 1994.

ENGEL, K.; HADWIGER, M.; KNISS, J.M.; RESZ-SALAMA, C. and WEISKOPF, D. **Real-Time Volume Graphics**. A K Peters, Wellesley, Massachusetts, 2006. ISBN 1568812663.

FAN, Z. Flow simulation and visualization on GPU clusters. 2008.

GEBHARD, U.; HEIN, H. and SCHMIDT, U. Numerical investigation of fluidic micro-oscillators. **Journal of Micromechanics and Microengineering**, v. 6, 115–117, 1996.

GIANNITSIS, A.T. Microfabrication of biomedical lab-on-chip devices. A review. **Estonian**

Journal of Engineering, v. 17, 109–139, 2011.

HABICH, J.; ZEISER, T.; HAGER, G. and WELLEIN, G. Performance analysis and optimization strategies for a {D3Q19} lattice boltzmann kernel on nvidia {GPUs} using {CUDA}. **Advances in Engineering Software**, v. 42, n. 5, 266 – 272, 2011. {PARENG} 2009.
URL: *http://www.sciencedirect.com/science/article/pii/S0965997810001274*

HARRIS, S.M. **An Introduction to the Theory of the Boltzmann Equation**. Holt, Rinehart and Winston, New York, United States of America, 1^{st} ed., 1971. ISBN 9780030827891.

HE, X. and LUO, L. Lattice Boltzmann model for the incompressible Navier-Stokes equation. **Journal of Statistical Physics**, v. 88, 1997.

JACKSON, M.J. **Microfabrication and Nanomanufacturing**. CRC Press, 1^{st} ed., 2006. ISBN 978-0824724313.

JANUSZEWSKI, M. Sailfish manual reference. http://sailfish.us.edu.pl/, Jan 2015.

KANG, J.; HEO, H. and SUH, Y. LBM simulation on mixing enhancement by the effect of heterogeneous zeta-potential in a microchannel. **Journal of Mechanical Science and Technology**, v. 22, n. 6, 1181–1191, 2008.
URL: *http://dx.doi.org/10.1007/s12206-008-0301-4*

KIRK, D.B. and HWU, W.W. **Programming massively parallel processors**. Morgan Kaufmann, 2010.

LATT, J. How to implement your DdQq dynamic with only q variables per node (insted of 2q). Technical report, Tufts University Medford, USA, 2007.

LI, B. and KWOK, D. Y. A Lattice Boltzmann model with high Reynolds number in the presence of external forces to describe microfluidics. **Heat and Mass Transfer**, v. 40, n. 11, 843–851, 2004.
URL: *http://dx.doi.org/10.1007/s00231-003-0442-z*

MA, A.; CAI, J.; CHENG, Y.; NI, X.; TANG, Y. and XING, Z. Performance optimization strategies of high performance computing on GPU. **Advanced Parallel Processing**, pp. 150–164, 2009.

MAIRHOFER, J.; ROPPERT, K. and ERTL, P. Microfluidic systems for pathogen sensing: a review. **Sensors**, v. 9, 4804–4823, 2009.

MANZ, A.; GRABER, N. and WIDMER, H.M. Miniaturized total chemical analysis systems: A novel concept for chemical sensing. **Sensors and Actuators B: Chemical**, v. 1, 244–248, 1990.

MARTYS, N.S. and CHEN, H. Simulation of multicomponent fluids in complex three-dimensional geometries by the lattice Boltzmann method. **Physical Review E**, v. 53, 743–750, Jan 1996. **URL:** *http://link.aps.org/doi/10.1103/PhysRevE.53.743*

MATLAB. Color maps. http://www.mathworks.com/help/matlab/ref/colormap.html, 2014.

MATTILA, K.; HYVÄLUOMA, J.; ROSSI, T.; ASPNÄS, M. and WESTERHOLM, J. An efficient swap algorithm for the lattice Boltzmann method. **Computer Physics Communications**, pp. 200–210, 2006.

MCNAMARA, G.R. and ZANETTI, G. Use of the Boltzmann equation to simulate lattice-gas automata. **Physical Review Letters**, pp. 2332–2335, 1988.

MEI, R.; SHYY, W.; YU, D. and LUO, L. Lattice Boltzmann method for 3-D flows with curved boundary. **Journal of Computational Physics**, v. 161, 680–699, 2000.

MICIKEVICIUS, P. 3D finite difference computation on GPUs using CUDA. In **GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units**. 2009.

MOHAMAD, A.A. **The Lattice Boltzmann equation for fluid dynamics and beyond**. Springer, 2011.

NOUAR, C.; OLDROUIS, M.; SALEM, A. and LEGRAND, J. Developing laminar flow in the entrance region of annuli-review and extension of standard resolution methods for the hydrodynamic problem. **International Journal of Engineering Science**, v. 33, 1517–1534, 1995.

NVIDIA. CUDA C best practices guide. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#axzz35l0JMtLn, Feb 2014a.

NVIDIA. CUDA C programming guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz35l0JMtLn, Feb 2014b.

NVIDIA. Fermi compute architecture whitepaper 1.1. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2014c.

OBRECHT, C.; KUZNIK, F.; TOURANCHEAU, B. and ROUX, J.J. Global Memory Access Modelling for Efficient Implementation of the Lattice Boltzmann Method on Graphics Processing Units. v. 6449, 151–161, 2011a.
URL: *http://dx.doi.org/10.1007/978-3-642-19328-6_16*

OBRECHT, C.; KUZNIK, F.; TOURANCHEAU, B. and ROUX, J.J. A new approach to the lattice Boltzmann method for graphics processing units. **Computers and Mathematics with Applications**, v. 61, 3628–3638, 2011b.

OBRECHT, C.; KUZNIK, F.; TOURANCHEAU, B. and ROUX, J.J. The TheLMA project: Multi-GPU implementation of the lattice Boltzmann method. **International Journal of High Performance Computing Applications**, v. 25, 295–303, 2011c.

OLIVEIRA, F.M.C. and FERREIRA, L.O.S. Simulation of the alcohol-oil mixture in a micromixer using the Lattice Boltzmann method on a GPU device. In **Proceedings of the 14th Brazilian Congress of Thermal Sciences and Engineering**. Rio de Janeiro, Brazil, 2012.

OLIVEIRA, F.M.C. and FERREIRA, L.O.S. Performance analysis of serial and parallel implementation for a 2D microfluidics simulator using the Lattice Boltzmann method. In **Proceedings**

of the 22^{nd} **Internacional Congress of Mechanical Engineering**. Ribeirão Preto, Brazil, 2013.

OLIVEIRA, F.M.C. and VOLPE, L.M. Sistema interativo de visualização de dados volumétricos. http://www.dca.fee.unicamp.br/courses/IA369E/2s2013/projects/oliveira-volpe/index.html, Dec 2013.

OLIVEIRA, F.M.C. Lattibol_0.3.31f source code. https://github.com/FabiolaOliveira/lattibol_0.3.31f, Mar 2015.

OLIVEIRA, F.M.C.; VOLPE, L.M. and FERREIRA, L.O.S. Performance analysis of parallel CPU and GPGPU implementation of a 2D microfluidics simulator using the Lattice Boltzmann method. In **Proceedings of the XXXIV Iberian Latin-American Congress on Computational Methods in Engineering**. Pirenópolis, Brazil, 2013.

PACHECO, P. **An introduction to parallel programming**. Elsevier, 2011.

PALMER, B. and NIEPLOCHA, J. Efficient algorithms for ghost cell updates on two classes of MPP architectures. 2002.

QIAN, Y.H.; D'HUMIERES, D. and LALLEMAND, P. Lattice BGK models for Navier-Stokes equation. **Europhysics Letters**, v. 17, 479–484, 1992.

QIN, D.; XIA, Y.; ROGERS, J.A.; JACKMAN, R.J.; ZHAO, X. and WHITESIDES, G.M. Microfabrication, microstructures and microsystems. **Microsystem Technology in Chemistry and Life Science**, v. 194, 1–20, 1998.

QT. Qt project. http://qt-project.org/, 2014.

SANDERS, J. and KANDROT, E. **CUDA by Example: an introduction to general-purpose GPU programming**. Addison-Wesley, 2011.

SCHREIBER, M. GPU based simulation and visualization of fuids with free surfaces. 2010.

SHAN, X. and CHEN, H. Lattice Boltzmann model for simulating flows with multiple phases and components. **Physical Review E**, v. 47, 1815–1820, 1993.

SHREINER, D. **OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions** 3.0 **and** 3.1. Addison-Wesley Professional, 7^{th} ed., 2009. ISBN 0321552628, 978-0321552624.

STÜRMER, M.; GÖTZ, J.; RICHTER, G.; DÖRFLER, A. and RÜDE, U. Fluid flow simulation on the Cell Broadband Engine using the lattice Boltzmann method. **Computers & Mathematics with Applications**, v. 58, n. 5, 1062 – 1070, 2009. Mesoscopic Methods in Engineering and Science. **URL:** *http://www.sciencedirect.com/science/article/pii/S0898122109002442*

SUCCI, S. **Lattice Boltzmann method: fundamentals and engineering applications with computer codes**. Numerical Mathematics and Scientific Computation. Oxford Science Publications, 2001.

SUKOP, M.C. and THORNE JR., D.T. **Lattice Boltzmann modeling: an introduction to geoscientists and engineers**. Springer, 2005.

TANEDA, S. Experimental investigation of the wakes behind cylinders and plates at low Reynolds numbers. **Journal of the Physical Society of Japan**, v. 11, 302–307, 1956.

TANEDA, S. Visualization of separating Stokes flows. **Journal of the Physical Society of Japan**, v. 46, 1935–1942, 1979.

TELEA, A.A. **Data Visualization: Principles and Practice**. A K Peters, 2008. ISBN 1568813066.

TÖLKE, J. and KRAFCZYK, M. TeraFLOP computing on a desktop PC with GPUs for 3D CFD. **International Journal of Computational Fluid Dynamics**, v. 22, 443–456, 2008.

UNIVERSITY OF GENEVA. Common VersatileMulti-purpose Library for C++. http://tech.unige.ch/cvmlcpp/, 2012.

VALDERHAUG, T.K. The Lattice Boltzmann simulation on multi-GPU systems. 2011.

VAN DEN BERG, A. and LAMMERINK, T.S.J. Micro total analysis systems: microfluidic aspects, integration concept and applications. **Microsystem Technology in Chemistry and Life Science**, v. 194, 21–49, 1998.

VOLKOV, V. and KRAFCZYK, M. Benchmarking GPUs to tune dense linear algebra. In **Proceedings of the 2008 ACM/IEEE conference**. 2008.

WIKIBOOKS. OpenGL programming/modern OpenGL tutorial Arcball. http://en.wikibooks.org/wiki/OpenGL_Programming/ Modern_OpenGL_Tutorial_Arcball, 2014.

WOIAS, P. Micropumps—past, progress and future prospects. **Sensors and Actuators B Chemical**, v. 105, 28–38, 2004.

WOLF-GLADROW, D.A. **Lattice-gas cellular automata and Lattice Boltzmann models - an introduction**, v. 1725. Springer, 2005.

XIAN, W. and TAKAYUKI, A. Multi-GPU performance of incompressible flow computation by lattice Boltzmann method on GPU cluster. **Parallel Computing**, 2011.

YONG, Y.; YANG, C.; JIANG, Y.; JOSHI, A.; SHI, Y. and YIN, X. Numerical simulation of immiscible liquid-liquid flow in microchannels using lattice Boltzmann method. **Science China Chemistry**, v. 54, n. 1, 244–256, 2011.
URL: *http://dx.doi.org/10.1007/s11426-010-4164-z*

ZHANG, J. Lattice Boltzmann method for microfluidics: models and applications. **Microfluid Nanofluid**, v. 10, 2011.

ZOU, Q. and HE, X. On pressure and velocity boundary conditions for the lattice Boltzmann BGK model. **Physics of Fluids**, v. 9, 1997.

ANNEX A Published papers

Two papers were published during this masters: one in Proceedings of the 22^{nd} Internacional Congress of Mechanical Engineering (COBEM, 2013), Ribeirão Preto, Brazil, 2013, and another in Proceedings of the XXXIV Iberian Latin-American Congress on Computational Methods in Engineering (CILAMCE, 2013), Pirenópolis, Brazil, 2013. Copies of these papers follow next.

PERFORMANCE ANALYSIS OF SERIAL AND PARALLEL IMPLEMENTATION FOR A 2D MICROFLUIDICS SIMULATOR USING THE LATTICE BOLTZMANN METHOD

Fabíola Martins Campos de Oliveira
Luiz Otávio Saraiva Ferreira
Department of Computational Mechanics, Mechanical Engineering Faculty, State University of Campinas, Campinas, Brazil
fabiola@fem.unicamp.br, lotavio@fem.unicamp.br

Abstract. *Recent advancements on multicore processors technology led to parallel computing, decreasing runtime of several interesting problems for engineering, and making possible an expansion of the problem domain. One example is the fluid mechanics, an area of great economic and academic interest, whose simulations have high computational cost. This said, a good method for simulating fluid flows with proven advantage for use in parallel computing is the Lattice Boltzmann Method. As its algorithm is highly parallelizable, simulations based on this method tend to gain efficiency when more cores are used. As a first step, this work gets a cpu-based algorithm for measuring efficiency running on a single-core processor and, later, this algorithm is converted into a parallelized cpu-based code running on a multicore processor. At last, the performance of both fluidic simulators based on the Lattice Boltzmann Method running on a single-core processor and on a multicore processor is compared.*

Keywords: *Lattice Boltzmann method, microfluidics, parallel computing*

1. INTRODUCTION

From 1986 until 2002, software developers and final users could rely on technical advances of microprocessors to increase performance of their programs, having an average of 50% more speed per year (Pacheco, 2011). After 2002, this performance decreased to 20% per year due to technical issues, like the difficulty of dissipating heat of high-density transistors microprocessors. To overcome this, the next step for industries to achieve higher performance was adding more than one processor on a single chip. The main consequence of this decision for programmers was that their old programs would no longer benefit from new technologies as before, since single processor programs do not recognise multiple processors. Since 2005, when most companies started offering multicore processors, serial codes needed to be rewritten in order to increase performance using these new processors. There is a trend for developing translation programs, which would automatically convert a serial code into a parallel code, but since this approach has only been good for specific cases, it is needed to find more efficient algorithms for each case (Pacheco, 2011).

For mechanical engineering, an area that widely takes advantage of parallelization is fluid mechanics. Usually, traditional computational fluid dynamics (CFD) methods demand heavy computational resources in order to simulate fluid flows properly. A more powerful method for solving fluid dynamics problems (Mohamad, 2011) that is effectively parallelizable is the Lattice Boltzmann method (LBM). Compared to finite differences method, LBM have proved to double performance (Chen *et al.*, 1994) LBM models the fluid as particles probabilities distribution functions, that collide and propagate over a lattice domain (Oliveira and Ferreira, 2012). This method easily handles features that traditional CFDs can not deal with or is very slow, like complex boundaries and multicomponent multiphase flows (Succi, 2001).

This work measures performance of a basic solver using the Lattice Boltzmann method, with solid walls, comparing a pure serial code and a parallel version using Message-Passing Interface (MPI), an extension to languages like C and C++ suitable for distributed-memory systems, meaning that each core has its own amount of memory, and therefore communication among cores is required. As results showed that the parallel code is advantageous, there is expectation on reusability of code for future implementations.

2. MESSAGE-PASSING INTERFACE (MPI) PROGRAMMING

MPI is an explicit parallel extension, what means that the work of each core must be specified, being a more powerful tool than higher level languages, like OpenMP (Pacheco, 2011). The MPI extension includes type definitions, functions and macros and is well suitable for distributed-memory systems, offering ways for communication among cores. There are mainly two ways to parallelize a program: using task-parallelism or data-parallelism (Pacheco, 2011). Task-parallelism divides the problem in several tasks that are distributed among cores, while data-parallelism divides the problem data among cores, and they execute similar tasks on its own data. Writing parallel programs involves coordination of cores: they usually need to communicate with each other, sending information or data. It is also desired for the program to have load balancing, meaning that all cores should receive aproximately the same amount of work, so there are not idle cores during execution of the program. Another type of coordination is syncronization: cores sometimes need to wait for all the other cores to reach the end of a point in code before proceed.

Data can be divided in three ways: using a block partition, when first data block *data/processes* is assigned to first pro-

cess, second data block *data/processes*, to second process, and so on; using a cyclic partition, like round-robin scheduling or a block-cyclic partitioning, when blocks of data are assigned to process in a round-robin manner. Most basic MPI functions works with block partition, but, to have cyclic or block-cyclic partitioning, a MPI derived data type can be created in order to communicate different partitioning.

Lastly, the terms concurrent, parallel and distributed computing have slight differences. While concurrent computing is most used for a program whose multiple tasks can be run at any time, parallel computing means that multiple tasks of a program cooperate to solve a problem. Distributed computing is used for a program that needs to help other programs to solve a problem.

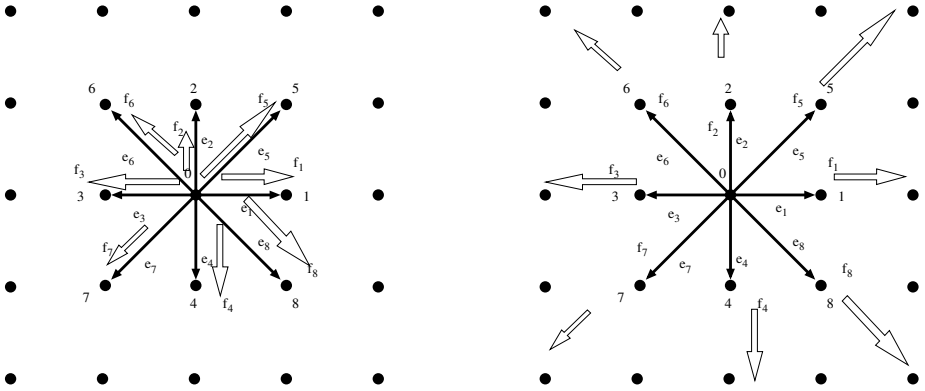
3. NUMERICAL MODELS AND THE LATTICE BOLTZMANN METHOD

The Lattice Boltzmann method (LBM) derived from the Lattice Gas Cellular Automata (LGCA) method of fluid-flow simulation (Wolf-Gladrow, 2005). LBM recovers Navier-Stokes equations in the macroscopic scale based on Boltzmann kinetic theory (Succi, 2001).

In both LGCA and LBM methods, simulation is separated in two steps: streaming and collision. Discretizing the original Boltzmann equation on time, space and momentum gives the LBM Eq. (1) (Aidun and Clausen, 2010; Zhang, 2011):

$$f_a(\mathbf{x} + e_a\Delta t, t + \Delta t) = f_a(x,t) - \frac{[f_a(x,t) - f_a^{eq}(x,t)]}{\tau} \tag{1}$$

in which x is the position of the particle, e_a is its microscopic velocity, t is the time, Δt is the time-step of the simulation, f_a is the particle probability distribution function on direction a, $f_a(\mathbf{x} + e_a\Delta t, t + \Delta t) = f_a(x,t)$ is the streaming part, f_a^eq is the equilibrium probability function, τ is the relaxation parameter and $\frac{[f_a(x,t) - f_a^{eq}(x,t)]}{\tau}$ is the collision term, which is the simplified model introduced in 1954 by Bhatnagar, Gross and Krook and known as BGK aproximation. One of the most used LBE models is the D2Q9 (2 dimensions and 9 velocities), in which the microscopic velocity e_a $(a = 0, \ldots, 8)$ is restricted to 8 directions plus a rest particle, 3 magnitudes, and there is a single particle mass, as shown in Fig. 1a (Oliveira and Ferreira, 2012). It was assumed on Eq. (1) that particle mass $= 1$, so that microscopic velocities and momenta are equivalent.



(a) D2Q9 lattice and microscopic velocities

(b) Streaming step of D2Q9 lattice

Figure 1: LBM model and streaming step. Each velocity e_a (black arrows) has an associated frequency f_a (white arrows)

Units of length and time measurement are the lattice unit (lu) and the lattice time (lt), respectively. Velocity magnitudes of e_1 through e_4 is $1lu/lt$, and velocity magnitudes of e_5 through e_8 is $\sqrt{2}lu/lt$.

The sum of distribution functions of a lattice node gives the macroscopic fluid density:

$$\rho = \sum_{a=0}^{8} f_a \tag{2}$$

and the macroscopic velocity is computed as Eq. (3):

$$u = \frac{1}{\rho}\sum_{a=0}^{8} f_a e_a \tag{3}$$

On the streaming step, each function f_a goes to the nearest neighbor lattice node pointed by the corresponding arrow, as shown in Fig. 1b, resulting on updated values of f for each node. Next step is the collision, and first it is need to perform calculation of the equilibrium distribution function f^{eq} for each node of the lattice, using Eq. (4):

$$f_a^{eq}(x) = \omega_a \rho(x) \left[1 + 3\frac{e_a u}{c^2} + \frac{9}{2}\frac{(e_a u)^2}{c^4} - \frac{3}{2}\frac{u^2}{c^2} \right] \tag{4}$$

in which ω_a is the weight for each particle: $4/9$ for $a = 0$, $1/9$ for $a = 1, 2, 3, 4$ and $1/36$ for $a = 5, 6, 7, 8$ and c is the lattice sound speed, usually $1lu/lt$. After that, BGK aproximation is calculated using this result. Besides the two steps of the method, boundary conditions should be applied on each iteration, so that it is possible to consider the effect of solid walls in the flow. The most common boundary condition for walls is the bounceback condition, as shown in Fig. 2. Periodic condition can be applied on domain edges.

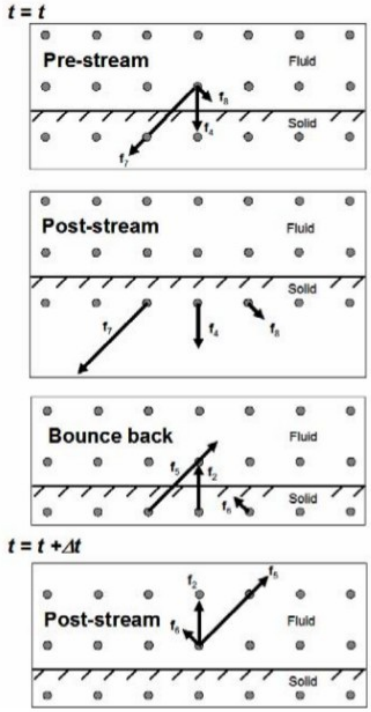


Figure 2: Bounceback movement of probabilities distribution functions f_a (Sukop and Thorne Jr., 2005)

4. IMPLEMENTATION

In this proposed algorithm for parallelizing the Lattice Boltzmann method, data-parallelism is applied to serial code, dividing the domain into smaller column blocks, as in Fig. 3. It was used block-cyclic partitioning, with block size equal to number of processes.

Steps of streaming, collision and boundary conditions can be performed independently, but after streaming it is needed to communicate left and right borders of each subdomain between processes in order to get the correct result. Since each subdomain has its own first and last column and they are exchanged after streaming, the whole domain need to be corrected, as Fig. 4 shows.

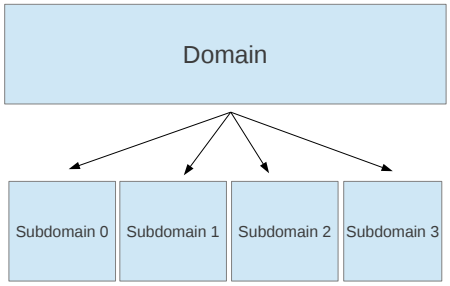


Figure 3: Domain division among processes

Collision and boundary conditions do not need any correction because, in these steps, lattice nodes do not access neighbour nodes, so calculations only depend on the node itself. This code uses a feature for saving 50% of memory in iterations by performing twice calculations to avoid temporary arrays; more details in (Latt, 2007). MPI functions are used to broadcast initial data to all processes, divide work for them, measure time and, on each iteration, exchange borders to right places. Output data is a set of vti extension files, which is an extension associated with ParaView VTK ImageData, for visualizing data, saved by each process individually and so also parallelized. The simulation can be watched in a vtk viewer program, such as ParaView.

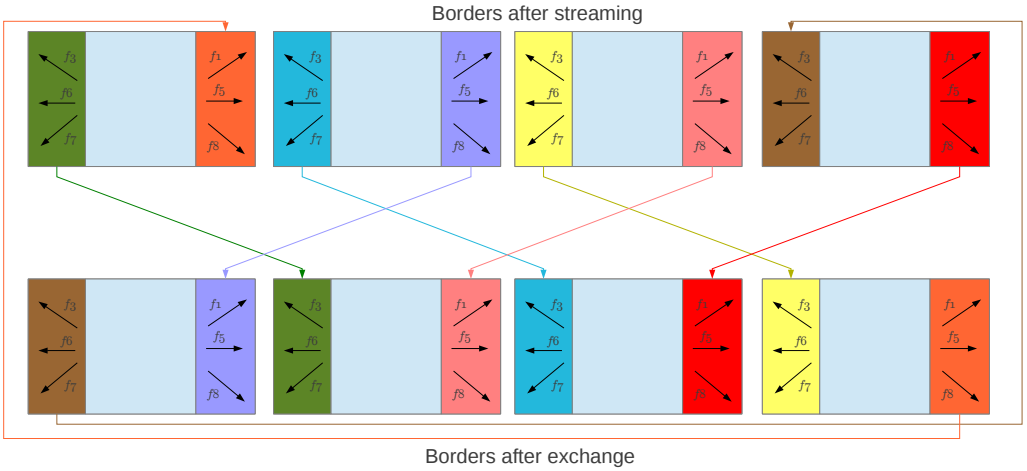
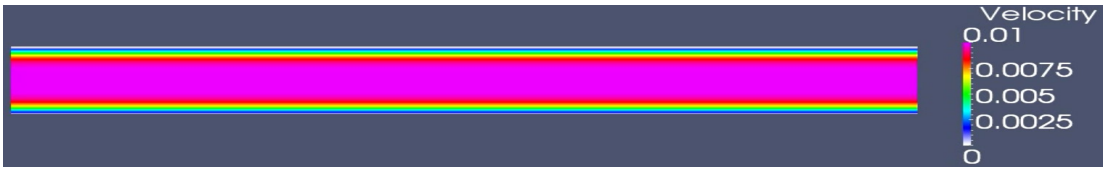


Figure 4: Subdomains after streaming (above) and after exchange of borders (below)

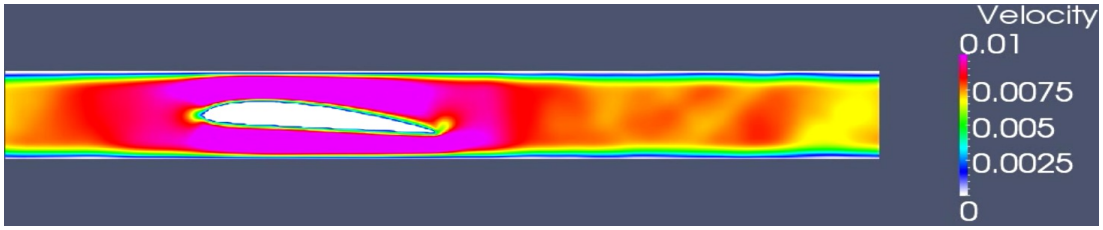
5. RESULTS

In order to validate the simulator, three cases of literature were tested: flow between parallel plates, flow through a cylinder and flow through an airplane airfoil (Fig. 5). As initial condition, a velocity of $0.01lu/lt$ was applied to every fluid node of domain. Periodic boundary was set in left and right edge of domain and in all cases there were plates in upper and lower edges. The microprocessor used to run these codes was Intel® Core^{TM} i7 CPU 950 at 3.07GHz $\times$ 8 (four cores, eight threads).



(a) Flow between parallel plates after 900 iterations

(b) Flow past a cylinder after 400 iterations



(c) Flow through an airfoil after 3000 iterations

Figure 5: Simulation cases

For performance comparison between serial and parallel code, the following sizes 512×64, 1024×128, 2048×256, 4096×512 and 8192×1024 lattice cells were used for each case in serial and in parallel code with two, four and eight processes, since the most recommended number of processes is the maximum number of processor's cores. The domain is described by a bitmap input file, that can be generated with a simple drawing software. Each pixel of the input bitmap file becomes a node of the lattice. 500 iterations were run and 100 output files as well as simulation time were saved. Figure 6 shows speedup (serial time divided by parallel time) for all cases.

From Fig. 6, it is possible to notice that the parallel code with two processes was around 20% faster than serial code. However, with four or eight processes, one can double performance with parallel code compared to serial run. There is almost no gain for eight processes because the microprocessor used in this test had four cores and eight threads, which confirms that the recommended number of processes is the same number of cores of CPU.

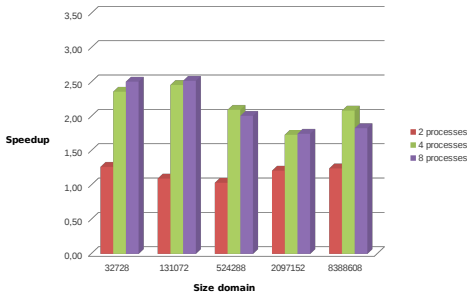
6. CONCLUSION

A 2D Lattice Boltzmann fluid simulator was implemented using MPI and its performance was compared to serial code. Three cases of literature in two dimensions were used: flow between parallel plates, flow past a cylinder and flow past an airfoil. Run times were measured and, with these results, speedups and efficiency were calculated in order to evaluate performance gain. Results showed that this parallel implementation using CPU with MPI is advantageous, and increases with number of processes, until the maximum number of cores of a microprocessor. MPI is worth to achieve performance gain, especially in distributed-memory systems, like clusters, when its use is essential.

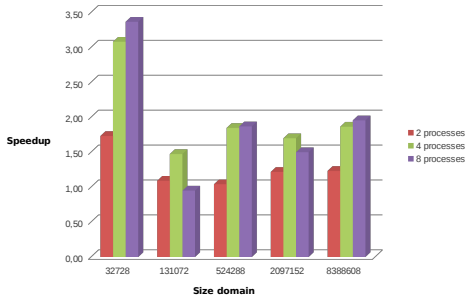
Next step is to run this parallel code adapted for a cluster of CPUs using MPI, so that it is possible to simulate larger domains that requires more memory and achieve even higher speedups, as number of cores increases in a cluster. Another step is parallelize same code to run on Graphics Processing Units (GPUs) to increase the gain performance.

7. ACKNOWLEDGEMENTS

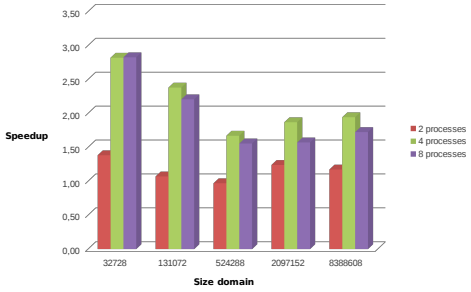
The authors are supported through grants from CAPES, CNPq grant $310474/2009-4$ and FAPESP grant $10/09717-7$.



(a) Speedup for parallel plates



(b) Speedup for cylinder



(c) Speedup for airfoil

Figure 6: Speedup for simulated cases

8. REFERENCES

Aidun, C.K. and Clausen, J.R., 2010. "Lattice-Boltzmann method for complex flows". *Annual review of fluid mechanics*, Vol. 42, pp. 439–472.

Chen, S., Doolen, G.D. and Eggert, K.G., 1994. "Lattice-Boltzmann, a versatile tool for multiphase fluid dynamics and other complicated flows". *Los Alamos Science*, Vol. 22, pp. 99–111.

Latt, J., 2007. "How to implement your DdQq dynamic with only q variables per node (insted of 2q)". Technical report, Tufts University Medford, USA.

Mohamad, A.A., 2011. *The Lattice Boltzmann equation for fluid dynamics and beyond*. Springer.

Oliveira, F.M.C. and Ferreira, L.O.S., 2012. "Simulation of the alcohol-oil mixture in a micromixer using the Lattice Boltzmann method on a GPU device". In *Proceedings of the 14th Brazilian Congress of Thermal Sciences and Engineering*. Rio de Janeito, Brazil.

Pacheco, P., 2011. *An introduction to parallel programming*. Elsevier.

Succi, S., 2001. *Lattice Boltzmann method: fundamentals and engineering applications with computer codes*. Numerical Mathematics and Scientific Computation. Oxford Science Publications.

Sukop, M.C. and Thorne Jr., D.T., 2005. *Lattice Boltzmann modeling: an introduction to geoscientists and engineers*. Springer.

Wolf-Gladrow, D.A., 2005. *Lattice-gas cellular automata and Lattice Boltzmann models - an introduction*, Vol. 1725. Springer.

Zhang, J., 2011. "Lattice Boltzmann method for microfluidics: models and applications". *Microfluid Nanofluid*, Vol. 10.

9. RESPONSIBILITY NOTICE

The authors are the only responsible for the printed material included in this paper.

PERFORMANCE ANALYSIS OF PARALLEL CPU AND GPGPU IMPLEMENTATION OF A 2D MICROFLUIDICS SIMULATOR USING THE LATTICE BOLTZMANN METHOD

Fabíola Martins Campos de Oliveira

Lucas Monteiro Volpe

Luiz Otávio Saraiva Ferreira

fabiola.bass@gmail.com

lucasmvolpe@gmail.com

lotavio@fem.unicamp.br

STATE UNIVERSITY OF CAMPINAS, Faculty of Mechanical Engineering, Department of Computational Mechanics

Rua Mendeleyev 200, CEP: 13083-860, Campinas, SP, Brazil

Abstract. *Recent advancements on GPGPU (General-Purpose Graphics Processing Unit) technology led to a new way of parallel computing, decreasing runtime of several interesting problems for engineering, and making possible an expansion of the problem domain. One example is the fluid mechanics, an area of great economic and academic interest, whose simulations have high computational cost. Thus, a method for simulating fluid flows, which is favorable for use in parallel computing and that is becoming popular is the Lattice Boltzmann Method. As its algorithm is highly parallelizable, simulations based on this method tend to gain efficiency when GPGPUs are utilized. As a first step, this work implements a cpu-based algorithm on C language for measuring efficiency running on a CPU multicore processor and, later, this algorithm is converted into a parallelized gpgpu-based code running on a GPU (Graphics Processing Unit) using CUDA C (Computer Unified Device Architeture) as programming language. At last, the performance of both fluidic simulators based on the Lattice Boltzmann Method running on the processor and on the GPGPU is compared.*

Keywords: *Lattice Boltzmann Method, Two-dimensional flow, Single-component flow, GPGPU, MPI*

1 INTRODUCTION

Phenomena such as interfacial slip and wetting, negligible on macroscopic systems but essential on microfluidic devices, have difficulties to be incorporated on traditional computational fluid dynamics (CFD) methods, but are conveniently handled by simulation programs based on the Lattice Boltzmann Method (LBM). This method was introduced by McNamara and Zanetti (1988) and, since then, it is being developed and successfully used for solving fluid dynamics problems (Mohamad, 2011).

Fluids are modeled on LBM as fictitious particles that collide and propagate over a lattice, and resulted from the junction of Lattice Gas Cellular Automata to Boltzmann theory of rarefied gas dynamics (Wolf-Gladrow, 2005). It is an atomistic model, in opposition to the continuum mechanics approach of the Navier-Stokes equations. Its main advantages over the conventional CFD methods are the efficient parallelization (Obrecht, 2012), easy handling of multicomponent multiphase flows, reactive flows, and complex boundaries (Succi, 2001). It is being used as a general purpose solver for phenomena like heat transfer, electric fields, magnetic fields, diffusion processes, flows in porous media, and shallow flows (Zhang, 2011).

This work investigates the parallelization efficiency of LBM by comparing the performance of two different implementations of a two-dimensional (2D) LBM simulator targeted for microfluidic systems design: a CPU (Central Processing Units) based implementation and a GPU (Graphics Processing Unit) based implementation.

2 METHODS

Both CPU and GPU implementations of LBM use a nine-velocity vector structure (D2Q9) associated to a single-component BGK model.

2.1 LBM basics

LBM simulation is performed in two steps: streaming and collision. Time, space and momentum discretization of the Boltzmann equation, and projection onto a discrete spatial lattice results on the equation of the Lattice Boltzmann Method (Zhang, 2010, Aidun, 2010) Eq.(1):

$$f_a(\mathbf{x} + \mathbf{e}_a \Delta t, t + \Delta t) = f_a(\mathbf{x}, t) - \frac{[f_a(\mathbf{x}, t) - f_a^{eq}(\mathbf{x}, t)]}{\tau} \tag{1}$$

in which $\mathbf{x}$ is the position of the particle, $\mathbf{e}_a$ is its microscopic velocity, t is the time, $\mathbf{\Delta}t$ is the time step of simulation, $f_a(\mathbf{x}+\mathbf{e}_a\Delta t, t+\Delta t) = f_a(\mathbf{x}, t)$ is the streaming part and $[f_a(\mathbf{x}, t) - f_a^{eq}(\mathbf{x}, t)]/\tau$ is the collision term.

The collision term of Eq.1 came from a simplified model (BGK) introduced in 1954 by Bhatnagar, Gross and Krook.

The discretized Boltzmann equation has the advantage, over its original form, that it may be used for dense fluids. The original Boltzmann equation has application limited to short-range interactions in low-density gas flow (Aidun, 2010). The most used bidimensional LBE model is the

D2Q9 (2 dimensions and 9 velocities), in which microscopic velocity $\mathbf{e}_a(a = 0, ..., 8)$ is restricted to 8 directions plus a rest particle, has only 3 magnitudes, and there is a single particle mass, as shown on the left side of Fig.(1). As on Eq.(1) is assumed unity particle mass, the microscopic velocities and momenta are equivalent.

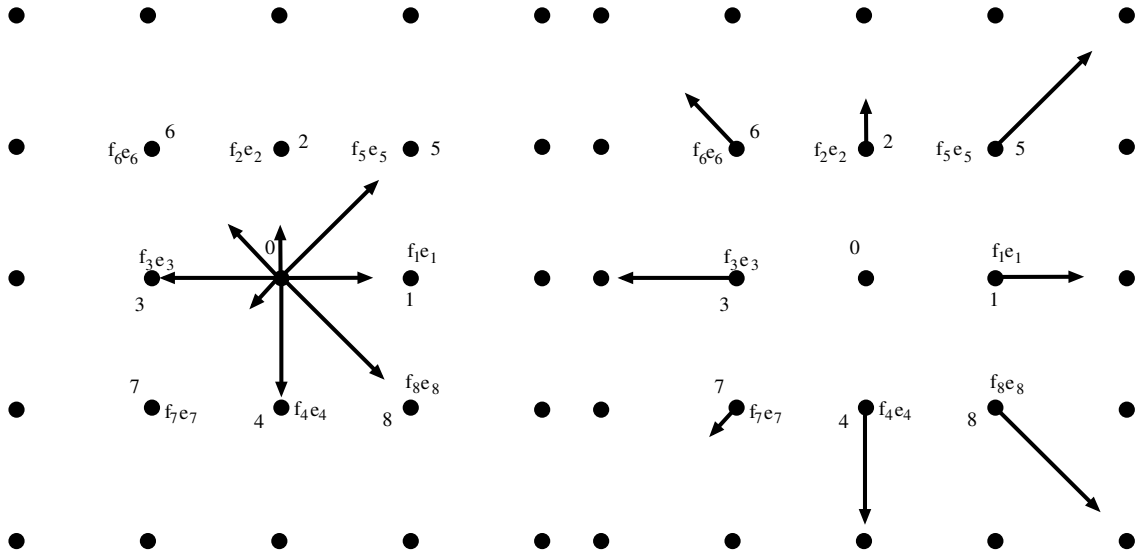


Figure 1: Left: Microscopic velocities of D2Q9 lattice. Each velocity $\mathbf{e}_a$ has an associated frequency f_a. Right: Streaming step on D2Q9 lattice.

The unit of length is the *lattice unit (lu)*, and the time is measured in *lattice time (lt)*. Velocity magnitudes from $\mathbf{e}_1$ to $\mathbf{e}_4$ are $1lu/lt$, and from $\mathbf{e}_5$ to $\mathbf{e}_8$ are $\sqrt{2}lu/lt$. If the distribution function is thought as a frequency of occurrence, its sum results on the macroscopic fluid density Eq.(2):

$$\rho = \sum_{a=0}^{8} f_a \tag{2}$$

and Eq.(3) represents the macroscopic velocity u:

$$\mathbf{u} = \frac{1}{\rho} \sum_{a=0}^{8} f_a \mathbf{e}_a \tag{3}$$

Each density f_a is moved to the nearest neighbor lattice node pointed by the corresponding arrow on the streaming step, as shown on the right side of Fig.(1), changing the values of f for every lattice node. Calculation of the value of the equilibrium distribution function f^{eq} for each node of the lattice, using Eq.(4), is the next step:

$$f_a^{eq}(\mathbf{x}) = w_a \rho(\mathbf{x}) \left[1 + 3\frac{\mathbf{e}_a \cdot \mathbf{u}}{c^2} + \frac{9}{2}\frac{(\mathbf{e}_a \cdot \mathbf{u})^2}{c^4} - \frac{3}{2}\frac{\mathbf{u}^2}{c^2} \right] \tag{4}$$

in which w_a is the weight for each particle: $4/9$ for $a = 0$, $1/9$ for $a = 1, 2, 3, 4$, and $1/36$ for $a = 5, 6, 7, 8$, and c is the lattice sound speed.

The BGK approximation for the collision term of Eq.(1) is calculated for each node of the lattice on the last step. Boundary conditions imposed are described on the next subsection.

2.2 Boundary conditions

Four boundary conditions were used in this simulation: constant velocity at inlet of domain, constant pressure at outlet of domain, periodic condition at top and bottom edges of domain and bounceback for solid nodes.

Constant velocity was implemented setting the particle probabilities distribution functions f_a entering into domain to the result of equilibrium function with initial values of velocity and density from the input parameters. At the outlet of domain, constant pressure was simulated discarding probabilities functions that points outside the domain and getting the values of the last but one node. This way, the outlet works as the exit of a pipe. Periodic domain at the top/bottom edges was implemented through the replacement of the probabilities functions leaving the bottom of domain for those on the top, and functions leaving the top for those of the bottom of domain. Thereby, top and bottom edges nodes will be the same, and the topology of domain becomes cylindrical (Sukop, 2007). At last, bounceback condition for solid nodes was used with all probabilities functions being reversed in the presence of a solid node. The same boundary conditions were implemented on both CPU and GPU code. The next subsections approach aspects of each implementation.

2.3 CPU implementation

The CPU code was implemented in C language to run on a CPU multicore processor using the Message-Passing Interface (MPI) extension. MPI has type definitions, functions and macros so that it is possible to specify work for each core of a microprocessor or cluster of CPUs (Pacheco, 2011). It was designed for distributed-memory systems, when each core or node has its own memory not acessible for the others. Thus, cores need to communicate in order to exchange information to solve a problem together. Most parts of the code uses data-parallelism, meaning that the domain is divided among the cores and each core works on its part of the domain. In the end of the streaming step, the left and the right borders of the domain are sent to the right core that should contain that border. In other cases, like reading the input parameters and printing results on the screen, task-parallelism is applied, as only one core does this work and the others can do another task, aiming higher performance. Some boundary conditions also use task-parallelism.

First, the code reads the input parameters formed by initial velocities in x and y axis, viscosity and density of the fluid and how many iterations and saved files the user wants to have. Also, a monochromatic bitmap is read to extract its width and height dimensions and solid nodes, being black pixels a solid node, and white pixels, fluid nodes. Parameters read are showed at the screen as well as duration time, time for saving files and performance (in MLUPS - Millions of Lattice Updates Per Second) for each saved iteration. These data is also stored in a data file. After that, data is initialized according to the Lattice Boltzmann Method, and simulation starts. For each step, all cores save their part of data into a vtk file, that can be seen with viewer programs, such as Paraview. After that, all cores perform the streaming step, border exchange step, boundary conditions step and collision step. When each of these steps end, a barrier is placed between them so that all cores

reach this point of code before proceed, so that there is not one core updating in the streaming step while other core is already sending and receiving old data from the first core. Output files contain information about the magnitude of velocity of each node. In the next subsection, implementation details of GPU code are explored.

2.4 GPU implementation

The GPU code was implemented in CUDA C language to run on a GPU. In recent years, GPUs has increased its computational capability related to number of operations per second as well as memory bandwidth (Sanders, 2011). This higher performance is due to simpler controller and greater quantity of cores, which apply the same operation in a large amount of data. CUDA C is an extension for C language and was developed for general-purpose programming, bringing this high performance in graphics computing to other areas, which benefit from data-parallelism.

In this code, streaming, boundary conditions and collision steps on each node are performed in parallel. When using GPU, there is no need for correcting the domain after the streaming step, all data are in its right position (Obrecht, 2011). Input and output data are treated by the CPU. While GPU performs LBM steps, CPU saves the output file, gaining efficiency.

The input parameters are read in the same way of CPU code, and are printed on screen. Data are initialized on CPU, and copied to GPU for the simulation. For each step, CPU saves output data in a vtk file while GPU performs LBM steps. After each set of iterations that the user wants to save, data from GPU are copied back to CPU. Output files also contain velocity values for all lattice nodes.

2.5 Performance evaluation

For simulation of the CPU code, the microprocessor Intel(R) Core (TM) i7 CPU 950 at 3.07 GHz x 8 (four cores, eight threads) was used, in a computer with 6 GB of RAM memory. It has 49 GFLOPS of capability. For the GPU code, the same processor and RAM memory amount were used with a NVIDIA GeForce GTX 560 Ti GPU with 384 cores and 1 GB of RAM memory GDDR5. Its capability is 1.5 TFLOPS.

The Haagen-Poiseuille flow on a slit and flow past a cylinder under unstable Reynold (Re) number were used to validate the fluidic behavior of the simulator and to benchmark both CPU and GPU implementations. The performance was measured on Millions of Lattice Updates Per Second (MLUPS) for domains of dimensions varying from 64 by 16 to 8192 by 2048 cells. Figure 2 shows the fully developed velocity profile for the flow between parallel plates (Haagen-Poiseuille flow) for initial and inlet velocities in x axis of $0.1lu/lt$, viscosity of $0.04lu^2/lt$ and density of 1 lattice node$/lu^2$. It can be seen the parabolic curve that agrees with literature (Succi, 2001) with velocity around $0.15lu/lt$ at position 300 lu (Mohamad, 2011). Figure 3 shows the unstable flow past a cylinder with Re number equal to 105 (Sukop, 2007). It is possible to see the vortex shedding common to this flow.

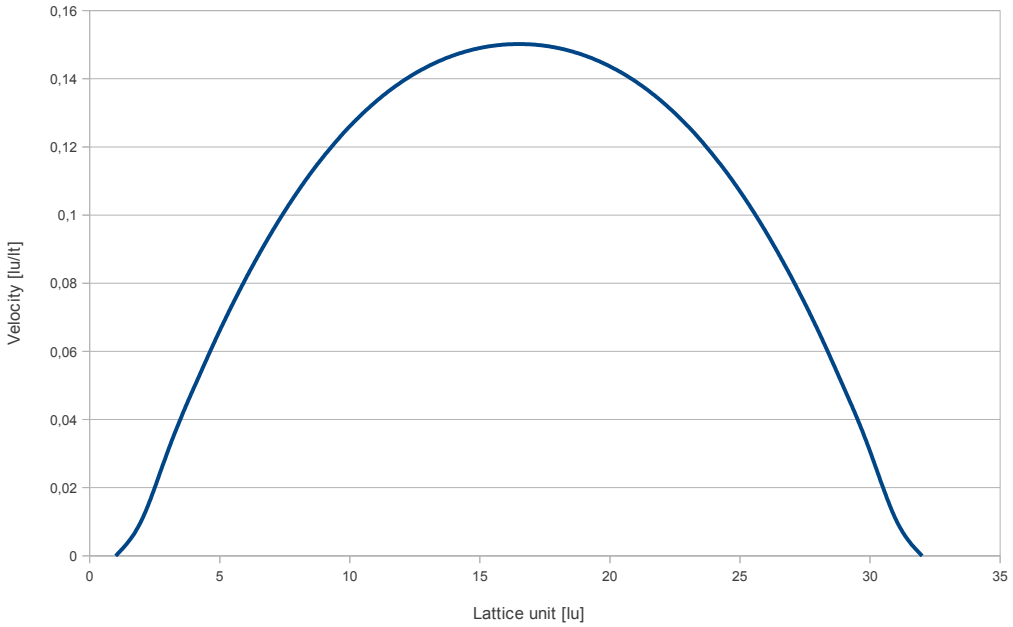


Figure 2: Velocity profile for Haagen-Poiseuille flow.

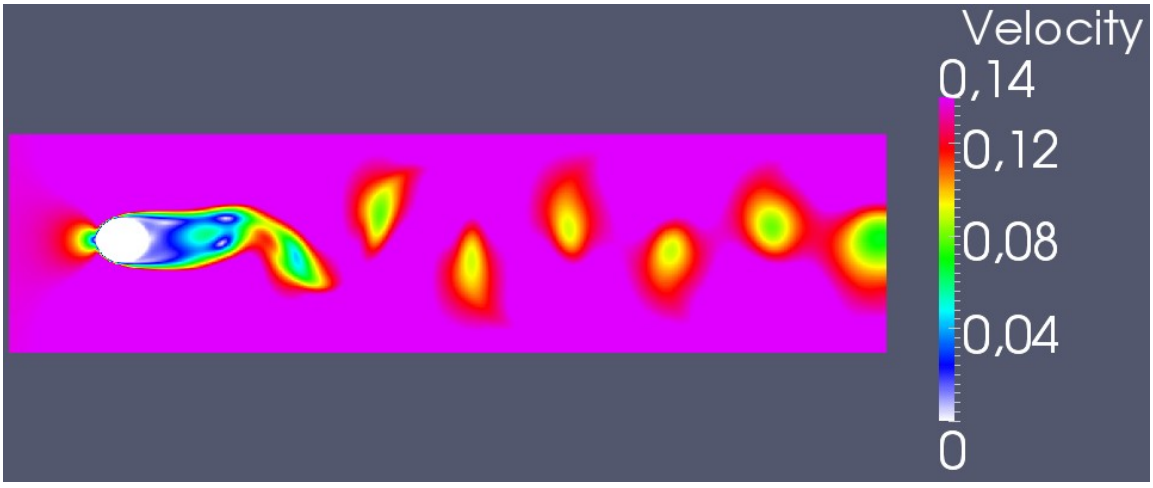


Figure 3: Vortex shedding and von Kármán street at Re = 105 **in unstable flow past a cylinder. The white circle at left is the cylinder**

3 RESULTS AND DISCUSSION

After the simulator validation, a performance evaluation was done with increasing lengths and widths of the domain, so it can be shown how performance of both implementations are affected by the amount of lattice nodes. Four processes were used for the CPU code, as it is the most recommendable value for the microprocessor used in this simulation.

The Haagen-Poiseuille flow was chosen in this stage, as both validation flows would give similar resuts because they have no big difference in the amount of solid nodes. Figure 4 shows the performance for all tests, and a curve fitting them for CPU and GPU implementations. While CPU code had an average of 30 MLUPS, GPU code reached around 620 MLUPS for one million and eight millions of lattice nodes. Performance on GPU also increases with the domain size until a maximum, different from CPU run, that does not have many changes.

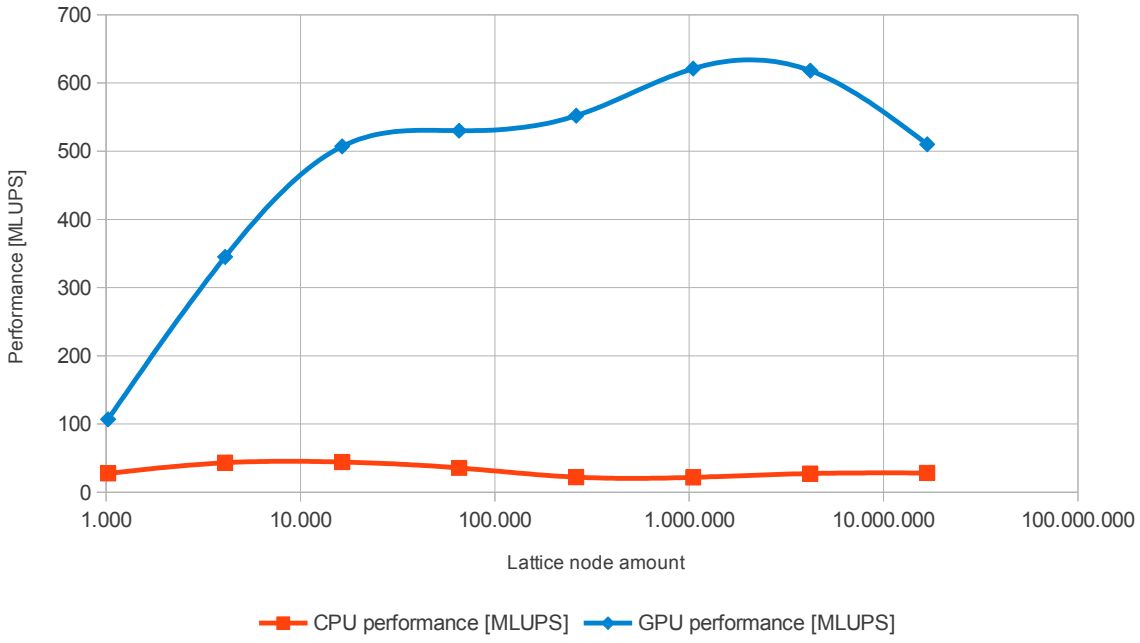


Figure 4: Performance for CPU and GPU implementations. CPU code performs 146 FLOP/lu and GPU code, 136 FLOP/lu (FLOP - floating point operation).

Speedup results are showed in Fig.(5), reaching maximum value of 28 for one million nodes, and minimum value of 4 for one thousand nodes. It is remarkable that implementations of the Lattice Boltzmann Method, a massively data-parallel problem, can be run with higher performance on GPUs parallelization than on CPUs, even using parallel extensions for CPU multicore processors.

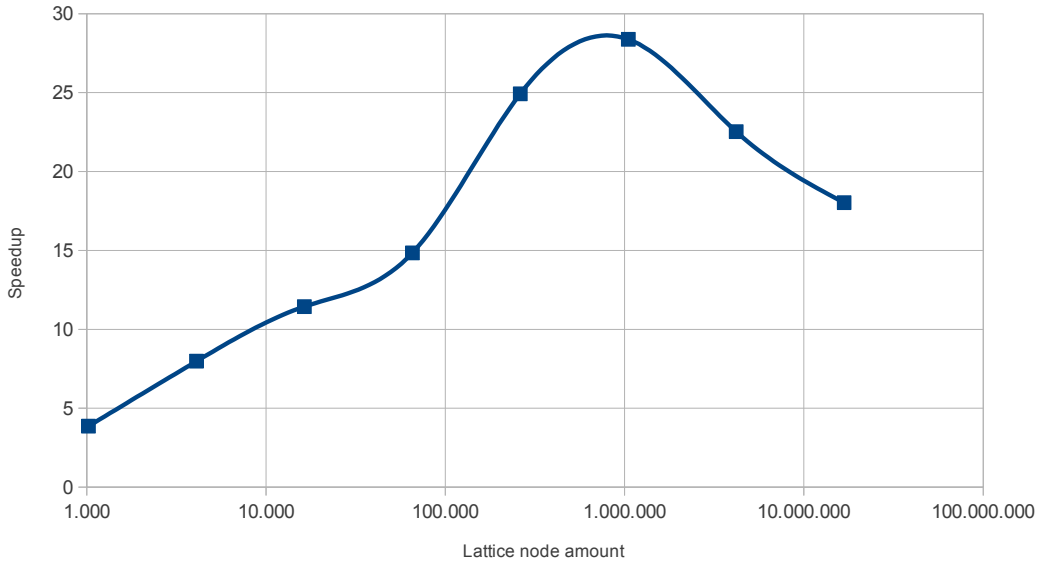


Figure 5: Speedup comparing CPU and GPU code.

4 CONCLUSION

The Lattice Boltzmann Method was implemented for fluid simulation in two dimensions with two distinct approaches: C language with MPI for execution in CPU, and CUDA C language for execution in CPU and GPU. Validation tests were performed in order to verify correctness of both implementations, with two literature flows: Haagen-Poiseuille flow and unstable flow past a cylinder. Simulation results were compared for performance evaluation, showing that, with increasing domain sizes, GPU code has great advantage over CPU code for this method, which is highly data-parallel, being around 18 times faster than CPU code and, for some cases, 28 times faster.

Although CPU code is slower than GPU code, part of its implementation will be blended with GPU code aiming execution on a cluster of GPUs, in which MPI will be fundamental for both intra-node and inter-node parallelism.

REFERENCES

Aidun, C. K. & Clausen, J. R., 2010. Lattice-Boltzmann Method for Complex Flows. *Annual Review of Fluid Mechanics.*

McNamara, G. R. & Zanetti, G., 1998. Use of the Boltzmann equation to simulate lattice-gas automata. *Physical Review Letters.*

Mohamad, A. A., 2011. *Lattice Boltzmann Method Fundamentals and Engineering Applications with Computer Codes*. Springer.

Obrecht, C., Kuznik, F., Tourancheau, B. & Roux, J.-J., 2011. A new approach to the lattice Boltzmann method for graphics processing units. *Computers & Mathematics with Applications*, vol. 61, n. 12, pp. 3628–3638.

Pacheco, P., 2011. *An introduction to parallel programming.* Morgan Kauffmann.

Sanders, J. & Kandrot, E., 2011. *CUDA by Example: an introduction to general-purpose GPU programming*. Addison-Wesley.

Succi, S., 2001. *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond.* Oxford Science Publications.

Sukop, M., & Jr, D. T., 2007. *Lattice Boltzmann modeling: an introduction for geoscientists and engineers*. Springer.

Wolf-Gladrow, D. A., 2005. *Lattice-Gas Cellular Automata and Lattice Boltzmann Models - An Introduction*. Springer.

Zhang, J., 2011. Lattice Boltzmann method for microfluidics: models and applications. *Microfluidics and Nanofluidics*, 1–28.

ACKNOWLEDGEMENTS

This work was supported by CAPES, and by FAPESP grant $2010/09717-7$.