Universidade do Minho
Escola de Engenharia

Simão Pedro Salgado Bernardes Pereira

Parallelization of Flow Modelling Code on a GPU

Tese de Mestrado
Mestrado Integrado em Engenharia de Polímeros

Trabalho efectuado sob a orientação de
Professor Doutor João Miguel Amorim N. C. Nóbrega
Professor Doutor Fernando M. C. Tavares de Pinho

Novembro de 2011

Simão Pedro Salgado Bernardes Pereira: *Paralellization of Flow Modelling Code on a GPU,* Master of Science Thesis in Polymer Engineering, University of Minho, Guimarães, Portugal © November 2011

Supervisors:
Professor João Miguel de Amorim Novais da Costa Nóbrega
Professor Fernando Manuel Coutinho Tavares de Pinho

Location:
Escola de Engenharia da Universidade do Minho
Guimarães, Portugal

An expert is a person who has
made all the mistakes that can
be made in a very narrow field.

NIELS BOHR (1885-1962)

Acknowledgements

I would like to appreciate the support and help given by my supervisor Prof. Miguel Nóbrega during this work and for his encouragement to always go further. The trust my supervisor placed on me to conclude this work filled me with confidence and will to succeed. I am also thankful to the co-supervisor Prof. Fernando Pinho, for his contributions to this work. I cannot forget to mention the support I received from the people at the Delft Institute of Technology, with special emphasis to Kees Lemmens, Prof. Kees Vuik and Rohit Gupta. A very special thanks to all my friends and girlfriend for providing me great moments in life. Last but not least, my full gratitude to my parents for their daily love, education and support. I owe my achievements to them.

The present work was developed under the IPROEXT project PTDC/EME-MFE/113988/2009, thus I would like to acknowledge the Foundation for Science and Technology (FCT) for the financial support.

Abstract

Computational Fluid Dynamics (CFD) tools have a large tradition in modelling the flow of complex materials, including polymer melts, and have been used as valuable tools to aid the design of processing equipment. The evolution of hardware power has allowed to increase the complexity of the problems to be solved, in an adequate time scale required by industrial problems. However, nowadays the demand on larger and more complex physical systems is increasing, and frequently leads to unacceptable computation times. The massive parallel Graphics Processing Units (GPUs) are currently being used to accelerate scientific computations over what can be sometimes achieved by Central Processing Units (CPUs) in parallel, and their impressive performance has attracted the attention of physicists and engineers.

This work comprises the implementations of a serial and a parallel unstructured two-dimensional finite volume codes to model the flow of Newtonian and generalized Newtonian fluids, by using the Bird-Carreau viscosity model. The performance of the parallel implementation over the serial one is compared by assessment of the lid-driven cavity and Poiseuille flow problems. A comparison between Newtonian and non-Newtonian fluids is carried out for Poiseuille flow between parallel plates.

The results show that the parallel code can run up to 14x faster than the CPU counterpart when using a single core for the Poiseuille flow, reducing the computational time from 35h to 2h30min, for the simulation with the finest mesh, whereas for the lid-driven cavity case the speed up factor was 11. This investigation will serve as a guide for future implementation/parallelization of more complex CFD codes, that are being developed by the research group involved in this work.

Resumo

A Mecânica de Fluidos Computacional (MFC) tem vindo a ser usada como ferramenta para modelar o escoamento the materiais complexos, tais como polímeros fundidos, sendo por isso uma ferramenta útil no dimensionamento de equipamento de processamento. A evolução do poder computacional tem permitido resolver problemas de maior complexidade numa escala temporal adequada aos requisitos industriais. Contudo, a necessidade de resolver problemas de maior dimensões e envolvendo sistemas físicos mais complexos tem vindo a aumentar, levando por vezes a tempos de computação inaceitáveis. O enorme paralelismo das Unidades de Processamento Gráfico (ou Graphics Processing Units), que excedem por vezes as capacidades das Unidades Centrais de Processamento em paralelo, tem vindo a ser usado para acelerar o cálculo científico e os incríveis desempenhos obtidos têm cativado o interesse de físicos e engenheiros.

Este trabalho consistiu no desenvolvimento de dois códigos, um em série e outro em paralelo, para modelar o escoamento de fluidos Newtonianos e não-Newtonianos usando o método dos volumes finitos em malhas não estruturadas. O desempenho dos dois códigos é comparado em dois problemas: nomeadamente o escoamento entre placas paralelas e no interior de uma cavidade com parede em movimento. O modelo Newtoniano generalizado com função de viscosidade de Bird-Carreau é comparado com o modelo Newtoniano no caso do escoamento entre placas paralelas.

Os resultados mostram que o código paralelizado em GPU é até 14x mais rápido que o código a correr em CPU, no caso do escoamento entre placas paralelas, reduzindo o tempo de computação de 35h para 2h30min para a malha mais fina, enquanto que o escoamento dirigído por uma parede no interior de uma cavidade quadrada foi acelerado até cerca de 11x. As conclusões desta investigação servirão de guia para a implementação/paralelização futura de códigos mais complexos, que estão em desenvolvimento pelo grupo de investigação envolvido neste trabalho.

Contents

List of Figures

List of Tables

List of Symbols

A_{edge} length of the edge.

H channel height.

L channel length.

S_ϕ property source term.

ΔP pressure drop.

$\Delta\eta$ distance between vertices of an edge.

$\Delta\xi$ distance between centres of adjacent cells.

Δt time step.

Γ diffusivity constant.

$\Psi(r)$ flux limiter function.

α_p pressure relaxation constant.

$\dot{\gamma}$ shear rate.

η dynamic viscosity.

η_0 zero shear viscosity.

η_∞ infinite shear viscosity.

λ Carreau time constant.

$\mathbf{A}$ matrix of coefficients.

$\mathbf{e}_\eta$ unit vector of edge vertices.

$\mathbf{e}_\xi$ unit vector of adjacent cells.

$\mathbf{n}$ normal vector.

$\mathbf{r}_{UD}$ distance bewtween cell ϕ_U and ϕ_D.

$\mathbf{u}$ velocity vector.

ϕ property.

ϕ^{n+1} property in the next time step.

ϕ^n property in the present time step.

ϕ_{C_0} property at the cell with index 0.

ϕ_{C_1} property at the cell with index 1.

ϕ_D property at the Downwind cell.

ϕ_U property at the Upwind cell.

ϕ_{V_0} property at the vertex with index 0.

ϕ_{V_1} property at the vertex with index 1.

ϕ_{edge} property at the edge.

ϕ_{nb} property of neightbour.

ρ density.

τ stress tensor.

a_P contribution to the A matrix from the P cell.

a_{np} contribution to the A matrix from the neightbours.

m power law consistensy index.

n power law index.

p pressure tensor.

p' pressure correction.

p^* guess pressure.

r Sweby's r factor.

u^* guess velocity.

u_{bnd} boundary velocity.

List of Acronyms

CAE	Computer Aided Engineering.
CFD	Computational Fluid Dynamics.
CPU	Central Processing Unit.
DP	Double Precision.
FFT	Fast Fourier Transform.
FVM	Finite Volume Method.
GNF	Generalized Newtonian Fluids.
GPU	Graphics Processing Unit.
MPI	Message Passage Interface.
OpenMP	Open Multi-Processor.
PTT	Phan-Thien-Tanner.
SIMPLE	Semi-Implicit Method for Pressure-Linked Equations.
SP	Single Precision.

Chapter 1

Introduction

This chapter presents a synopsis of the employment of Computer Aided Engineering (CAE) on the field of polymer processing, the evolution of Computational Fluid Dynamics (CFD) approaches and a brief revision of the current state of the art on computations performed with graphics cards, with a special emphasis on CFD applications. The final sections of this chapter comprise the main objectives of this work and the description of the thesis organisation and contents.

1.1 Framework

Since the early nineteen sixties the use of modelling and simulation tools has helped engineers to solve numerous problems in several fields, supported by the developments on computer technology. Over the last three decades, the huge increase on computational capability and the development of innovative modelling methodologies, have improved the feasibility and speed of these tools, leading to the appearance of more powerful and reliable numerical codes, that transformed the traditional engineering design procedure from experimental to numerical based operations. In the near past, the continuous innovation requirements and the demands to solve more complex and large dimensional problems, has led to the development of parallel computing methodologies, in order to solve the problems of interest within acceptable computation times. Among others, this evolution was observed on fields related to the polymer transformation processes.

Currently, new developments are envisaged in computational science fields, mainly due to the attractive performance of graphic cards when dealing with massively parallel algorithms. These new approaches have already been evaluated with success in other areas, and justify its assessment in numerical codes developed to aid the polymer processing area.

1.2 CAE in Polymer Processing

Intrinsically, any polymer transformation process involves the transport of mass, energy and momentum. Considering the injection moulding and extrusion as two of the most relevant polymer transformation processes, it is crucial to have tools that can aid engineers to improve the process productivity. Furthermore, due to the need of constant innovation, the cycle of designing new products, new materials and shaping tools can be accelerated by the use of CAE tools. For several years, a huge effort has been made to model the phenomena involved in polymer transformation processes, aiming to optimise the processing parameters, geometries, materials used and the properties of the final products. Therefore, the industrial relevance of modelling polymer transformation processes is to obtain optimised parameters in a more cost effective manner, than the exclusive experimental one [1]. In these fields, the most relevant problems involve the flow of polymer melts, where the material rheology plays a major role [2].

Considering the two major processing techniques used to obtain polymer based products, the main target in the injection moulding process is to understand the effect of the processing conditions on the morphology developed during the transformation process, because it influences the final properties and performance of the moulded parts [3, 4]. Likewise, in the extrusion process, efforts have been made to model the process from the polymer pellets to the finished products, involving the extruder screw [5,6], die [7,8] and calibrator [9] design. The main purposes of employing numerical modelling tools to aid the extrusion process are: the prediction of end-use product properties, minimisation of processing difficulties, and development of new materials, among others. In both cases, the capability to model accurately the physical phenomena involved is crucial to solve the main problems encountered.

The use of numerical tools in these fields has been increasing progressively for over 50 years, but nowadays the industry requires solutions for more complex problems and, at the same time, demands smaller response times. The most promising way to accomplish both objectives is the parallelization of numerical codes, especially when solving complex problems that usually involve large computational times.

Apart from the macroscale modelling approaches, that enable to predict macroscopic properties, nowadays other lower scale techniques are emerging (atomistic and mesoscale), along with multiscale modelling procedures, that have been following the trends of nanotechnology [10]. The calculation times involved in these classes of problems is currently very large, thus these approaches require additional contributions to the demand for the implementation of parallel algorithms, as it is predictably the only way to intensify the employment of lower scale simulations in the near future.

1.3 Computational Fluid Dynamics

Most of the engineering challenges involving fluid flow, heat and mass transfer and associated phenomena fall into the CFD category, which can be solved through the employment of numerical techniques. Among others, this includes a wide range of applications, like aerodynamics of cars and airplanes, combustion processes, multiphase systems or flows of complex fluids. The most employed numerical techniques used to solve CFD problems can be divided into three different numerical methods: the Finite Difference, the Finite Element and the Spectral Methods. Over the last three decades the Finite Volume Method (FVM), a special form of the finite difference formulation, has been largely used, due to the economy on computational resources, when compared with the finite element counterpart [11, 12].

The employment of any numerical method to solve flow problems always requires the solution of two conservation equations (mass and momentum) and, for specific materials, an additional constitutive equation has to be considered. In the case of problems involving heat transfer the solution of the energy equation is also required [1, 11].

Regarding the flow of polymeric materials, the FVM has been widely used to model the flow of Newtonian and Generalized Newtonian Fluids, such as the power law and Carreau-Yasuda inelastic models. Therefore, the best practises recommend the implementation of Newtonian approaches using pressure-velocity coupling algorithms such as Semi-Implicit Method for Pressure-Linked Equations (SIMPLE) or SIMPLER, before moving forward to more complex viscoelastic flows [13]. However, for the solution of similar problems assuming a viscoelastic behaviour for the material, using models like the Phan-Thien-Tanner (PTT) or Oldroyd-B, some numerical difficulties can be encountered [14].

Despite the difficulties encountered in modelling the flow of viscoelastic fluids, due to the above mentioned numerical complexities, some successful cases have been reported in the literature, using structured and unstructured meshes [14–16]. The employment of unstructured meshes enables the possibility of solving problems involving more complex geometries and/or moving-bodies.

On the other hand, there are still some challenges to solve regarding the large Central Processing Unit (CPU) times usually required to obtain accurate solutions, since precise calculations involve always the need of highly refined meshes, thus increasing the problem dimension and, consequently, the required computational power [17]. Accordingly, the increase of computing speed and memory capacity over the last decade enabled the possibility of solving more accurately, more complex engineering problems.

The significant demands of CFD codes able to tackle computationally heavy problems, coupled to the emerging parallel computer hardware, led to the development of powerful parallel computing approaches. Implementations of parallel CFD codes to model the flow of incompressible Newtonian [18–20] and non-Newtonian [21] fluids have already been successfully achieved. Currently, parallel computing is a common solution for intensive computer simulations, such as those required by complex problems, exemplified by ubiquitous weather predictions.

Over the last five years, a new alternative for parallel computing has emerged, known as Graphics Processing Unit (GPU) computing. The availability of both the hardware and accessible programming tools has attracted the attention of engineers, physicists and mathematicians. Porting numerical code to run on GPUs is a current practice nowadays, but depending on the massive parallelism for massive data applications, the performance can differ significantly [22–24]. A brief description enhancing some of the potentialities and perspectives of GPUs is given below, with a special focus on CFD applications.

1.4 Parallel and GPU Computing

According to Moore's law, the number of transistors in a chip doubled every eighteen months increasing the corresponding processing speed, and for many decades the performance of processors followed this law. However, the increase of processing speed has been limited by power consumption reasons (also related to the heat generation) leading to a new generation of processors that include more than one processing unit, opening the field of parallel computing. In this framework, the growth in complexity of physical problems led to the development of new strategies to overcome the above mentioned difficulties [24, 25].

The traditional way of solving physical problems on a computer is known as serial computing, where several tasks are performed sequentially, thus one task can start only after the end of the previous one. An alternative to this approach is to solve several tasks concurrently, by using several processing units, which is known as parallel computing. The most common way of parallel computing is distributed computing, common in clusters, where independent computers (often called nodes) are connected through a network, used for communication purposes. The most common parallel languages are Message Passage Interface (MPI), Pthreads and Open Multi-Processor (OpenMP) [26]. Parallel computing must involve parallel hardware computing platforms, parallel algorithms and parallel applications. Since nowadays the power of a unique processor is unlikely to change substantially, there has been a huge effort to produce multi-core processors rather than to increase clock speed. On the other hand, recent developments in graphics processing units are taking parallel computing to another level, since GPUs have a higher number of multiprocessors than encountered in CPUs [27]. This is also because GPUs do not have to perform well over such a wide range of applications such as the CPU, therefore it can be said that GPUs are in a sense scale down versions of CPUs optimised for specific tasks. These facts are encouraging a huge effort made by physicists and engineers, to port the data parallelism involved in the numerical codes to be computed in GPUs, mainly due to their availability and low cost over performance ratio and since that appeals to what GPUs do well, which is to chum data.

GPUs were originally developed for rendering high quality images, mainly for the industry of computer games. The game consumers' demand for real-time, high-definition 3D graphics, with realistic scenarios, led to the development of a huge powerful computer graphic architecture, in some cases able to simulate physical phenomena in real time. Examples of these involve the illustration of phenomena with liquids or gases [28], food, elastic, plastic and melting objects [29] and animations for interactive surgical training of the cardiovascular system [30]. All these have been implemented on GPU for real-time applications, mainly in video games. Some of these real-time animations are achieved by applying physical models, such as the Navier-Stokes equations [31, 32]. The list of applications successfully ported to GPUs is large, and is increasing on a daily basis. Parallel computing involving linear algebra operations such as matrix-matrix multiplication or Fast Fourier Transform (FFT) [33], matrix-vector multiplications [34] and QR (Quadratic Residue) decomposition [35] have been implemented and improved, over the last 5 years. However these well know parallelized routines, involve simple operations, and do not solve per se complex computational problems. Developing new algorithms and new methodologies to solve complex problems in parallel architectures is a current challenge.

The massive parallelism available on GPUs offers tremendous performance in many applications. However as reported in a recent survey, in the field of CFD there are some performance limitations upon comparison with other applications [25]. In CFD the first reported implementation of the compressible Euler equations on GPUs can be found in Hagen and Lie [36], indicating speed-ups up to 25x, when compared with CPU code. Brandvik *et al.* [37] reported speed-ups of 29x and 16x when solving the Euler equations using structured grids, for two dimensional and three dimensional flows, respectively. Elsen *et al.* [38] solved also the Euler equations for hypersonic flows with speed-ups of 40x (simple geometries) or 20x (complex geometries). All the previous reported performances were attained by using Single Precision (SP) codes, because the first generation of GPU hardware did not allow Double Precision (DP) computations. For some time, mixed precision techniques were used to solve this problem as reported by Goddeke *et al.* [39]. The first DP GPU calculations of the incompressible Navier-Stokes equations concerned structured grids and speed-ups of 8.5x were achieved when comparing with the CPU implementation [40]. Corrigan *et al.* [41] reported speed-ups of 7.4x using DP with an unstructured cell-centred mesh FVM code, solving the Euler equations. The highest speed-ups were obtained on the implementation of an unstructured mesh vertex-based FVM code developed by Kampolis et al [42], where maximum speed-ups of 28x (single precision), 25x (mixed precision) and 20x (double precision) for a 2D and 3D Navier-Stokes solver, were achieved. More recently Kampolis *et al.* [43] improved that code and achieved a maximum speed up of 46x, with the mixed precision technique, when solving unsteady laminar and turbulent flows.

1.5 Objectives

Over the last decade, some researchers of the Polymer Engineering Department at University of Minho, have been involved on the development of computational rheology tools that are able to model the flow of polymer melts. These tools have been successfully applied to aid the design of processing tools. These numerical codes have been used to solve progressively more complex (both in terms of geometry and constitutive models) and larger dimension problems (involving highly refined meshes). These developments have increased significantly the computation times required, mainly because the numerical codes are based on a serial implementation. Therefore, developments towards parallel computing approaches are necessary, in order to solve these time consuming problems within a reasonable time frame, suitable for an end-user engineer. In this framework, the potentialities of the promising GPUs on parallel computing were already assessed in several individual problems (like FFT and matrix-vector operations), but they should be further tested, such as in computational rheology.

The two main objectives of this MSc Thesis are to develop a serial unstructured 2D version of a cell-centred FVM code for modelling the heat transfer and flow of Newtonian fluids and, subsequently, to parallelize it and evaluate the advantages of its implementation on a GPU over that of its CPU sibling. With this work, a general perspective about the performance gains resulting from running an unstructured flow code on a GPU should be achieved. Note also that the development of the serial finite volume code was carried out in conjunction with another MSc student, Hadi Goldansaz.

1.6 Thesis Organisation and Structure

The present work is focused on the implementation of a cell-centred finite volume code on CPU and GPU to solve the momentum equations for Generalized Newtonian Fluids (GNF) using a SIMPLE based algorithm, to couple velocity and pressure on unstructured meshes. The computational approach employed is described in Chapter 2 along with the implementation details of the CPU code. Chapter 3 introduces some features of the GPU programming model and important aspects about the architecture of graphics cards, and describes the implementation of the numerical code on GPU. The codes are assessed in Chapter 4, using benchmark fluid mechanics problems for Newtonian fluids such as the lid driven cavity flow and Poiseuille flow for validation purposes. A comparison between the performance of the CPU and GPU codes is presented on Chapter 5. This is accomplished by timing the most important routines in order to observe which are the codes' bottlenecks. The Poiseuille flow is also used to assess the implementation of non-Newtonian viscosity function of the GNF model. This thesis ends with Chapter 6, where the main conclusions of the work and proposals for future research are presented.

Chapter 2

Unstructured Finite Volume Code

This chapter describes the methodology employed for the numerical solution of the governing equations describing fluid flow. The approach used follows a conventional Finite Volume Method for 2D unstructured meshes, with a cell-centred arrangement.

2.1 Governing Equations

The finite volume method is a numerical technique that has been largely used to solve the governing equations of fluid flow for Newtonian and non-Newtonian fluids. The idea behind the method is to discretise the governing equations over a control volume and time within a domain to form a system of algebraic equations that can be solved using an iterative method [11, 12]. In this work, the discretisation of the governing equations was made using an unstructured two-dimensional mesh based on triangles as control volumes, with a cell-centred arrangement. The advantage of using unstructured meshes, in comparison with structured meshes, lays in the fact that more complex geometries can be studied without the need for unnecessary mesh refinements [11, 44]. An example of an unstructured mesh using the cell-centred arrangement for all variables is presented in Figure 2.1, applied to the geometry of an extrusion profile. In this case all the main flow variables are stored at the centre of the cells represented by black dots.

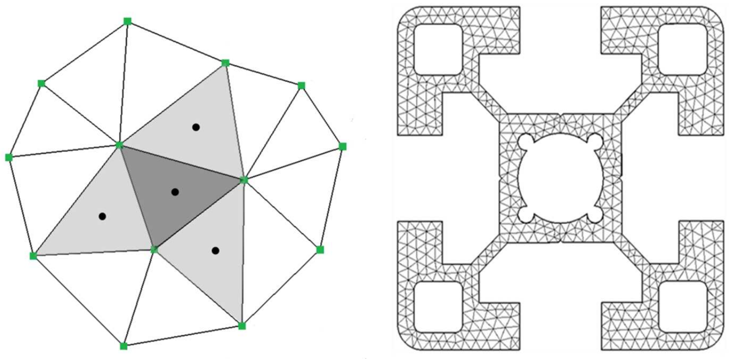


Figure 2.1: Cell-centred control volume (left) and two-dimensional mesh of an extrusion profile (right).

The governing equations for isothermal flow of an incompressible fluid take the form of Equation 2.1 for mass conservation and of Equation 2.2 for momentum, in conjunction with a constitutive equation for the stress tensor.

$$\frac{\partial u_j}{\partial x_j} = 0 \tag{2.1}$$

$$\rho\frac{\partial u_i}{\partial t} + \rho\frac{\partial u_j u_i}{\partial x_j} = -\frac{\partial p}{\partial x_i} + \frac{\partial \tau_{ij}}{\partial x_j} \tag{2.2}$$

The simplest constitutive equation for incompressible fluids is purely viscous and is given by Equation 2.3. If this equations is implicitly integrated in the momentum equation the Navier-Stokes equation for variable viscosity fluids results [11,44].

$$\tau_{ij} = \eta(\dot{\gamma})\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) \tag{2.3}$$

This model is valid for Newtonian and Generalized Newtonian Fluids, *i.e.*, fluids with variable viscosity without any elastic or memory effect. For the Newtonian model function $\eta(\dot{\gamma})$ takes on a constant value, while for the Generalized Newtonian the shear viscosity depends on the shear rate $(\dot{\gamma})$, an invariant quantity related to the rate of strain tensor [11,45], where

$$|\dot{\gamma}| = \sqrt{\frac{\dot{\gamma}_{ij}\dot{\gamma}_{ij}}{2}} \tag{2.4}$$

$$\dot{\gamma}_{ij} = \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \tag{2.5}$$

The viscosity function used in the Generalized Newtonian model to account for the shear-thinning behaviour of polymer melts is a Bird-Carreau model (Eq. 2.6).

$$\eta(\dot{\gamma}) = \eta_\infty + \frac{\eta_0 - \eta_\infty}{(1 + (\lambda\,|\dot{\gamma}|)^2)^{\frac{1-n}{2}}} \tag{2.6}$$

This model uses four empirical parameters and accounts for the low shear and high shear Newtonian regions plateau (η_0 and η_∞, respectively). Its shear viscosity variation with shear rate is shown in Figure 2.2. For the case of a Generalized Newtonian Fluid, the viscosity should be computed at the end of each iteration of the algorithm, since it depends on the rate of deformation.

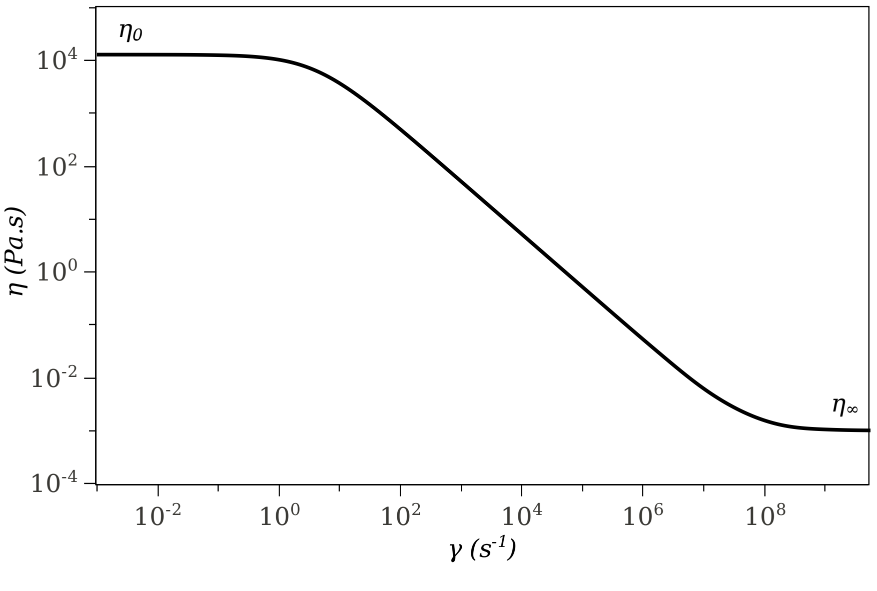


Figure 2.2: Shear viscosity as function of shear rate for the Bird-Carreau model.

2.2 Computational Approach

In the finite volume method the computational domain is divided into control volumes and here we adopt the cell-centred scheme, in which all variables are stored at the centre of the control volume. This is called the collocated arrangement and the pressure and velocity fields are coupled via a special scheme. The 2D control volumes, or cells, considered in this work were triangles, each containing three edges and three vertices. On the other hand two cells straddle every edge (apart from the boundary edges that only have one cell) which contains two entries, see Figure 2.1. These geometrical relationships that arise from the mesh topology are important for the data structure of the code.

2.2.1 General Conservation Equation

A practical approach in the finite volume method is to describe the solution of all governing equations in terms of a single convection-diffusion equation. This generic transport equation for property ϕ equals its rate of change and convection (left-hand side), with its diffusion and possible source terms (right-hand side) [11].

$$\frac{\partial(\rho\phi)}{\partial t} + \nabla \cdot (\rho \mathbf{u} \phi) = \nabla \cdot (\Gamma \nabla \phi) + S_\phi \tag{2.7}$$

Volume integration of Equation 2.7 over the control-volume yields

$$\int_V \rho \frac{\partial \phi}{\partial t} dV + \int_V \nabla \cdot (\rho \mathbf{u} \phi) dV = \int_V \nabla \cdot (\Gamma \nabla \phi) dV + \int_V S_\phi dV \tag{2.8}$$

Integration of the diffusive and convective terms in Equation 2.8 benefits from Gauss's divergence theorem (Eq. 2.9); the integral of the divergence of a vector over a volume is equivalent to the integral over area of the surface projection of that vector.

$$\int_V div(\phi) dV = \int_A (\mathbf{n} \cdot \phi) dA \tag{2.9}$$

Applying Gauss's theorem to Equation 2.8 and summing the contribution from all edges to each control volume yields

$$\int_V \rho \frac{\partial \phi}{\partial t} dV + \sum_{edges} \int_A \rho \mathbf{u}(\mathbf{n} \cdot \phi) dA = \sum_{edges} \int_A \Gamma(\mathbf{n} \cdot \nabla \phi) dA + \int_V S_\phi dV \quad (2.10)$$

The discretisation of each of the terms presented in Equation 2.10 is described separately hereafter. The convective flux at an edge is usually approximated (Eq. 2.11) by the mass flow rate normal to that edge multiplied by some averaged of the convected variable at the edge and is computed always assuming that the mass flow rate per unit area ($\rho \mathbf{u}$) is known from the previous iteration [46]. Thus, a correct evaluation of that representative value of the scalar property ϕ at the edge is crucial.

$$\int_A (\rho \mathbf{u} \cdot \mathbf{n}) \, \phi_{edge} dA \cong (\rho \mathbf{u} \cdot \mathbf{n}) \, \phi_{edge} A_{edge} \quad (2.11)$$

The simplest way to compute the edge property in a convective flux is to use a first-order approximation, by using the value of ϕ at the upstream cell. The use of an upwind-differencing scheme (UDS) to compute the edge scalar property results in a numerically stable method, but the disadvantage is the introduction of the so called false diffusion which can produce less accurate results for coarse meshes. More accurate estimations of scalar properties at the edge can be made by using a second-order approximation (Eq. 2.12), thus requiring the estimation of the gradient of ϕ at the upstream cell. Details on gradient construction techniques are discussed later. In Equation 2.12 the vector $\mathbf{r}_{Uedge}$ is the distance between cell P and the edge.

$$\phi_{edge} = \phi_U + \nabla \phi_U \cdot \Delta \mathbf{r}_{Uedge} \quad (2.12)$$

Another way to estimate the convective fluxes is by means of Total Variational Diminishing (TVD) schemes, which ensure that the interpolation scheme does not increase the total variation of the solution, eliminating numerical oscillations. For a TVD scheme the edge property must be computed by

$$\phi_{edge} = \phi_U + \frac{1}{2}\Psi(r)(\phi_D - \phi_U) \tag{2.13}$$

where $\Psi(r)$ is the flux limiter function. A detailed description of TVD schemes for unstructured meshes can be found in Darwish and Moukalled [47], which recommend the use of an r factor given by Equation 2.14, also known as Sweby's r factor, to compute the flux limiter in some schemes. It is important to notice that the position of upwind and downwind cells to a green central cell varies according to the edge velocity computed by the numerical code (Figure 2.3).

$$r = \frac{(2\nabla\phi_U \cdot \mathbf{r}_{UD})}{(\phi_D - \phi_U)} \tag{2.14}$$

In Equation 2.14, the vector $\mathbf{r}_{UD}$ is the distance between Upstream (U) and Downstream (D) cells (see Figure 2.3). One way to identify a TVD scheme is to analyse the monotonicity region of a r-Ψ plot (Figure 2.4). Any flux-limiter function that lies inside the monotonicity region yields a TVD scheme [47]. A list of flux limiters functions implemented in the numerical code is presented in Table 2.1 and its nomenclature will be used along the text.

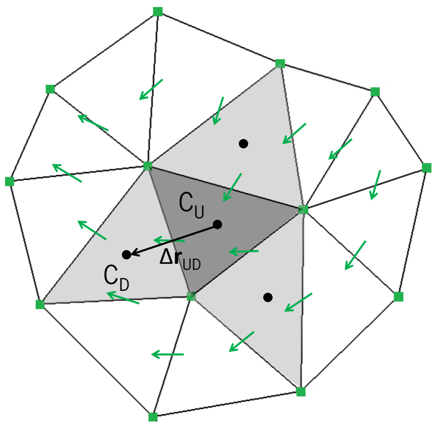


Figure 2.3: Flow direction determines upstream and downstream cells.

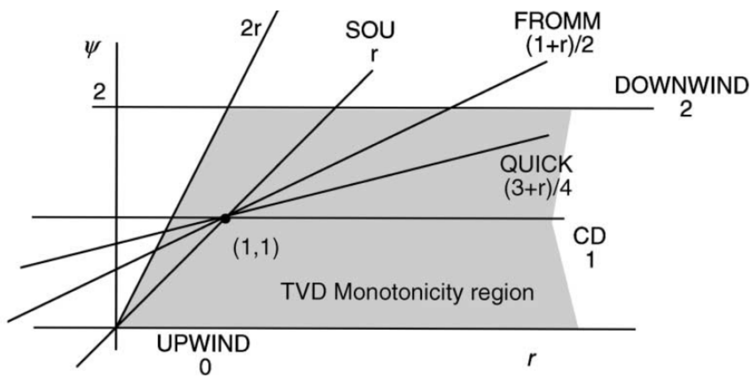


Figure 2.4: TVD monotonicity region.

Table 2.1: List of flux limiters according to the numerical scheme.

ID	**Scheme**	**Flux Limiter**(Ψ)
1	DOWNWIND	2
2	CD	1
3	SOU	r
4	FROMM	(1+3)/2
5	SUPERBEE	$\max[0, \min(2r, 1), \min(r, 2)]$
6	MINMOD	$\max[0, \min(r, 1)]$
7	OSHER	$\max[0, \min(2r, 1)]$
8	MUSCL	$(r + \lvert r \rvert)/(1 + \lvert r \rvert)$
9	QUICK	$\max[0, \min(2r, (1 + 3r)/4, (3 + r)/4), 2]$
10	UPWIND	0

Regarding the diffusive term, the integration over the area of the cell is approximated by the product between the "average" values of the diffusivity (Γ) by the flux vector, taking into account the orientation of the cell face of the control volume on the dot product (Eq. 2.15).

$$\int_A \Gamma(\mathbf{n} \cdot \nabla\phi)dA \cong \Gamma(\mathbf{n} \cdot \nabla\phi)\Delta A_{edge} \tag{2.15}$$

The representative gradient of ϕ is then discretised using the central difference method which is a second-order accurate scheme in uniform meshes according to Taylor's series [11]. In order to account for the non-orthogonality of the mesh, it is required to add a cross-diffusion term following the geometric description of Figure 2.5. The interested reader should consult [11] for more details concerning the calculation of the diffusion term. The final form of the diffusive contribution to the discretised transport equation is given by Equation 2.16.

$$\Gamma(\mathbf{n}\cdot\nabla\phi)\Delta A_{edge} = \Gamma\Delta A_{edge}\left[\left(\frac{\mathbf{n}\cdot\mathbf{n}}{\mathbf{n}\cdot\mathbf{e}_\xi}\frac{\phi_{C_1}-\phi_{C_0}}{\Delta\xi}\right)-\left(\frac{\mathbf{e}_\xi\cdot\mathbf{e}_\eta}{\mathbf{n}\cdot\mathbf{e}_\xi}\frac{\phi_{V_1}-\phi_{V_0}}{\Delta\eta}\right)\right] \tag{2.16}$$

In this equation, the values at nodes (ϕ_V) are obtained, in this work, by simple averaging the neighbouring cells; however more accurate results might be obtained through distance-weighted averages of the centre of cell values [46].

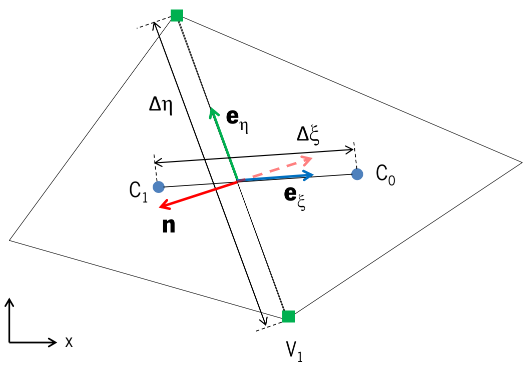


Figure 2.5: Geometric arrangement on cell-centred control volume.

Time integration of Equation 2.10 uses a pseudo-transient approach, by using an implicit scheme only for relaxation purposes of momentum equations. The unsteady term in Equation 2.7 is approximated by the change of total amount of fluid property in the control volume over time and can be computed based on the values of the previous and current time steps [11].

$$\int_t\int_V \rho\frac{\partial\phi}{\partial t}dV dt = \rho\Delta V\left(\phi^{n+1}-\phi^n\right) \tag{2.17}$$

The approximated transport equation can be summarise in Equation 2.18 resultant after dividing both sides of the equation by Δt. Note that the source term is simply approximated as the product of the volume of the control volume by a volume average value of the S_ϕ function, usually taken to be represented by the value of the S_ϕ at the cell centre.

$$\rho \Delta V \frac{(\phi^{n+1} - \phi^{n})}{\Delta t} + \sum_{edges} (\rho \mathbf{u} \cdot \mathbf{n})\, \phi_{edge} \Delta A_{edge} = \sum_{edges} \Gamma(\mathbf{n} \cdot \nabla \phi) \Delta A_{edge} + S_\phi \Delta V \quad (2.18)$$

2.2.2 Gradient Reconstruction

For the computation of the convective term and other contributions presented later like the pressure gradient in the momentum equation (Eq. 2.2), the methodology employed in this work was the least-square gradient reconstruction. This methodology is used to estimate the gradient at the cell by gathering information from the surrounding neighbours (Figure 2.6) and can be assembled in a matrix Equation (Eq. 2.19) [11].

$$\begin{bmatrix} \Delta x_{AP} & \Delta y_{AP} \\ \Delta x_{BP} & \Delta y_{BP} \\ \Delta x_{CP} & \Delta y_{CP} \end{bmatrix} \begin{bmatrix} \frac{\partial \phi_P}{\partial x} \\ \frac{\partial \phi_P}{\partial y} \end{bmatrix} = \begin{bmatrix} \phi_A - \phi_P \\ \phi_B - \phi_P \\ \phi_C - \phi_P \end{bmatrix} \quad (2.19)$$

This represents a system of linear equations in the form of $\mathbf{Ax} = \mathbf{B}$ which must be solved for $\mathbf{x}^T = \left[\frac{\partial \phi_P}{\partial x} \frac{\partial \phi_P}{\partial y}\right]$. The gradient vector $\mathbf{x}$ can be computed after some basic linear algebra operations as given by Equation 2.20.

$$\mathbf{x} = \left(\mathbf{A}^T \mathbf{A}\right)^{-1} \mathbf{A}^T \mathbf{B} \quad (2.20)$$

In the case of highly stretched meshes the least-square gradient does not provide enough accuracy, therefore other techniques must be considered for gradient reconstruction such as QR decomposition method [11].

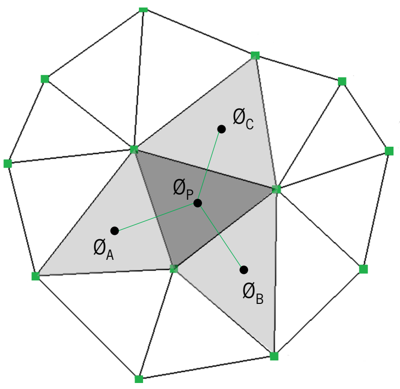


Figure 2.6: Control volume and its neighbour cells.

2.2.3 Boundary Conditions

The boundary conditions are essential to obtain the correct solution of the physical problem. When solving partial partial differential equations the most common boundary conditions are Dirichlet and Neumann boundary conditions. The former gives the value of the unknown function at the boundary while the latter specifies the value of its derivative at the boundary. For instance, in a heat transfer problem, the Dirichlet boundary condition imposes temperature while the Newmann boundary condition corresponds to an imposed wall heat flux.

Regarding confined fluid flow problems, the boundary conditions are commonly designated as inlet, outlet, symmetric, constant pressure or wall. A combination of different boundary conditions is commonly used for solving subsonic flows, but one must be aware that certain combinations are not allowed, such as those that lead to an indeterminate solution. For more details on boundary conditions in the finite volume method the reader can consult the following textbooks [11, 12].

- **Inlet** - Essentially one can use an imposed constant velocity or a known velocity profile. The former boundary condition is considered to be poor when solving steady-state problems, because one must account for the entrance region where the velocity profile develops. Normally, when a constant velocity is imposed a longer entrance region must be provided when compared with an inlet boundary with a known velocity profile, thus increasing the computational time. The inlet boundary conditions implemented in the code are listed below. The analytical solutions for the velocity profiles in a Poiseuille flow between parallel plates with wall to wall distance H, both for Newtonian (Eq. 2.22) and power law (Eq. 2.23) fluids, can be found in the literature [1]. The velocity profiles presented consider a Cartesian referential x-y coordinate system, with an origin at the symmetry plane. The Bird-Carreau model (Eq. 2.6) does not allow an analytical solution for the velocity profile, therefore it is common practise to use a closer known solution, in this case the velocity profile for a power law fluid.

- Constant velocity,

$$u_{bnd} = U \tag{2.21}$$

- Fully developed profile (Newtonian fluid),

$$u_{bnd}(y) = \frac{H^2 \Delta P}{8 \eta L} \left[1 - \left(\frac{2y}{H} \right)^2 \right] \tag{2.22}$$

- Fully developed profile (power law fluid),

$$u_{bnd}(y) = \frac{H}{2\,(n+1)} \left(\frac{H \Delta P}{2mL} \right) \left[1 - \left(\frac{2y}{H} \right)^{(n+1)} \right] \tag{2.23}$$

- **Outlet** - The outlet boundary conditions are required due to the order of the differential governing equations and a common approach to deal with this it is to consider a null gradient of the velocity normal to the flow direction, also designated by fully developed outflow. This assumes that no changes occur in the flow direction at the outlet, therefore this boundary must be located far enough from the region of interest to avoid flow disturbances affecting the solution inside the domain. Also it is essential to guarantee mass conservation within the domain, therefore a flow rate correction is applied to the outlet velocity profile after solving the momentum equations. This flow correction to the outlet velocity is the ratio between the imposed inlet and the extrapolated outlet flow rates.

  – Fully developed outflow

$$\frac{\partial \mathbf{u}}{\partial \mathbf{n}} = 0 \tag{2.24}$$

- **Wall** - In most confined fluid problems this boundary condition is usually applied. The simplest assumption is to consider the no-slip condition, meaning that the velocity components at the wall have the wall velocity, a zero value in the absense of wall motion ($u = 0, v = 0$).

- **Symmetry** - For situations that possess geometric symmetry conditions, it is possible to employ this boundary condition, provided it is known that the flow is also symmetric which allows computational time savings, since just part of the domain is considered. In these situations the velocity at the symmetry boundary is given by:

$$u_{bnd} = u_P - \mathbf{u}_P \cdot \mathbf{n} \tag{2.25}$$

  Note that there are many real situations where symmetric geometries have asymmetric flows.

- **Constant Pressure** - Another way of solving fluid problems is to assign pressure rather than velocities to the boundaries. A combination of inlet velocity and outlet constant pressure (normally zero) is also possible.

It is important to note that if the velocity at the boundary is imposed the pressure must be extrapolated from the interior cells, and if the pressure is fixed the velocities must be extrapolated at each iteration of the calculation process.

The discretisation of the transport equation (Eq. 2.7) at cells next to boundaries follows the same approach for interior cells, by simply replacing the known values at the boundary and treating those terms explicitly, meaning that boundary contributions are included in the source term S_ϕ.

2.2.4 Discretised System of Equations

At this stage, a linear system of equations (Eq. 2.26) is obtained and its numerical solution can be obtained by using an iterative solver. Essentially the system of equations is assembled by the implicit contributions to the cell P (a_P) and a summation over the contributions of the neighbours (a_{nb}). The source term (S_ϕ) contributions arise from all other terms such as sources, pressure gradient in momentum equation (Eq. 2.2), or cross-diffusion in Equation 2.16, or contributions from boundaries.

$$a_P\phi_P = \sum_{nb} a_{np}\phi_{nb} + S_\phi \tag{2.26}$$

2.3 Flow Field Calculation

The methodology presented in Subsection 2.2 is not enough to solve the fluid flow governing equation, due to the nonlinearities of the problem, the necessity to satisfy both the mass and momentum equations, but also due to the absence of an equation to compute the pressure. One way to approach these problems is to use The Semi-Implicit Method for Pressure-Linked Equations (SIMPLE), introduced by Patankar and Spalding in 1972 [11, 48], is an iterative strategy used to bypass the problems mentioned above.

For problems that involve 2D laminar incompressible flows the equations to be solved are the mass conservation (Eq. 2.27) and the linear momentum equations both on x (Eq. 2.28) and y (Eq. 2.29) directions.

$$div(\mathbf{u}) = 0 \tag{2.27}$$

$$\rho\frac{\partial u}{\partial t} + \nabla\cdot(\rho\mathbf{u}u) = \nabla\cdot(\eta\nabla u) - \frac{\mathrm{d}p}{\mathrm{d}x} \tag{2.28}$$

$$\rho\frac{\partial v}{\partial t} + \nabla\cdot(\rho\mathbf{u}v) = \nabla\cdot(\eta\nabla v) - \frac{\mathrm{d}p}{\mathrm{d}y} \tag{2.29}$$

For problems with a known pressure gradient distribution the solution for the momentum equations can be easily computed. On the other hand, in the most familiar case where the pressure gradient is not known an equation to compute the pressure field must be derived. The procedure to solve fluid problems by employing the SIMPLE algorithm is given below.

2.3.1 The SIMPLE Procedure

The initial step is to discretise the x and y momentum equations by following methods described in Section 2.2. The x momentum is presented once again with a description of each term, accordingly to the computational approach given in Section 2.2.

$$\underbrace{\rho\frac{\partial u}{\partial t}}_{\text{Unsteady Term}} + \underbrace{\nabla\cdot(\rho\mathbf{u}u)}_{\text{Convective Term}} = \underbrace{\nabla\cdot(\eta\nabla u)}_{\text{Diffusive Term}} - \underbrace{\frac{\mathrm{d}p}{\mathrm{d}x}}_{\text{Source Term}} \tag{2.30}$$

Please note the similarities between the transport governing equation (Eq. 2.7) and the x momentum equations by simple replacing the ϕ with u. Is also important to emphasise that the pressure gradient is treated as a source term, and was computed based on the least square gradient reconstruction method presented in the Subsection 2.2.2. Without further ado, the discretised momentum equations for x (Eq. 2.31) and y (Eq. 2.32) are presented.

$$\rho\Delta V\frac{u^{n+1}-u^{n}}{\Delta t} + \sum_{edges}\rho u\Delta A_{edge}(\mathbf{u}\cdot\mathbf{n}) = \sum_{edges}\eta\Delta A_{edge}(\nabla u\cdot\mathbf{n}) - \frac{\mathrm{d}p}{\mathrm{d}x}\Delta V \tag{2.31}$$

$$\rho\Delta V\frac{v^{n+1}-v^{n}}{\Delta t} + \sum_{edges}\rho v\Delta A_{edge}(\mathbf{u}\cdot\mathbf{n}) = \sum_{edges}\eta\Delta A_{edge}(\nabla v\cdot\mathbf{n}) - \frac{\mathrm{d}p}{\mathrm{d}y}\Delta V \tag{2.32}$$

The discretised equations are then assembled in the standard linear form:

$$a_P u_P = \sum_{nb} a_{nb}u_{nb} - \frac{\mathrm{d}p}{\mathrm{d}x}\Delta V \tag{2.33}$$

$$a_P v_P = \sum_{nb} a_{nb} v_{nb} - \frac{\mathrm{d}p}{\mathrm{d}y} \Delta V \tag{2.34}$$

Please recall that the velocity variables are stored in the centre of the cells and in order to solve the momentum equations, it is required to know from the previous iteration the edge velocity, *i.e.*, the convective flow at the edge. It is important also to note that the velocities obtained from momentum equations must satisfy the continuity equation. Thus, the discretised continuity equation (Eq. 2.35) is also presented, where $(\mathbf{u} \cdot \mathbf{n})$ is the velocity normal to the edge.

$$\sum_{edges} (\mathbf{u} \cdot \mathbf{n}) A_{edge} = 0 \tag{2.35}$$

Computing the convective flux at the cell edge by simple averaging cell-centred velocities promotes velocity-pressure decoupling. Therefore, Rhie and Chow [49] proposed and interpolation (Eq. 2.36) responsible to link the pressure field to the velocity field, by using values from the previous iteration to estimate the convective flux at the edge. For the computation of the pressure gradients in Equation 2.36 the least square gradient reconstruction was once again used.

$$\begin{aligned}(\mathbf{u} \cdot \mathbf{n})_{edge} = \left(\frac{\mathbf{u}_{C_0} + \mathbf{u}_{C_1}}{2}\right) \cdot \mathbf{n} + \frac{1}{2}\left(\frac{\Delta V_{C_0}}{a_{C_0}} + \frac{\Delta V_{C_1}}{a_{C_1}}\right)\left(\frac{p_{C_0} - p_{C_1}}{\Delta \xi}\right) \\ - \frac{1}{2}\left[\frac{\Delta V_{C_0}}{a_{C_0}} \nabla p_{C_0} + \frac{\Delta V_{C_1}}{a_{C_1}} \nabla p_{C_1}\right] \cdot \mathbf{e}_{\xi}\end{aligned} \tag{2.36}$$

However, the real difficulty lies in the unknown pressure field and on the non-linearity of the convective term of the linear momentum equations, the reason to introduce the SIMPLE algorithm [48]. The idea behind the SIMPLE is to start with a guessed velocity field (u^{0*}) in order to compute the convective fluxes and a guessed pressure field (p^*) to solve the linear momentum conservation equations, obtaining a new velocity field (u^*). This new velocity field does not satisfy continuity, but since the pressure field was a guess, a pressure correction equation can be derived from the continuity equation, to get a conservative field in terms of mass. The new pressure field is used to correct the pressure and velocity fields and the face fluxes.

The process converges when sufficient velocity and pressure corrections are performed, in order to satisfy the chosen stopping criteria for the SIMPLE algorithm. The new velocity field (u^*) obtained with the solution of the linear momentum conservation equation can be used to compute the new normal face velocities using 2.36:

$$(\mathbf{u}\cdot\mathbf{n})^* = \left(\frac{\mathbf{u}^*_{C_0}+\mathbf{u}^*_{C_1}}{2}\right)\cdot\mathbf{n} + \frac{1}{2}\left(\frac{\Delta V_{C_0}}{a_{C_0}}+\frac{\Delta V_{C_1}}{a_{C_1}}\right)\left(\frac{p^*_{C_0}-p^*_{C_1}}{\Delta\xi}\right) - \frac{1}{2}\left[\frac{\Delta V_{C_0}}{a_{C_0}}\nabla p^*_{C_0}+\frac{\Delta V_{C_1}}{a_{C_1}}\nabla p^*_{C_1}\right]\cdot\mathbf{e}_\xi \quad (2.37)$$

If one has the correct pressure field the correct velocity field would be obtained by

$$(\mathbf{u}\cdot\mathbf{n}) = \left(\frac{\mathbf{u}^*{}_{C_0}+\mathbf{u}^*_{C_1}}{2}\right)\cdot\mathbf{n} + \frac{1}{2}\left(\frac{\Delta V_{C_0}}{a_{C_0}}+\frac{\Delta V_{C_1}}{a_{C_1}}\right)\left(\frac{p_{C_0}-p_{C_1}}{\Delta\xi}\right) - \frac{1}{2}\left[\frac{\Delta V_{C_0}}{a_{C_0}}\nabla p^*_{C_0}+\frac{\Delta V_{C_1}}{a_{C_1}}\nabla p^*_{C_1}\right]\cdot\mathbf{e}_\xi \quad (2.38)$$

where asterisks ($*$) denote the "exact" values. Subtracting Equation 2.38 from 2.37 gives

$$(\mathbf{u}\cdot\mathbf{n})' = (\mathbf{u}\cdot\mathbf{n}) - (\mathbf{u}\cdot\mathbf{n})^* = \frac{1}{2}\left(\frac{\Delta V_{C_0}}{a_{C_0}}+\frac{\Delta V_{C_1}}{a_{C_1}}\right)\left(\frac{p'_{C_0}-p'_{C_1}}{\Delta\xi}\right) \quad (2.39)$$

where the superscript $'$ stands for the corrected fields. The mass balance for each cell is then given by

$$\sum_{edges}\left(\left((\mathbf{u}\cdot\mathbf{n})-(\mathbf{u}\cdot\mathbf{n})^*\right)A_{edge}\right) = \sum_{edges}\left(\left(\frac{1}{2}\left(\frac{\Delta V_{C_0}}{a_{C_0}}+\frac{\Delta V_{C_1}}{a_{C_1}}\right)\left(\frac{p'_{C_0}-p'_{C_1}}{\Delta\xi}\right)\right)A_{edge}\right) \quad (2.40)$$

Assuming that the correct velocities will satisfy mass conservative, *i.e.*, no need for pressure corrections, the following algebraic equation can be obtained for each cell

$$\sum_{edges} \left((\mathbf{u} \cdot \mathbf{n})^{*} A_{edge}\right) = \sum_{edges} \left(\left(\frac{1}{2} \left(\frac{\Delta V_{C_0}}{a_{C_0}} + \frac{\Delta V_{C_1}}{a_{C_1}} \right) \left(\frac{p'_{C_0} - p'_{C_1}}{\Delta \xi} \right) \right) A_{edge} \right) \quad (2.41)$$

which contains only fluid and geometric properties and the pressure corrections are the unknowns. The equation can be rewritten as

$$a_P p'_P = \sum_{nb} a_{nb} p'_{nb} + \sum_{edges} \left((\mathbf{u} \cdot \mathbf{n})^{*}\right) A_{edge} \quad (2.42)$$

i.e., a system of equations to compute the pressure correction field. Once this is known, the corrected normal face velocity can be obtained using Equation 2.39 and the corrected cell velocities by Equation 2.33 neglecting the velocity correction of the neighbour cells, that is given by

$$u'_P = -\frac{\mathrm{d}p'}{\mathrm{d}x} \frac{\Delta V}{a_P} \quad (2.43)$$

Due to the neglect of the neighbour's contribution to the velocity correction in Equation 2.43, the SIMPLE algorithm [11,48], comprises an error that vanishes when the problem is close to the convergence, since all the velocity and pressure corrections become null at the final solution. Finally, the pressure field of the next iteration can be obtained by

$$p = p^{*} + p' \quad (2.44)$$

However, the pressure-correction equation is prone to divergence unless some under-relaxation is used. According to literature [48], it is common practice to relax the pressure-correction with a constant parameter α (with $\alpha < 1$). Therefore, the equation to compute the correct pressure (Eq. 2.44) is replaced by the following equation

$$p = p^{*} + \alpha_p p' \quad (2.45)$$

where α_p is the under-relaxation coefficient for pressure. Due to the non-linearity of the convective term the momentum equation an under-relaxation is also used, following the pseudo-transient approach previously introduced. An important remark regarding the pressure-correction equation, is that its coefficient matrix is symmetric and diagonally dominant. These two important properties of the matrix enable the use of the conjugate gradient iterative method, presented later. The procedure for the calculation of the flow field has been presented, thus a summary of the sequential steps is given in the algorithm of Table 2.2. This algorithm is known for its slow convergence rate due to the neglect of the neighbour velocity corrections in equation (2.43), therefore other variants of the method have been developed to essentially minimise the number of iterations.

Table 2.2: SIMPLE algorithm.

Algorithm 2.1 SIMPLE

```
1. Guess a pressure ($p^*$) field.
2. Solve momentum equations to obtain $u^*$ (Eq. 2.33) and $v^*$.
3. Solve pressure-correction equation ($p'$) (Eq. 2.42).
4. Correct pressure (Eq. 2.45).
5. Correct u (Eq. 2.43) and v velocities.
6. Solve other scalar quantities (Ex: Temperature) or update local viscosity (Ex: GNF fluid).
7. return to step 2 assuming the correct pressure ($p$) as a guess pressure ($p^*$).
```

These other methods are the SIMPLER, SIMPLEST or SIMPLEC. Moreover other numerical techniques approach the SIMPLE or SIMPLER as preconditioners to accelerate Krylov subspace where velocity and pressure are assembled into a single matrix and solved concurrently [50].

Exploring other techniques to improve SIMPLE convergence is not within the scope of this work. The numerical procedure to solve fluid problems has been given, and below we discuss the details of the implementation of the code on CPU. However, prior to that the solvers used to calculate the systems of equations are discussed.

2.4 Iterative Solvers

The linear system of equations (Eq. 2.26) previously presented is now given in a more familiar form (Eq. 2.46) where $\mathbf{A}$ is the matrix of coefficients a_P and a_{nb}, $\mathbf{x}$ is the solution and $\mathbf{b}$ is the right hand-side of the system of equations.

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{2.46}$$

In order to solve this linear system one can use direct methods or iterative methods. The latter have received a huge amount of attention for large linear systems, because they can be faster and facilitate parallelism. One condition for an iterative method to converge is that the matrix must be diagonally dominant, meaning that the sum of the magnitudes of non-diagonal entries must not exceed the magnitude of the diagonal [51]. Over the past few years, several linear algebra packages such as BLAS [52] or LAPACK [53] have been developed to enable essentially code portability and high performance computing. Nowadays commercial packages such as Intel's Math Kernel Library (MKL) [54] or open-source PETCS [55] take advantage of BLAS subprograms for parallel implementation of linear algebra routines or iterative solvers, enabling parallel implementations of iterative solvers with minimal effort. Furthermore, the matrices arising from the discretisation process in unstructured meshes are usually sparse (see Subsection 2.4.5). Thus, iterative sparse systems were used to solve the system of equations.

In the code implemented on CPU, the iterative sparse solvers take advantage of MKL library routines for sparse matrix-vector multiplications (*SpMV*), dot product (*ddot*) and scalar multiplication and vector update (*dxapy*). As an example, the following operation computes the matrix-vector product of a sparse matrix ($\mathbf{A}$) and stores the result back in $\mathbf{y}$.

$$\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$$

or in the case of the scalar multiplication and vector update (*dxapy*) the operation is described by

$$\mathbf{y} \leftarrow \alpha\mathbf{x} + \mathbf{y}$$

where α is a scalar and $\mathbf{x}$, $\mathbf{y}$ are vectors. Apart from these simple operations, there are others, and the the interest reader can consult Intel's MKL documentation [54] for more details regarding other operations. They are easily found in many algorithms of iterative solvers, as will be shown below. These routines are essentially optimised for Intel processors and can be used in parallel, taking advantage of the multi-core CPU architecture.

2.4.1 The Jacobi Method

The Jacobi iterative method is considered to be a point-iterative technique for solving linear systems. Basically this method rearranges the system of equations in a way that the contributions due to (x_i) are on the left side of the linear system and the other contributions are placed on the right-hand side. The previous solution (iteration $k-1$) is assumed to be known on the right-hand side, while the current solution (iteration k) is computed according to Equation 2.47 [11].

$$x_i^k = \sum_{\substack{j=1 \\ j \neq i}}^{n} \left(\frac{-a_{ij}}{a_{ii}} \right) x_i^{k-1} + \frac{b_i}{a_{ii}} \qquad (i = 0, 1, 2, ..., n) \tag{2.47}$$

2.4.2 The Conjugate Gradient Method

The Conjugate Gradient (CG) algorithm in Table 2.3 is one of the most famous iterative techniques to solve linear systems as long as the matrix is symmetric and positive definitive (or diagonally dominant). Details about this technique are not given here but full details can found in Shewchuk [56].

As mentioned previously the algorithms for iterative solvers have been designed to use simple and highly optimised operations such as sparse matrix-vector multiplications (*SpMV*), dot product (*ddot*) and scalar multiplication and vector update (*dxapy*), among others. By taking a closer look at step 4, the reader can identify a *SpMV* operation (Ap_i) followed by two dot products operations $((r_i, r_i)$ and $(Ap_i, p_i))$. Note that the result of the *SpMV* is a vector and its result is stored in α_i. To summarise, at each step of the iterative solver, the CG algorithm makes use of 1 *SpMV*, 4 *ddot* and 4 *dxapy* operations (step 8 requires two).

Table 2.3: Conjugate Gradient (CG) algorithm.

Algorithm 2.2 Conjugate Gradient

```
1. Compute $r_0 = b - Ax_0$
2. $p_0 = r_0$
3. For $i = 0, 1, 2, ...$ until convergence, Do
4.     $\alpha_i = (r_i, r_i)/(Ap_i, p_i)$
5.     $x_{i+1} = x_i + \alpha_i p_i$
6.     $r_{i+1} = r_i - \alpha_i p_i$
7.     $\beta_i = (r_{i+1}, r_{i+1})/(r_i, r_i)$
8.     $p_{i+1} = r_i - \beta_i p_i$
9. endDo
```

2.4.3 The Bi-Conjugate Gradient Stabilised Method

The Bi-Conjugate Gradient Stabilised Method (BICGSTAB) is an iterative method that can be applied to general matrices, in contrast to the CG method (Table 2.3).

Both methods use Krylov subspace methods to compute a solution for a system of equations [51]. Essentially, Krylov subspace methods find a solution by minimising the residuals over a subspace created with information given by the matrix. Just like CG method, the BICGSTAB described in Table 2.4 also makes use of BLAS operations. They can be summarised in 1 *SpMV*, 6 *ddot* and 9 *dxapy* operations. Other operations can be used to simplify even more the computations such as the product of a vector by a scalar (see step 10).

The idea was to clarify the reader about the implementation of the BLAS operations and how can they be identified in the algorithms of the iterative solvers. This will be also used later when coding the GPUs.

Table 2.4: Bi-Conjugate Gradient Stabilised Method (BICGSTAB) algorithm.

```
Algorithm 2.3 Bi-CGSTAB
1. Compute $r_0 = b - Ax_0, r_0^*$ arbitrary
2. $p_0 = r_0$
3. For $i = 0, 1, 2, ...$ until convergence, Do
4.      $\alpha_i = (r_i, r_0^*)/(Ap_i, r_0^*)$
5.      $s_i = r_i - \alpha_i p_i$
6.      $\omega_i = (As_i, s_i)/(As_i, As_i)$
7.      $x_{i+1} = x_i + \alpha_i p_i + \omega_i s_i$
8.      $r_{i+1} = s_i - \alpha_i p_i$
9.      $\beta_i = \frac{(r_{i+1}, r_0^*)}{(r_i, r_0^*)} \times \frac{\alpha_i}{\omega_i}$
10.     $p_{i+1} = r_{i+1} + \beta_i(p_i - \omega_i Ap_i)$
11. endDo
```

2.4.4 Stopping Criteria

The previous subsections dealt with the iterative methods, however no stopping criteria were mentioned. Essentially, two stopping criteria are presented, but others can be used depending on the iterative method and on its convergence behaviour. The simplest stopping criterion is the norm of the residuals must fall below a predefined value (Eq. 2.48). Criterion 1 does not scale with problem size and requires a proper choice for the tolerance ϵ. The other stopping criterion implemented (Eq. 2.49) is based on the ratio between the initial and current residual. This ratio must decrease until a predefined valued ϵ in a number of iterations. The main drawback of criterion 2 is if the initial estimate is better than the computed solutions, the iterative solver might not converge.

- Criterion 1:

$$\left|\left|\mathbf{b} - \mathbf{A}\mathbf{x}^i\right|\right|_2 \leq \epsilon \tag{2.48}$$

- Criterion 2:

$$\frac{\left|\left|\mathbf{b} - \mathbf{A}\mathbf{x}^i\right|\right|_2}{\left|\left|\mathbf{b} - \mathbf{A}\mathbf{x}^0\right|\right|_2} \leq \epsilon \tag{2.49}$$

2.4.5 Sparse Matrix Format

A matrix is usually considered sparse when there is a considerable number of zero entries, thus it is computationally expensive to store huge matrices with a large number of zeros, that often arise from finite differences or finite volume discretisations [51]. The introduction of sparse matrices is crucial for the meshes employed, because only the non-zero values of the matrices are stored. There are several ways to store data for the system of equations, but over the years several efficient formats have been proposed to improve performance and portability of the codes. The most common data storage scheme is the Compressed Sparse Row (CSR), which requires the storage of an array with real numbers with the non-zeros values of the matrix $\mathbf{A}$ into $\mathbf{A_{sp}}$, an array of integers $\mathbf{A_i}$ containing the column indices's of the non-zero elements $\mathbf{a_{ij}}$, plus an array of pointers $\mathbf{A_j}$ indicating the index of the element in the $\mathbf{A_{sp}}$ array that is the first non-zero element in a row j. As an example, consider below the matrix $\mathbf{A}$ represented in the CSR format with zero-based index [51]. Note that an array, in C programming language, begins with index zero, therefore the first entries in $\mathbf{A_i}$ and $\mathbf{A_j}$ are zero which correspond to the column index zero and to the first non-zero element of $\mathbf{A_{sp}}$, respectively.

$$\mathbf{A} = \begin{bmatrix} 1.0 & 0.2 & 0.0 & 0.0 \\ 0.0 & 1.5 & 0.5 & 0.8 \\ 1.7 & 0.0 & 3.0 & 0.0 \\ 0.0 & 0.0 & 2.3 & 2.0 \end{bmatrix}$$

$$\mathbf{A_{sp}} = [1.0, 0.2, 1.5, 0.5, 0.8, 1.7, 3.0, 2.3, 2.0]$$

$$\mathbf{A_i} = [0, 1, 1, 2, 3, 0, 2, 2, 3]$$

$$\mathbf{A_j} = [0, 2, 5, 7, 9]$$

The sparse storage format can influence the code performance, especially in parallel implementations, therefore for the GPU implementation other sparse formats are presented later and references to the performance differences with different storage formats are given.

2.5 Code Description

The flow chart of the developed numerical serial code is illustrated in Figure 2.7 and shows the code organisation, which employs the SIMPLE algorithm, previously introduced, to solve fluid problems.

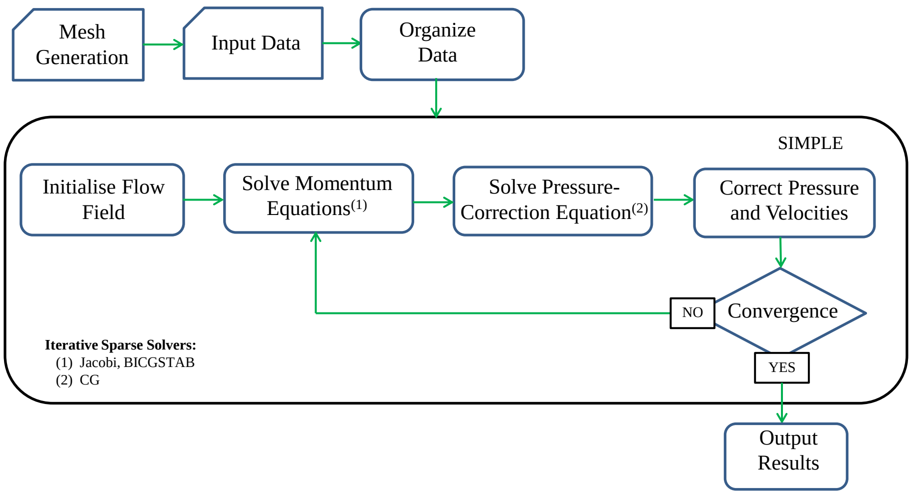


Figure 2.7: Flow chart for the serial code on CPU.

The process of mesh generation was performed with an open-source software (*gmsh*) [57] whose description and main details can be found in Appendix A. In order to run the program, two input files are required. One is the geometry file to generate the mesh and another is a text file with basic information such as boundary conditions, convergence criteria, fluid properties among others. A simple description of the second input file and its features is given in Appendix B.

The developed numerical code is not fully presented due to its long extension. Instead, the curious reader with interest on how the serial code was built, with focus on the most important routines and relevant implementation details must consult Appendix C. The results were visualised on the *gmsh* which also enables post-processing data, for fast result analysis. *Paraview* [58], another open source software was used, for more detailed post-processing analysis. *Paraview* has a stream tracer filter useful to plot streamlines and a brief description on how to use it is presented in Appendix

D. The developed numerical code, implemented on the CPU to run on a single processor (core), was written in the C programming language and compiled with the GNU compiler gcc. Nowadays, compilers try to optimise the code with internal tricks in order to accelerate the numerical codes without user interference. An optional optimisation flag -O3 can be included at compilation time, to reduce code size, execution time and many other optimisations are possible. The difference in execution time of numerical code with the -O3 optimisation regarding non-optimised one compilation can differ by up to a factor of 3, approximately. For more details about the gcc compiler and the optimisation options please refer to the gcc documentation [59]. The CPU used for the computations is an Intel Core i7-950 with 4 cores and 8 threads and in all cases the compilation of the CPU code were carried out with the -O3 optimisation flag.

Chapter 3

GPU Computing

This chapter introduces the GPU computing environment and presents the implementation details of the developed numerical modelling code on GPU, together with some relevant optimisations for realistic performance gains.

3.1 Graphics Processing Units

The desire for real-time, high-definition and realistic graphics in the gaming industry has lead to evolution of the graphics cards into devices with enormous parallel computing capabilities. GPUs are now considered to be highly parallel multi-core processors with tremendous floating point performance peak (GFLOPS[1]) and higher memory bandwidth[2] (GB/s) when compared to a CPU. Modern computers have nowadays more than one core, moving towards multi-core systems, however the performance of the GPU can be considerably higher than a single CPU while being available for a comparable price. The revolutionary era of GPU computing started with the G80 architecture introduced by NVIDIA in 2006 [27, 60]. Programming a GPU has become an easy task since support for C programming language became available[3]. Therefore, using GPUs in non-graphical applications has become

[1]Floating point operations per second (FLOPS) is a measure of processor's performance. 1 GFLOPS = 10^9 FLOPS.

[2]Bandwidth is the amount of data that can be transferred in a given amount of time.

[3]In more recent architectures other common languages can also be used like Java, Fortran, Perl, etc.

a common practice essentially in scientific computing to accelerate numerical codes with incredible speed-ups, up to 100x in some applications [61, 62]. The demands for GPUs outside the graphic applications has lead to a revision of the G80 architecture and in 2008 the second generation of these devices emerged (GT200) basically increasing the number of cores (up to 240) and introducing double-precision support, necessary for most scientific computations. The downside of the latter new feature was the performance limitation when using double-precision in the computations, when compared to the incredible speed-ups using single precision, as their cards where mainly targeted at the gaming industry [63]. Apart from this, in order to achieve high performance, applications required huge amount of data parallelism and data coalesced accesses[4] to the memory in order to take advantage of graphics cards' architecture. The first two generations of graphics cards explored several applications and encountered some limitations. The extensive feedback given by the GPU computing community focused on the improvement, for instance, of double-precision performance, cache hierarchy, more shared memory, among others, as will be discussed along the text and consequently the requirements of computational science lead to the development of an architecture essentially dedicated to scientific computations, the Fermi architecture. The first two generations of graphics cards are not explored in detail since the G80 architecture does not support double-precision computations and the GT200 has performance limitations when operating in double-precision. Since accurate computational fluid dynamics requires double-precision floating point operations, the focus is on the Fermi architecture [63].

3.2 GPU Architecture and CUDA Programming Model

The introduction of the Compute Unified Device Architecture (CUDA) introduced by NVIDIA in 2006 partly helped recently the computational power

[4]Coalesced accesses to memory takes place when all threads access consecutive memory addresses, requiring a single memory transaction. This is only possible with regular data structures.

of GPUs to exceed significantly the power of multi-core CPUs, in some applications. CUDA is a general parallel computing architecture with its own programming and execution models [60]. The parallel architecture of the GPU allows the use of CUDA to solve many computational intensive problems faster than with the CPU counterpart. A description of the GPU architecture is given, focusing essentially on the Fermi architecture. Furthermore the CUDA programming and execution models are presented. The first time reader may need to read this chapter more than once, since to understand the GPU architecture it helps to understand the execution model and *vice versa*.

The Fermi architecture introduced the first GPU suitable for scientific computations, while the previous generations of GPU architectures lacked some important features in order to be suitable for a wide range of scientific computations, such as double precision support [63]. Basically, the GPU architecture is composed by a number of Streaming Multiprocessors (SMs) each containing a number of Streaming Processor (SP) cores, usually referred to as CUDA cores. The Fermi architecture features up to 512 CUDA cores organised into SMs of 32 cores each. This arrangement of 32 cores (green) per each of the 16 SMs is illustrated in Figure 3.1, where a general view of the Fermi architecture is presented.

The GPU has its own DRAM memory that can be up to 6 GB depending on the graphic card, and exchanging data between the host (CPU) and the device (GPU) memories is done via a PCI-Express interface. On the device (GPU) there are different memories that can be used and controlled by the programmer, from fast to slow memories, depending on the data to be accessed. In order to use fast memories a certain regularity in the data is required and not all algorithms can take advantage of these fast memories, just like not everybody or road condition can take advantage of a Ferrari. Even if one can afford it, driving a Ferrari in a road full of potholes is not recommended. Something similar happens with the Fermi architecture, because for algorithms with random memory accesses the use of the fast memories becomes rather difficult. The only way of the previous algorithms retrieve data is by accessing directly the DRAM, also designated as global memory.

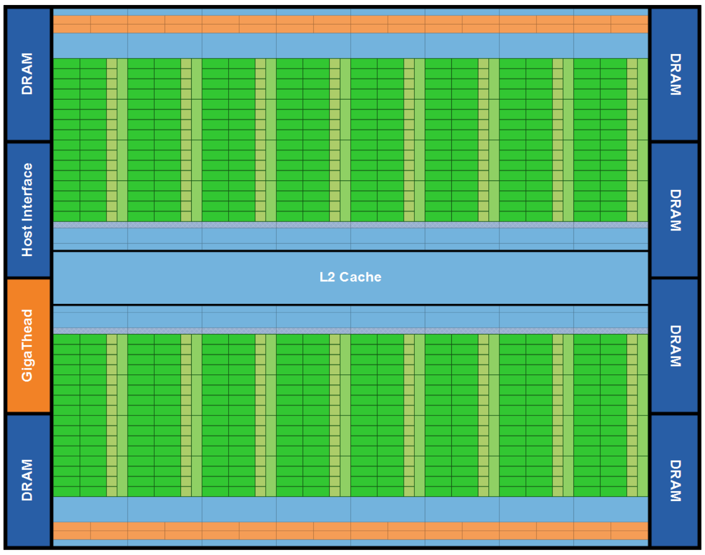


Figure 3.1: Fermi architecture (adapted from [63]).

All Streaming Multiprocessors (SMs) have access to the global memory, but disadvantage of accessing directly the global memory is the high latency,[5] of several hundreds of clock cycles. In other words, global memory accesses are slow. Luckily, in the Fermi architecture the SMs are positioned around a L2 cache[6] (see Figure 3.1) of 768 KB which enables speedy data sharing between threads[7] in different SMs. More details on memory hierarchy are discussed later. The different kinds of memories with various sizes and distinct access times are associated with bandwidth and latency metrics. Latency plays an important role, especially when dealing with small amounts of data. One of the strategies implicit in GPUs' architecture is to utilise the computational power of the cores by increasing the parallelism to hide the memory latency. This can only be achieved when dealing with huge amounts of data. The downside can be a memory bound problem, where the bandwidth becomes a bottleneck in respect to the processor capabilities. The Fermi architecture

[5]Latency can be seen as the time between a task initialisation and the time it begins to execute it. Latency is also designated as a delay of the start-up time of a task.

[6]Cache is a high-speed memory used to store data that is likely to be used again. In this case the L2 cache is a copy of data from global memory which enables fast memory accesses to all threads.

[7]In CUDA, a thread is an element of data to be processed.

also has a GigaThread global scheduler that distributes the amount of work to the thread schedulers in SMs.

A closer look at each SM (Figure 3.2) shows features such as 32 CUDA cores, a dual warp scheduler, two instruction units, 16 load/store units for memory operations, 64 KB configurable L1 cache and shared memory, 4 Special Function Units (SFU) for instructions such as *sin*, *cos* or square root and other features. In the Fermi architecture, each SM schedules threads in groups of 32 parallel threads called warps, which are the minimum size of the data processed. The dual warp scheduler and the two dispatch units enable the launch of two warps concurrently. A warp scheduler can issue an instruction to only half of the CUDA cores, to 16 load/store units or to 4 SFUs. In order to execute an instruction for all threads in a warp two clock cycles are required for an integer or floating point arithmetic instruction. In the case of double precision instructions the dual instruction dispatch is not supported.

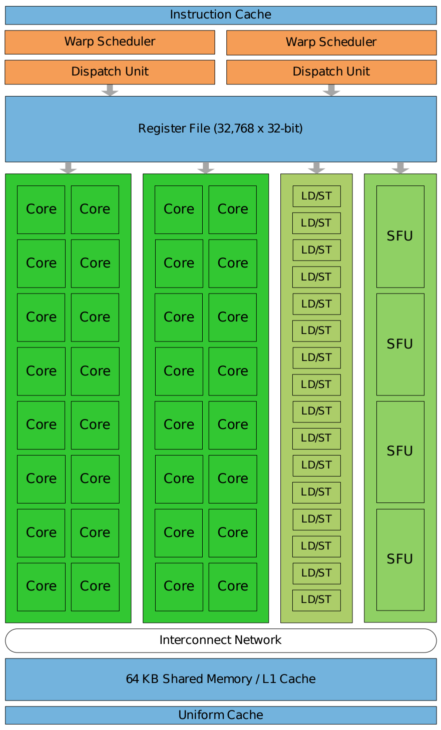


Figure 3.2: Streaming multiprocessor (SM) (adapted from [63]).

The graphic cards used in this work were the NVIDIA GeForce GTX 480 and the Tesla C2070. A comparison between the two graphic cards is given in Table 3.1. It can be seen that the GTX 480 has lower double precision floating point performance peak in comparison with the Tesla. On the other hand the GTX 480 has higher memory bandwidth which can be an advantage for memory bound problems such as sparse matrix-vector multiplication. Memory bound applications are limited by the speed of the memory accesses and not by the speed of computations.

Table 3.1: Features of Fermi GPUs used for the computations.

Feature	**GTX 480**	**Tesla C2070**
CUDA cores	480	448
Core Frequency	1.4 GHZ	1.15 GHZ
GFLOPS (single)	1345	1030
GFLOPS (double)	168	512
Memory	1.5 GB GDDR5	6 GB GDDR5
Memory Bandwidth	177 GB/s	144 GB/s
Power Consumption	250 W	238 W
IEEE single/double	yes/yes	yes/yes

One of the limitations of GTX 480 is the total dedicated memory which limits substantially the dimension of the problems to be solved on this GPU. Despite the fact that the GTX 480 has more limitations in respect to the Tesla, the former is approximately 10 times cheaper than the latter.

3.2.1 Memory Hierarchy

The memory hierarchy in GPUs is composed of different memories ranging from small size memories with low latency to large size memories with high latency. The global memory is the off-chip memory mentioned before and is accessible to all threads in each SMs with the disadvantage of long access times. Access to data in global memory have the lifetime of the application. For read only memory access a constant memory can be used however its size (64 KB) is quite limited and the lifetime is also that of the application. For faster memory accesses, the user can use the shared memory, which is an on-chip read/write memory, a sort of cache controlled by the programmer. On Fermi cards, shared memory can automatically cache global memory accesses without programmer's intervention. The shared memory can only be accessed by threads within a block[8] and has latencies of 1-2 clock cycles. Benefits from shared memory can only be obtained if the number of arithmetic operations is higher than the number of read and write memory accesses. In the case of memory bound problems the use of shared memory might not introduce considerable benefits [60, 63]. Registers are also on-chip read/write memory (32 KB for the GPUs mentioned above), which are incredibly fast (latency of 1 clock cycle) and have the life time of a thread. Another type of memory usually used is the texture memory but this was not explored in this work. The interested reader should consult the CUDA programming guide for more details [60].

In the Fermi architecture, there is an extended memory hierarchy essentially for parallel algorithms unable to use the fast shared memory. The previous architectures (G80 and GT200) have essentially a 16KB shared memory per SM and the global memory (DRAM). In the Fermi GPUs, each SM has 64 KB configurable memory, that can be a combination of 16 KB/48 KB of shared memory and 16 KB/48 KB of L1 cache. The shared memory enables thread cooperation within threads in the same block, but since not all algorithms can take advantage of this feature, a L1 cache exists for algorithms where the access paths to memory are unknown. Moreover there is a unified L2 cache (768 KB) created to enable efficient and high speed data transfer across the GPU. This memory is adequate for irregular memory accesses. In the Fermi architecture, the patterns for global memory access are more efficient in terms of bandwidth that the previous GPUs, since memory accesses are cached in the L1 and/or L2 caches in lines of 128 bytes. A cache line is a 128-byte aligned segment in device memory. Considering a warp accessing aligned and non-sequential float values [60], only 1 memory transaction is required due to the caching. In previous architectures data must be coalesced, so non-sequential data would require more memory transactions

[8]In the CUDA programming model threads are grouped in blocks that are assigned to each SM. A more detailed description is given in the next subsection.

in comparison with Fermi's memory model. However if memory accesses are misaligned and one thread might need to access another 128-cache segment, more memory accesses are required. In the case of double-precision, at least two memory accesses are required. Caching in L2 memory can be important in random memory accesses because it reduces the over-fetch [60, 63].

To summarise, global memory access in Fermi architecture is automatically cached and the differences in performance observed in comparison to cases where shared memory is used is small. Nevertheless, the guidelines for performance optimisation state that, whenever possible, one should replace global memory accesses by shared memory accesses [63]. The developed numerical code has random access patterns, being suitable for the L1 and L2 caches, thus the capabilities of shared memory were not explored.

3.2.2 Execution Model

CUDA is a general purpose parallel computing architecture developed to exploit the parallel architecture of GPUs, based on a scalable programming model and on an instruction set architecture. In other words, CUDA is simultaneously a hardware and a software architecture that enables the execution parallel programs on GPUs [27,63]. Furthermore it provides a C/C++ language interface to the hardware, enabling programmers to focus on the parallel implementation to exploit the parallel computing capabilities of the hardware. The challenge behind the scalable programming model, is to increase the performance of parallel applications to take advantage of the increasing number of cores in the near future. The idea behind the instruction set architecture is to divide the problem into smaller problems and consequently into finer pieces that can be solved cooperatively, in parallel, by all threads within a block. Essentially each block is an amount of work that is assigned to each SM and the threads within the block are executed simultaneously by all SPs in each SM. Increasing the number of cores and SM enables to solve more blocks of work concurrently, enabling an automatic scalability with the number of CUDA cores [60].

Developing CUDA applications consists of writing a sequential host (CPU) program that runs on the CPU to launch device (GPU) functions, designated by kernels, written in CUDA. A kernel is a piece of code that runs on the GPU that is executed in parallel by a set of threads. Invoking a kernel uses a new executing syntax ($<<<, >>>$) and requires launching configuration parameters, the thread block dimensions and the grid dimensions. There is also a thread hierarchy, in which, threads are organised into blocks of threads (or thread blocks) and threads are identified up to 3 dimensions. One can imagine a Rubik's cube, for instance, to represent a block of threads, where each smaller cube is a thread. Thread blocks are then organised in a grid that can be be one-dimensional or two-dimensional. Threads, blocks and grids can be indexed by using built-in variables such as `ThreadIdx.x`, which specifies the index of the thread in the x direction of the block. In the same way the block index inside the grid can be accessed by the variable `BlockIdx.x` and the grid dimensions along x is know by the using `gridDim.x` [27, 60].

The device (GPU) is considered to be a coprocessor with its own physical memory. The host (CPU) control function calls for device memory allocation, data transfer between host and device, and kernel calls. This is a heterogeneous programming model (Figure 3.3) that simultaneously enables running some code on the device and some other code on the host. Kernel calls by the host are sequential, but asynchronous with the host threads. In order to synchronise GPU kernels with the host one can use the command `cudaThreadSynchronize()` to guarantee that the device has completed all preceding requested tasks [27, 60].

As already mentioned a kernel launches a grid of thread blocks, with a group of threads per thread block. Each block is assigned to a SM, but threads are executed in groups of 32 called warps. Each SP in a SM executes the same instruction for all threads, also known as Single Instruction Multiple Thread (SIMT) computations. The maximum number of threads per block for the Fermi architecture, both in one, two or three dimensions is 1024 and the number of blocks that can be launched in a grid is of 65535 for one dimension [60].

A general idea on the CUDA programming model and GPU architecture was given with the intent to introduce some of the important implementation

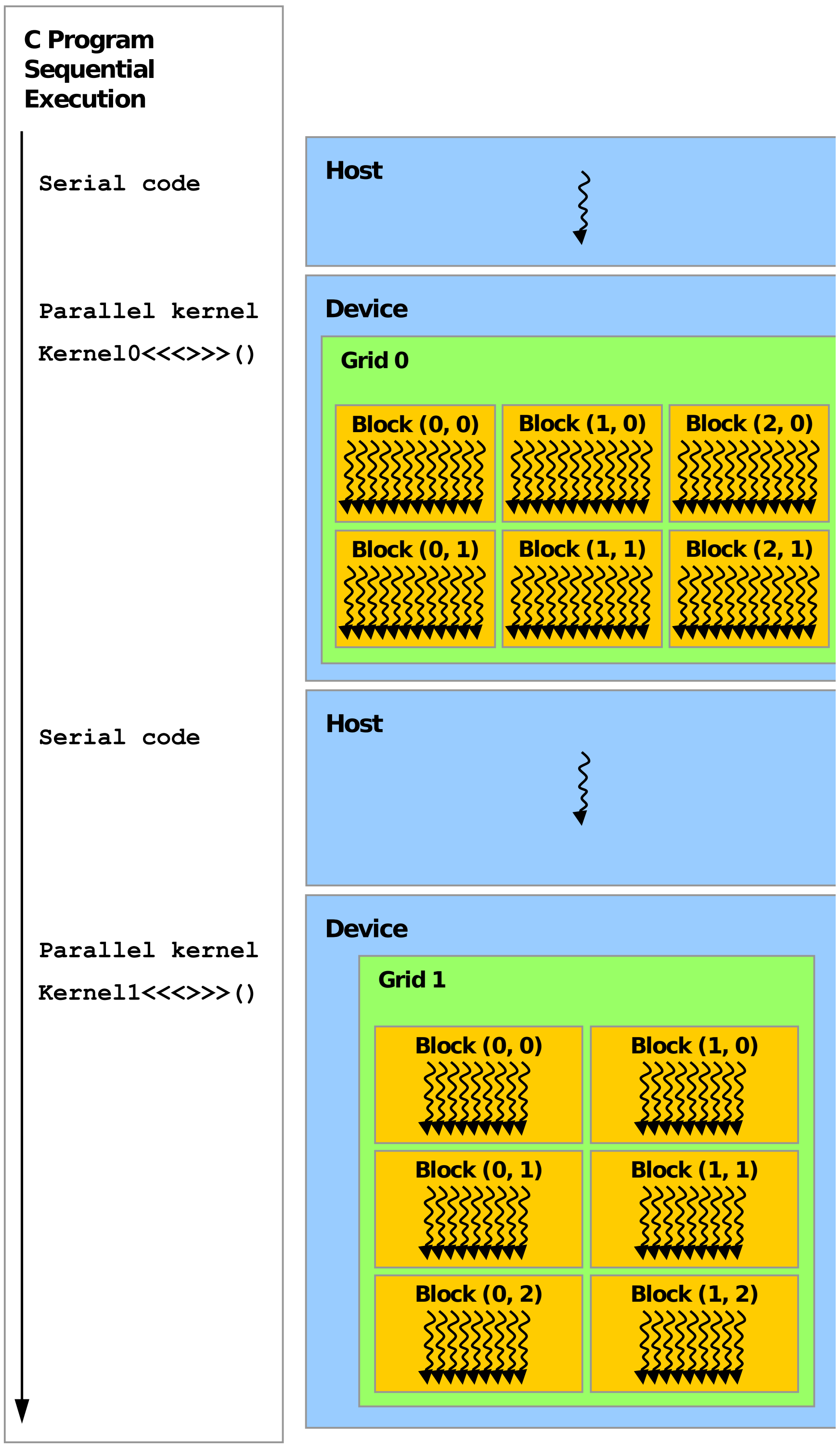


Figure 3.3: CUDA execution model (adapted from [60]). Each winding arrow represents a thread.

details of the parallel code. The interested reader should consult [27, 60] for more details on GPU architecture and programming.

3.3 Unstructured Finite Volume Code on GPU

The parallel implementation of the code on a GPU is not a hard task, but getting performance benefits from the parallel implementation is not so straightforward. The most basic strategy is to remove the bottlenecks of the serial code, by porting to the GPUs the most time consuming routines. Likewise, the former step must be done carefully to avoid unnecessary memory transfers between host and device. Sometimes it compensates to run pieces of code on the GPU rather than on the CPU, even if that does not show performance gains, just to avoid data transfer between host and device. Moreover, memory optimisations are the most important area for performance benefits by maximising bandwidth. Highest performance can be achieved when porting the code completely thus avoiding host-device communication as much as possible [64].

The implementation of the code on the CPU was already explained in Chapter 2 and the idea behind the GPU implementation is to port to the device only the heavy computations. Essentially the code enters an iterative process of solving momentum equations, pressure-correction equation and perform field corrections until sufficient convergence is reached. This is the idea underlined in the SIMPLE algorithm which is known by its slow convergence rate. Therefore it is desired to port these routines to the GPU. Solving momentum equations involves the initial step of assembly and solving the system of equations with an iterative solver. This is repeated at every iteration of SIMPLE. One could port only the iterative solver to GPU and perform the assembly of the equations on the CPU, since it is well known that the iterative solvers are almost always code bottlenecks. In order to avoid data transfer between host and device at every iteration of SIMPLE, the approach is to implement the complete SIMPLE algorithm within the parallel architecture.

The flow chart of the code implemented on the GPU is given in Figure 3.4

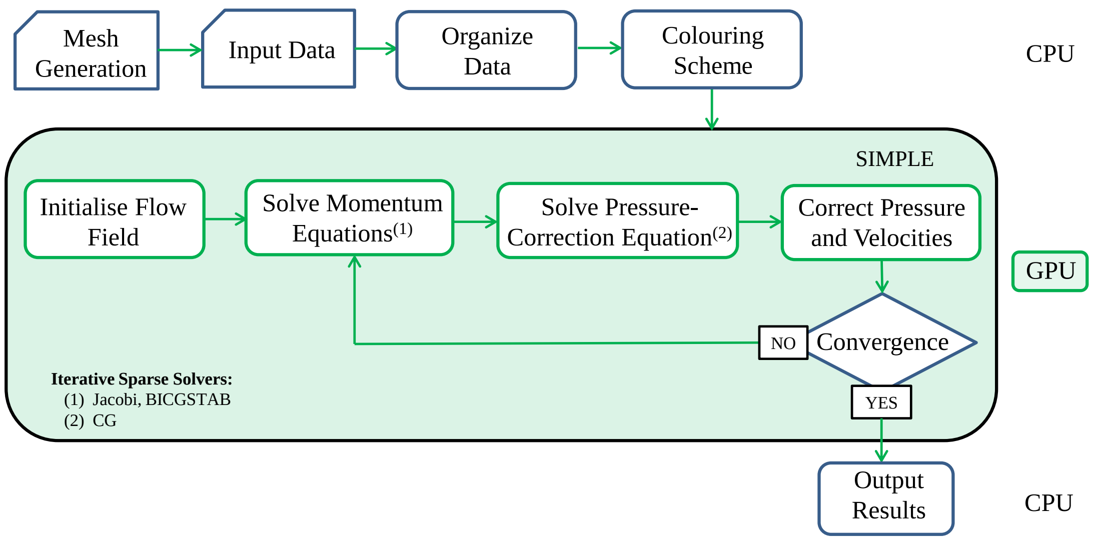


Figure 3.4: Flow chart for the parallel code on GPU.

and illustrates the routines that run on the GPU and on the CPU (compare with chart of Figure 2.7 for the CPU). Some routines are lightweight and are executed only once, hence is not worthy to port them into the device. These are the mesh generation, the data structuring arising from the mesh and the geometry which are required along the code, but never change which are all computed on the the CPU. The necessary data is then copied to the GPU global memory. The following subsections describe the strategies adopted for the assembly of the algebraic system of equations and its numerical solution.

3.3.1 Equations Assembly

The assembly of the equations on parallel architectures is not so simple as on a single CPU core. Thus, an additional routine had to be implemented on the CPU for the parallel code, to deal with the race conditions (*i.e.* simultaneous access to the same memory address) in the assembly of the system of equations.

A loop that goes through all edges of the computational mesh collects contributions from each edge to both the triangles (control volume) straddling that edge. In this way the complete system of equations is assembled.

In the CPU implementation this does not create any problem, but in the parallel implementation race conditions at runtime might occur, without any control by the programmer. More than one edge contribution is being assembled at the same time and if at least two edges try to add a contribution to the same control volume (triangle), a race condition occurs because two threads try to read and write to the same memory location at the same time. This leads to information loss, which promotes incorrect results. Since in the CUDA execution model no information is given about which thread performs a specific task, the fastest thread will perform the task.

One way to avoid this problem in parallel architectures is to adopt a colouring scheme when assembling the system of equations [42,65]. The idea is to colour each edge (see Figure 3.5) in a triangle in a way that all contributions from a edge colour are independent. This guarantees that each triangle receives a contribution from only one edge from a colour assembly step. Since the domain is composed only by triangles, three colouring steps are necessary to assemble the system of equations. Since every edge has two adjacent cells first the contributions to the cells with index zero are assembled, followed by the cells with index one (see Appendix C for details on data structure). A sample corresponding to the assembly of the diffusive term on the GPU is given in Appendix E. The kernel launches for each colour are

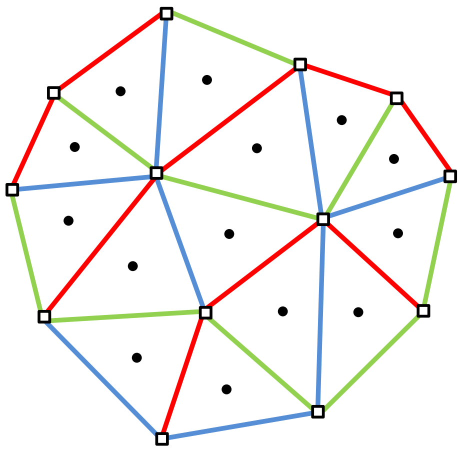


Figure 3.5: Colouring scheme adopted to assemble the system of equations of the GPU.

exemplified here and are composed by a grid with a certain `NumberOfBLocks` and `ThreadsPerBlock`. The number of blocks in this case is the lowest integer number resulting from the division of the number of edges (`dev_n_edge`) by the number of `ThreadsPerBlock`. The number of `ThreadsPerBlock` has to be defined by the user (for instance 256).

```
GetDiffusion<<NumberOfBlocks,ThreadsPerBlock>>>(colour), ...);
GetDiffusion<<NumberOfBlocks,ThreadsPerBlock>>>(colour), ...);
GetDiffusion<<NumberOfBlocks,ThreadsPerBlock>>>(colour), ...);
```

The grid dimensions and the number of blocks are defined at runtime. The performance is dependent on the choice of the number of threads for each block and the number of blocks, especially when dealing with memory optimisations. This is related to the occupancy[9] of the SMs. The problems solved on the GPU are normally executed with a large number of blocks and number of threads per block in order to hide latency.

The implementation uses constant memory for small amounts of data that are accessed continuously and all other memory accesses are to global memory. In previous GPU architectures a huge amount of effort had to be done for the optimisation of memory accesses. In the Fermi architecture the global memory accesses are cached into the L2 or L1 memories due to the extended memory hierarchy. It is important to remark that in the particular case of unstructured meshes the irregularity of the data does not promote coalesced accesses to memory and this was one of the reasons leading to the introduction of L2 and L1 caches in the Fermi architecture. The unstructured code is characterised by random memory accesses, therefore global memory accesses are automatically cached (in Fermi cards) in these small caches.

Regarding the developed numerical code on GPU, a description of some implementation details is presented in Appendix E.

[9]Occupancy is a metric used to relate the number of active warps and the maximum of possible active warps on a SM. The idea is to keep the hardware busy by executing other warps while others are paused.

3.3.2 Iterative Solvers on GPU

The iterative methods for solving sparse linear systems were introduced in Section 2.4 and special attention was devoted to the sparse matrix-vector multiplication (SpMV) operations due to their importance in computational science [34,51]. Iterative sparse solvers implemented in parallel architectures have been reported in the literature [66, 67], especially for the conjugate gradient method.

Traditionally, the SpMV is the most computationally intensive operation in an iterative sparse solver. Therefore efforts have been made to improve the performance of matrix-vector operations on parallel implementations. The SpMV is commonly represented in the operation $\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{y}$ where $\mathbf{A}$ is a large sparse matrix and $\mathbf{x}$ and $\mathbf{y}$ are column vectors [51]. The key feature of SpMV operations is the large number of load instructions relative to the floating point instructions and this is the reason they are considered to be a memory bound operation. The performance of SpMV is therefore dictated by memory bandwidth efficiency and data storage formats for the matrix. An interesting paper by Bell and Garlard [34] presents the performance dependence on the data storage scheme in different sparse matrices with different number of non-zero entries. There are several sparse matrix representations that differ in the storage requirements and the way the data is accessed. The choice of the sparse format essentially depends on the characteristics of the matrix and the number of non-zero entries. There are a variety of matrix storage formats, and the most common are cited such as the diagonal format (DIA), ELLPACK format (ELL), coordinate format (COO), compressed sparse row format (CSR) or hybrid ELL/COO format (HYD). A more complete list can be found in the literature [34, 51]. Essentially, for unstructured grids the HYB format enables performance gains according to Bell and Garlard [34], but in this work the different data storage schemes were not explored.

The iterative solvers introduced in Section 2.4 and implemented on the GPU use the CSR format presented in the Subsection 2.4.5. An existent implementation of SpMV (`cusparseDcsrmv`) in NVIDIA's CUSPARSE library is used to perform matrix-vector operations. CUBLAS is another library used within the iterative solvers for dot-product (`cublasDdot`) and vector-scalar multiplications and vector update (`cublasDaxpy`). These basic routines are presented for instance in the conjugate-gradient algorithm (Table 2.3) or in the bi-conjugate gradient algorithm (Table 2.4).

Chapter 4

Code Assessment

In the present chapter, the developed numerical codes are initially used to solve two classical problems in fluid mechanics, the Poiseuille flow between parallel plates and the lid-driven cavity, both for Newtonian fluids, which serve the purpose of validation of the implemented incompressible Navier-Stokes solver. The results presented here were computed with the serial code, unless otherwise stated.

4.1 Poiseuille Flow Between Parallel Plates

The pressure-driven laminar flow of a fluid along a cylindrical pipe or between parallel plates is known as Poiseuille flow. For Newtonian fluids these flows have a full analytical solution for the velocity and pressure fields. In the case of the parallel plate flow, one assumes that the length of the channel is much larger than the distance separating the two plates, thus allowing the flow to achieve the fully developed state and the channel width is much larger that the wall-to-wall distance so that edge effects are neglected. The flow is driven by a pressure gradient along the length of the channel while the viscous drag retards it. A geometrical description of the parallel plate domain used to perform the numerical assessment is presented in Figure 4.1, where H is the distance separating the two plates and L is the length of the channel.

Initially, the problem was solved considering a Newtonian fluid for Re=0.1,

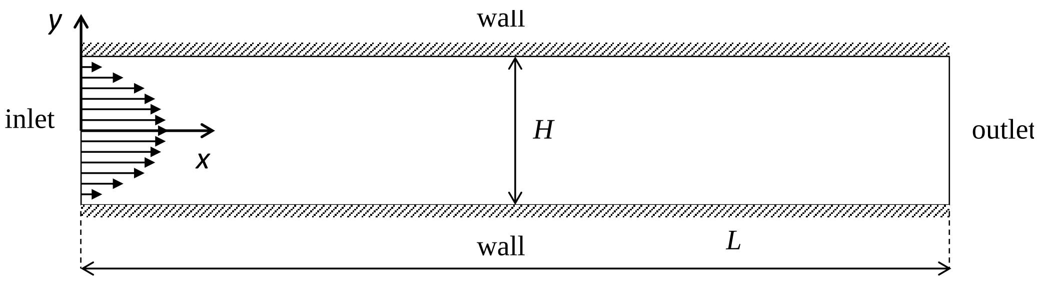


Figure 4.1: Poiseuille flow and boundary conditions description.

where the Re is computed based on the mean fluid velocity, distance separating the two plates and the kinematic viscosity. A fully developed velocity profile (Eq. 2.22) was used as inlet boundary condition and as outlet the fully developed outflow (Eq. 2.24) condition was applied. The dimensions of the channel were defined as $L = 0.5m$ and $H = 0.1m$, for length and height, respectively. The computational meshes for the problem, are illustrated in Figure 4.2, with reference to the number of cells. All meshes are unstructured but with a "quasi-uniform" distribution of cell sizes, *i.e.*, no attempt was made to refine the mesh at any specific location.

The numerical solutions for the u-velocity at L=0.25m are presented here and were obtained by employing the SIMPLE algorithm and the UDS scheme to estimate the convective fluxes. The stability of the UDS scheme was the main criterion for its choice, although it is a first-order numerical scheme. All the computations were performed using the pseudo-transient approach, considering that the flow initially is at rest. The momentum equations were under-relaxed with a time-step of 5.0x10^{-4}s and pressure corrections were relaxed with a constant value of 0.013. The momentum equations were solved with the Jacobi method and the pressure-correction equation with the conjugate gradient method. Both iterative solvers used the convergence criterion 2, given by Equation 2.49 with an accuracy (ϵ) of 10^{-2}. The convergence criterion of SIMPLE is based on the normalised residuals of the momentum and pressure equations. The maximum residual is taken to be the maximum of the first 5 iterations and the normalised residuals must fall below 5.0x10^{-4} for momentum and pressure equations. The dimensionless velocity profiles for the different meshes were compared with the analytical solution, for both

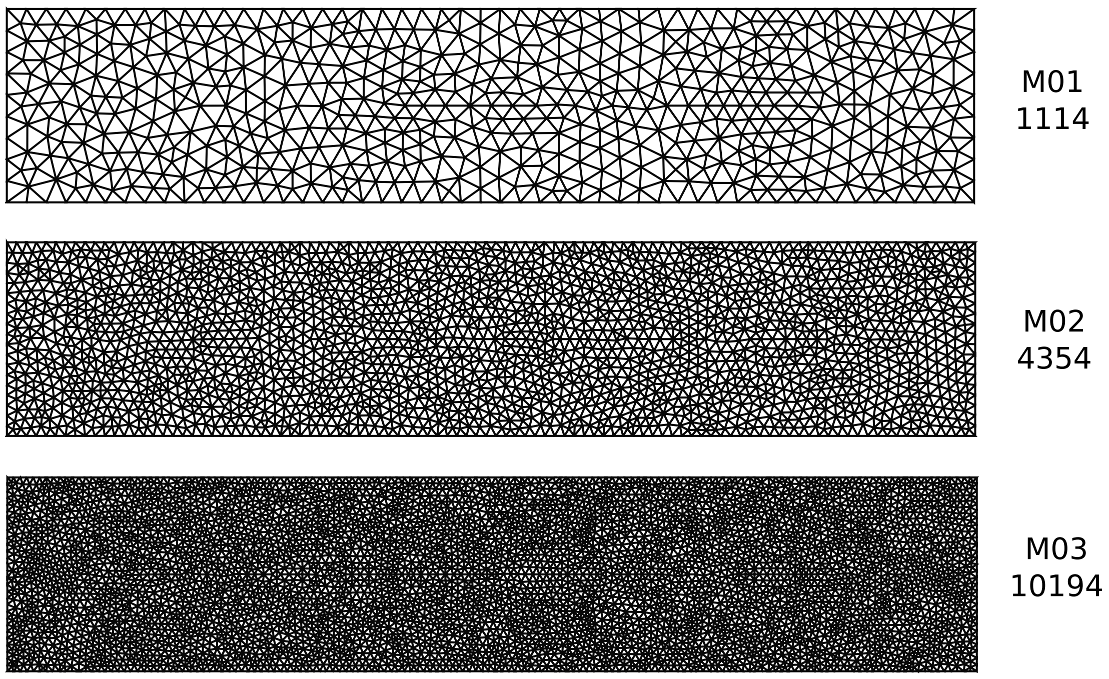


Figure 4.2: Meshes used for the Poiseuille case study.

CPU and GPU developed codes. The results are presented in Figures 4.3 and 4.4.

Both CPU and GPU developed codes estimated successfully the parabolic u-velocity profiles. The results also show that by refining the mesh, the computed velocities approach the analytical solution. Table 4.1 lists the errors of the numerical predictions of the velocity profile relative to the analytical solution as obtained both by the CPU and GPU codes. The errors are computed using Equation 4.1, by arithmetic average of the errors in the velocity profile of a number of points collected from a narrow region in the middle of the channel.

$$error = \frac{1}{n} \sum_{i=1}^{n} \frac{\left| u_{i_{numerical}} - u_{i_{analytical}} \right|}{u_{i_{analytical}}} \tag{4.1}$$

As expected, the errors decrease with the mesh refinement. Thus, both the numerical codes implemented on the CPU and GPU were capable of solving a simple Newtonian fluid problem.

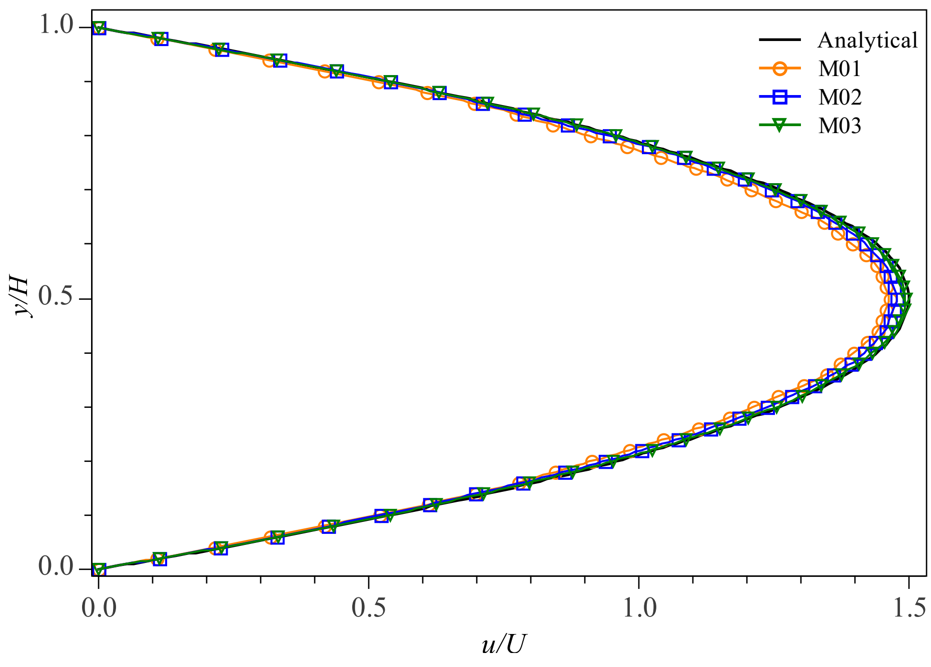


Figure 4.3: Dimensionless velocity profiles calculated on CPU for Newtonian fluid flow at *Re*=0.1.

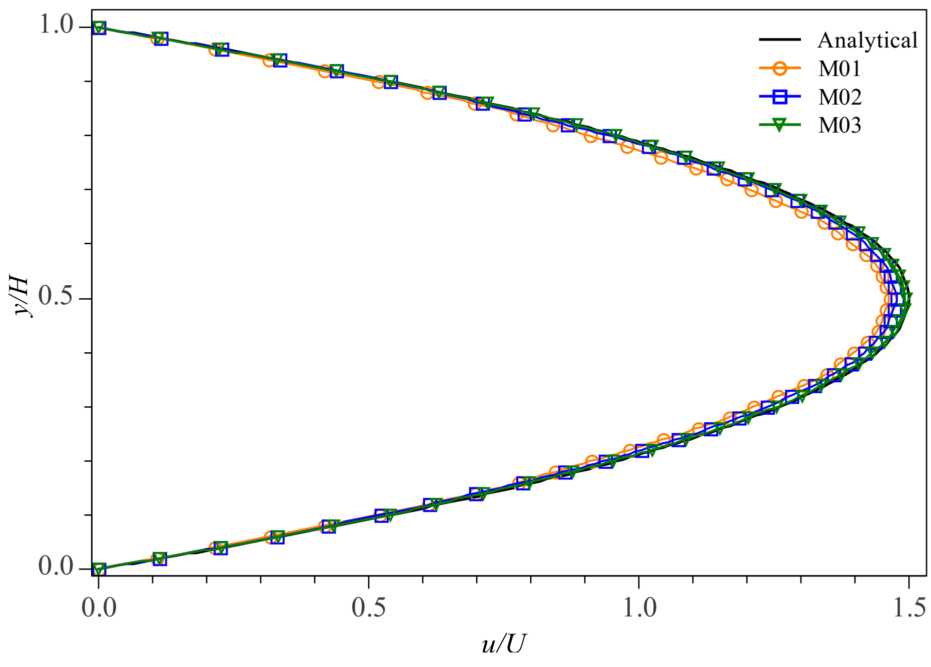


Figure 4.4: Dimensionless velocity profiles calculated on GPU for Newtonian fluid flow at *Re*=0.1.

Table 4.1: Average error in the velocity profile for the various meshes.

	M01	**M02**	**M03**
CPU	2.29%	1.50%	0.48 %
GPU	2.29%	1.50%	0.48 %

Moreover, the same case study presented before is used to perform the assessment of the generalized Newtonian model implemented in the numerical codes. The Bird-Carreau model (Eq. 2.6) was used to account for the shear-thinning behaviour, as it happens in the case of polymer melts. This model uses four empirical parameters and accounts for the low shear and high shear rate Newtonian regions. One parameter is the power law index n, as in the simplest Generalized Newtonian Fluid model, the Ostwald-de Waale power law model for viscosity. The parallel plate domain was defined to be similar from the previous one, while in the current domain a constant velocity was employed as inlet boundary condition. The dimensions of the channel were defined as $L = 1.0m$ and $H = 0.1m$, for length and height, respectively.

The problem was solved for Re=0.01, computed as described previously for Newtonian fluids, whereas in the case of the variable viscosity fluid, the Reynolds number was computed based on the mean fluid velocity, distance separating the two plates and the zero-shear viscosity. The computations were carried out with different values for the power index n and the parameters used for the Bird-Carreau model were: $\eta_0 = 1.3\text{x}10^4$ Pa.s, $\eta_{\text{inf}} = 1.0\text{x}10^{-3}$ Pa.s and $\lambda = 5.4\text{x}10^{-2}$s. The computational mesh domain used for all the computations is presented in Figure 4.5 with number of cells equal to 8902.

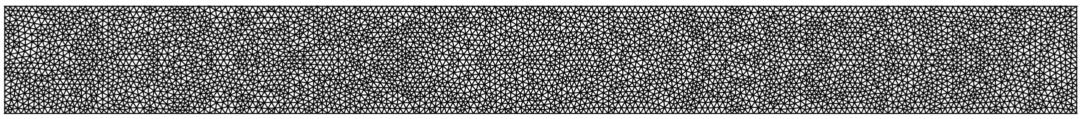


Figure 4.5: Mesh for validation of the Bird-Carreau model.

Regarding the computational details, the pseudo-transient approach was again used and the flow initial condition is at rest. The convective fluxes were estimated by UDS scheme, momentum equations were under-relax by

$\Delta t = 5.0\text{x}10^{-6}$s and pressure-correction relaxed by $\alpha_p = 0.013$. The iterative solver used to solve momentum and pressure-correction equations were Jacobi and CG, respectively, both with an accuracy ($\epsilon < 10^{-3}$). The SIMPLE converged after the normalised residuals felt below $5.0\text{x}10^{-4}$ for momentum and pressure equations.

The numerical solutions for the u-velocities at L=0.5m were compared with a reliable and well-validated structured finite volume code [15] developed inside the research group involved in this work. This comparison was performed since the Bird-Carreau model does not have an analytical solution for the velocity profiles. The results are plotted in Figure 4.6 and good agreement is attained. The unstructured FVM code is capable of performing computations with a GNF fluid. In the same figure plots for different velocity profiles, using the Bird-Carreau model are presented, by changing the power law index (n). As can be seen, by decreasing the power index (n) the profile tends to a plug, the typical behaviour of a shear-thinning fluid. This last problem validates the code for variable viscosity fluids.

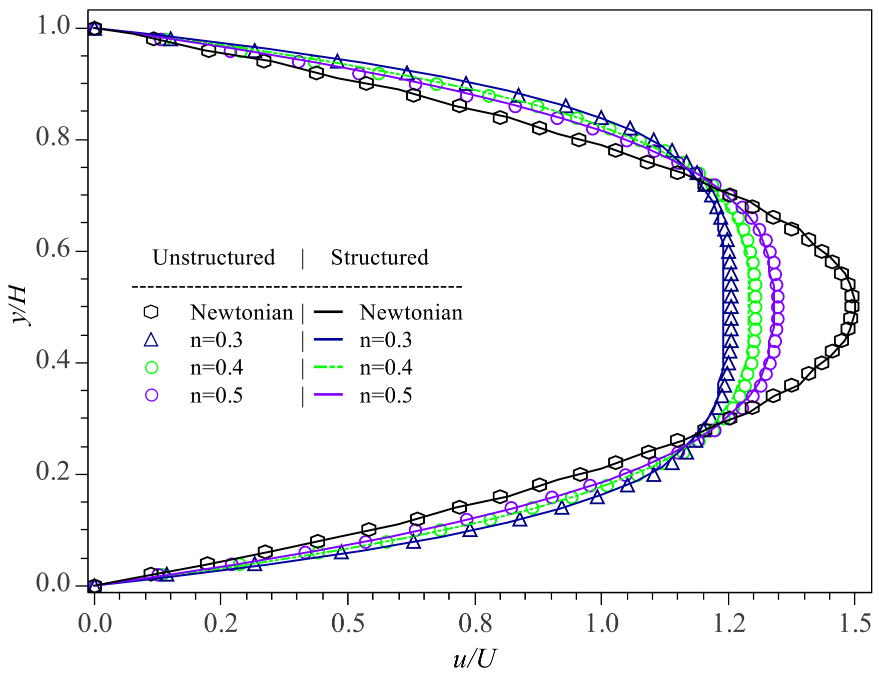


Figure 4.6: Velocity profiles for Newtonian and Bird-Carreau models.

4.2 Lid-Driven Cavity Flow

The lid-driven cavity flow is a benchmark problem to validate the Navier-Stokes equations solvers and for the assessment of numerical techniques. The problem considers incompressible flow in a square cavity, where an upper lid moves with a known velocity (U). The description of the problem and boundary conditions are illustrated in the Figure 4.7, with referent to the side length (represented by H) of the square cavity. This test problem is well documented in the literature [68–70], using different solution procedures and Reynolds numbers ranging from 100 to 10000. The major difficulty with this problem is to capture the flow near the corners, where vortices appear.

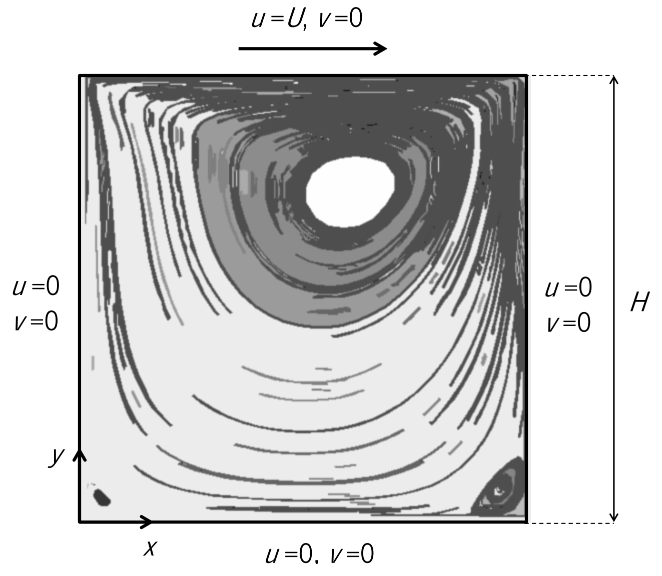


Figure 4.7: Driven cavity flow problem with boundary conditions.

In order to validate the numerical code with predictions for this case study, computations at different Reynolds numbers were performed and different numerical schemes are used to estimate the convective fluxes. The results were compared with values reported in the literature for validation purposes and are presented below.

The numerical solutions for the cavity flow at Re=100, Re=400 and Re=1000 are presented. The solutions were obtained by employing the SIMPLE algorithm and different TVD schemes to estimate the convective fluxes.

All the computations were performed using the pseudo-transient approach, considering that the flow initial condition is at rest. The side length dimensions were defined to be $H = 0.1m$ and the top wall moves at a speed of $0.001ms^{-1}$, $0.004ms^{-1}$ and $U = 0.01ms^{-1}$, corresponding to flows at Re=100, Re=400 and Re=1000, respectively. The momentum equations were under-relaxed with different time-steps of 0.8s for Re=100, 0.02s for Re=400, 0.005s for Re=1000, while the pressure corrections were relaxed with a constant value of 0.02. The momentum equations were solved with Jacobi method and pressure equation with conjugate gradient method. The iterative solvers use both the convergence criterion 2, given by equation 2.49 with an accuracy (ϵ) of 10^{-1}. The SIMPLE convergence criterion is based on the normalised residuals of momentum and pressure equations. The maximum residual is taken to be the maximum of first 5 iterations and the normalised residuals must fall bellow 10^{-4} for momentum and pressure equations. More details on the computational approach can be found in Chapter 2. The computational meshes for the problem, are illustrated in Figure 4.8 and were generated using *gmsh*. A description of the procedure for mesh generation can be found in Appendix A.

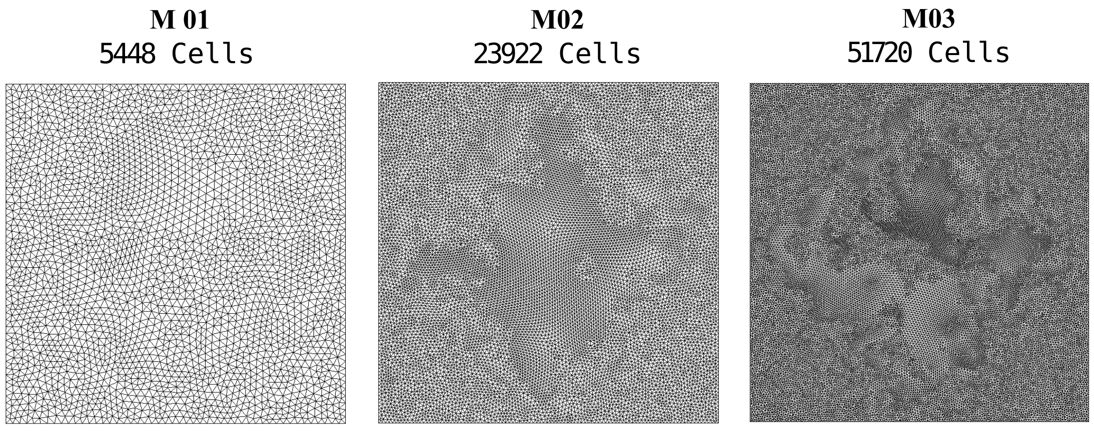


Figure 4.8: Meshes used for the lid-driven cavity case study.

Most of the assessment work was performed using mesh M03, whereas the remaining meshes were employed on the performance evaluation presented in Chapter 5. The numerical results are presented by means of velocity plots and streamlines. The computed u-velocity along the central vertical centre line (x=0.5) and the v-velocity along the horizontal centre line (y=0.5) are compared with the values reported by Ghia *et al.* [68]. The velocity profiles are plotted in Figures 4.9, 4.10 and 4.11 for the different Reynolds numbers.

The velocity profiles match very well with the reference values reported in the literature, although slight differences can be found between different TVD schemes, specially for higher Reynolds. As expected, the computed values with the first-order upwind scheme (UDS) deviate more from the reference values in comparison with the other schemes with are second-order accurate.

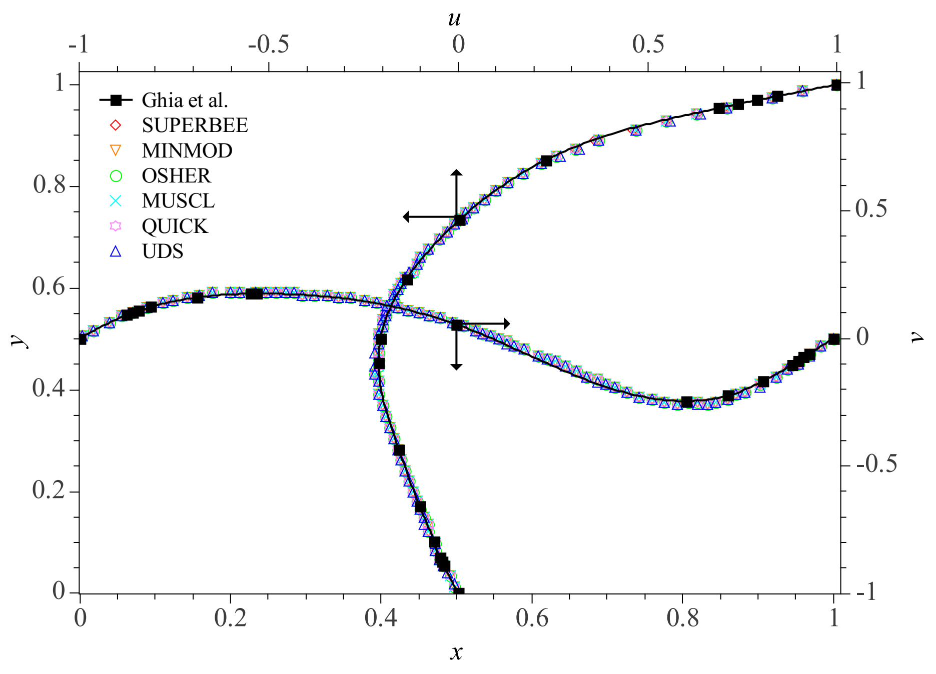


Figure 4.9: Velocity profiles for Re=100 using M03 mesh.

For assessment purposes it is also useful to present the flow field using streamlines, because it enables the visualization of the global flow field. The streamlines for the three different Reynolds numbers are plotted in Figure 4.12, and were obtained using the Paraview's [58] implementation of Range-Kutta 4/5 method in the Stream Tracer filter. The streamlines computed by Ghia *et al.* [68] are also presented in Figure 4.13 for visual compari-

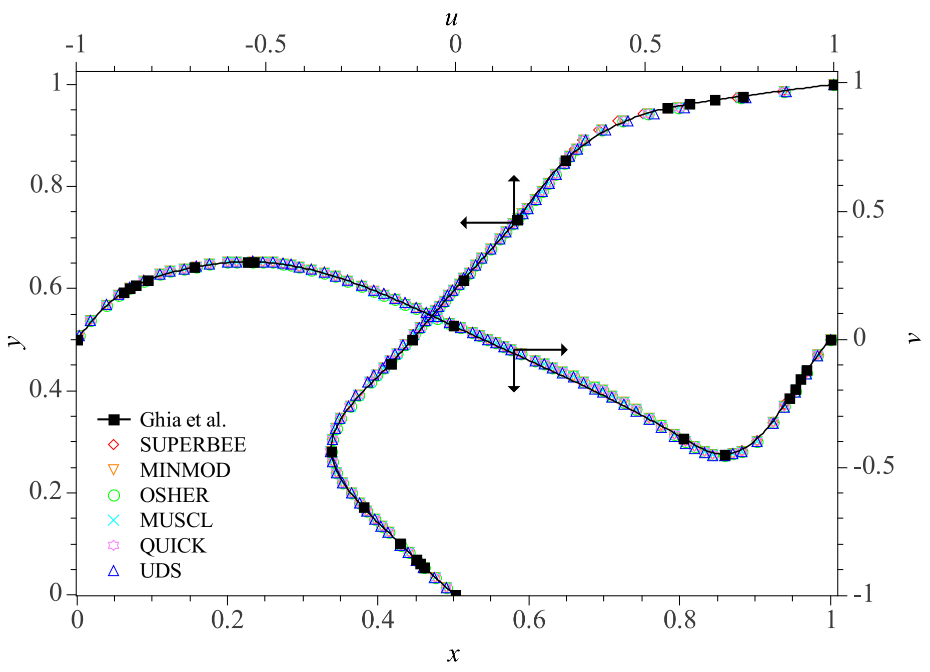


Figure 4.10: Velocity profiles for Re=400 using M03 mesh.

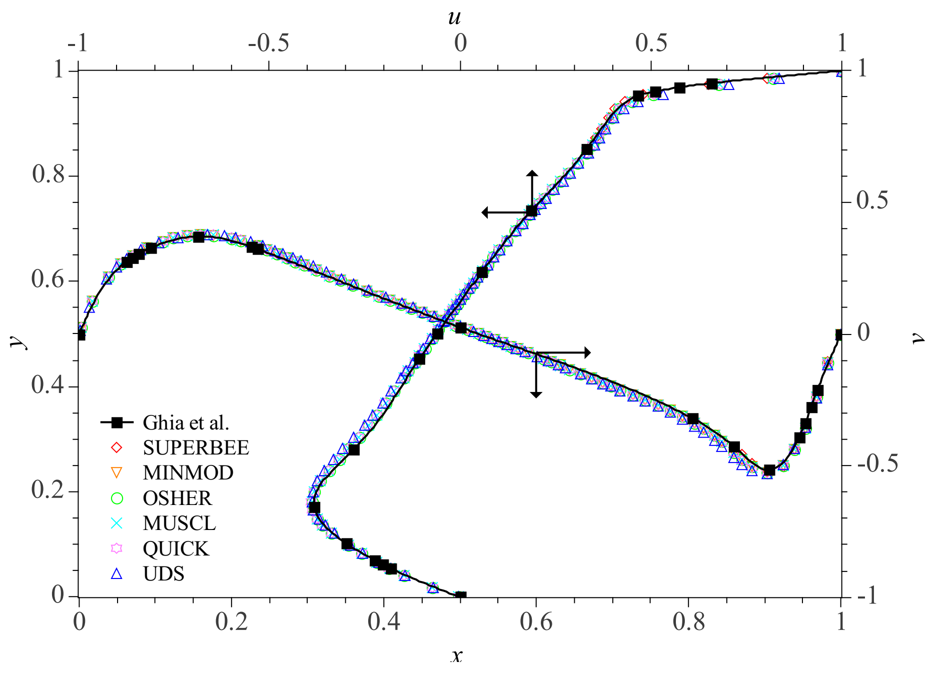


Figure 4.11: Velocity profiles for Re=1000 using M03 mesh.

son with the computed flows in this thesis. The streamlines predicted by the code match very well with the the literature with the shape and location of the vortices.

The results predicted by the developed codes, matched in all cases the data from the literature, a clear indication of the correctness of the implementations. The results presented in this subsection were computed with the serial code and for brevity reasons the same results computed with the parallel code were not shown.

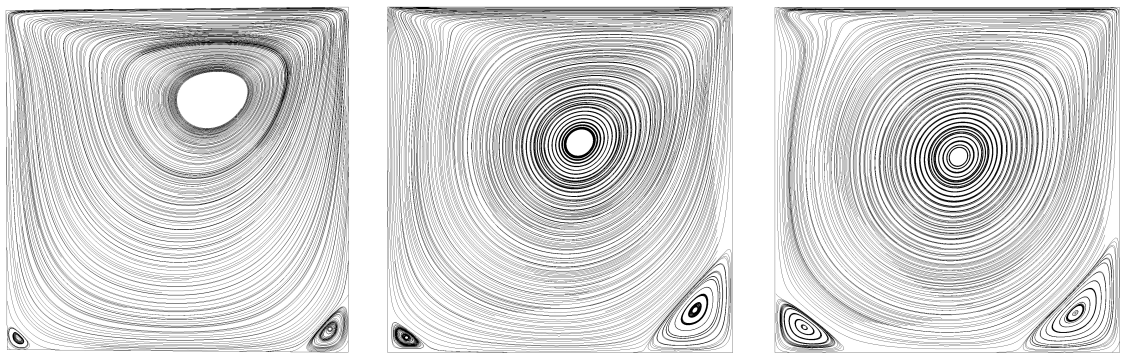


Figure 4.12: Streamlines for *Re*=100, *Re*=400 and *Re*=1000 (from left to right).

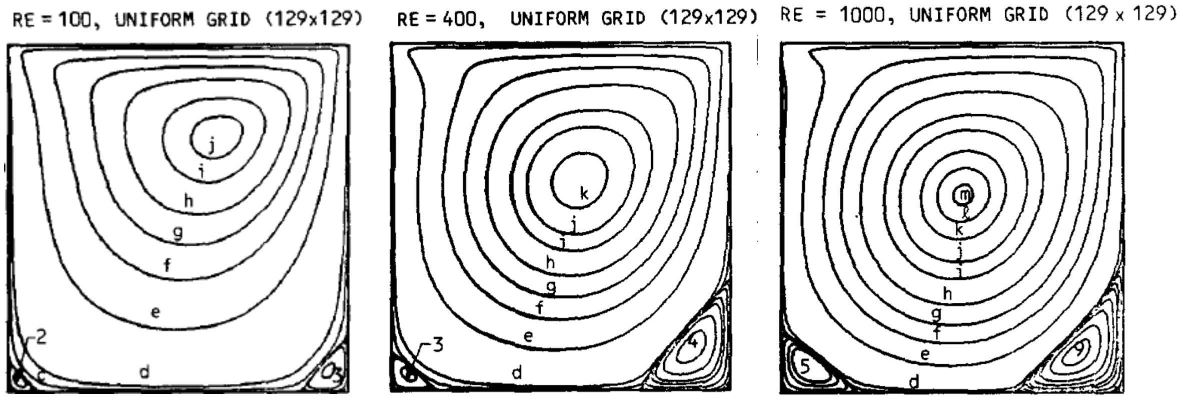


Figure 4.13: Streamlines adapted from Ghia *et al.* [68].

Chapter 5

Performance Analysis

A performance comparison between the developed CPU and GPU codes is presented here, with the goal of exploring the benefits of porting code to GPUs. The fluid mechanics benchmark problems of the previous chapter are used to perform computations with different mesh sizes to evaluate how the performance scales with mesh refinement.

5.1 Introduction

Since the GPUs are suitable for massively parallel algorithms, especially those with high data parallelism, the performance comparison is based on mesh refinement scalability. By increasing the number of cells, the amount of load and the computation time also increase. The developed numerical codes follow the description given in Chapters 2 and 3 for CPU and GPU implementations, respectively. More details concerning the implementation of each code can be found in Appendix C and Appendix E. In order to analyse the performance of both codes a summary of the most important routines is given in Table 5.1. The bold green crosses are used to indicate that those routines run fully on the GPU, whereas the black crosses refer to those running on the CPU. Note that the GPU implementation requires 3 additional routines in comparison with the CPU counterpart.

Following the nomenclature given to each routine, a small description

Table 5.1: Summary of most important routines and corresponding code implementations.

Routine	**CPU**	**GPU**
Mesh generation	**X**	**X**
Cross diffusion format	**X**	**X**
Colouring scheme	—	**X**
Copy to device	—	**X**
Solve momentum equations	**X**	X
Solve pressure equation	**X**	X
SIMPLE corrections	**X**	X
Copy to host	—	**X**

of each one is given below. The reader can be familiarised with the most relevant steps of the algorithm without going into details. This nomenclature is further used to address the time measurements obtained with both CPU and GPU codes. This process of identifying the most relevant routines in order to perform timings can be designated as code profiling.

- **Mesh generation**
  The pre-processing step of mesh generation is performed with open-source software (*gmsh*). Time measurements also include the organisation of mesh data.

- **Cross diffusion format**
  This routine stores information about the neighbours of each vertex. It is required in the estimation of the diffusive term.

- **Colouring scheme**
  This routine runs on CPU, but it is necessary only for the implementation on GPU. Edge colouring is a technique used to avoid race conditions in the assembly of the system of equations.

- **Copy to device**
  Copy to the device all necessary data to build the system of equations and to solve it. This routine is executed only once.

- **Solve momentum equations**
  Includes the process of assembly of the momentum equations and timings of the iterative solver (Jacobi or BICGSTAB). The assembly of equations is done fully on the GPU to avoid data transfer between host and device.

- **Solve pressure-correction equation**
  Similar to the previous routine, it includes time measurements of the assembly of the pressure-correction equation and of the iterative solver (normally CG). It is also carried out completely on the GPU.

- **SIMPLE corrections**
  This routine corrects pressure and velocity fields accordingly to SIMPLE algorithm. It runs also on the GPU.

- **Copy to host** Copies to the host memory the results of the simulation to allow post-processing by adequate software such as *gmsh* or *Paraview*. This routine is executed only once when convergence criteria for the SIMPLE algorithm are met.

Time measurements of the most relevant routines, previously discussed, for the CPU and GPU implementations were obtained with the command `gettimeofday()`. In the case of the GPU implementation the command `cudaThreadSynchronize()` must precede the timing command in order to guarantee that the GPU has finished the computations [60], since kernel calls are asynchronous to host threads.

For the performance analysis, the computations were carried with an Intel i7-950 (CPU) using only a single core and each time with one of these two Fermi GPUs, a GTX 480 and a Tesla C2070. More details on the GPUs capabilities were described in Chapter 3, specifically in Table 3.1. The GTX 480 was installed in the same machine as the i7-950 and performance comparison is performed directly. Some computations where performed with a Tesla GPU installed in another machine with a different CPU, therefore results of computations with this GPU are presented always in comparison with the GTX 480. The optimised CPU code was compiled with gcc and the -O3 optimisation flag while the GPU code was compiled with nvcc compiler, part of CUDA 4.0 toolkit. All the results presented were achieved with double-precision computations.

5.2 Poiseuille Flow

The Poiseuille flow case study assessed in the previous chapter is now used to perform heavy computations both on the CPU and GPU. For problems with a small dimension, parallel computing might not be an option because the CPU can tackle them within a very short time. However for large dimensional problems with a large number of cells the computational time increases considerably since it varies nonlinearly with the dimension. As consequence, the level of parallelism can increase and the use of parallel hardware like GPUs enable the solution of large problems within acceptable times. Details of the meshes used in the performance analysis of the Poiseuille flow problem are presented in Table 5.2 with reference to the number of cells and nodes.

Table 5.2: Computational meshes used in the Poiseuille case study with number of cells and nodes.

Mesh	**M01**	**M02**	**M03**	**M04**	**M05**	**M06**	**M07**
Cells	1114	4354	10194	18186	28512	41326	55474
Nodes	616	2296	5276	9332	14555	21022	28156

Mesh	**M08**	**M09**	**M10**	**M11**	**M12**	**M13**	**M14**
Cells	72338	92404	114890	140488	168818	197338	228584
Nodes	36648	46741	58044	70903	85130	99448	115131

All computations were performed according to the description given in Section 4.1, *i.e.*, considering a Newtonian fluid and the geometrical description of Figure 4.1. The nomenclature introduced at the beginning of this chapter will be useful to identify the most time consuming routines and how they scale with mesh refinement. The relative times of each routine for the CPU and GPU codes are plotted in Figures 5.1 and 5.2.

In this case study, the most time consuming task is to the x momentum equation, up to 86% of the total computational time. The relative time

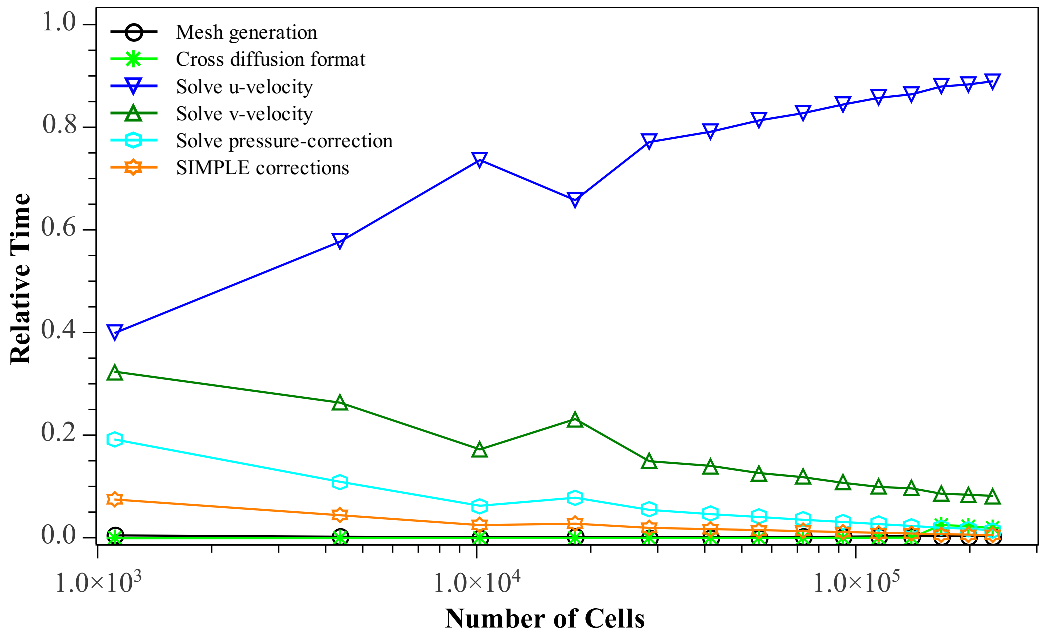


Figure 5.1: Poiseuille code profiling on CPU.

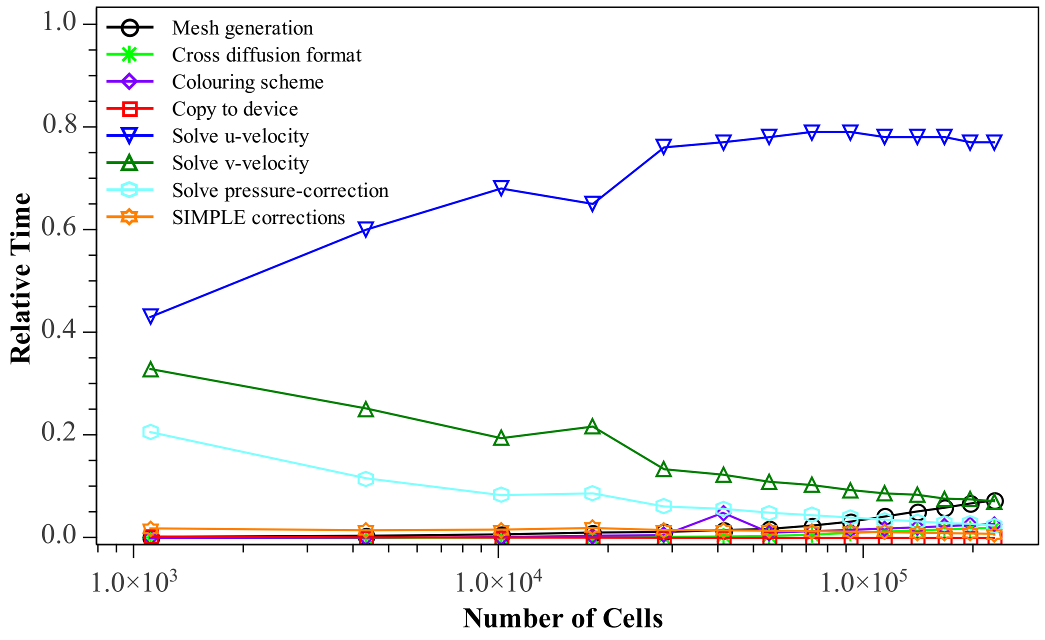


Figure 5.2: Poiseuille code profiling on GPU (GTX 480).

includes the assembly of equations and the Jacobi iterative solver. The percentage of time to solve momentum equations depends on the accuracy of the Jacobi. Due to the slow convergence of Jacobi, a tight stopping criteria might require several iterations to compute the momentum equation, which is this case here, leading to this large relative time. One can also see that the most time consuming routines are solving the momentum equations, solving the pressure-correction equation and performing SIMPLE corrections. This justifies the need for the complete implementation of the SIMPLE algorithm on the GPU. Since the routine that copies data to the device executes only once, at the beginning of the algorithm, its relative time is negligible. The interest reader should consult Appendix F for absolute time measurements of each routine. Regarding the data transfer between host and device, one can see in Table F.2 that upon mesh refinement the time to transfer mesh data varies from 0.20 to 0.26 seconds. Copying to the device a small or a large amount of data takes essentially the same time, since latency plays a relevant role here. For instance, if only the iterative solver would be ported to the GPU one could include twice this time to transfer data back and forth to the device at each iteration of the SIMPLE algorithm for solving only one equation. The Jacobi for instance is used twice to solve x and y momentum equations and CG to solve pressure-correction equation, thus it is not worthy to implement only the iterative solver on the GPU, but the complete algorithm must be implemented to obtain computing gains.

The performance comparison between the serial and parallel code are summarised on Table 5.3, by presenting the total computational time in seconds for the CPU and GPU and the respective speed up. Likewise, an illustration on how the speed up scales with mesh refinement is given in Figure 5.3. The speed up is the ration between the total time of computations on the CPU and GPU, indicating how fast the GPU code runs in comparison to the CPU counterpart. One can see that for small dimensional problems the speed up is relatively small and even below 1 for M01. This means that in the case of mesh M01 the CPU is faster than the GPU. This means it takes longer to start up a small task by several “people” than by a single individual, however if the work load is considerably large the tasks can be

Table 5.3: Timings in seconds of the Poiseuille flow problem with different mesh sizes and respectively speed up.

Mesh	**M01**	**M02**	**M03**	**M04**	**M05**	**M06**	**M07**
CPU	30.62	207.24	840.30	1455.50	3518.10	6457.91	10089.25
GPU	69.97	151.42	262.21	369.64	667.34	943.45	1401.23
Speed Up	**0.44**	**1.37**	**3.20**	**3.94**	**5.19**	**6.84**	**7.20**

Mesh	**M08**	**M09**	**M10**	**M11**	**M12**	**M13**	**M14**
CPU	16528.90	23029.33	34597.23	49184.95	69955.75	93609.73	126850.09
GPU	1916.57	2535.14	3389.73	4568.36	5890.22	7544.30	9189.64
Speed Up	**8.62**	**9.08**	**10.21**	**10.77**	**11.88**	**12.41**	**13.80**

easily divided and the multiple workers do it faster that an individual. Thus the increase in the amount of work raises the speed-up ratio considerably higher and up to about 14x. This is a significant result since for the mesh M14 the execution time decreases from 35h to 2h30min.

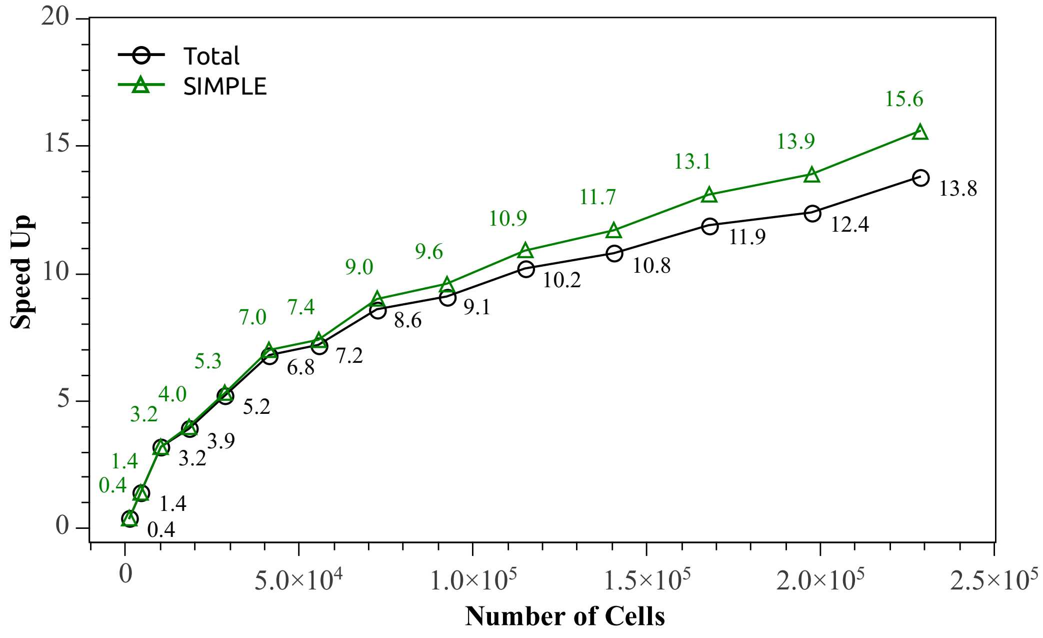


Figure 5.3: Total and SIMPLE speed-ups.

5.3 Lid-Driven Cavity Flow

The lid-driven cavity flow of Chapter 4 is now used to compare the performance between the CPU and GPU codes. The performance analysis for the case study was carried out for Re=100 by following the computational details given in Section 4.2 for the same problem. In order to evaluate how performance scales with mesh refinement several computational meshes were used as in Table 5.4.

Table 5.4: Different mesh sizes used for the performance analysis of the lid-driven cavity flow problem.

Mesh	**M01**	**M02**	**M03**	**M04**	**M05**	**M06**	**M07**
Cells	5448	23922	51720	95442	143380	210624	298774
Nodes	2823	12160	26159	48120	72189	105911	149944

An initial performance analysis is presented by taking a closer look on how these routines scale with mesh refinement. The code profiling for the CPU implementation is shown in Figure 5.4, where clearly the largest part of the computational time is used for solving the momentum equations, mainly due to the matrix-vector operations, part of the iterative solver. The amount of time spent in each routine depends both on the choice of the under-relaxation parameter for the momentum equations, the pressure-correction relaxation constant and the accuracy chosen for the convergence of the iterative solver. The code profiling for the GPU (GTX 480) implementation is given in Figure 5.5, where again the most time consuming routines are those for solving momentum equations. Note also that with the mesh refinement the relative time to solve the respective equations tends to decrease and the time for mesh generation tends to increase. This indicates the need to port to GPU the mesh generation algorithm if very refined meshes are to be used frequently. Since this flow is now clearly a 2D flow, the solution of v momentum equation consumes the same time as u momentum equation, in contrast with the 1D Poiseuille flow for which the solution of u momentum equation took considerably longer than the solution of v momentum. Another remark regarding the colouring scheme is that the time spent on this routine increases with

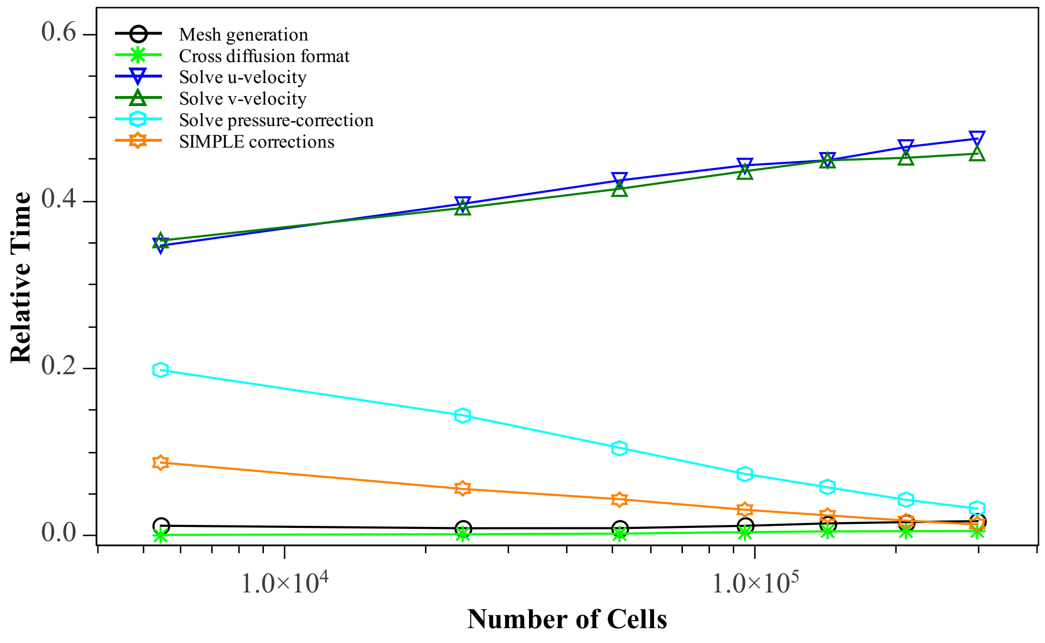


Figure 5.4: Lid-driven cavity code profiling on CPU.

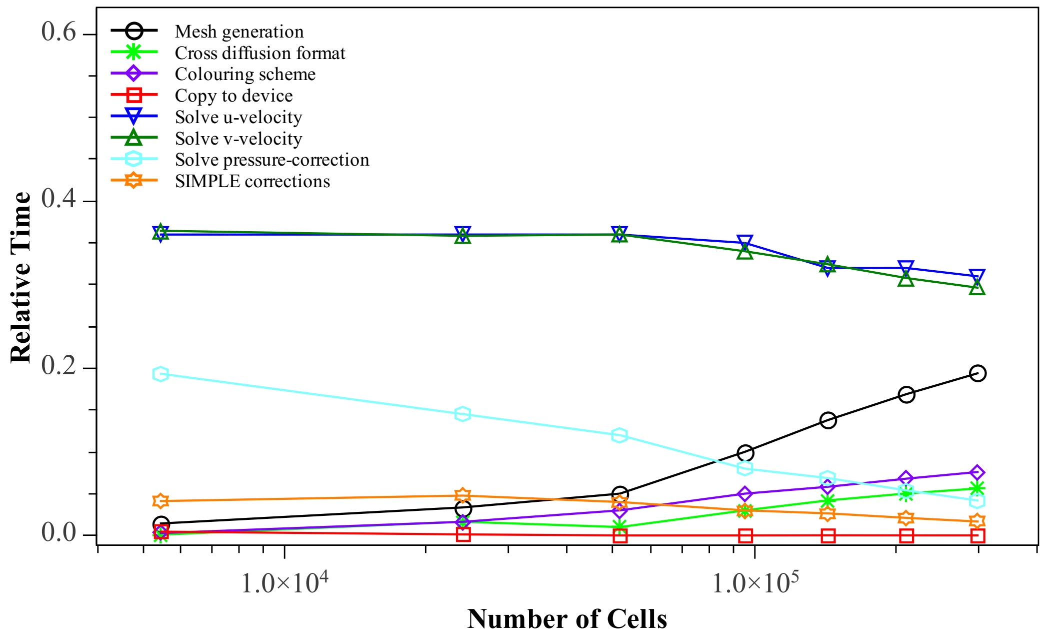


Figure 5.5: Lid-driven cavity code profiling on GPU.

mesh refinement. Eventually, for more refined meshes the algorithm could be redesigned or could be ported to the GPU.

Time measurements in seconds for all the routines are presented in Appendix F. These results were useful to indicate which were the bottlenecks of the code and are an indicator for future optimisations. Timing results in seconds for both serial and parallel code are summarised in Table 5.5 for all computational meshes. The percentage of time taken by the SIMPLE algorithm is also presented. As shown, SIMPLE takes around 98% of the total time spent on the CPU. However, when running SIMPLE on the GTX 480, that amount of load decreases with mesh refinement.

The performance benefits obtained scale with the mesh size, as shown in Figure 5.6. The total speed up is up to 10.5 when comparing the CPU (i7-950) with the GPU (GTX 480).

Table 5.5: Summary of timings for the lid-driven cavity with different mesh sizes.

Mesh	**M01**	**M02**	**M03**	**M04**	**M05**	**M06**	**M07**
Cells	**5448**	**23922**	**51720**	**95442**	**143380**	**210624**	**298774**
SIMPLE i7-950 (s)	60.4	620.8	2261.1	6554.6	13757.6	28777.0	57157.8
Total i7-950 (s)	61.1	627.3	2286.8	6659.8	14033.4	29405.8	58487.3
SIMPLE GTX 480 (s)	49.6	156.1	344.7	736.1	1271.9	2209.3	3751.3
Total GTX 480 (s)	50.8	165.5	383.0	893.0	1670.0	3097.9	5568.3
SIMPLE Tesla C2070 (s)	104.6	334.0	697.6	1426.4	2418.7	4028.1	6821.9
% SIMPLE (i7-950)	98.7%	99.0%	98.9%	98.4%	98.0%	97.9%	97.7%
% SIMPLE (GTX 480)	97.6%	94.3%	90.0%	82.4%	76.2%	71.3%	67.4%

Computations were also performed with a Tesla C2070 GPU, so far the best graphic card for scientific computations. In order to compare both graphic cards the parts of the code that run on the CPU were not taking into account since the GPUs were installed in two different machines with different CPUs. Only the SIMPLE algorithm, that runs fully on the GPU, is used to compare the performance of both graphic cards. The performance of the Tesla C2070 is worse than that of the GTX 480 and the difference might be explained by the memory bound nature of the problem, especially due to sparse-matrix vector multiplications. Since the GTX 480 has more memory bandwidth than the Tesla C2070 this helps to explain the unexpected higher performance of the GTX 480. Moreover, the PCI communication speed can also influence the Tesla performance since it was installed in a not very recent machine with another graphic card used for visualisation purposes which can reduce the PCI communication speed by half, decreasing by half the PCI bandwidth. This can play a significant role, since at every step of the iterative solvers the dot product is computed to estimate the residual, returning a single value to the host memory. These are plausible reasons why the Tesla performed worse than GTX 480 since time measurements of data transfer between CPU and GPU were 10x slower with the Tesla (see Appendix F).

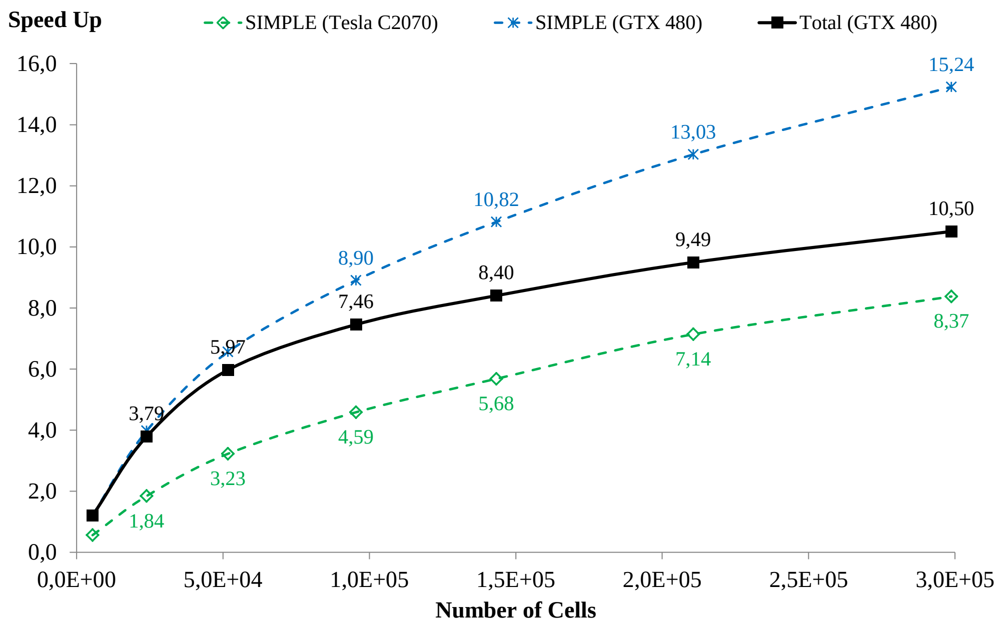


Figure 5.6: Total Speed Up of GTX 480 in double-precision and SIMPLE Speed Up on GTX 480 and Tesla C2070.

Chapter 6

Conclusions and Further Work

The objectives of this work are summarised in this Chapter with reference to the work developed and the main achievements attained. Proposals and recommendations for future work are also given.

6.1 Conclusions

The two main objectives of this work were the development of a serial unstructured 2D cell-centred finite volume code and especially of the corresponding parallel version code on GPU for the computation of flows of Newtonian and Generalized Newtonian Fluids. With older GPU architectures, ir order to take advantage of GPUs capabilities, algorithms required a high level of data parallelism and regular accesses to memory, along with the need to use the fast shared memory to see considerable performance gains. With the most recent GPU hardware, such as the Fermi architecture used in this thesis, new improvements were made, as the extended memory hierarchy and automatically cached global memory accesses. Thus, the results obtained were also expected to evaluate the advantages of the GPU in terms of computational speed of porting an irregular data code, as it happens when using unstructured meshes, which leads to random memory accesses to global memory.

The developed numerical codes were assessed by solving two benchmark fluid dynamics problems, the Poiseuille flow between parallel plates and the lid-driven cavity flow. The results obtained in terms of computational speed up, show well that the GPUs massively parallel hardware are capable of accelerating algorithms, even those with irregular patterns in terms of memory accesses, such as the case of an unstructured code. The performance gains obtained between the CPU and GPU implementations are quantified in speed-ups of up to 14x and 11x for the Poiseuille flow and lid-driven cavity flow studies, respectively. Considering the GPU availability, cost and easiness to program, the results obtained are an indication that they are very useful tools for the future trends of parallel computing in computational fluid dynamics.

6.2 Further Work

The presented work can be improved and extended by contributions from a variety of fields. Some proposals for future works are given below with recommendations to possible improvements of running finite volume code on graphic cards.

Regarding the parallel implementations on graphics cards it would be interesting to compare the used of shared memory or texture memory whenever possible in Fermi's architecture. The possible performance gains obtained by using these fast memories would not be considerably large since global memory accesses on Fermi architectures are automatically cached. Nevertheless it would be interesting to compare them.

Different sparse storage schemes should be implemented in order to improve the bandwidth of the sparse-matrix vector multiplication such as ELLPACK or HYB. Accordingly to Bell and Galard [34], for unstructured meshes these seem to be the most suitable.

Although the BICGSTAB was implemented no performance comparison was carried out with this iterative solver. Thus, a future study should comprise the evaluation of the BICGSTAB to solve momentum equations in terms of speed up when comparing to the Jacobi iterative solver.

More specifically, on the CFD front the parallel code could be further developed by the inclusion of more complex constitutive equations for non-Newtonian fluids and its evaluation in benchmark flows.

Bibliography

[1] Z. Tadmor and C.G. Gogos. *Principles of Polymer Processing.* Wiley-Interscience, New Jersey, 2nd edition, 2006.

[2] M. Gahleitner. Melt rheology of polyolefins. *Progress in polymer science*, 26(6):895–944, 2001.

[3] L.F.A. Douven, F.P.T. Baaijens, and H.E.H. Meijer. The computation of properties of injection-moulded products. *Progress in Polymer Science*, 20(3):403–457, 1995.

[4] R. Pantani, I. Coccorullo, V. Speranza, and G. Titomanlio. Modeling of morphology evolution in the injection molding process of thermoplastic polymers. *Progress in Polymer Science*, 30(12):1185–1222, 2005.

[5] A. Gaspar-Cunha and J.A. Covas. Design of extrusion screws using an optimisation approach. In *ANTEC 2001 Conference Proceedings*, page 5, 2001.

[6] A. Gaspar-Cunha and J.A. Covas. Designing screws for polymer extrusion. In *4 th International ESAFORM Conference on Material Forming*, pages 793–796, 2001.

[7] J.M. Nóbrega, O.S. Carneiro, P.J. Oliveira, and F.T. Pinho. Flow Balancing in Extrusion Dies for Thermoplastic Profiles Part I: Automatic Design. *International Polymer Processing*, 18(3):298–306, 2003.

[8] J.M. Nóbrega and O.S. Carneiro. Optimising cooling performance of calibrators for extruded profiles. *Plastics, Rubber and Composites*, 35(9):387–392, 2006.

[9] O.S. Carneiro, J.M. Nóbrega, F.T. Pinho, and P.J. Oliveira. Computer aided rheological design of extrusion dies for profiles. *Journal of Materials Processing Technology*, 114(1):75–86, 2001.

[10] Q. Zeng, A. Yu, and G. Lu. Multiscale modeling and simulation of polymer nanocomposites. *Progress in Polymer Science*, 33(2):191–269, 2008.

[11] H.K. Versteeg and W. Malalasekera. *An Introduction to Computational Fluid Dynamics The Finite Volume Method*. Pearson Education Limited, 2nd edition, 2007.

[12] J. Blazek. *Computational Fluid Dynamics: Principles and Applications, Second Edition: (Book with accompanying CD) (Hardcover)*. Elsevier Science; 2 edition, 2006.

[13] J.Y. Yoo and Na Y. A numerical study of the planar contraction flow of a viscoelastic fluid using the SIMPLER algorithm. *Journal of Non-Newtonian Fluid Mechanics*, 39(1):89 – 106, 1991.

[14] S. Edussuriya and C. Bailey. A cell-centred finite volume method for modelling viscoelastic flow. *Journal of Non-Newtonian Fluid Mechanics*, 117(1):47–61, 2004.

[15] P.J. Oliveira, F.T. Pinho, and G.A. Pinto. Numerical simulation of non-linear elastic flows with a general collocated finite-volume method. *Journal of non-newtonian fluid mechanics*, 79(1):1–43, 1998.

[16] N. Phan-Thien and H.-S. Dou. Viscoelastic flow past a cylinder: drag coefficient. *Computer Methods in Applied Mechanics and Engineering*, 180(3-4):243–266, 1999.

[17] K. Nakahashi, Y. Ito, and F. Togashi. Some challenges of realistic flow simulations by unstructured grid CFD. *International Journal for Numerical Methods in Fluids*, 43(6-7):769–783, 2003.

[18] B. Bihari, S. Ramakrishnan, V. Shankar, and S. Palaniswamy. Massively parallel implementation of an explicit CFD algorithm on unstructured grids. In *Fifteenth International Conference on Numerical Methods in Fluid Dynamics*, pages 438–443. Springer, 1997.

[19] P.F. Fischer and A.T. Patera. Parallel Simulation of Viscous Incompressible Flows. *Annual Review of Fluid Mechanics*, 26(1):483–527, 1994.

[20] C.H. Tai and Y. Zhao. Parallel unsteady incompressible viscous flow computations using an unstructured multigrid method. *Journal of Computational Physics*, 192(1):277–311, 2003.

[21] H.-S. Dou and N. Phan-Thien. Parallelisation of an unstructured finite volume code with PVM: viscoelastic flow around a cylinder. *Journal of Non-Newtonian Fluid Mechanics*, 77(1-2):21–51, 1998.

[22] G. Chen, G. Sun, Y. Xu, and B. Long. Integrated research of parallel computing: Status and future. *Chinese Science Bulletin*, 54(11):1845–1853, 2009.

[23] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[24] T. Rauber and G. Runger. *Parallel Programming: For Multicore and Cluster Systems*, volume 54. Springer-Verlag New York Inc, 2010.

[25] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn, and T.J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[26] K.A. Hawick, A. Leist, and D.P. Playne. Mixing Multi-Core CPUs and GPUs for Scientific Simulation Software. Technical report, CSTN-091, Computer Science, Massey University, 2009.

[27] D.B. Kirk and W.-M. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach (Applications of GPU Computing Series)*. Morgan Kaufmann, 1st edition, 2010.

[28] V. Mihalef, D. Metaxas, and M. Sussman. Animation and control of breaking waves. *Proceedings of the 2004 ACM SIGGRAPHEurographics symposium on Computer animation SCA 04*, page 315, 2004.

[29] M. Müller, R. Keiser, A. Nealen, M. Pauly, M. Gross, and M. Alexa. Point based animation of elastic, plastic and melting objects. *Proceedings of ACM SIGGRAPH / Eurographics Symposium on Computer Animation 2004*, pages 141–151, 2004.

[30] M. Müller, S. Schirm, and M. Teschner. Interactive blood simulation for virtual surgery based on smoothed particle hydrodynamics. *Technology and health care : official journal of the European Society for Engineering and Medicine*, 12(1):25–31, 2004.

[31] M. Johnson, D.P. Playne, and K.A. Hawick. Data-parallelism and gpus for lattice gas fluid simulations. In *Proceedings of PDPTA*, pages 210–216, 2010.

[32] M. Harris. Fast fluid dynamics simulation on the gpu. In Randima Fernando, editor, *GPU Gems*, chapter 38. Addison Wesley Professional, 2004.

[33] X. Cui, Y. Chen, and H. Mei. Improving performance of matrix multiplication and FFT on GPU. In *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*, pages 42–48. IEEE, 2010.

[34] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. Technical report, NVIDIA Corporation, December 2008.

[35] A. Kerr, D. Campbell, and M. Richards. *QR decomposition on GPUs.* ACM Press, New York, New York, USA, 2009.

[36] T. Hagen, K.A. Lie, and Natvig J.R. Solving the euler equations on graphics processing units. In *International Conference on Computational Science (4)*, pages 220–227, 2006.

[37] T. Brandvik and G. Pullan. Acceleration of a 3D Euler Solver using Commodity Graphics Hardware. In *46th AIAA Aerospace Sciences Meeting*. American Institute of Aeronautics and Astronautics, 2008.

[38] E. Elsen, P. Legresley, and E. Darve. Large calculation of the flow over a hypersonic vehicle using a GPU. *Journal of Computational Physics*, 227(24):10148–10161, 2008.

[39] D. Goddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, H. Wobker, C. Becker, and S. Turek. Using GPUs to improve multigrid solver performance on a cluster. *International Journal of Computational Science and Engineering*, 4(1):36–55, 2008.

[40] J.M. Cohen and J. Molemake. A Fast Double Precision CFD Code using CUDA. In *21st International Conference on Parallel Computational Fluid Dynamics (ParCFD2009)*, pages 414–429, 2009.

[41] A. Corrigan, F.F. Camelli, and R. Löhner. Running unstructured grid-based cfd solvers on modern graphics hardware. *International Journal for Numerical Methods in Fluids*, 66(2):221–229, 2011.

[42] I.C. Kampolis, X.S. Trompoukis, V.G. Asouti, and K.C. Giannakoglou. CFD-based analysis and two-level aerodynamic optimization on graphics processing units. *Computer Methods in Applied Mechanics and Engineering*, 199(9-12):712–722, 2010.

[43] V.G. Asouti, X.S. Trompoukis, I.C. Kampolis, and K.C. Giannakoglou. Unsteady CFD computations using vertex-centered finite volumes for unstructured grids on Graphics Processing Units. *International Journal for Numerical Methods in Fluids*, 67(2):232–246, 2010.

[44] L. Sun, S. Mathur, and J Murthy. An Unstructured Finite-Volume Method for Incompressible Flows with Complex Immersed Boundaries. *Numerical Heat Transfer, Part B: Fundamentals*, 58(4):217–241, 2010.

[45] F. A. Morrison. *Understanding Rheology*. Oxford University Press, 2001.

[46] S.R. Mathur and J.Y. Murthy. A Pressure-Based Method for Unstructured Meshes. *Numerical Heat Transfer Part B Fundamentals*, 31(2):195–215, 1997.

[47] M. Darwish and F. Moukalled. TVD schemes for unstructured grids. *International Journal of Heat and Mass Transfer*, 46(4):599–611, 2003.

[48] S.V. Patankar. *Numerical Heat Transfer and Fluid Flow.* Taylor & Francis, 1984.

[49] C.M. Rhie and W.L. Chow. Numerical study of the turbulent flow past an airfoil with trailing edge separation. *AIAA Journal*, 21(11):1525–1532, 1982.

[50] C. Vuik and A. Saghir. The Krylov accelerated SIMPLE (R) method for flow problems in industrial furnaces. *Journal for Numerical methods in*, 33(7):1027–1040, 2000.

[51] Y. Saad. *Iterative Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics, 2nd edition, 2003.

[52] J.J. Dongarra, J. Du Croz, S. Hammerling, and I.S. Duff. Algorithm 679: A set of level 3 basic linear algebra subprograms: model implementation and test programs. *ACM Trans. Math. Softw.*, 16:18–28, 1990.

[53] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*, volume 79. SIAM, 1999.

[54] Intel Corporation. Intel Math Kernel Library – Reference Manual, 2009.

[55] S. Balay, K. Buschelman, V. Eijkhout, W.D. Gropp, D. Kaushik, M.G. Knepley, L.C. McInnes, B.F. Smith, and H. Zhang. PETSc User's Manual. Technical report, Argonne National Laboratory, 2010.

[56] J.R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1994.

[57] C. Geuzaine and J.-F. Remacle. Gmsh: A 3-d finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, 2009.

[58] A. Henderson. *ParaView User's Guide (v3.10).* Kitware, Inc, 2011.

[59] R.M. Stallman and the GCC Developer Community. *Using the GNU Compiler Collection.* For GCC version 4.6.1. GNU, 2011.

[60] NVIDIA. *CUDA C Programming Guide Version 4.0.* NVIDIA Corporation, Santa Clara, CA, USA, 2011.

[61] J.P. Walters, V. Balu, S. Kompalli, and V. Chaudhary. Evaluating the use of gpus in liver image segmentation and hmmer database searches. In *IEEE International Symposium on Parallel & Distributed Processing (IPDPS'09)*, pages 1–12, 2009.

[62] J.E. Stone, J.C. Phillips, P.L. Freddolino, D.J. Hardy, L.G. Trabuco, and K. Schulten. Accelerating molecular modeling applications with graphics processors. *J. Comput. Chem.*, 28(16):2618–2640, 2007.

[63] NVIDIA. *Fermi Compute Architecture Whitepaper.* NVIDIA Corporation, 2009.

[64] NVIDIA. *CUDA C Programming Best Practices Guide Version 4.0*, 2011.

[65] C. Cecka, A. J. Lew, and E. Darve. Assembly of finite element methods on graphics processors. *International Journal for Numerical Methods in Engineering*, 85(5):1–6, 2010.

[66] A. Gaikwad and I.M. Toke. Parallel iterative linear solvers on gpu: A financial engineering case. In *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, volume 0 of *PDP '10*, pages 607–614, Washington, DC, USA, 2010. IEEE Computer Society.

[67] M. Ament, G. Knittel, D. Weiskopf, and W. Strasser. A parallel preconditioned conjugate gradient solver for the poisson problem on a multi-gpu platform. *2010 18th Euromicro Conference on Parallel Distributed and Networkbased Processing*, pages 583–592, 2010.

[68] U. Ghia, K. Ghia, and C. Shin. High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method1. *Journal of Computational Physics*, 48(3):387–411, 1982.

[69] O. Botella and R. Peyret. Benchmark spectral results on the lid-driven cavity flow. *Computers & Fluids*, 27(4):421–433, 1998.

[70] C. Bruneau and M. Saad. The 2D lid-driven cavity problem revisited. *Computers & Fluids*, 35(3):326–348, 2006.

[71] C. Geuzaine and M. N. Artyomov. *Gmsh Reference Manual*, 2009.

Appendix A

Mesh Generation

The process to generate a two-dimensional mesh using *gmsh* starts by defining a geometry, in this case a square geometry. This example is the mesh generation process used for the assessment of the lid-driven cavity flow problem. In order to generate a mesh, the user needs to create a geometry file with the .geo extension. The script for the square.geo file is presented below with some comments. The command lines to run *gmsh* executable and generate the mesh file are given below for Linux and Windows systems.

Linux

```
./gmsh square.geo -2 -o square.msh
```

Windows

```
gmsh.exe square.geo -2 -o square.msh
```

The output (-o) is a mesh file with .msh extension. This is a simple mesh generation example; for other options and more complex meshes the reader is refered to the *gmsh* reference manual [71]. Snapshots of gmsh's graphical user interface for the geometry file square.geo and the mesh file square.msh are presented by Figure A.1 and Figure A.2, respectively.

```
/*****************************************************
This script generates a square with dimensions Lx and Ly
and generates a two-dimensional mesh.
*****************************************************/

// Dimensions in meters
Lx = 0.1;
Ly = 0.1;

Point(1) = {0.0, 0.0, 0.0};
Point(2) = {Lx,  0.0, 0.0};
Point(3) = {Lx,  Ly,  0.0};
Point(4) = {0.0, Ly,  0.0};

Line(101) = {1, 2};
Line(102) = {2, 3};
Line(103) = {3, 4};
Line(104) = {4, 1};

// Number of points along the boundary line
size=60;

// Mesh refinement is done by increasing
// the number of points along the boundary line.

Transfinite Line{101} = size;
Transfinite Line{102} = size;
Transfinite Line{103} = size;
Transfinite Line{104} = size;

Line Loop(201) = {101, 102, 103, 104};

Plane Surface(301) = {201};
```

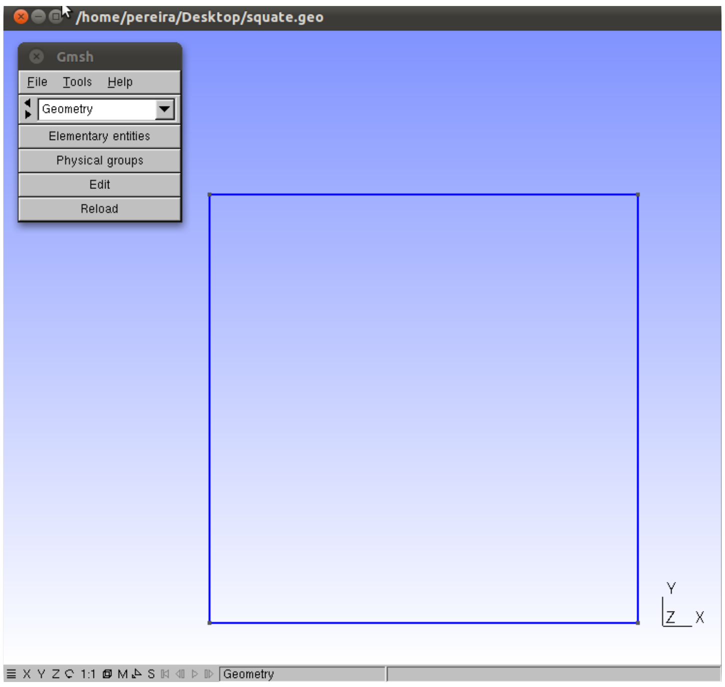


Figure A.1: Snapshot of geometry file square.geo. generated in *gmsh*.

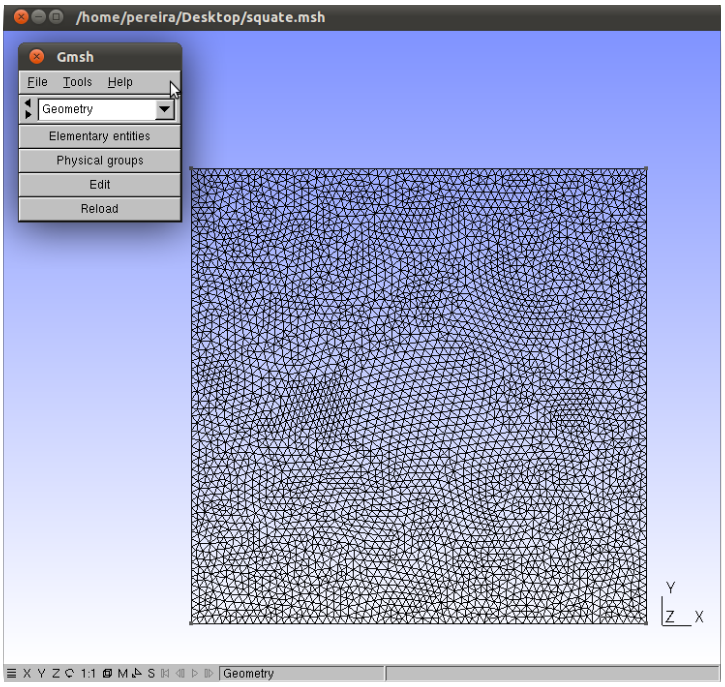


Figure A.2: Snapshot of mesh file square.msh generated in *gmsh*.

Appendix B

Input File

The input file is a requirement of the developed numerical code (the same input file is used for the CPU and GPU codes) and a brief description of its features is given, in case someone might want to use the program in a near future. The file format was divided in sections by using capital letters for organisation purposes, which are described hereafter.

Regarding the material properties, the user must declare the density and viscosity for fluid problems involving Newtonian fluids. For problems involving generalized Newtonian fluids, the Bird-Carreau model (2.6) introduced in Section 2.1 requires 4 parameters, the zero shear viscosity (η_0), the infinite viscosity (η_∞), the n from power law and the Carreau relaxation time (λ). The consistency constant (K) from the Power Law model (2.23) can also be used whenever the user chooses as inlet boundary condition, a fully developed velocity profile for a Power Law fluid. For problems involving heat-transfer the specific heat and thermal conductivity are required.

The simulation parameters correspond to the relaxation of the unsteady term (2.17) in momentum equations and to the relaxation of pressure correction in the SIMPLE algorithm. The latter parameter is the inverse of the α_p in pressure-correction equation (2.45).

The section of the iterative solvers has 4 input parameters, the SolverID, the maximum error, the maximum number of iterations and the percentage error decrease. The SolverID corresponds to the pair of iterative solvers used to solve momentum and pressure-correction equation, respectively, allowing the following combinations:

```
SolverID 1     Jacobi and Conjugate−Gradient
SolverID 2     Jacobi and Jacobi
SolverID 3     BICGSTAB and Conjugate−Gradient
```

The maximum error corresponds to the residual value of SIMPLE algorithm; whenever the normalised residuals of momentum and pressure correction equations fall below this value, the program terminates. The other two parameters correspond to the internal stopping criteria of the iterative solvers. Normally the iterative solver exits when the relative error reduces up to a certain value (see stopping criteria 2 in Subsection 2.4.4).

The following section is optional and enables the user to initialise a temperature, a velocity and pressure fields. Furthermore, the boundary conditions are defined by a label, a type and a value. The number 4 corresponds to the existing number of boundaries. A list of possible boundary type for temperature, velocity and pressure is given below. In case of inlet boundaries with fully developed velocity profiles, the user must input a pressure value to the corresponding pressure boundary (type 31).

```
Temperature
0         Imposed temperature
1         Imposed flux
Velocity
11        Imposed velocity (Wall)
12        Fully developed outflow
13        Symmetry
14        Fully developed velocity profile (Newtonian)
15        Inflow imposed velocity
16        Fully developed half−velocity profile (Newtonian)
17        Fully developed velocity profile (Power–Law)
18        Fully developed half−velocity profile (Power–Law)
Pressure
31        Read only pressure value
32        Linear pressure field along x
```

There are 10 flux limiter choices for the estimation of the convective term (see Section 2.2 for details) and the complete list is presented in Table 2.1.

The last two lines of the input file are concerned with the use of SIMPLE, the under-relaxation of momentum equations and which fluid model to be used (N for Newtonian and C for Carreau). The flag 1 or 0 stands for true or false, respectively. The program is also able to solve a simple pure heat transfer, for that turn off (0) the SIMPLE flag and turn on (1) the pure heat transfer flag.

As final remark, the file extension of the input file is .ini.

```
/****************************************************
Example of input file to run the lid-driven cavity
flow problem.
****************************************************/
MATERIAL_PROPERTIES
[Density(kg/m3)]                            1000.0
[SpecificHeat(J/Kg.K)]                      4000.0
[ThermalConductivity(W/mK)]                 0.58
[Viscosity(Pa.s)]                           0.001
[ZeroShearViscosity(Pa.s)]                  1.0
[InfiniteShearViscosity(Pa.s)]              0.0
[CarreauRelaxationTime]                     3.21
[nPowerLaw]                                 0.3
[KPowerLaw]                                 1.0

SIMULATION_PARAMETERS
[UnsteadyTimeStep]                          0.8
[PressureRelaxation]                        50.0

ITERATIVE_SOLVER
[SolverID]                                  1
[MaxError]                                  1.0E-4
[MaxIterations]                             500
[PercentualErrorDecrease]                   1.0E-1

INITIAL_VARIABLES
[Temperature]                               0.0
[VelX]                                      0.0
[VelY]                                      0.0
[Pressure]                                  0.0

BOUNDARY_VALUES
4
[Label]_[Type]_[Value]_[Type]_[Vx]___[Vy]__[Type]__[Pressure]
101       0      0.0       11     0.0     0.0     31        0.0
102       0      0.0       11     0.0     0.0     31        0.0
103       0      0.0       11     0.001   0.0     31        0.0
104       0      500.0     11     0.0     0.0     31        0.0

FluxLimiter      10

[SIMPLE, UnderRelaxation, FluidModel]       1    1    N
[PureHeatTransfer]                          0
```

Appendix C

Implementation Details - CPU

The developed numerical code is considerably extensive to be completely presented, instead, the main features and relevant details are examined. The purpose is to enable the reader to understand how the code was built and how it works, without going through all the lines of the code.

One way to start, is by explaining its organisation, by addressing the data structure. For clarity and organisation reasons, several structures are used but the most important are exposed here; the triangle, the edge and the boundary edge structures, each containing information about its specific type (see list of structures below). For instance, each triangle has three edges, three vertices, the variables that are stored in the centre of each cell, among other less relevant data. The same for the edges which contain two adjacent triangles and two vertices each. This way of storing the information was quite relevant for the implementation of the code.

```
// triangle data structure
typedef struct {
   int edge[3];          // three edges
   int vertex[3];        // three vertices
   double value[8];      // variables: [0] temp,[1] Vx,[2] Vy,[3]-pressure
   ...
} type_triangles ;
// edge data structure
typedef struct  {
   int vertex[2];        // two vertices
   int cell[2];          // cell[0] and cell[1]
   ...
} type_edge;
```

```
// boundary edge data structure
typedef struct  {
   int vertex[2];       // two vertices
   int cell[2];         // cell[0] and cell[1]
   ...
} type_boundary_edge ;
```

A global image of the main file of the developed numerical code is given below, with emphasis to the most important routines. Some comments (in gray) help the reader going through the code.

```
/*******************************************
Flow Modelling Code on CPU
SPPereira
*******************************************/
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#include "math.h"
#include "wchar.h"
#include "sys/time.h"
...
int main(int argc, char *argv[]) {
   ...
   GenerateMesh(...);                // generate mesh file
   ReadInputFile(...);               // read input
   GetGeometricValues(...);          // compute geometric parameters
   AllocateCoefficients(...);        // alloc. & sparse matrix organisat.
   CrossDiffusionStoreData(...);     // stores info. about vertex neighb.
   ...
   // the SIMPLE algorithm
   do {
      // solve x-momentum equation
      AssembleDiffusiveTerm(...);
      AssembleConvectiveTerm(...);
      UnsteadyTerm(...);
      SourceContribution(...);
      IterativeSolver(...);          // Jacobi or BICGSTAB
      // solve y-momentum equation
      AssembleDiffusiveTerm(...);
      AssembleConvectiveTerm(...);
      UnsteadyTerm(...);
      SourceContribution(...);
      IterativeSolver(...);
      ...
      // solve pressure-correction equation
      AssemblePressureEquation(...);
      IterativeSolver(...);          // CG
      ...
      // velocity and pressure corrections
      FieldCorrections(...);
   } while(stopping criteria not satisfied);
   ...
}
```

An important aspect yet to be discussed is how to assemble the system of equations. A closer look at the routine `AssembleDiffusiveTerm(...)` will enable the reader to understand how the assembly process was performed. Please consult Section 2.2 for the estimation of the diffusive term. The source code presented below is essentially a loop that collects information from cell[0] and cell[1], the adjacent cells of each edge. In this way the diffusive and convective term are assembled. Please recall the linear form of assembling the system of equations (2.26) where the a_P is the diagonal, a_{nb} the off-diagonal and S_ϕ the source contributions. The unsteady term and source term were assembled in a similar way, but instead, a loop over all the triangles was performed.

```
/***************************************************************
Assembly of Equations on the CPU - Contribution from Diffusive Term
***************************************************************/
int  GetDiffusionInteriorCoeffVelocity(...) {
   int i;
   for (i=0; i<n_edge; i++) {
      ...
      // contribution from cell[0]
      a_coef[...] += edge[i].NDC * edgeViscosity;         // diagonal
      a_coef[...] -= edge[i].NDC * edgeViscosity;         // off-diagonal
      b_src  [...] -= edge[i].CDC * edgeViscosity *edge[i].CrossDiffusion;
      // contribution from cell[1]
      a_coef[...] += edge[i].NDC * edgeViscosity;
      a_coef[...] -= edge[i].NDC * edgeViscosity;
      b_src  [...] += edge[i].CDC * edgeViscosity*edge[i].CrossDiffusion;
   }
return 0;
}
```

After the assembly process, an iterative solver was used to compute the solution to the system of equations. The Jacobi or BICGSTAB methods can be used to solve momentum equations, and the conjugate-gradient method to solve the pressure-correction equation. The SIMPLE algorithm converges when sufficient convergence is reached for all its equations.

Appendix D

Paraview

Paraview is a powerful open-source parallel program largely used for scientific visualisation of small and large data sets. It contains several quantitative and qualitative tools that enable the quick build up of visualisation sets required by the user. The parallel nature of its implementation enables the use of distributed systems to analyse enormous data sets, that are not possible with a single processor. Therefore, Paraview is commonly used in Supercomputers, including my laptop. The interest reader should take a look at the website for more informations (www.paraview.org).

One way to use Paraview's post-processing tools is by creating a .vtk file with the corresponding data sets. VTK stands for Visualization ToolKit, another open-source software system for 3D computer graphics, image processing and visualisation. VTK is the basis of advanced visualisation software such as Paraview. The choice for VTK's files format was based on the simplicity of generating an ASCI .vtk file and because it is a general format widely used in many post-processing programs.

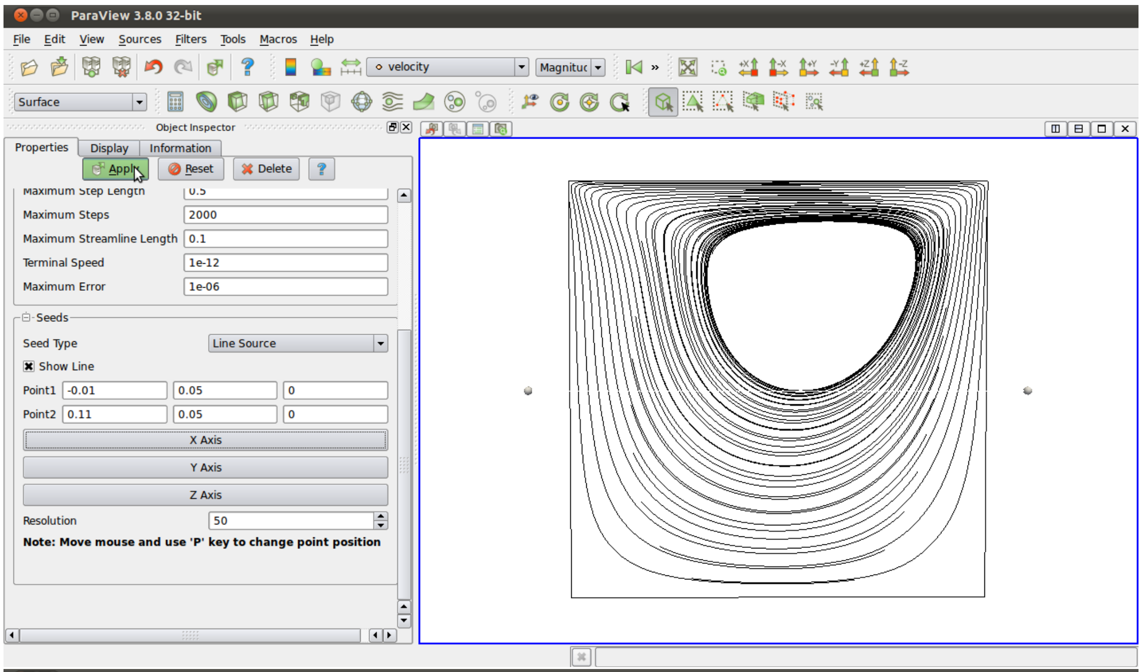
In this work, Paraview was used to plot streamlines by using the *Stream Tracer* filter. In order to use the *Stream Tracer* filter, the velocity data sets must be given in order to the vertices (Point Data) and not in order to the cell centres (Cell Data). Since the developed numerical code uses the cell-centred arrangement, where variables are stored at the centre of the cells, a pre-processing step is required to interpolate the variables from the cell centres to the vertices. This was done by simple averaging data from the neighbour cells of each vertex. Is important to remark that the velocity must be normalised. As an example, a brief description on how to plot the streamlines for the lid-driven cavity flow problem is given. After loading the output file from the numerical code with the .vtk extension, the first step is to normalise the velocities by using the Calculator filter.

Filters → Alphabetical → Calculator

In the Calculator filter, the velocities were normalised by using the command *norm*(velocity). The apply button was pressed to proceed with the calculation. The *Stream Tracer* uses a Runge-Kutta integrator type to compute the streamlines and its use is illustrated in Figure D.1 by two snapshots of two stream traces. The following path opens the *Stream Tracer* object inspector.

Filters → Alphabetical → Stream Tracer

The first stream trace (Figure D.1) was generated by choosing, in the seeds menu list, the line source option as seed type and the x axis as line. Another stream trace was used but this time by using the y axis as line source (see bottom snapshot in Figure D.1). Other stream traces could be used to catch the vorticity in the corners. For more details on how to plot stream lines and other features of Paraview the interested user should read the documentation [58].


ParaView 3.8.0 32-bit
File Edit View Sources Filters Tools Macros Help
velocity
Magnituc
Surface
Object Inspector
Properties Display Information
Apply Reset Delete

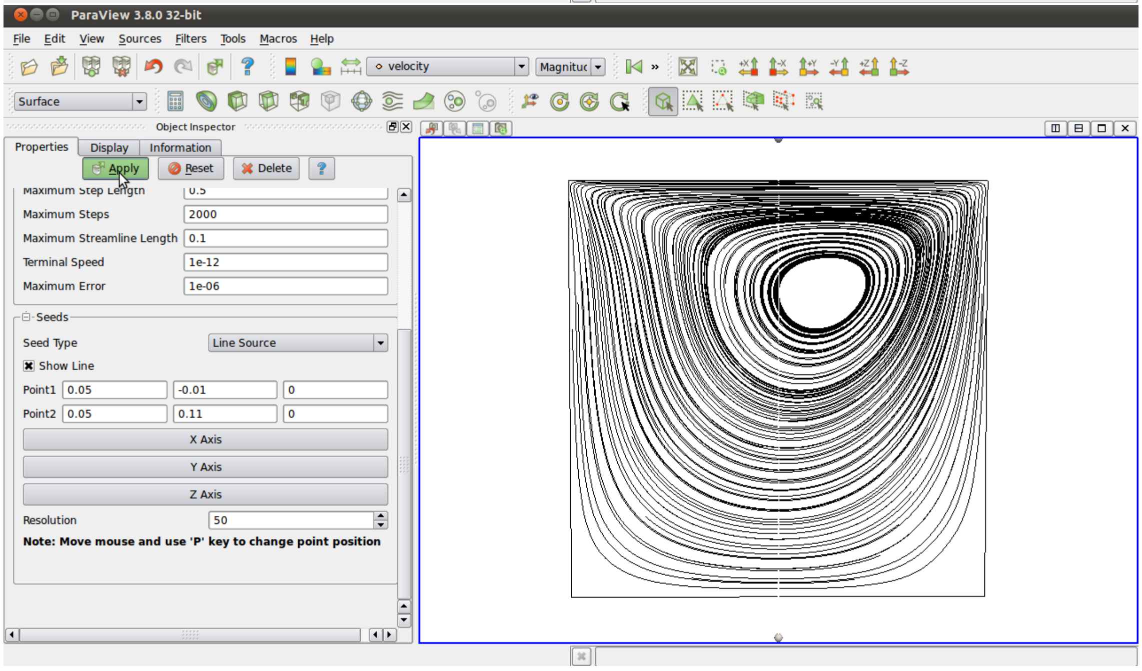
Maximum Step Length 0.5
Maximum Steps 2000
Maximum Streamline Length 0.1
Terminal Speed 1e-12
Maximum Error 1e-06

Seeds

Seed Type Line Source
[✓ Show Line]
Point1 -0.01 0.05 0
Point2 0.11 0.05 0

Resolution 50

Note: Move mouse and use 'P' key to change point position


ParaView 3.8.0 32-bit
File Edit View Sources Filters Tools Macros Help
velocity
Magnituc
Surface
Object Inspector
Properties Display Information
Apply Reset Delete

Maximum Step Length 0.5
Maximum Steps 2000
Maximum Streamline Length 0.1
Terminal Speed 1e-12
Maximum Error 1e-06

Seeds

Seed Type Line Source
[✓ Show Line]
Point1 0.05 -0.01 0
Point2 0.05 0.11 0

Resolution 50

Note: Move mouse and use 'P' key to change point position

Figure D.1: Two snapshots of Paraview's graphical user interface, when employing the *Stream Tracer* filter.

Appendix E

Implementation Details - GPU

The code developed to run on the GPU has some similarities to the CPU code in Appendix C. The reader is advised to check also the implementation details on the CPU. As already mentioned in Section 3.3, and additional routine was required to be able to assemble the system of equations on parallel architectures. This routine was designated by colouring scheme and lead to the introduction of a new variable in the edge data structure, the colour (or an integer that corresponds to a colour).

```
// edge data structure
typedef struct  {
   int vertex[2];                // two vertices
   int cell[2];                  // cell[0] and cell[1]
   int color;
   ...
} type_edge;
```

In Chapter 3 it was mentioned that the device has its own DRAM memory, thus we have to use the specific syntax for declaration of variables that will exist in the device memory. Trying to access a device variable declared as a host variable and *vice versa* is a memory violation and actually this is a common mistake. In order to avoid these typing errors a suffix or prefix `dev` was added to each device variable. The use of variable qualifiers such as `__device__` `__constant__` specifies the memory location on the device of a variable, in this case the fast constant memory. Below are some examples of variables declared on the constant memory, because they are used all over the numerical code and accesses to these variables are extremely fast. The function `cudaMemcpyToSymbol(...)` was used to copy the host variable to the device memory.

```
__device__ __constant__ int NElem_dev;        // number of cells
__device__ __constant__ int dev_n_edge;       // number of edges
__device__ __constant__ int dev_n_bnd_edge;   // number of boundary edges
__device__ __constant__ double dev_density;   // density
__device__ __constant__ double dev_TimeStep;  // time step to relax mom. eqs.
...
cudaMemcpyToSymbol("NElem_dev", &NElem, sizeof(int), 0,
    cudaMemcpyHostToDevice);
cudaMemcpyToSymbol("dev_n_edge", &n_edge, sizeof(int), 0,
    cudaMemcpyHostToDevice);
cudaMemcpyToSymbol("dev_n_bnd_edge", &n_bnd_edge, sizeof(int), 0,
    cudaMemcpyHostToDevice);
cudaMemcpyToSymbol("dev_density", &density, sizeof(double), 0,
    cudaMemcpyHostToDevice);
cudaMemcpyToSymbol("dev_TimeStep", &TimeStep, sizeof(double), 0,
    cudaMemcpyHostToDevice);
```

In the case of arrays or arrays of structures, the variables need to be allocated on the device memory by using the function `cudaMalloc(...)`, similar to the `malloc(...)` for allocation on the host. Only then the data can be copied (to the global memory) from the host to the device memory. An example of memory allocation on the device, of the triangles and edges type structures, is given below. The function `cudaMemcpy((...)` is used to copy the structures from the host memory to the device memory.

```
cudaMalloc( (void**)&dev_triangle, (n_triang) * sizeof(type_triangles) );
cudaMalloc( (void**)&dev_edge, (n_edge) * sizeof(type_edge) );
...
cudaMemcpy(dev_edge, edge, (n_edge) * sizeof(type_edge),
    cudaMemcpyHostToDevice );
cudaMemcpy(dev_triangle, triangle, (NElem) * sizeof(type_triangles),
    cudaMemcpyHostToDevice );
```

A global image of the main file of the code on GPU is given below, with emphasis to the most relevant routines. Comments (in gray) help the reader understand the code. The reader can find some similarities with the CPU implementation in terms of sequence of computations and function calls. The code developed to run on the GPU is more extensive, and several kernel calls are required to assemble the diffusive term, for instance, while in the CPU a simple function with a for loop was required. This strategy was adopted to avoid race conditions, as already mentioned.

```
/*****************************************
Flow Modelling Code on GPU
SPPereira
*****************************************/
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#include "math.h"
#include "wchar.h"
#include "sys/time.h"
// header files for CUDA
#include "cuda.h"
#include "cuda_runtime.h"
#include "device_functions.h"
#include "device_launch_parameters.h"
#include "cutil_inline.h"
#include "cusparse.h"
#include "cublas.h"
...
# define ThreadsPerBlock 256
int main(int argc, char *argv[]) {
   // computations on the CPU
   ...
   GenerateMesh(...);            // generate mesh file
   ReadInputFile(...);           // read input
   GetGeometricValues(...);      // compute geometric parameters
   AllocateCoefficients(...);    // alloc. & sparse matrix organisat.
   CrossDiffusionStoreData(...); // stores info. about vertex neighb.
   ColoringScheme(...);          // colors groups of independent edges
   CopyToDevice(...);            // alloc. & copies data to device memory
   ...
   // computations on the GPU
   // the SIMPLE algorithm on GPU
   do {
      // solve x-momentum equation
      AssembleDiffTermZero<<<NumberOfBlocks,ThreadsPerBlock>>>(blue,...);
      AssembleDiffTermZero<<<NumberOfBlocks,ThreadsPerBlock>>>(green,...);
      AssembleDiffTermZero<<<NumberOfBlocks,ThreadsPerBlock>>>(red,...);
      AssembleDiffTermOne <<<NumberOfBlocks,ThreadsPerBlock>>>(blue,...);
      AssembleDiffTermOne <<<NumberOfBlocks,ThreadsPerBlock>>>(green,...);
      AssembleDiffTermOne <<<NumberOfBlocks,ThreadsPerBlock>>>(red,...);
      AssembleConvTermZero<<<NumberOfBlocks,ThreadsPerBlock>>>(blue,...);
      AssembleConvTermZero<<<NumberOfBlocks,ThreadsPerBlock>>>(green,...);
      AssembleConvTermZero<<<NumberOfBlocks,ThreadsPerBlock>>>(red,...);
      AssembleConvTermOne <<<NumberOfBlocks,ThreadsPerBlock>>>(blue,...);
      AssembleConvTermOne <<<NumberOfBlocks,ThreadsPerBlock>>>(green,...);
      AssembleConvTermOne <<<NumberOfBlocks,ThreadsPerBlock>>>(red,...);
      UnsteadyTerm(...);
      SourceContribution(...);
      IterativeSolver(...);           // Jacobi or BICGSTAB

      // solve y-momentum equation
      AssembleDiffTermZero<<<NumberOfBlocks,ThreadsPerBlock>>>(blue,...);
      AssembleDiffTermZero<<<NumberOfBlocks,ThreadsPerBlock>>>(green,...);
      AssembleDiffTermZero<<<NumberOfBlocks,ThreadsPerBlock>>>(red,...);
      AssembleDiffTermOne <<<NumberOfBlocks,ThreadsPerBlock>>>(blue,...);
      AssembleDiffTermOne <<<NumberOfBlocks,ThreadsPerBlock>>>(green,...);
      AssembleDiffTermOne <<<NumberOfBlocks,ThreadsPerBlock>>>(red,...);
      AssembleConvTermZero<<<NumberOfBlocks,ThreadsPerBlock>>>(blue,...);
      AssembleConvTermZero<<<NumberOfBlocks,ThreadsPerBlock>>>(green,...);
      AssembleConvTermZero<<<NumberOfBlocks,ThreadsPerBlock>>>(red,...);
      AssembleConvTermOne <<<NumberOfBlocks,ThreadsPerBlock>>>(blue,...);
      AssembleConvTermOne <<<NumberOfBlocks,ThreadsPerBlock>>>(green,...);
      AssembleConvTermOne <<<NumberOfBlocks,ThreadsPerBlock>>>(red,...);
      ...
      UnsteadyTerm(...);
      SourceContribution(...);
      IterativeSolver(...);
      ...
      // solve pressure-correction equation
      AssemblePressureEqZero<<<NumberOfBlocks,ThreadsPerBlock>>>(blue,...);
      AssemblePressureEqZero<<<NumberOfBlocks,ThreadsPerBlock>>>(green,...);
      AssemblePressureEqZero<<<NumberOfBlocks,ThreadsPerBlock>>>(red,...);
      AssemblePressureEqOne <<<NumberOfBlocks,ThreadsPerBlock>>>(blue,...);
      AssemblePressureEqOne <<<NumberOfBlocks,ThreadsPerBlock>>>(green,...);
      AssemblePressureEqOne <<<NumberOfBlocks,ThreadsPerBlock>>>(red,...);
      IterativeSolver(...);           // CG
      ...
      // velocity and pressure corrections
      FieldCorrections(...);
   } while(stopping criteria not satisfied);
   ...
}
```

The source for the assembly of the system of equations is presented below. The reader is advised to compare with the source code from the CPU implementation (see Appendix C). This kernel assembles the diffusive term for the cells with index zero (recall that each edge has two adjacent cells) and the assembly process is made by collecting the cell contributions from all edges. A remark to the use the built in variables such as `ThreadIdx.x`, `BlockIdx.x` and `gridDim.x` to address an index to each thread `tid`. It is a very simple kernel and this way was adopted to assemble the momentum and pressure-correction equations. Note that only those contributions from cells with a specific colour are accounted for.

```
/*************************************************************
Assembly of Equations on the GPU - Contribution from Diffusive Term
*************************************************************/
__global__ void  AssembleDiffTermZero(int color ,...) {
     int tid=threadIdx.x + blockIdx.x * blockDim.x;
     if ((tid<dev_n_edge) && (dev_edge[tid].colorZero==color)) {
          ...
          // Diagonal Contribution
          A_dev[...] += (NDC*dev_viscosity);
          // Off-Diagonal Contribution
          A_dev[...] =- (NDC*dev_viscosity);
          // Source Contribution
          rhs_dev[...] -= (CDC*dev_viscosity*dev_edge[tid].CrossDiffusion);
     }
}
```

The assembled equations were solved with an iterative method as in the CPU implementation. The iterative solvers on the GPU use the CUSPARSE and CUBLAS library to perform sparse matrix-vector multiplications (*SpMV*), dot product (*ddot*) and scalar multiplication and vector update (*dxapy*). These are libraries similar to the BLAS and Intel's MKL but to be used on the GPU.

Appendix F

Code Profiling

F.1 Poiseuille Flow

Table F.1: Timings in seconds on the CPU of the most important routines for the Poiseuille flow problem with different mesh sizes.

Mesh	**M01**	**M02**	**M03**	**M04**	**M05**	**M06**	**M07**
Mesh Generation	0.17	0.63	1.79	3.95	7.74	14.52	23.88
Cross Diffusion Format	0.002	0.03	0.18	0.58	1.45	3.20	5.87
Solve u-Velocity	12.26	119.58	648.12	957.20	2713.65	5108.03	8199.63
Solve v-Velocity	9.92	54.67	145.24	336.97	528.70	912.86	1277.55
Solve Pressure-Correction	5.88	22.87	53.16	114.94	194.48	303.26	419.12
SIMPLE Corrections	2.31	9.30	21.53	41.39	71.40	115.08	162.04
Total	**30.62**	**207.24**	**840.30**	**1455.50**	**3518.10**	**6457.91**	**10089.25**

Mesh	**M08**	**M09**	**M10**	**M11**	**M12**	**M13**	**M14**
Mesh Generation	41.41	72.45	128.48	208.93	311.81	446.18	623.02
Cross Diffusion Format	12.92	24.94	43.28	66.92	99.68	136.85	187.63
Solve u-Velocity	13673.20	19432.78	29637.41	42488.41	61476.23	82658.93	112789.04
Solve v-Velocity	1974.71	2483.00	3461.17	4776.98	6074.38	7934.80	10480.95
Solve Pressure-Correction	595.98	731.86	955.52	1181.22	1432.95	1749.64	1992.02
SIMPLE Corrections	229.19	285.54	369.22	459.92	557.78	679.80	773.54
Total	**16528.91**	**23029.33**	**34597.23**	**49184.95**	**69955.75**	**93609.73**	**126850.10**

Table F.2: Timings in seconds on the GPU (GTX 480) of the most important routines for the Poiseuille flow problem with different mesh sizes.

Mesh	**M01**	**M02**	**M03**	**M04**	**M05**	**M06**	**M07**
Mesh Generation	0.17	0.65	1.83	3.92	7.94	14.40	24.87
Cross Diffusion Format	0.001	0.03	0.16	0.52	1.41	2.79	5.73
Colouring Scheme	0.006	0.09	0.48	1.61	3.85	8.00	14.61
Copy to Device	0.20	0.20	0.21	0.20	0.21	0.21	0.23
Solve u-Velocity	30.00	90.88	179.29	238.49	513.23	726.12	1098.79
Solve v-Velocity	23.00	38.15	50.93	80.14	90.62	116.22	153.18
Solve Pressure-Correction	14.41	17.59	21.85	32.10	41.53	53.6	69.93
SIMPLE Corrections	1.31	2.28	4.28	7.19	10.44	14.17	19.69
Total	**69.97**	**151.41**	**262.20**	**369.64**	**677.34**	**943.45**	**1401.23**
Mesh	**M08**	**M09**	**M10**	**M11**	**M12**	**M13**	**M14**
Mesh Generation	46.21	80.50	141.63	235.37	347.51	506.91	671.06
Cross Diffusion Format	12.50	24.89	42.61	66.84	98.5	139.25	185.10
Colouring Scheme	24.92	40.29	62.22	94.03	134.51	190.63	251.49
Copy to Device	0.22	0.22	0.23	0.24	0.24	0.25	0.25
Solve u-Velocity	1504.67	1996.94	2660.26	3555.43	4590.03	5816.18	7071.40
Solve v-Velocity	197.60	236.51	293.95	384.61	450.65	566.86	658.58
Solve Pressure-Correction	85.09	100.71	120.86	148.25	170.95	207.10	222.48
SIMPLE Corrections	25.60	31.07	32.39	47.34	55.47	66.67	73.30
Total	**1916.57**	**2535.14**	**3389.73**	**4568.36**	**5890.22**	**7544.30**	**9189.64**

F.2 Lid-Driven Cavity Flow

Table F.3: Timings in seconds on the CPU of the most important routines for the lid-driven cavity flow problem with different mesh sizes.

Mesh	**M01**	**M02**	**M03**	**M04**	**M05**	**M06**	**M07**
Mesh Generation	0.72	5.52	20.3174	77.5762	205.3436	473.1130	1014.08
Cross Diffusion Format	0.05	1.02	5.37	27.50	70.27	155.40	314.96
Solve u-Velocity	21.22	248.85	970.94	2952.60	6301.01	13686.45	27762.14
Solve v-Velocity	21.57	246.07	949.78	2902.65	6304.76	13302.94	26727.20
Solve Pressure-Correction	12.13	90.43	240.0251	492.31	810.34	1258.2545	1886.00
SIMPLE Corrections	5.34	35.02	99.58	205.73	339.62	526.70	778.76
Total	**61.14**	**627.34**	**2286.82**	**6659.79**	**14033.40**	**29405.84**	**58487.29**

Table F.4: Timings in seconds on the GPU (GTX 480) of the most important routines for the lid-driven cavity flow problem with different mesh sizes.

Mesh	**M01**	**M02**	**M03**	**M04**	**M05**	**M06**	**M07**
Mesh Generation	0.73	5.54	20.81	86.83	230.28	522.065	1080.76
Cross Diffusion Format	0.14	2.70	12.62	43.01	97.25	210.46	422.81
Colouring Scheme	0.05	2.71	4.54	26.71	70.02	155.56	312.68
Copy to Device	0.23	0.20	0.22	0.23	0.23	0.23	0.25
Solve u-Velocity	18.28	60.08	138.35	308.72	540.93	978.33	1712.21
Solve v-Velocity	18.50	59.34	136.22	303.90	541.76	954.15	1650.56
Solve Pressure-Correction	9.82	24.04	44.18	75.74	114.64	167.25	232.78
SIMPLE Corrections	2.09	7.89	15.48	28.33	44.10	65.12	92.66
Total	**50.76**	**165.53**	**382.96**	**893.03**	**1669.95**	**3097.92**	**5568.26**

Table F.5: Timings in seconds on the GPU (Tesla C2070) of SIMPLE and copy to device routines for the lid-driven cavity flow problem with different mesh sizes.

Mesh	**M01**	**M02**	**M03**	**M04**	**M05**	**M06**	**M07**
Copy to Device	2.90	2.91	2.91	2.96	2.95	2.99	3.02
SIMPLE	104.59	333.96	697.62	1426.36	2418.67	4028.09	6821.94