Sofia University "St. Kliment Ohridski"
Faculty of Mathematics and Informatics

Department of Numerical Methods and Algorithms

GPU implementation of the FEM for the Navier-Stokes equations

Master's thesis
in Computational Mathematics and Mathematical Modelling

Author: Vasil D. Pashov, FN 25938
Supervisor: Assist. Prof. Ph.D. Tihomir B. Ivanov

November, 2021

Abstract

Nowadays graphics processing units (GPUs) are capable of performing general purpose tasks. Some areas such as Artificial Intelligence, Cryptocurrencies have successfully taken advantage of the GPU hardware and programming model. The goal of this work is to construct a FEM based algorithm for solving the incompressible Navier-Stokes equations, which is suitable, though not limited to, the purposes of computer graphics and the VFX industry.

Three different approaches were researched and the pros and cons with respect to our goals, performance and suitability for GPU (and multi threaded CPU) implementation were discussed. Numerical experiments with a classical model problem (the 2D DFG Benchmark) were conducted in order to evaluate the applicability of the algorithms. The algorithm which completes our goals best splits the differential operator into three parts and each of the resulting equations is solved via a method which takes advantage of the GPU processing power.

Contents

Abbreviations

API Application Programming Interface

CG Conjugate Gradient
CPU Central Processing Unit
CSR Compressed Sparse Row

FEM Finite Element Method
FPS Frames Per Second

GPU Graphics Processing Unit

IC0 Zero Fill-in Incomplete Cholesky

PCG Predonditioned Conjugate Gradient

SM Streaming Multiprocessor
SPD Symmetric Positive Definite

TBB Thread Building Blocks

VFX Visual Effects

Chapter 1

Introduction

In the last 20 years Graphic Processing Units (GPUs) have come a long way from having a fixed rendering pipeline used only for the purpose of computer graphics, to becoming fully capable of performing general purpose tasks. This evolution made it possible to use the raw computing power of GPUs in areas other than computer graphics such as Artificial Intelligence, Cryptocurrencies, Computational Fluid Dynamics and so on.

Though it is true that GPUs have more cores than a regular CPU[1] a comparison between both based purely on the count of cores is not appropriate due to the vast difference in the hardware architecture. Different algorithms can benefit to a different degree of a GPU implementation. Again due to the difference in the architecture one will not fully benefit from the computational power of the GPU simply by rewriting an existing algorithm in GPU specific API. The GPU computing model can be described as Single Instruction Multiple Threads (SIMT) meaning that threads, which are assigned to a specific task, will execute the same instruction, but on different sets of data. This makes the GPU perfect for problems which have high arithmetic intensity with low data dependencies.

[1]For example, high end processors such as AMD Ryzen™ Threadripper™ 3990X and AMD EPYC™ 7742 have 64 cores (allowing for 128 threads to be run), while high end graphic cards such as NVidia GeForce RTX™ 3090 have 10496 CUDA cores.

1.1 GPU implementation of the FEM

Considering the potential speed benefits of a GPU implementation of the FEM, a number of articles have been written on the subject. We can outline three possible directions. First the domain must be discretized. This is the part where the GPU could be least helpful due to the nature of the problem. The problem is partially discussed in [10]. Second, elementwise computations and assembling of global matrices are reviewed in [7]. Given that the elementwise computations are independent of each other we can see significant gain from a GPU implementation, however this is true only if the matrices are stored in dense format. Most of the sparse matrix formats create data dependencies, which make the assembly less scalable. Last, solving of the sparse linear system produced by the method should be discussed. Arguably this is always the bottleneck of the method. For large systems iterative methods are preferred, most often Conjugate Gradient, BiConjugate Stabilized or GMRES methods are used. Since this is the most computationally intensive task, most research is done for it [3], [16], [21].

In particular, we are interested in a possible GPU implementation of the FEM for the Navier-Stokes equations. The Navier-Stokes equations model the flow of viscous fluids. They are used in many areas e.g. weather forecasting, modelling ocean currents, engineering, medicine, computer graphics, etc. Unfortunately a general solution for them has not been found up to this day, moreover, proving that one exists was stated as a millenium problem by the Clay Mathematics Institute. Thus numerical methods for solving the Navier-Stokes equations are needed. The aim of this work is to implement an efficient and scalable GPU algorithm for solving the Navier-Stokes equations over unstructured 2D grids. Although the presented algorithms are general purpose and could be applied for various problems, our focus will be tilted more to applications in computer graphics and the VFX industry. Usually in computer graphics performance is valued more than accuracy, as long as the result is believable. In order to achieve a given artistic vision most of the softwares used in computer graphics even allow to generate results which are not physically correct at all (e.g. water flowing upwards, negative pressure, etc. are sometimes allowed). Another requirement is that the video sequences must be produced with a fixed frame rate; the most commonly used frame

rates are 24, 30, 60 FPS[2]. A classical book on fluid simulations for the purpose of computer graphics is [5]. According to it, the preferred methods are a combination of finite difference method and finite volume method (i.e. the marker and cell method with structured, staggered grid [12]) [3]. Thus we find it an interesting task to provide a FEM alternative for unstructured meshes.

Even though the literature concerning the FEM for Navier-Stokes equations is vast, there are relatively few papers about GPU implementation of it. Most of the research (e.g. [13], [24]) is based around the marker and cell method initially presented in [12] [4]. [1] focuses on GPU solver for the Reynolds-averaged Navier–Stokes equations. It presents both implicit and explicit time stepping schemes using Runge-Kutta method. It touches on the topic of matrix assembly. The method of choice for solving the presented linear system is the Generalized Minimal Residual Method (GMRES) with various preconditioning techniques. The type of element is not specified. [14] focuses on GPU solver for Streamline upwind pressure-stabilizing Petrov-Galerkin formulation of the incompressible Navier–Stokes equations. It uses the GPBi-CG method to directly solve the system which results from applying the FEM. The paper does not specify whether the time stepping scheme is explicit or implicit nor does it discuss the type of the chosen element.

In this work we will present three possible algorithms, based on the FEM on an unstructured grid, for solving the Navier-Stokes equations and compare them. Two of the methods use operator splitting and the other one does not. The method which seems most promising is implemented both on CPU (in C++) and GPU (in CUDA). The CPU implementation is multithreaded. The CG method for solving linear systems is presented and an additional preconditioned version is also presented with comparisons between both. Two approaches (single kernel and mega kernel) for implementing the

[2]The corresponding time steps for these frame rates are $\frac{1}{24}s$, $\frac{1}{30}s$, $\frac{1}{60}s$. Smaller time steps are allowed, but the intermediate results are usually thrown away.

[3]For computer graphics gases and liquids are usually simulated with different algorithms. Gases are simulated with the so-called grid solver, while liquids are simulated with a FLIP solver. The reason to use different algorithms is not because they produce vastly different results from a physical standpoint, it is because of the visual appearance. Rendering images and videos, however, is out of the scope of this work. The algorithm presented here could be categorized as a grid solver, but it is applicable for simulation of liquids, as well as, gases.

[4]The marker and cell method could be interpreted as a very specific variant of the FEM with structured grid of $Q_{-1}Q_0$ elements.

CG method on GPU are presented and compared. The semi-Lagrangian method for solving advection problems is presented alongside with a suitable data structure which improves its performance when unstructured meshes are used. The source code which is provided also contains a straightforward implementation of matrix assembly and computation of IC0 precontitioner.

1.2 Choice of elements for solving the Navier-Stokes equations

A vast study of the FEM applied to the Navier–Stokes equations is provided in [22]. Here we shall mention the advices given with respect to the choice of elements. We start with the notation. $P_m P_n$ means that the velocity (each component) is approximated by continuous piecewise complete polynomials of degree m and pressure by continuous piecewise complete polynomials of degree n. $P_m P_{-n}$ means that the velocity (each component) is approximated by continuous piecewise complete polynomials of degree m and the pressure is approximated via piecewise-discontinuous polynomials (C^{-1}) of degree n. For quadrilaterals/hexahedra, the designation $Q_m Q_n$ means that the velocity (each component) is approximated by a continuous piecewise polynomial of degree m in each direction on the quadrilateral and likewise for the pressure, except that the polynomial degree is n. Again for the same families, $Q_m P_{-n}$ indicates the same velocity approximation with a pressure approximation that is a discontinuous complete piecewise polynomial of degree n (not of degree n in each direction - it is as if the pressure was to be represented on a triangle within the quadrilateral, with "extrapolation" as necessary). The designation P^+ or Q^+ adds some sort of "bubble function" to the polynomial approximation for the velocity. These are sometimes called "enriched" elements. In 2D, in general, quadrilaterals have better accuracy than the triangular elements (considering polynomials of the same order). Triangluar elements are recommended in case of sophisticated geometry. $Q_2 P_{-1}$ isoparametric quadrilateral element is considered to be one of the most accurate elements in 2D. From the family of the triangular elements the recommended elements are $P_2^+ P_1$ and $P_2^+ P_{-1}$. The latter provides "elimination" of u and v at the center node and two pressures at element level, which makes it as cost effective as the $P_1 P_0$ element. The $P_1 P_0$ on the other hand is not recommended at all. The situation in 3D is more complicated. Should the problem

require unstructured meshes, tetrahedral should be considered first, due to the high computational cost of hexahedral elements. An easy and low cost option is the $P_1^+P_1$ MINI element. Better options, however, are the P_2P_1 and $P_2(P_1 + P_0)$ elements. The use of hexahedral elements is discouraged, but Q_1Q_0, the LBB stable Q_2P_{-1} and Q_2Q_{-1} are some viable options.

1.3 Time discretization techniques for the Navier-Stokes equations

There are various formulations of the Navier-Stokes equations, many of them are described in [22]. Unfortunately, there is no universal numerical time differentiation approach applicable to all of the formulations. For example, for the direct approach, which we shall present in Chapter 2, only implicit schemes make sense [22]. On the other hand [8] and [5] both propose operator splitting (also know as fractional time stepping) techniques which could be combined with explicit time differentiation schemes. The fractional time stepping techniques are also presented in Chapter 2. [8] proposes fractional time stepping which splits the differential operator in time into 2 steps: advection combined with diffusion and a Pressure Poisson equation. [5] proposes a fractional time stepping scheme, which splits the differential operator with respect to time into 3 steps: advection, Pressure Poisson equation and diffusion equation.

1.4 Goals and structure of the MSc Thesis

We formulate the following goals for the presented MSc Thesis:

- to construct a FEM-based algorithm for solving the incompressible Navier-Stokes equations using unstructured meshes, which is suitable for (though not limited to) the purposes of computer graphics; to this end our main requirements would be:
  - computational efficiency;
  - good scalability;
  - suitability for GPU implementation i.e. high artimethic intensity with low data dependencies;

  - stability;

- to conduct numerical experiments, based on a classical model problem (the DFG benchmark [25]), in order to study the scalability and effectiveness of the proposed method both on CPU and GPU;
- to outline some of the possible pros and cons of a GPU implementation;
- to outline directions for further development of the subject.

The MSc thesis is structured as follows. Chapter 2 presents the differential formulation of the problem and compares three different finite element formulations, which can be used to solve it. Two of them split the differential operator in time and the other one is a direct approach. The CSR sparse matrix format is presented, CG, PCG and semi-Lagrangian methods are outlined. Chapter 3 presents implementation details about the algorithms used to solve the finite element formulations presented in Chapter 2. It shows how the algorithms outlined in Chapter 2 are multithreaded on CPU and on GPU and comparison between the two is presented. Appendix A presents an extended version of the statistical data presented in Chapter 3. Appendix B provides an implementation of GPU synchronization primitive used by CG method. Older GPUs need it in order to implement the CG method efficiently.

1.5 Source Code

The source code for this work is free and can be found at GitHub.

- For the aim of this work a general purpose library for solving linear equations with sparse matrices was written. It can be found at `https://github.com/vasil-pashov/sparse_matrix_math`[5].
- The FEM implementation can be found at `https://github.com/vasil-pashov/Navier_Stokes_FEM`

[5] Check the develop branch of the repo for the latest changes.

Chapter 2

Finite element method for the Navier–Stokes equations

2.1 Governing equations

For reference, we will present here the Navier–Stokes equations, for incompressible Newtonian fluid [18]

$$\frac{\partial \mathbf{u}(\mathbf{x},t)}{\partial t} + (\mathbf{u}(\mathbf{x},t)\cdot\nabla)\,\mathbf{u}(\mathbf{x},t) + \nabla p(\mathbf{x},t) - \nu\Delta\mathbf{u}(\mathbf{x},t) = \mathbf{f}(\mathbf{x},t) \tag{2.1}$$

$$\nabla\cdot\mathbf{u}(\mathbf{x},t) = 0 \tag{2.2}$$

where:

- $\mathbf{u}\in\mathbb{R}^n$ is velocity
- $p\in\mathbb{R}$ is mechanical pressure
- ν is viscosity
- $\mathbf{f}$ represents the body forces acting on the fluid e.g. gravity

For the sake of brevity, explicit function parameters will be omitted and body forces will not be considered.

2.2 Model problem

We shall base our studies on the two dimensional DFG benchmark model problem initially presented in [25]. It considers flow in a channel with an obstacle. The channel is rectangular with length 2.2 and height 0.41. The obstacle is a circle with center at $(0.2, 0.2)$ and diameter 0.1. Fig. 2.1 shows the geometric setup for the benchmark. No-slip boundary condition is imposed on $\Gamma_1 \cup \Gamma_3 \cup \Gamma_5$. Do nothing condition is imposed on Γ_2. The problem is considered in the time interval $J = [0, T]$. The viscosity is $\nu = 0.001$. On Γ_4 we impose Dirichlet boundary condition which determines the type of the flow. Two types of flow will be studied, laminar and turbulent. For laminar flow $\mathbf{u} = \left(\frac{1.2y(0.41-y)}{0.41^2}, 0\right), \quad (\mathbf{x}, t) \in \Gamma_4 \times J$ is imposed. For turbulent flow $\mathbf{u} = \left(\frac{6y(0.41-y)}{0.41^2}, 0\right), \quad (\mathbf{x}, t) \in \Gamma_4 \times J$ is imposed (see Fig. 2.2).

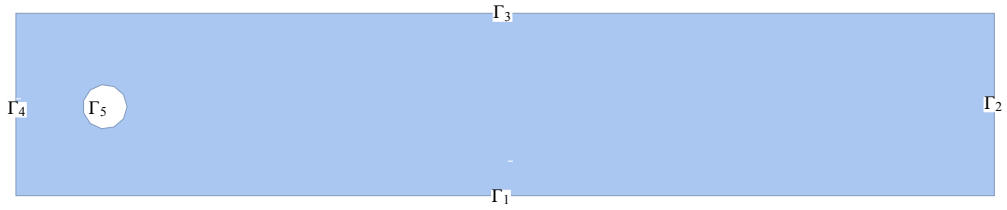


Figure 2.1: 2D DFG Benchmark setup

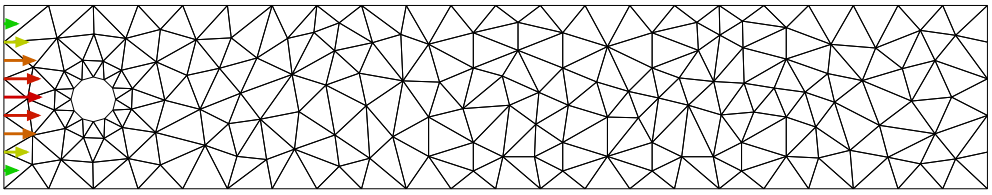


Figure 2.2: Dirichlet condition imposed on Γ_4. It has the same parabolic profile for laminar and turbulent flow. The only difference is in the length of the velocity vector.

The differential formulation is now presented. For laminar flow it is:

$$\begin{aligned}
&\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla p + \nu \Delta \mathbf{u}, && (\mathbf{x}, t) \in \Omega \times J \\
&\nabla \cdot \mathbf{u} = 0, && (\mathbf{x}, t) \in \Omega \times J \\
&\mathbf{u} = 0, && (\mathbf{x}, t) \in (\Gamma_1 \cup \Gamma_3 \cup \Gamma_5) \times J \\
&\mathbf{u} = \left(\frac{1.2y\,(0.41 - y)}{0.41^2}, 0 \right), && (\mathbf{x}, t) \in \Gamma_4 \times J \\
&\nu \frac{\partial \mathbf{u}}{\partial \mathbf{n}} - p\mathbf{n} = 0, && (\mathbf{x}, t) \in \Gamma_2 \times J
\end{aligned} \tag{2.3}$$

For turbulent flow it is:

$$\begin{aligned}
&\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla p + \nu \Delta \mathbf{u}, && (\mathbf{x}, t) \in \Omega \times J \\
&\nabla \cdot \mathbf{u} = 0, && (\mathbf{x}, t) \in \Omega \times J \\
&\mathbf{u} = 0, && (\mathbf{x}, t) \in (\Gamma_1 \cup \Gamma_3 \cup \Gamma_5) \times J \\
&\mathbf{u} = \left(\frac{6y(0.41 - y)}{0.41^2}, 0 \right), && (\mathbf{x}, t) \in \Gamma_4 \times J \\
&\nu \frac{\partial \mathbf{u}}{\partial \mathbf{n}} - p\mathbf{n} = 0, && (\mathbf{x}, t) \in \Gamma_2 \times J
\end{aligned} \tag{2.4}$$

2.3 Finite Element Method

Now, we shall present three FEM-based discretizations of the Navier-Stokes equations. We shall compare them in terms of our main goals – to find a method that is efficient and easy to parallelize.

2.3.1 Direct approach

One possible way to attack the problem is to consider Eq. (2.3) or Eq. (2.4), derive a weak formulation and then apply the FEM directly to this weak formulation. Based on [22] implicit time integration scheme must be used. A big disadvantage is that this approach requires solving a nonlinear system of equations. The method also requires assembling the convection matrix at each iteration of the nonlinear solver, however, this is not a trivial task to parallelize. The derivation of the FEM will be presented, as well as, the

result of a numerical experiment, mainly to outline what the limitations of this approach are for our purposes.

2.3.1.1 Weak formulation

We start by taking the dot products of both sides of the first equation from Eq. (2.3) with a test function $\mathbf{v} \in V : \{\mathbf{v} \in H^1 : \mathbf{v}(\mathbf{x}, t)|_{\Gamma_D} = 0\}$ and both sides of the second equation from Eq. (2.3) with a test function $p \in L^2(\Omega)$, where $\Gamma_D = \Gamma_1 \cup \Gamma_3 \cup \Gamma_4 \cup \Gamma_5$

$$\left(\frac{\partial \mathbf{u}}{\partial t}, \mathbf{v}\right) + (\mathbf{u} \cdot \nabla \mathbf{u}, \mathbf{v}) = -(\nabla p, \mathbf{v}) + \nu(\Delta \mathbf{u}, \mathbf{v}), \quad \mathbf{v} \in V \tag{2.5}$$

$$(\nabla \cdot \mathbf{u}, q) = 0, \qquad q \in L^2 \tag{2.6}$$

$$\mathbf{u} = \mathbf{u}_{\boldsymbol{\Gamma_4}}, \qquad (\mathbf{x}, t) \in \Gamma_4 \times J \tag{2.7}$$

Now for Eq. (2.5) we perform integration by parts and obtain:

$$\left(\frac{\partial \mathbf{u}}{\partial t}, \mathbf{v}\right) + (\mathbf{u} \cdot \nabla \mathbf{u}, \mathbf{v}) =$$

$$(p, \nabla \cdot \mathbf{v}) - \cancelto{\mathbf{v}|_{\Gamma_D} = 0}{(p\mathbf{n}, \mathbf{v})_{\Gamma_D}} - \cancelto{\left(\nu \frac{\partial \mathbf{u}}{\partial \mathbf{n}} - p\mathbf{n}\right)|_{\Gamma_2} = 0}{(p\mathbf{n}, \mathbf{v})_{\Gamma_2}}$$

$$- \nu(\nabla \mathbf{u} : \nabla \mathbf{v}) + \cancelto{\mathbf{v}|_{\Gamma_D} = 0}{\nu(\mathbf{n} \cdot \nabla \mathbf{u}, \mathbf{v})_{\Gamma_D}} + \cancelto{\left(\nu \frac{\partial \mathbf{u}}{\partial \mathbf{n}} - p\mathbf{n}\right)|_{\Gamma_2} = 0}{\nu(\mathbf{n} \cdot \nabla \mathbf{u}, \mathbf{v})_{\Gamma_2}}, \quad \forall \mathbf{v} \in V$$

This leads us to the final form of the weak formulation. Find $\mathbf{u}$ and p such that:

$$\mathbf{u} \in U : \left\{\mathbf{v} \in H^1 : \mathbf{v}(\mathbf{x}, t)|_{\Gamma_1 \cup \Gamma_3 \cup \Gamma_5} = 0\right\}$$
$$p \in L^2(\Omega)$$

$$\left(\frac{\partial \mathbf{u}}{\partial t}, \mathbf{v}\right) + (\mathbf{u} \cdot \nabla \mathbf{u}, \mathbf{v}) + \nu(\nabla \mathbf{u} : \nabla \mathbf{v}) = (p, \nabla \cdot \mathbf{v}), \quad \mathbf{v} \in V \tag{2.8}$$

$$(\nabla \cdot \mathbf{u}, q) = 0, \qquad q \in L^2 \tag{2.9}$$

$$\mathbf{u} = \mathbf{u}_{\boldsymbol{\Gamma_4}}, \qquad (\mathbf{x}, t) \in \Gamma_4 \times J \tag{2.10}$$

where the Frobenius inner product of two second rank tensors is a real number, defined as:

$$\mathbf{A} : \mathbf{B} = \sum_{i=1}^{n} \sum_{j=1}^{m} A_{i,j} B_{i,j} = A_{i,j} B_{i,j}$$

2.3.1.2 Finite Element Method

Let:

$$\{\mathbf{\Phi}_i\}_{i=1}^{2N_v} = \left\{ \begin{bmatrix} \phi_1 \\ 0 \end{bmatrix}, \ldots, \begin{bmatrix} \phi_{N_v} \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ \phi_1 \end{bmatrix}, \ldots, \begin{bmatrix} 0 \\ \phi_{N_v} \end{bmatrix} \right\}$$

be the set of basis functions for the finite dimensional space U_h where we will seek the approximate solution $\mathbf{u}_h$ for the velocity function $\mathbf{u}$ where N_v is the number of velocity nodes in the mesh, which do not lie on $\Gamma_1 \cup \Gamma_3 \cup \Gamma_5$ and let:

$$\{\chi_i\}_{i=1}^{N_p}$$

be the set of basis functions for the finite dimensional space Q_h where we will seek the approximate solution p_h for the pressure function p where N_p is the number of pressure nodes in the mesh. Then we can rewrite Eq. (2.8) in terms of the approximate solution. We look for $\mathbf{u}_h$ in the form $\mathbf{u}_h = \sum_{i=1}^{2N_v} \mathbf{\Phi}_i q_i$ and p_h in the form $p_h = \sum_{i=1}^{N_p} \chi_i p_i$ and, thus, obtain:

$$\sum_{j=1}^{2N_v} \frac{dq_j}{dt} \left(\mathbf{\Phi}_j, \mathbf{\Phi}_i\right) + \sum_{j=1}^{2N_v} q_j \left(\mathbf{u}_h \cdot \nabla \mathbf{\Phi}_j, \mathbf{\Phi}_i\right) =$$

$$\sum_{j=1}^{N_p} p_j \left(\chi_j, \nabla \cdot \mathbf{\Phi}_i\right) - \nu \sum_{j=1}^{2N_v} q_j \left(\nabla \mathbf{\Phi}_j : \nabla \mathbf{\Phi}_i\right), \quad \forall i \in [1, 2N_v]$$

More precisely, the latter equations should hold for all i, corresponding to nodes, which don't lie on Γ_D. The system is closed by imposing the Dirichlet boundary conditions on Γ_4. However, we shall neglect this for the time being, and explain later how we impose the Dirichlet boundary conditions in practice. From Eq. (2.9) we have:

$$\sum_{j=1}^{2N_v} q_j \left(\nabla \cdot \mathbf{\Phi}_j, \chi_i\right) = 0, \quad \forall i \in [1, N_p]$$

Now putting these in matrix form, we finally obtain:

$$\begin{bmatrix} \mathbf{M} & 0 & 0 \\ 0 & \mathbf{M} & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \dot{\mathbf{q}}_1 \\ \dot{\mathbf{q}}_2 \\ \mathbf{p} \end{bmatrix} = \begin{bmatrix} -\nu\mathbf{K} - \mathbf{C}(\mathbf{u_h}) & 0 & \mathbf{B}_1^T \\ 0 & -\nu\mathbf{K} - \mathbf{C}(\mathbf{u_h}) & \mathbf{B}_2^T \\ \mathbf{B}_1 & \mathbf{B}_2 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{q}_1 \\ \mathbf{q}_2 \\ \mathbf{p} \end{bmatrix} \quad (2.11)$$

where $\mathbf{M}$ denotes the mass matrix, which can be lumped.

$$\mathbf{M} = \begin{bmatrix} (\phi_1, \phi_1) & (\phi_2, \phi_1) & \cdots & (\phi_{N_v}, \phi_1) \\ (\phi_1, \phi_2) & (\phi_2, \phi_2) & \cdots & (\phi_{N_v}, \phi_2) \\ \vdots & \vdots & \cdots & \vdots \\ (\phi_1, \phi_{N_v}) & (\phi_2, \phi_{N_v}) & \cdots & (\phi_{N_v}, \phi_{N_v}) \end{bmatrix}$$

$\mathbf{C}$ denotes the convection matrix, which depends on the fluid velocities:

$$\mathbf{C}(\mathbf{u}_h) = \begin{bmatrix} (\mathbf{u}_h \cdot \nabla\phi_1, \phi_1) & (\mathbf{u}_h \cdot \nabla\phi_2, \phi_1) & \cdots & (\mathbf{u}_h \cdot \nabla\phi_{N_v}, \phi_1) \\ (\mathbf{u}_h \cdot \nabla\phi_1, \phi_2) & (\mathbf{u}_h \cdot \nabla\phi_2, \phi_2) & \cdots & (\mathbf{u}_h \cdot \nabla\phi_{N_v}, \phi_2) \\ \vdots & \vdots & \cdots & \vdots \\ (\mathbf{u}_h \cdot \nabla\phi_1, \phi_{N_v}) & (\mathbf{u}_h \cdot \nabla\phi_2, \phi_{N_v}) & \cdots & (\mathbf{u}_h \cdot \nabla\phi_{N_v}, \phi_{N_v}) \end{bmatrix}$$

$\mathbf{K}$ denotes the stiffness matrix:

$$\mathbf{K} = \begin{bmatrix} (\nabla\phi_1 \cdot \nabla\phi_1) & (\nabla\phi_2 \cdot \nabla\phi_1) & \cdots & (\nabla\phi_{N_v} \cdot \nabla\phi_1) \\ (\nabla\phi_1 \cdot \nabla\phi_2) & (\nabla\phi_2 \cdot \nabla\phi_2) & \cdots & (\nabla\phi_{N_v} \cdot \nabla\phi_2) \\ \vdots & \vdots & \cdots & \vdots \\ (\nabla\phi_1 \cdot \nabla\phi_{N_v}) & (\nabla\phi_2 \cdot \nabla\phi_{N_v}) & \cdots & (\nabla\phi_{N_v} \cdot \nabla\phi_{N_v}) \end{bmatrix}$$

$$\mathbf{B}_1 = \begin{bmatrix} \left(\frac{\partial\phi_1}{\partial x_1}, \chi_1\right) & \left(\frac{\partial\phi_2}{\partial x_1}, \chi_1\right) & \cdots & \left(\frac{\partial\phi_{N_v}}{\partial x_1}, \chi_1\right) \\ \left(\frac{\partial\phi_1}{\partial x_1}, \chi_2\right) & \left(\frac{\partial\phi_2}{\partial x_1}, \chi_2\right) & \cdots & \left(\frac{\partial\phi_{N_v}}{\partial x_1}, \chi_2\right) \\ \vdots & \vdots & \cdots & \vdots \\ \left(\frac{\partial\phi_1}{\partial x_1}, \chi_{N_p}\right) & \left(\frac{\partial\phi_2}{\partial x_1}, \chi_{N_p}\right) & \cdots & \left(\frac{\partial\phi_{N_v}}{\partial x_1}, \chi_{N_p}\right) \end{bmatrix}$$

$$\mathbf{B}_2 = \begin{bmatrix} \left(\frac{\partial\phi_1}{\partial x_2}, \chi_1\right) & \left(\frac{\partial\phi_2}{\partial x_2}, \chi_1\right) & \cdots & \left(\frac{\partial\phi_{N_v}}{\partial x_2}, \chi_1\right) \\ \left(\frac{\partial\phi_1}{\partial x_2}, \chi_2\right) & \left(\frac{\partial\phi_2}{\partial x_2}, \chi_2\right) & \cdots & \left(\frac{\partial\phi_{N_v}}{\partial x_2}, \chi_2\right) \\ \vdots & \vdots & \cdots & \vdots \\ \left(\frac{\partial\phi_1}{\partial x_2}, \chi_{N_p}\right) & \left(\frac{\partial\phi_2}{\partial x_2}, \chi_{N_p}\right) & \cdots & \left(\frac{\partial\phi_{N_v}}{\partial x_2}, \chi_{N_p}\right) \end{bmatrix}$$

2.3.1.3 Time discretization

For the time discretization Crank-Nicolson method is used:

$$\begin{bmatrix} \mathbf{M} & 0 & 0 \\ 0 & \mathbf{M} & 0 \\ 0 & 0 & 0 \end{bmatrix} \left(\begin{bmatrix} \mathbf{Q}_1^{i+1} \\ \mathbf{Q}_2^{i+1} \\ \mathbf{P}^{i+1} \end{bmatrix} - \begin{bmatrix} \mathbf{Q}_1^{i} \\ \mathbf{Q}_2^{i} \\ \mathbf{P}^{i} \end{bmatrix} \right) \frac{1}{\Delta t} =$$
$$\frac{1}{2} \begin{bmatrix} -\nu\mathbf{K} - \mathbf{C}(\mathbf{u}_h^{i+1}) & 0 & \mathbf{B}_1^T \\ 0 & -\nu\mathbf{K} - \mathbf{C}(\mathbf{u}_h^{i+1}) & \mathbf{B}_2^T \\ \mathbf{B}_1 & \mathbf{B}_2 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{Q}_1^{i+1} \\ \mathbf{Q}_2^{i+1} \\ \mathbf{P}^{i+1} \end{bmatrix}$$
$$+\frac{1}{2} \begin{bmatrix} -\nu\mathbf{K} - \mathbf{C}(\mathbf{u}_h^{i}) & 0 & \mathbf{B}_1^T \\ 0 & -\nu\mathbf{K} - \mathbf{C}(\mathbf{u}_h^{i}) & \mathbf{B}_2^T \\ \mathbf{B}_1 & \mathbf{B}_2 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{Q}_1^{i} \\ \mathbf{Q}_2^{i} \\ \mathbf{P}^{i} \end{bmatrix}$$

which leads to the following nonlinear system of equations:

$$\begin{bmatrix} \mathbf{M} + \frac{\Delta t}{2}\left(\nu\mathbf{K} + \mathbf{C}(\mathbf{u}_h^{i+1})\right) & 0 & -\frac{\Delta t}{2}\mathbf{B}_1^T \\ 0 & \mathbf{M} + \frac{\Delta t}{2}\left(\nu\mathbf{K} + \mathbf{C}(\mathbf{u}_h^{i+1})\right) & -\frac{\Delta t}{2}\mathbf{B}_2^T \\ -\frac{\Delta t}{2}\mathbf{B}_1 & -\frac{\Delta t}{2}\mathbf{B}_2 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{Q}_1^{i+1} \\ \mathbf{Q}_2^{i+1} \\ \mathbf{P}^{i+1} \end{bmatrix} =$$
$$\begin{bmatrix} \mathbf{M} - \frac{\Delta t}{2}\left(\nu\mathbf{K} + \mathbf{C}(\mathbf{u}_h^{i})\right) & 0 & \frac{\Delta t}{2}\mathbf{B}_1^T \\ 0 & \mathbf{M} - \frac{\Delta t}{2}\left(\nu\mathbf{K} + \mathbf{C}(\mathbf{u}_h^{i})\right) & \frac{\Delta t}{2}\mathbf{B}_2^T \\ \frac{\Delta t}{2}\mathbf{B}_1 & \frac{\Delta t}{2}\mathbf{B}_2 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{Q}_1^{i} \\ \mathbf{Q}_2^{i} \\ \mathbf{P}^{i} \end{bmatrix} \quad (2.12)$$

This system is solved via Picard iteration.

2.3.1.4 Dirichlet Boundary Conditions

In order to impose the boundary conditions on Γ_D we shall assemble the matrices as if the nodes on Γ_D were "regular" nodes, then the matrix and the right hand side of Eq. (2.12) will be tweaked. Let us denote the matrix on the left-hand-side of Eq. (2.12) with A and the vector on the right-hand-side of Eq. (2.12) with $\mathbf{b}$. Then Algorithm 1 shows how to tweak Eq. (2.12) in order to apply the Dirichlet boundary conditions and preserve the symmetry of the matrix.

Algorithm 1 Impose Dirichlet Boundary Conditions

```
1: procedure ImposeDirichlet(dirichletIndices, values, A, b)
2:     for all j ∈ dirichletIndices do
3:         Set the j − th row of A to 0
4:         A[j][j] ← 1
5:         b[j] ← values[j]
6:         for all i ≠ j row in A do
7:             b[i] ← b[i] − values[j] ∗ A[i][j]
8:             A[i][j] ← 0
```

2.3.2 Operator Splitting

As it was mentioned in the introduction according to [22] only implicit time discretization methods for Eq. (2.11) make sense. On the other hand, using implicit time discretization methods for Eq. (2.11) would require solving nonlinear systems at each time step which is an extremely computationally expensive process. Furthermore, it would not be trivial to parallelize the method, presented in the previous chapter, for our purposes. Thus another technique, which is commonly applied and allows for the use of explicit time approximation will be presented. The trick is to split the differential operator (with respect to time). The method proposed in [8] splits the operator in two, while the one proposed in [5] splits it in three. Both splitting techniques will require solving an equation in which the second derivative of the pressure will be present. This means that the element's polynomial space for the pressure must be at least of first order.

2.3.2.1 Chorin Split

The first approach which will be presented will split the differential operator in time into two parts. One part which governs the velocity and another which governs the pressure. We start by approximating Eq. (2.1) with forward difference and then adding and subtracting a dummy term in the forward difference.

$$\frac{\mathbf{u}^{i+1} - \mathbf{u}^i}{\Delta t} + \mathbf{u}^i \cdot \nabla \mathbf{u}^i + \nabla p^i - \nu \Delta \mathbf{u}^i = 0$$
$$\frac{\mathbf{u}^{i+1} + \mathbf{u}^{i+\frac{1}{2}} - \mathbf{u}^{i+\frac{1}{2}} - \mathbf{u}^i}{\Delta t} + \mathbf{u}^i \cdot \nabla \mathbf{u}^i + \nabla p^i - \nu \Delta \mathbf{u}^i = 0$$

Now we can split the equations. At one fraction of time the pressure will be computed and at the other everything else.

$$\frac{\mathbf{u}^{i+\frac{1}{2}} - \mathbf{u}^i}{\Delta t} = \nu \Delta \mathbf{u}^i - \mathbf{u}^i \cdot \nabla \mathbf{u}^i \tag{2.13}$$
$$\frac{\mathbf{u}^{i+1} - \mathbf{u}^{i+\frac{1}{2}}}{\Delta t} = -\nabla p^i \tag{2.14}$$

Now we can further transform Eq. (2.14), by taking the divergence of both sides. From Eq. (2.2) we know that $\nabla \cdot \mathbf{u}^{i+1} = 0$. And we get:

$$\nabla \cdot \mathbf{u}^{i+\frac{1}{2}} = \Delta p^i \Delta t$$

known as the Pressure Poisson Equation.

The most controversial part of this splitting are the boundary conditions. For the velocity we shall impose the corresponding no-slip boundary conditions on $\Gamma_1 \cup \Gamma_3 \cup \Gamma_5$, on the outflow boundary Γ_2 we shall impose $\mathbf{n} \cdot \nabla \mathbf{u}^i = 0$ and on the inflow boundary Γ_4 a constant velocity $\mathbf{u}_{\Gamma_4}$ is imposed. For the pressure we shall impose $\mathbf{n} \cdot \nabla p^i = 0$ on $\Gamma_1 \cup \Gamma_3 \cup \Gamma_4 \cup \Gamma_5$. Imposing $\mathbf{n} \cdot \nabla p^i = 0$ has no physical meaning and is believed to lead to bad approximation of the pressure near the boundary [18]. For the pressure on the outflow boundary we shall impose $p^i = 0$. This is done in order to mimic the do-nothing boundary condition.

Finally, the equations to which we shall apply the FEM are:

$$\frac{\mathbf{u}^{i+\frac{1}{2}} - \mathbf{u}^i}{\Delta t} = \nu \Delta \mathbf{u}^i - \mathbf{u}^i \cdot \nabla \mathbf{u}^i, \quad (\mathbf{x}, t) \in \Omega \times J \tag{2.15}$$

$$\nabla \cdot \mathbf{u}^{i+\frac{1}{2}} = \Delta p^i \Delta t, \quad (\mathbf{x}, t) \in \Omega \times J \tag{2.16}$$

$$\mathbf{u}^{i+1} = \mathbf{u}^{i+\frac{1}{2}} - \Delta t \nabla p^i, \quad (\mathbf{x}, t) \in \Omega \times J \tag{2.17}$$

$$\mathbf{n} \cdot \nabla \mathbf{u}^i = 0, \quad (\mathbf{x}, t) \in \Gamma_2 \times J \tag{2.18}$$

$$\mathbf{u}^i = 0, \quad (\mathbf{x}, t) \in (\Gamma_1 \cup \Gamma_3 \cup \Gamma_5) \times J \tag{2.19}$$

$$\mathbf{u}^i = \mathbf{u}_{\Gamma_4}, \quad (\mathbf{x}, t) \in \Gamma_4 \times J \tag{2.20}$$

$$\mathbf{n} \cdot \nabla p^i = 0, \quad (\mathbf{x}, t) \in (\Gamma_1 \cup \Gamma_3 \cup \Gamma_4 \cup \Gamma_5) \times J \tag{2.21}$$

$$p^i = 0, \quad (\mathbf{x}, t) \in \Gamma_2 \times J \tag{2.22}$$

Weak formulation We begin with Eq. (2.15). As usual both sides of the equation are multiplied by a test function $\mathbf{v} \in V = \{\mathbf{v} | \mathbf{v} \in H^1 \wedge \mathbf{v}|_{\Gamma_D} = 0\}$

$$\left(\frac{\mathbf{u}^{i+\frac{1}{2}} - \mathbf{u}^i}{\Delta t}, \mathbf{v}\right) = \left(\nu \Delta \mathbf{u}^i, \mathbf{v}\right) - \left(\mathbf{u}^i \cdot \nabla \mathbf{u}^i, \mathbf{v}\right), \forall \mathbf{v} \in V$$

$$\left(\frac{\mathbf{u}^{i+\frac{1}{2}} - \mathbf{u}^i}{\Delta t}, \mathbf{v}\right) = \cancelto{\mathbf{v}|_{\Gamma_D} = \mathbf{0}}{\nu \left(\mathbf{n} \cdot \nabla \mathbf{u}, \mathbf{v}\right)_{\Gamma_D}} + \cancelto{(\mathbf{n} \cdot \nabla \mathbf{u}^i)\,|_{\Gamma_2} = 0}{\nu \left(\mathbf{n} \cdot \nabla \mathbf{u}, \mathbf{v}\right)_{\Gamma_2}}$$

$$- \nu \left(\nabla \mathbf{u}^i : \nabla \mathbf{v}\right) - \left(\mathbf{u}^i \cdot \nabla \mathbf{u}^i, \mathbf{v}\right), \forall \mathbf{v} \in V$$

$$\left(\mathbf{u}^{i+\frac{1}{2}}, \mathbf{v}\right) = \left(\mathbf{u}^i, \mathbf{v}\right) - \Delta t \left[\nu \left(\nabla \mathbf{u}^i : \nabla \mathbf{v}\right) + \left(\mathbf{u}^i \cdot \nabla \mathbf{u}^i, \mathbf{v}\right)\right], \forall \mathbf{v} \in V$$

Next we shall multiply both sides of Eq. (2.16) with a test function $q \in Q = \{q | q \in H^1 \wedge q|_{\Gamma_2} = 0\}$.

$$\left(\nabla \cdot \mathbf{u}^{i+\frac{1}{2}}, q\right) = \Delta t \left(\Delta p^i, q\right), \forall q \in Q$$

$$\left(\nabla \cdot \mathbf{u}^{i+\frac{1}{2}}, q\right) = \cancelto{(\mathbf{n} \cdot \nabla p^i)\,|_{\Gamma_D} = 0}{\Delta t \left(\mathbf{n} \cdot \nabla p^i, q\right)|_{\Gamma_D}} + \cancelto{q|_{\Gamma_D} = 0}{\Delta t \left(\mathbf{n} \cdot \nabla p^i, q\right)|_{\Gamma_D}} - \Delta t \left(\nabla p^i, \nabla q\right), \forall q \in Q$$

$$\left(\nabla \cdot \mathbf{u}^{i+\frac{1}{2}}, q\right) = -\Delta t \left(\nabla p^i, \nabla q\right), \forall q \in Q$$

Finally we take the dot product of both sides of Eq. (2.17) with a test function $\mathbf{v} \in V$.

$$\left(\mathbf{u}^{i+1}, \mathbf{v}\right) = \left(\mathbf{u}^{i+\frac{1}{2}}, \mathbf{v}\right) - \Delta t \left(\mathbf{v}, \nabla p^{i}\right)$$

Finite Element Method We shall seek the approximate solution $\mathbf{u}_h$ and p_h in a finite dimensional subspaces of V and Q. Let us use the same notation for the basis functions of those spaces as in Section 2.3.1.2.

$$\begin{aligned}
\begin{bmatrix} \mathbf{M} & 0 \\ 0 & \mathbf{M} \end{bmatrix} \begin{bmatrix} \mathbf{Q_1^{i+\frac{1}{2}}} \\ \mathbf{Q_2^{i+\frac{1}{2}}} \end{bmatrix} &= \begin{bmatrix} \mathbf{M} & 0 \\ 0 & \mathbf{M} \end{bmatrix} \begin{bmatrix} \mathbf{Q_1^{i}} \\ \mathbf{Q_2^{i}} \end{bmatrix} - \Delta t \begin{bmatrix} \nu\mathbf{K} + \mathbf{C}(\mathbf{u_h}) & 0 \\ 0 & \nu\mathbf{K} + \mathbf{C}(\mathbf{u_h}) \end{bmatrix} \begin{bmatrix} \mathbf{Q_1^{i}} \\ \mathbf{Q_2^{i}} \end{bmatrix} \\
\mathbf{K_p P}^{i} &= -\frac{1}{\Delta t} \begin{bmatrix} \mathbf{B_1} & \mathbf{B_2} \end{bmatrix} \begin{bmatrix} \mathbf{Q_1^{i+\frac{1}{2}}} \\ \mathbf{Q_2^{i+\frac{1}{2}}} \end{bmatrix} \\
\begin{bmatrix} \mathbf{M} & 0 \\ 0 & \mathbf{M} \end{bmatrix} \begin{bmatrix} \mathbf{Q_1^{i+1}} \\ \mathbf{Q_2^{i+1}} \end{bmatrix} &= \begin{bmatrix} \mathbf{M} & 0 \\ 0 & \mathbf{M} \end{bmatrix} \begin{bmatrix} \mathbf{Q_1^{i+\frac{1}{2}}} \\ \mathbf{Q_2^{i+\frac{1}{2}}} \end{bmatrix} - \Delta t \begin{bmatrix} \mathbf{B_{p,1}} \\ \mathbf{B_{p,2}} \end{bmatrix} \mathbf{P}^{i}
\end{aligned} \tag{2.23}$$

where $\mathbf{K_p}$ is given by:

$$\mathbf{K_p} = \begin{bmatrix} (\nabla\chi_1, \nabla\chi_1) & (\nabla\chi_2, \nabla\chi_1) & \dots & (\nabla\chi_{Np}, \nabla\chi_1) \\ (\nabla\chi_1, \nabla\chi_2) & (\nabla\chi_2, \nabla\chi_2) & \dots & (\nabla\chi_{Np}, \nabla\chi_2) \\ \vdots & \vdots & \vdots & \vdots \\ (\nabla\chi_1, \nabla\chi_{Np}) & (\nabla\chi_2, \nabla\chi_{Np}) & \dots & (\nabla\chi_{Np}, \nabla\chi_{Np}) \end{bmatrix}$$

$\mathbf{B_{p,1}}$ is given by:

$$\mathbf{B_{p,1}} = \begin{bmatrix} \left(\frac{\partial\chi_1}{\partial x_1}, \phi_1\right) & \left(\frac{\partial\chi_2}{\partial x_1}, \phi_1\right) & \dots & \left(\frac{\partial\chi_{N_p}}{\partial x_1}, \phi_1\right) \\ \left(\frac{\partial\chi_1}{\partial x_1}, \phi_2\right) & \left(\frac{\partial\chi_2}{\partial x_1}, \phi_2\right) & \dots & \left(\frac{\partial\chi_{N_p}}{\partial x_1}, \phi_2\right) \\ \vdots & \vdots & \dots & \vdots \\ \left(\frac{\partial\chi_1}{\partial x_1}, \phi_{N_v}\right) & \left(\frac{\partial\chi_2}{\partial x_1}, \phi_{N_v}\right) & \dots & \left(\frac{\partial\chi_{N_v}}{\partial x_1}, \phi_{N_v}\right) \end{bmatrix}$$

$\mathbf{B_{p,2}}$ is given by:

$$\mathbf{B_{p,2}} = \begin{bmatrix} \left(\frac{\partial \chi_1}{\partial x_2}, \phi_1\right) & \left(\frac{\partial \chi_2}{\partial x_2}, \phi_1\right) & \cdots & \left(\frac{\partial \chi_{N_p}}{\partial x_2}, \phi_1\right) \\ \left(\frac{\partial \chi_1}{\partial x_2}, \phi_2\right) & \left(\frac{\partial \chi_2}{\partial x_2}, \phi_2\right) & \cdots & \left(\frac{\partial \chi_{N_p}}{\partial x_2}, \phi_2\right) \\ \vdots & \vdots & \cdots & \vdots \\ \left(\frac{\partial \chi_1}{\partial x_2}, \phi_{N_v}\right) & \left(\frac{\partial \chi_2}{\partial x_2}, \phi_{N_v}\right) & \cdots & \left(\frac{\partial \chi_{N_v}}{\partial x_2}, \phi_{N_v}\right) \end{bmatrix}$$

The Dirichlet boundary conditions are imposed in practice in the usual way.

2.3.2.2 Advection–Diffusion split

The Chorin split is easy to implement but it has some flaws. One of them is the fact that the diffusion is approximated with an explicit scheme. This ties the time step to the space step (element size). In case more spatial detail is needed and the spatial mesh is refined we must lower the time step in order to have a stable method. This property is not desired, at least in computer graphics, because videos are produced with a specific frame per second ratio, so the frames produced when the time step is lowered will not be used, this is a waste of time and computational resources. One thing we could do is to approximate the diffusion–advection part of the equation with an implicit scheme. This would require solving a nonlinear system on each time step, which is also not desired. One can try to work around this problem by trying a semi-implicit scheme. Instead of doing that we shall do another split of the differential operator with respect to time, and split the diffusion and advection into separate equations. This split has two advantages – first, as we show later, there exists an efficient, embarrassingly parallel algorithm which can be used for the advection equation. Second, the diffusion can be approximated with an implicit scheme, without the need to solve nonlinear system at each time step. We shall derive the equation the same way that the Chorin split was derived. This time however we shall add and subtract two dummy variables $\mathbf{u^A}$ and $\mathbf{u^B}$.

$$\frac{\mathbf{u}^{i+1} - \mathbf{u}^i}{\Delta t} + \mathbf{u}^i \cdot \nabla \mathbf{u}^i + \nabla p^i - \nu \Delta \mathbf{u}^i = 0 \quad (2.24)$$

$$\frac{\mathbf{u}^{i+1} + \mathbf{u^A} - \mathbf{u^A} + \mathbf{u^B} - \mathbf{u^B} - \mathbf{u}^i}{\Delta t} + \mathbf{u}^i \cdot \nabla \mathbf{u}^i + \nabla p^i - \nu \Delta \mathbf{u}^i = 0 \quad (2.25)$$

Now we shall split Eq. (2.25) into three separate parts, keeping in mind that we want to use the implicit Euler method for the diffusion equation.

$$\frac{\mathbf{u^A} - \mathbf{u}^i}{\Delta t} + \mathbf{u}^i \cdot \nabla \mathbf{u}^i = 0$$
$$\frac{\mathbf{u^B} - \mathbf{u^A}}{\Delta t} = \nu \Delta \mathbf{u^B}$$
$$\frac{\mathbf{u}^{i+1} - \mathbf{u^B}}{\Delta t} = -\nabla p^i$$

The pressure term is treated the same way, by taking the divergence of both sides, as in the Chorin split. The system of equations which should be solved is:

$$\frac{\mathbf{u^A} - \mathbf{u}^i}{\Delta t} + \mathbf{u}^i \cdot \nabla \mathbf{u}^i = 0, \qquad (\mathbf{x}, t) \in \Omega \times J \tag{2.26}$$
$$\frac{\mathbf{u^B} - \mathbf{u^A}}{\Delta t} = \nu \Delta \mathbf{u^B}, \qquad (\mathbf{x}, t) \in \Omega \times J \tag{2.27}$$
$$\nabla \cdot \mathbf{u^B} = \Delta p^i \Delta t, \qquad (\mathbf{x}, t) \in \Omega \times J \tag{2.28}$$
$$\mathbf{u}^{i+1} = \mathbf{u^B} - \Delta t \nabla p^i, \qquad (\mathbf{x}, t) \in \Omega \times J \tag{2.29}$$
$$\mathbf{n} \cdot \nabla \mathbf{u}^i = 0, \qquad (\mathbf{x}, t) \in \Gamma_2 \times J \tag{2.30}$$
$$\mathbf{u}^i = 0, \qquad (\mathbf{x}, t) \in (\Gamma_1 \cup \Gamma_3 \cup \Gamma_5) \times J \tag{2.31}$$
$$\mathbf{u}^i = \mathbf{u}_{\Gamma_4}, \qquad (\mathbf{x}, t) \in \Gamma_4 \times J \tag{2.32}$$
$$\mathbf{n} \cdot \nabla p^i = 0, \qquad (\mathbf{x}, t) \in (\Gamma_1 \cup \Gamma_3 \cup \Gamma_4 \cup \Gamma_5) \times J \tag{2.33}$$
$$p^i = 0, \qquad (\mathbf{x}, t) \in \Gamma_2 \times J \tag{2.34}$$

Note that this is slightly different than the split applied in [5]. In their approach the advection is first, then the pressure projection is applied, then the diffusion is applied. This requires another pressure projection step after the diffusion in order to impose incompresibillity. This way both the advection and the diffusion are performed in a divergence free fluid. On the other hand, formally, only $\mathbf{u}^{i+1}$ is required to be noncompressible and that is why we do not apply the pressure projection step twice. Our numerical experiment does not show problems with the accuracy or the stability of our approach. More experiments are needed, however.

Semi-Lagrangian Advection One problem Eq. (2.26) has is the stability of the advection phase. The FEM discretization would not be stable in C

norm. There are various ways to improve the stability of the FEM, some of them are presented in [22]. However, there is another downside of using the finite element method for solving Eq. (2.26) – at each time step we must assemble the convection matrix. While the time needed for the assembly is not much compared to the time spent in the Conjugate Gradient method, it is a fact that matrix assembly procedures do not scale well when processor count is increased. Most of the algorithms used for multithreading assembly procedures are based on graph coloring, which is a NP-Complete problem, moreover the coloring would depend on the numbering of elements and nodes.

For solving the advection equation, we shall use the semi-Lagrangian method, a stable method [11] which falls under the category of embarrassingly parallel of algorithms. Which means that, at least in theory, the algorithm should scale linearly when processor count is increased. It does not require using any locking mechanisms and is suitable to be implemented on a GPU.

Solving Eq. (2.26) in a Lagrangian framework is trivial. Assume we were simulating each particle individually, then Eq. (2.26) would be trivially satisfied when we move all particles, which are being simulated. Finding the velocity for a particular particle is also trivial, because we would have all particles with their velocities explicitly stored in some structure. Working directly with particles has its downsides, however. We must either change our mesh at each point in time or we have to use some tree-like structure to handle the movement and creation of particles. To cope with this problem we shall use a mixture of Eulerian and Lagrangian frameworks (thus the name semi-Lagrangian). Let us demonstrate the idea with an example.

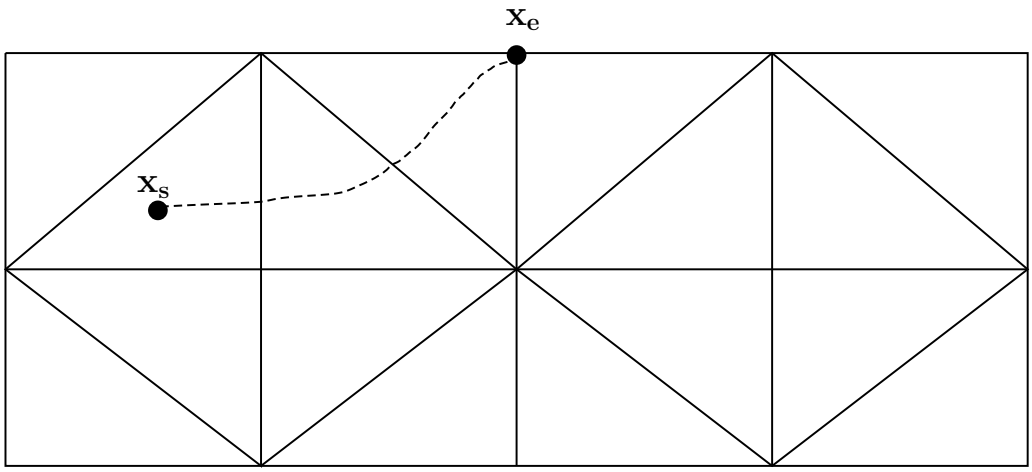


Figure 2.3: Semi-Lagrangian Method

Looking at Fig. 2.3 we want to find the velocity (or any other quantity which is being advected) at time $t + \Delta t$ at point $\mathbf{x_e}$. In a Lagrangian framework this would be the same as the velocity of the (same) particle which at time t was at $\mathbf{x_s}$ and has moved to $\mathbf{x_e}$ at time $t + \Delta t$. So our first problem is how to find $\mathbf{x_s}$. In order to do so we will linearize the velocity field go "backwards" in it. Let us elaborate, in a Lagrangian framework Eq. (2.26) boils down to:

$$\frac{\partial \mathbf{x}}{\partial t} = \mathbf{u}$$

which we choose to approximate with a backward difference.

$$\begin{aligned} \frac{\mathbf{x_e} - \mathbf{x_s}}{\Delta t} &= \mathbf{u}(\mathbf{x_e}, t) \\ \mathbf{x_s} &= \mathbf{x_e} - \Delta t \mathbf{u}(\mathbf{x_e}, t) \end{aligned}$$

Now that we have found out the starting position of our imaginary particle, we are presented with another problem. We are not working in a Lagrangian framework and we are not simulating particles and it is almost certain that $\mathbf{x_s}$ will not be a vertex in our Eluerian mesh. The obvious solution is to interpolate the value inside the element using the values we have. Moreover it can be proven [11] that the method is stable when we interpolate the value inside the element which surrounds it, the same cannot be stated

if we decide to extrapolate and use values from elements which do not contain $\mathbf{x_s}$. In case $\mathbf{x_s}$ lies outside of the mesh we find the closest point on the boundary and take its value instead. Note that the position we found by this procedure is not the exact position of the particle which would end up in $\mathbf{x_e}$ beacuse we have used the velocity at $(\mathbf{x_e}, t)$ to approximate the velocity at $(\mathbf{x_e}, t + \Delta t)$, (see [11] for an error estimate of this approach.)

The type of the mesh is of great importance when implementing the semi-Lagrangian algorithm. There are two properties of the mesh which we should consider. First is the shape of the element and the second is whether the mesh is structured. The term structured mesh is somewhat loose, by this we mean mesh whose elements have the same orientation and the same size, so that the indices of adjacent elements and nodes can be expressed via some explicit formula. An example of structured mesh with rectangular elements would be Fig. 2.4

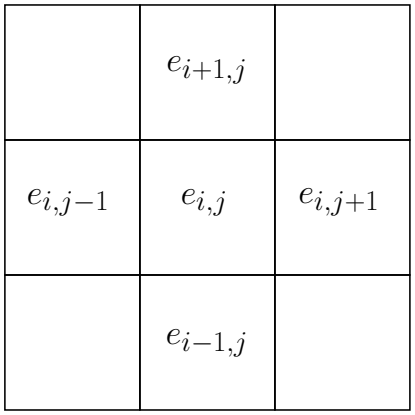


Figure 2.4: Structured mesh with rectangular elements.

The advantage in working with such meshes (especially ones with rectangular elements) is that it is easy to find in which element a point lies. It all boils down to simple integer arithmetics. However when the mesh is unstructured we must iterate through all elements in order to find where a point lies. In our case we are working with unstructured triangular mesh. For this reason KD Tree data structure was implemented in order to accelerate the process of finding which element of the mesh contain a given point. It has complexity of $O(\log n)$ for searching (assuming the tree is balanced) which element contains a point where n is the total number of elements. Also it has

another useful property: it can accelerate finding of the closest mesh point to another arbitrary point, although the worst case has complexity $O(n)$ it is rarely reached and on average it has complexity $O(\log n)$. Assuming a KD Tree is implemented and filled with the elements of the mesh, a pseudo code for the advection phase is presented.

Algorithm 2 Semi-Lagrangian Advection

```
1:  procedure ADVECT(KDTree, nodesIn, Δt)
2:      nodesOut ← nodesIn
3:      for all velocityNode ∈ nodesIn do
4:          x_s ← velocityNode.position − velocityNode.velocity Δt
5:          if pointInMesh(KDTree, x_s) then
6:              element ← findElement(KDTree, x_s)
7:              nodesOut.velocity ← interpolate(x_s, element.velocityNodes)
8:          else
9:              closestVelocityNode ← findClosestNode(KDTree, x_s)
10:             nodesOut.velocity ← closestVelocityNode.velocity
    return nodesOut
```

Finite Element Method Now that we have a stable algorithm for the advection phase we can state the Finite Element Method for the given three way split using implicit Euler method for the diffusion.

$$\mathbf{Q^A} = advect(KDTree, \mathbf{Q^i}, \Delta t) \tag{2.35}$$

$$\left(\begin{bmatrix} \mathbf{M} & 0 \\ 0 & \mathbf{M} \end{bmatrix} + \nu \Delta t \begin{bmatrix} \mathbf{K} & 0 \\ 0 & \mathbf{K} \end{bmatrix} \right) \begin{bmatrix} \mathbf{Q_1^B} \\ \mathbf{Q_2^B} \end{bmatrix} = \begin{bmatrix} \mathbf{M} & 0 \\ 0 & \mathbf{M} \end{bmatrix} \begin{bmatrix} \mathbf{Q_1^A} \\ \mathbf{Q_2^A} \end{bmatrix} \tag{2.36}$$

$$\mathbf{K_p P}^i = -\frac{1}{\Delta t} \begin{bmatrix} \mathbf{B_1} & \mathbf{B_2} \end{bmatrix} \begin{bmatrix} \mathbf{Q_1^B} \\ \mathbf{Q_2^B} \end{bmatrix} \tag{2.37}$$

$$\begin{bmatrix} \mathbf{M} & 0 \\ 0 & \mathbf{M} \end{bmatrix} \begin{bmatrix} \mathbf{Q_1^{i+1}} \\ \mathbf{Q_2^{i+1}} \end{bmatrix} = \begin{bmatrix} \mathbf{M} & 0 \\ 0 & \mathbf{M} \end{bmatrix} \begin{bmatrix} \mathbf{Q_1^B} \\ \mathbf{Q_2^B} \end{bmatrix} - \Delta t \begin{bmatrix} \mathbf{B_{p,1}} \\ \mathbf{B_{p,2}} \end{bmatrix} \mathbf{P}^i \tag{2.38}$$

where Eq. (2.35) uses Algorithm 2.

Again, the Dirichlet BC are imposed in the usual way.

2.3.3 Choice of elements

Now we shall present the elements used in this work. First of all, let us note that generating a FEM mesh is a large subject on its own and falls out of the scope of this work. Wolfram Mathematica was used to generate the FEM meshes used here. As of this date there is no built-in way to generate curved isoparametric elements with Wolfram Mathematica. For the direct approach we have used the $P_{-1}P_0$ non-conforming Crouzeix-Raviart element (see Fig. 2.5). While it does not satisfy the LBB condition it can be proven [18] that it is stable. It has the advantage of being divergence free and is computationally cheap. Using $P_{-1}P_0$ with the direct approach will produce a smaller matrix, compared to using any higher order element, thus the non-linear solver should perform better. For the two operator splitting methods P_2P_1 (see Fig. 2.6) Taylor-Hood element was used, it satisfies the LBB condition and according to [22] it is "the simplest second order element" and an "early favorite".

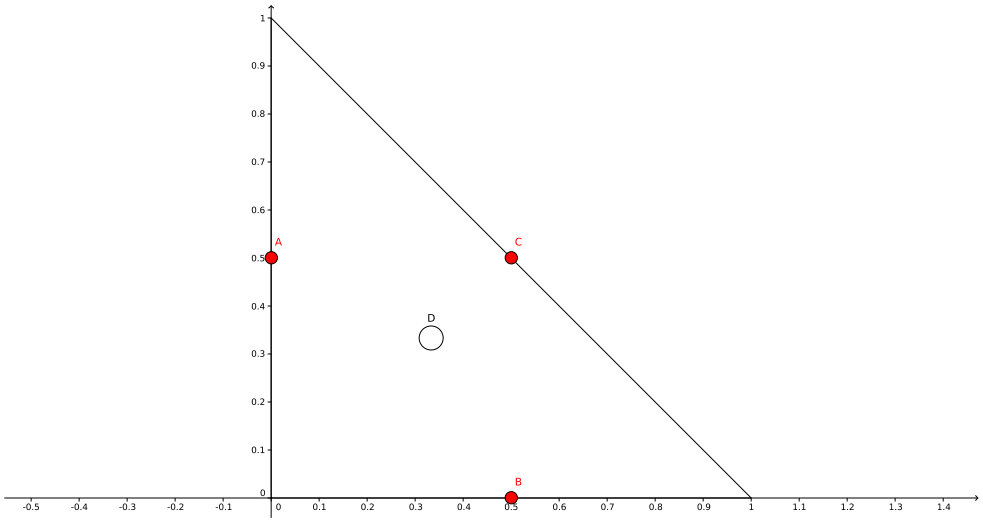


Figure 2.5: $P_{-1}P_0$ non-conforming Crouzeix-Raviart standard element. The 3 degrees of freedom for the velocity are marked with a solid color, the single degree of freedom for the pressure is marked with a circle.

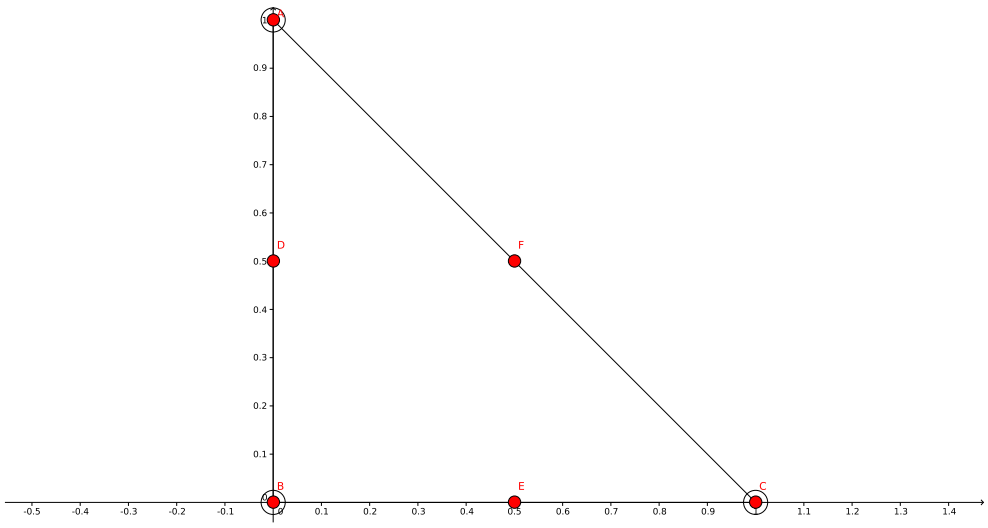


Figure 2.6: P_2P_1 Taylor-Hood standard element. The 6 degrees of freedom for the velocity are marked with a solid color, the 3 degrees of freedom for the pressure are marked with a circle.

2.4 The CSR sparse matrix format

We have chosen store the FEM matrices in the Compressed Sparse Row (CSR) format, which we shall describe briefly. For more information about other sparse matrix formats (see [23]). Let us denote the number of nonzero elements in a matrix with nnz and the number of rows with m. The CSR format consists of 3 arrays, one real array NZ of size nnz containing the nonzero entries of the matrix, one integer array Pos of size nnz containing the column for each nonzero entry and one integer array Start of size $m+1$ containing the start index in Pos for each row of the matrix, the last element of the Start is usually an integer number equal to nnz. We note that the rows are sorted by their index in increasing fashion, but there is no such requirement of the columns and they can be stored in an arbitrary fashion, however for this work we choose to sort the columns in each row in increasing fashion, because this imporoves the cache locality in the computer implementation. Let us illustrate the format with an example, by applying it to the following matrix (see Fig. 2.7).

$$\begin{bmatrix} 1 & 0 & 0 & 2 & 0 \\ 0 & 0 & 3 & 4 & 0 \\ 0 & 0 & 5 & 0 & 6 \\ 0 & 0 & 0 & 0 & 0 \\ 7 & 0 & 8 & 9 & 0 \end{bmatrix}$$

Figure 2.7: Example sparse matrix

The matrix stored in CSR format is shown in Fig. 2.8

NZ	1	2	3	4	5	6	7	8	9
Pos	0	3	2	3	2	4	0	2	3
Start	0	2	4	6	6	9			

Figure 2.8: The matrix from Fig. 2.7 stored in CSR format. The indexing of the columns and the rows starts from 0.

The memory needed to store a matrix in CSR format is $O(2 * nnz + m)$. The space saved by compressing repeating row indexes turns out to be non-negligible [23].

The number of nonzero elements in row i can be found by $Start[i + 1] - Start[i]$, the format allows for having empty rows, then $Start[i + 1] - Start[i] = 0$. To find the next row with nonzero entries one shall iterate to find the first i for which the $Start[i + 1] - Start[i] \neq 0$.

This format is not flexible. Adding and removing nonzero entries after the sparse matrix is initialized would require shifting the NZ and Pos arrays with $O(nnz)$ time complexity and updating the values in the $Start$ array for $O(m)$, yielding total of $O(2 * nnz) + O(m)$ operations which is considered to be slow. Despite the lack of flexibility this format allows for efficient implementation of matrix vector multiplication (see [23]).

2.5 Solving Linear Systems of Equations

Usually systems of equations which arise when applying numerical methods for differential equations have sparse matrices. This is true for the matrices in Eqs. (2.36) to (2.38) and for the matrix in Eq. (2.12). In the course of this work various methods for solving linear systems with sparse matrices

were studied. In this section we shall consider solving Eqs. (2.36) to (2.38) which have symmetric positive definite matrices. Systems with such matrices are most commonly solved using the Conjugate Gradient method, which we shall discuss in this section. For information about sparse matrix formats and other methods for solving linear systems (which might be needed should Eq. (2.12) be considered for solving the differential problem) refer to [23].

2.5.1 Conjugate Gradient

The Conjugate Gradient method is one of the most famous and most studied methods for solving linear systems. It is part of the Krylov family of methods. The method is a special case of Incomplete Orthogonalization Method. In the case of SPD matrices it can be shown that each new direction can be represented with a short recurrency, thus only 3 vectors should be stored at a time. For full derivation and detailed explanation of the method refer to [23]. Here we shall present only pseudocode for the method.

Algorithm 3 Conjugate Gradient Method solving $Ax = b$ with an initial guess x_0

procedure CG(A, b, x_0)
    $r_0 \leftarrow b - Ax_0$
    $p_0 \leftarrow r_0$
    for $j \leftarrow 0, 1, \ldots$ until convergence **do**
        $\alpha_j \leftarrow \frac{(r_j, r_j)}{(Ap_j, p_j)}$
        $x_{j+1} \leftarrow x_j + \alpha_j p_j$
        $r_{j+1} \leftarrow r_j - \alpha_j A p_j$
        $\beta_j \leftarrow \frac{(r_{j+1}, r_{j+1})}{(r_j, r_j)}$
        $p_{j+1} \leftarrow r_{j+1} + \beta_j p_j$
    return x_{j+1}

2.5.2 Zero Fill-in Incomplete Cholesky Preconditioner (IC0)

The point of preconditioning is to improve the condition number of the matrix. Using a good preconditioner can significantly lower the number of iterations needed by the method to converge. A good preconditioner for the CG

method should not change the properties of the matrix and keep it SPD. Let us assume that such a preconditioner is used. Multiplying two sparse matrices often produces a dense matrix, thus it is not wise to precondition the system by directly multiplying the matrix. A better approach incorporates the preconditioner in each iteration. There are few ways to do so in the CG with a preconditioner which is also SPD. The following algorithm is one of those proposed in [23].

Algorithm 4 Preconditioned Conjugate Gradient Method solving $Ax = b$ with an initial guess x_0

1: **procedure** PCG($A, M^{-1}b, x_0$)
2: $r_0 \leftarrow b - Ax_0$
3: $z_0 \leftarrow M^{-1}r_0$
4: $p_0 \leftarrow z_0$
5: **for** $j \leftarrow 0, 1, \ldots$ until convergence **do**
6: $\alpha_j \leftarrow \frac{(r_j, z_j)}{(Ap_j, p_j)}$
7: $x_{j+1} \leftarrow x_j + \alpha_j p_j$
8: $r_{j+1} \leftarrow r_j - \alpha_j Ap_j$
9: $z_{j+1} \leftarrow M^{-1}r_{j+1}$
10: $\beta_j \leftarrow \frac{(r_{j+1}, z_{j+1})}{(r_j, z_j)}$
11: $p_{j+1} \leftarrow z_{j+1} + \beta_j p_j$
12: **return** x_{j+1}

Given Algorithm 4 a logical choice for a preconditioner is the Cholesky decomposition of the matrix $A = LL^T$. However we are not guaranteed that L will be sparse even when A is. Thus we want to find $\widetilde{L}$ which is close to L and $\widetilde{L}\widetilde{L}^T$ which is close to A. We choose $\widetilde{L}$ such that it has the same nonzero pattern as the lower triangular part of A. This preconditioner can be computed via the standard Cholesky decomposition procedure by dropping elements $l_{i,j}$ for i and j such that $a_{i,j} = 0$. Now when the preconditioner and the matrix share the same non-zero pattern the preconditioner could be represented only by its values. We choose to store $\widetilde{L}^T$ in order to make the memory access more regular and improve the performance.

2.6 Numerical Experiment

Let us now show the plot of the solution of Eq. (2.3), afther the 100 iterations with step $\Delta t = 0.01$, produced by solving Eq. (2.12) and compare it to the solution of the same differential problem, but solved by solving Eq. (2.23) with $\Delta t = 0.01$ andEqs. (2.35) to (2.38) with $\Delta t = 0.01$.[1] We can observe that Eq. (2.23) and Eqs. (2.35) to (2.38) produce similar results, while solving Eq. (2.12) with $P_{-1}P_0$ element produces worse results, especially near the boundaries. It must be noted, however, that this comparison is not completely honest, because the two splitting methods use higher order elements. A solution of Eq. (2.3) with the LBB unstable P_1P_1 element could be found in [18]. It is also better than the result achieved by Eq. (2.12). This leads us to believe that an element of at least first degree polynomial space should be used for the pressure. The compuational cost of solving nonlinear systems and the bad results produced with $P_{-1}P_0$ element have discouraged us from further investigation of Eq. (2.12).

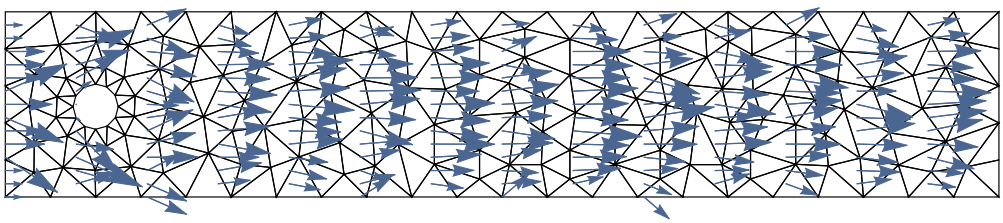


Figure 2.9: Solution for laminar flow via Eq. (2.12) after 100 iterations with $\Delta t = 0.01$

[1] There is a stylistic difference between Fig. 2.9, Fig. 2.10 and Fig. 2.11. This is because the solutions for the direct approach and the Chorin split were obtained by a prototype written in Wolfram Mathematica while the last plot was produced by the final version of the code written in C++ and CUDA.

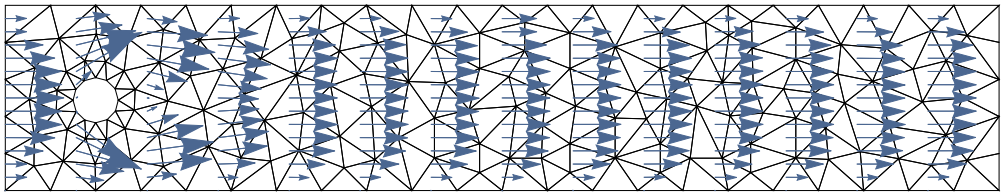


Figure 2.10: Solution for laminar flow via Eq. (2.23) after 100 iterations with $\Delta t = 0.01$

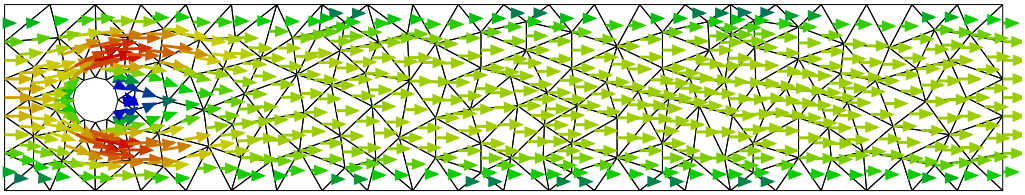


Figure 2.11: Solution for laminar flow via Eqs. (3.3) to (3.4) after 100 iterations with $\Delta t = 0.01$.

2.7 Conclusion

We have decided to further develop the three way splitting method. This is based on a few properties. First, the numerical experiment did not show any problems with the accuracy for the given task. Second, the semi-Lagrangian advection method is perfect for multithreaded CPU and GPU implementation. Third, the method can be used with an arbitrary step size and the step size is not tied to the spatial step (element size).

Chapter 3

Parallel implementation

In this chapter we shall present techniques used to improve the performance of the simulation. We shall present both CPU and GPU parallelization techniques and compare them. The performance will be measured on two machines Computer 1: CPU - Intel(R) Core(TM) i7-3632QM CPU @ 2.20GHz (4 cores, 8 threads), 2 blocks of 4GB SODIMM DDR3 Synchronous 800 MHz RAM and Computer 2: CPU - Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz (8 cores, 8 threads) and 4 blocks of 16GB DDR4 1333 MHz RAM, GPU 1 - NVidia A5000. We shall measure two scenarios - laminar and turbulent flow. Both scenarios will be tested with the following mesh on both machines.

	Mesh
Elements	305854
Velocity Nodes	613458
Pressure Nodes	153802

Let us first sum up the problem and the algorithm used to solve it. We are looking for an approximate solution to the Navier-Stokes equations for a

laminar (Eq. (3.1)) and a turbulent(Eq. (3.2)) flow.

$$\begin{aligned}
&\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u} = -\nabla p + \nu\Delta\mathbf{u}, && (\mathbf{x},t) \in \Omega \times J \\
&\nabla\cdot\mathbf{u} = 0, && (\mathbf{x},t) \in \Omega \times J \\
&\mathbf{u} = 0, && (\mathbf{x},t) \in (\Gamma_1 \cup \Gamma_3 \cup \Gamma_5) \times J \\
&\mathbf{u} = \left(\frac{1.2y\,(0.41-y)}{0.41^2}, 0\right), && (\mathbf{x},t) \in \Gamma_4 \times J \\
&\nu\frac{\partial \mathbf{u}}{\partial \mathbf{n}} - p\mathbf{n} = 0, && (\mathbf{x},t) \in \Gamma_2 \times J
\end{aligned} \tag{3.1}$$

$$\begin{aligned}
&\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u} = -\nabla p + \nu\Delta\mathbf{u}, && (\mathbf{x},t) \in \Omega \times J \\
&\nabla\cdot\mathbf{u} = 0, && (\mathbf{x},t) \in \Omega \times J \\
&\mathbf{u} = 0, && (\mathbf{x},t) \in (\Gamma_1 \cup \Gamma_3 \cup \Gamma_5) \times J \\
&\mathbf{u} = \left(\frac{6y(0.41-y)}{0.41^2}, 0\right), && (\mathbf{x},t) \in \Gamma_4 \times J \\
&\nu\frac{\partial \mathbf{u}}{\partial \mathbf{n}} - p\mathbf{n} = 0, && (\mathbf{x},t) \in \Gamma_2 \times J
\end{aligned} \tag{3.2}$$

To do so at each time step we apply Eqs. (3.3) to (3.6):

$$\mathbf{Q^A} = advect(KDTree, \mathbf{Q^i}, \Delta t) \tag{3.3}$$

$$\left(\begin{bmatrix}\mathbf{M} & 0\\ 0 & \mathbf{M}\end{bmatrix} + \nu\Delta t \begin{bmatrix}\mathbf{K} & 0\\ 0 & \mathbf{K}\end{bmatrix}\right)\begin{bmatrix}\mathbf{Q_1^B}\\ \mathbf{Q_2^B}\end{bmatrix} = \begin{bmatrix}\mathbf{M} & 0\\ 0 & \mathbf{M}\end{bmatrix}\begin{bmatrix}\mathbf{Q_1^A}\\ \mathbf{Q_2^A}\end{bmatrix} \tag{3.4}$$

$$\mathbf{K_p P}^i = -\frac{1}{\Delta t}\begin{bmatrix}\mathbf{B_1} & \mathbf{B_2}\end{bmatrix}\begin{bmatrix}\mathbf{Q_1^B}\\ \mathbf{Q_2^B}\end{bmatrix} \tag{3.5}$$

$$\begin{bmatrix}\mathbf{M} & 0\\ 0 & \mathbf{M}\end{bmatrix}\begin{bmatrix}\mathbf{Q_1^{i+1}}\\ \mathbf{Q_2^{i+1}}\end{bmatrix} = \begin{bmatrix}\mathbf{M} & 0\\ 0 & \mathbf{M}\end{bmatrix}\begin{bmatrix}\mathbf{Q_1^B}\\ \mathbf{Q_2^B}\end{bmatrix} - \Delta t\begin{bmatrix}\mathbf{B_{p,1}}\\ \mathbf{B_{p,2}}\end{bmatrix}\mathbf{P}^i \tag{3.6}$$

where the method *advect* is Algorithm 2.

3.1 CPU multithreading

Our end goal is to implement all algorithms on a GPU. However we need a baseline multithreaded CPU algorithm so that we can measure the pros and

cons of using GPU for simulations. In this section we shall present the steps which were taken in order to multithread the chosen method on CPU.

We shall measure the time it takes for the algorithm to compute 1s of simulation time with time step 0.01 i.e. 100 time steps. The baseline for the measurements is the single threaded algorithm. Tables 3.1 to 3.4 present the time spent in each step of the algorithm (Eq. (3.3) - Step 1, Eq. (3.4) - Step 2, Eq. (3.5) - Step 3 and Eq. (3.6) - Step 4 and the assembling of the corresponding matrices). The measurement is done using C++ timers (std::chrono::steady_clock) placed at the beginning and ending of each function. The data for executing the algorithm is gathered over 11 sequential runs of the algorithm on Computer 1 and Computer 2 for the given mesh.

	Computer 1						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	79.75s	138.33s	505.50s	44.47s	688.30s	20.80s	794.82s
SD	0.64s	3.83s	3.62s	1.30s	8.76s	0.38s	10.01s
Median	80.20s	141.05s	508.06s	45.39s	694.50s	21.07s	801.89s
Min	79.30s	135.62s	502.94s	43.55s	682.11s	20.54s	787.74s
Max	80.20s	141.05s	508.06s	45.39s	694.50s	21.07s	801.89s

Table 3.1: Execution time on 1 thread of Computer 1 for solving for laminar flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and Conjugate Gradient method used for solving the linear system.

	Computer 2						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	60.32s	101.12s	398.50s	30.31s	529.93s	11.82s	605.48s
SD	0.15s	0.57s	1.85s	0.05s	2.14s	0.04s	2.27s
Median	60.31s	100.96s	398.29s	30.33s	529.86s	11.84s	605.45s
Min	60.03s	100.56s	395.98s	30.19s	526.75s	11.76s	602.09s
Max	60.54s	102.54s	402.81s	30.37s	534.01s	11.87s	609.62s

Table 3.2: Execution time on 1 thread of Computer 2 for solving for laminar flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and Conjugate Gradient method used for solving the linear system.

	Computer 1						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	80.35s	196.81s	597.43s	55.31s	849.54s	20.45s	956.29s
SD	0.83s	5.57s	3.18s	0.77s	9.52s	0.08s	10.45s
Median	80.94s	200.74s	599.68s	55.86s	856.28s	20.51s	963.68s
Min	79.76s	192.87s	595.18s	54.76s	842.81s	20.40s	948.90s
Max	80.94s	200.74s	599.68s	55.86s	856.28s	20.51s	963.68s

Table 3.3: Execution time on 1 thread of Computer 1 for solving for turbulent flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and Conjugate Gradient method used for solving the linear system.

	Computer 2						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	61.22s	143.49s	541.06s	37.69s	722.23s	11.90s	798.78s
SD	0.31s	0.77s	3.23s	0.17s	3.97s	0.12s	4.25s
Median	61.35s	143.62s	542.91s	37.73s	724.74s	11.89s	801.48s
Min	60.74s	142.42s	535.43s	37.39s	716.07s	11.75s	792.43s
Max	61.71s	144.89s	544.70s	38.03s	727.36s	12.17s	804.32s

Table 3.4: Execution time on 1 thread of Computer 2 for solving for turbulent flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and Conjugate Gradient method used for solving the linear system.

Although these two examples are not comprehensive and more tests with different scenarios must be made, they give us a few hints about the behaviour of the program. The bulk of the time is spent in the Conjugate Gradient method, with Eq. (3.5) being the most computationally expensive. The matrix used when solving the Pressure Poisson equation is smaller than the other matrices, thus it is logical to assume that the method needs more iterations in order to converge. It looks like turbulent flows need more iterations of the Conjugate Gradient method, most likely because the flow is less "predictable".

3.1.1 Building the FEM matrices

The case we are studying is restricted to geometry which is not changing with time. In this case the matrices used by the FEM are constant and are built once at the beginning of the solution. The problem with assembling FEM matrices in a multithreaded fashion is similar to computing IC0 preconditioners in multithreaded fashion. It depends on the topology of the mesh. Usually graph coloring algorithms are used in order to find which elements will not interfere with one another and fill in the matrix for them in parallel. This technique is also not trivial. Given the fact that the matrices are assembled once and that the assembling takes a small fraction of time, a simpler technique was used - each matrix was computed on a separate thread. Since there are 5 matrices we need 5 threads. This way the assembling of the matrices takes time equal to the time for the most expensive matrix (given that

we have 5 or more threads at our disposal). This is a simplistic technique but it gives satisfactory results. Implementing graph coloring is left for further research.

3.1.2 Multithreading the advection phase

3.1.2.1 A brief introduction to the KD Tree data structure

KD Trees are binary trees which are very commonly used in ray tracing for acceleration of ray-mesh intersections [19] and in artificial intelligence for solving the k nearest neighbours (KNN) problem [4]. Originally they were used to handle large sets of points in a k dimensional space (thus the name KD Tree), but they are trivially extendable to handle different kinds of geometrical objects e.g. k dimensional spheres, triangles in 2D and 3D. We will focus on building KD Tree for the 2D space, which will be used to accelerate two types of queries: find in which triangle of a FEM mesh a given point lies and find the nearest node of our FEM mesh to a given point in space.

We begin with a procedure for building a KD Tree (see Algorithm 5). Each non-leaf node is a tuple of the form (Axis, Splitting Point, Left Child, Right Child), leaf nodes contain a list of elements assigned to them during the construction of the tree. Initially we begin with a root node, an axis aligned bounding box for the FEM mesh and a set of triangles representing the FEM mesh. The axis aligned bounding boxes are represented with their lower left and upper right vertices. At each point we choose an axis and a point along the axis. Then we imagine that there is a line orthogonal to the axis and containing the selected point. All elements on one side of the line are assigned to one of the children. The other elements (which are on the other side of the line) are assigned to the other (see Fig. 3.1). Should a triangle lie in both subspaces defined by the line it is assigned to both children. This continues recursively until the count of the triangles in a node gets below some cutoff value or the maximal allowed depth is reached.

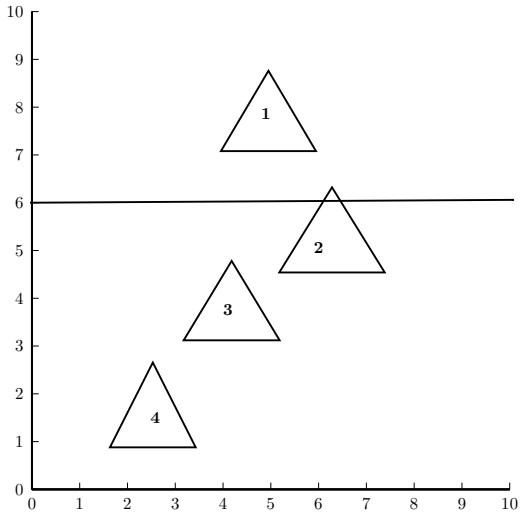


Figure 3.1: The chosen axis is the y axis and the splitting point is 6. A line passing through (0, 6) and orthogonal to the y axis separates the space into two subspaces. Triangles 1 and 2 will be in the right subtree, triangles 2, 3, 4 will be in the left subtree.

Algorithm 5 Build KD Tree

```
1:  procedure BuildKD(node, triangles, AABB, depth)
2:      if count(triangles) ≤ leafSize ∨ depth ≥ maxDepth then
3:          return makeLeaf(triangles)
4:      axis ← chooseAxis()
5:      split ← chooseSplitPoint()
6:      node.axis ← axis
7:      node.split ← split
8:      splitBox(AABB, axis, split, leftAABB, rightAABB)
9:      leftTriangles ← {}
10:     rightTriangles ← {}
11:     for all t ∈ triangles do
12:         if t.A[axis] ≤ split ∨ t.B[axis] ≤ split ∨ t.C[axis] ≤ split then
13:             insert(leftTriangles, t)
14:         if t.A[axis] ≥ split ∨ t.B[axis] ≥ split ∨ t.C[axis] ≥ split then
15:             insert(rightTriangles, t)
16:     root.left ← BuildKD(leftTriangles, leftAABB, depth + 1)
17:     root.right ← BuildKD(rightTriangles, rightAABB, depth + 1)
18:     return root
```

Next we present a procedure (see Algorithm 6) which finds which triangle (if any) contains a point. It is a straightforward task, we just need to traverse the tree. If the given point lies in the left bounding box we follow the left child, otherwise, we follow the right. When we reach a leaf, we need to iterate all triangles in it to see if the point lies in any of them. It is obvious that, given a balanced tree, the complexity of this task is $O(\log m)$ where m is the number of triangles in the tree.

Algorithm 6 Find in which triangle a point lies

```
1: procedure FindTriangle(node, point)
2:     if isLeaf(node) then
3:         for all triangle ∈ node.triangles do
4:             if point ∈ triangle then
5:                 return triangle
           return null
6:     if point[node.axis] ≤ node.split then
7:         return FindTriangle(node.left, point)
8:     else
9:         return FindTriangle(node.right, point)
```

Finding the nearest neighbour is a bit more complex than checking in which triangle a point lies, but the idea is basically the same. The problem is that the nearest mesh point could lie on the opposing side of the splitting axis and thus it is not enough to traverse the tree until reaching the first leaf (see Fig. 3.2). To find the nearest point we start traversing the tree recursively, but we need to visit both children of a given node. There is a common optimization which significantly improves the performance for the average case. Let us call the child whose subspace contains the point from the query the near child and the other child the far child. When traversing the tree we must always go to the near child and traverse down to the far child only when distance to the separating plane is less than the distance between the query point and the current nearest neighbour, Algorithm 7 presents this idea. Let us note that, although this improves the average case complexity to $O(\log n)$ where n is the number of points, the worst case, however, is still $O(n)$.

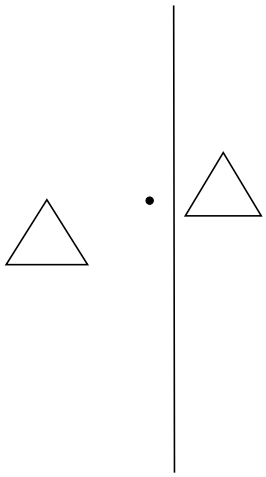


Figure 3.2: The nearest point of the mesh does not always lie in the same subspace, with respect to the separating axis, as the input point.

Algorithm 7 Find the nearest point from a FEM mesh to a given point in space

```
1:  procedure NearestNeighbour(node, point, currentResult)
2:      if isLeaf(node) then
3:          result ← currentResult
4:          for all triangle ∈ node.triangles do
5:              for all node ∈ triangle do
6:                  if distance(node, point) < result.distance then
7:                      result.distance ← distance(node, point)
8:                      result.point ← node
9:          return result
10:     near ← node.left
11:     far ← node.right
12:     if point[node.axis] > node.split then
13:         swap(near, far)
14:     result ← NearestNeighbour(near, point, currentResult)
15:     if result.distance > abs(point[node.axis] − node.split) then
16:         result ← NearestNeighbour(far, point, result)
17:     return result
```

3.1.2.2 KD Tree implementation details

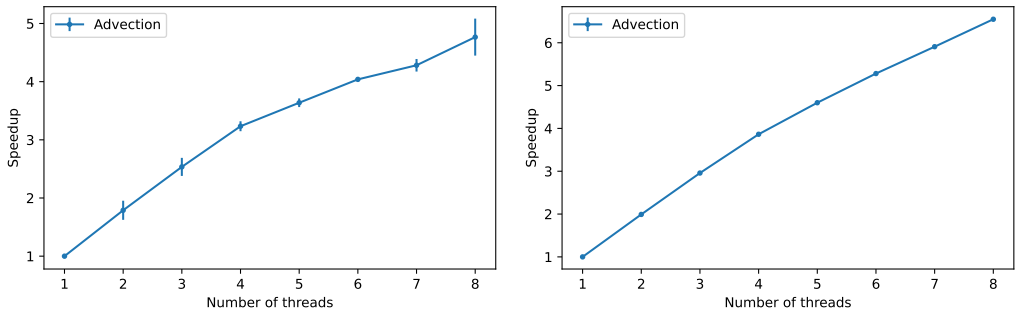
Before discussing the multithreading part we shall give some implementation details about the KD Tree and a commentary on how it should be improved. The tree is built in a recursive top-down fashion. It is stored in a linear array, the root is stored at index 0. If a node has index i in the array its left child is always stored at index $i + 1$, while its right child is stored at some other index which is not important. In our case for a given node we first build its left subtree and then the right subtree. Thus the index of the right child is the first free index after the last node of the left subtree. Such ordering is cache friendly, because when we reach a node its left child will always be on the same cache line. The maximum allowed depth is 29 and in case a node has less than 16 triangles in it, it is made a leaf. The cutoff value of 16 does not have any specific reasoning behind it, different values were tested and there were no significant changes in runtime. The most important part of building a KD tree for this case is to make it as balanced as possible. At each step a splitting point along the splitting axis must be chosen such that the difference between the number of triangles on both of its sides is as small as possible. There are two – questions how to choose the axis and how to choose the point. We select the axes in a round-robin fashion. Our strategy for choosing a split point is to split the bounding box of the parent node in the center, along the splitting axis, subdividing it into equal sized bounding boxes for each child. This strategy is rather simplistic and does not work well. For small meshes such as the given mesh it is tolerable and the run time is still dominated by the CG method, however should the given mesh become, say, 2 times larger the runtime increases unproportionally, making the method borderline unusable. A better approach should be implemented. The simplest approach is to use the Quickselect algorithm [9] in order to find the median of the points in the current tree and use it as a splitting point when subdividing into subtrees. The algorithm has best and average case complexity of $O(n)$ where n is the number of points, but worst case of $O(n^2)$. It was implemented and it significantly improved the tree traversal performance. However this increased the time needed to build the tree by a substantial amount. This happens because the algorithm must be executed for each node. Thus making it a viable choice when the simulation has a large amount of steps. Another option is to use the PICK algorithm [2], it is a well known algorithm for finding the median of an array in deterministic $O(n)$ time. We have not tried to implement it. On one hand it is not trivial

to implement on the other hand, although, it has asymptotically linear time, the constant (which is implied by the notation) is known to be large, thus making it slower than Quickselect with random pivot selection, until the data set becomes large enough. Due to this reason we think that it falls out of the scope of the current work. It would be interesting to implement it, however. Another approach worth trying is described in [6].

3.1.2.3 Splitting the work between threads

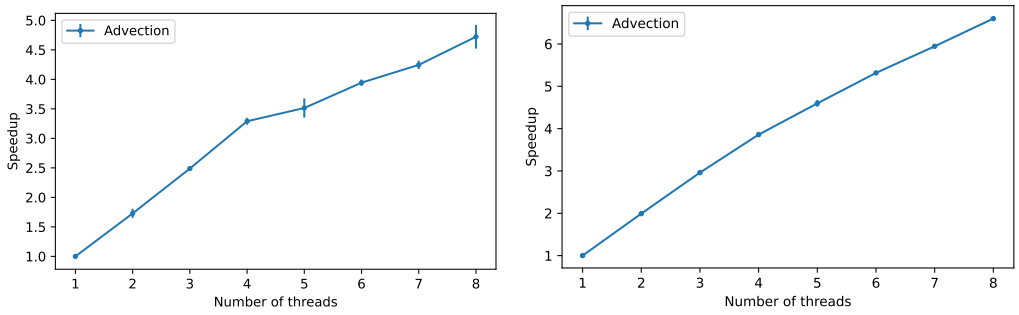
As it was already mentioned the semi-Lagrangian algorithm is trivial to parallelize. We shall use simple fork-join technique to do so. It is obvious that each quantity at each vertex (in our case only the velocity is advected, but in general there can be other quantities such as temperature, concentration, etc.) can be advected on its own without interacting with the other advected quantities and other mesh vertices. Thus we shall split all velocity nodes into groups and each group should be scheduled to a separate CPU thread. For the implementation we use the Thread Building Blocks (TBB) multithreading library provided by Intel. It does the job of spawning threads and distributing work between them, while we shall only pass the function which will be executed on each thread. Next we present how this scales with multiple threads see(Figs. 3.3 to 3.4). For more detailed view refer to Appendix A

3.1.2.4 Speedup results



(a) Computer 1 (b) Computer 2

Figure 3.3: Scaling of Eq. (3.3) solved via semi-Lagrangian method with $\Delta t = 0.01$ over the given mesh for laminar flow. Vertical bars represent 2 times the standard deviation of the measured speedup.



(a) Computer 1 (b) Computer 2

Figure 3.4: Scaling of Eq. (3.3) solved via semi-Lagrangian method with $\Delta t = 0.01$ over the given mesh for turbulent flow. Vertical bars represent 2 times the standard deviation of the measured speedup.

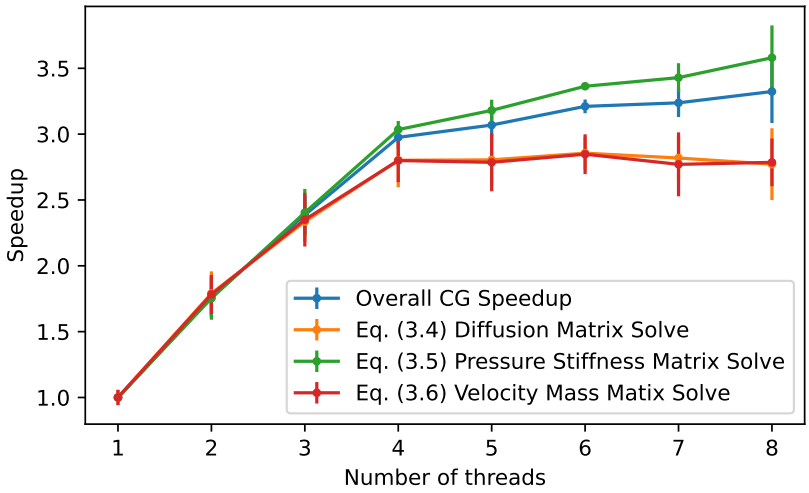
From the provided data we can see that the algorithm scales with increasing the number of threads. The standard deviation is low, meaning that the runtime does not fluctuate. One thing which should be noted is that according to the data, the achieved speedup for Computer 1 drops a bit when the mark of 5 threads is reached. After that the speedup still increases when the

number of threads increases, but it is further from the expected one. This phenomenon is due to Intel Hyper-Threading technology. Computer 1 is a quad core computer. This means that in theory only 4 tasks can be executed in parallel, thus the speedup is much better in the range [2;4] threads. However modern CPUs have very complicated architecture. One CPU core contains multiple arithmetic units, pipelines, cache and branch predictors, etc. It often happens that one thread cannot fully utilize the CPU. Imagine the simplest case possible, the thread does only integer operations. In this case all floating point arithmetic units are idle. Intel Hyper-Threading technology addresses this issue, by presenting each physical core as 2 logical cores. This way through clever task scheduling the CPU can be utilized better. This technique has its limitations and it is obvious that using threads more than the number of cores does not mean that the performance should double. On the other hand Computer 2 has 8 core processor and does not support Hyper-Threading. We can see that the algorithm, indeed, continues scaling in an almost linear fashion even beyond the 4 threads mark.

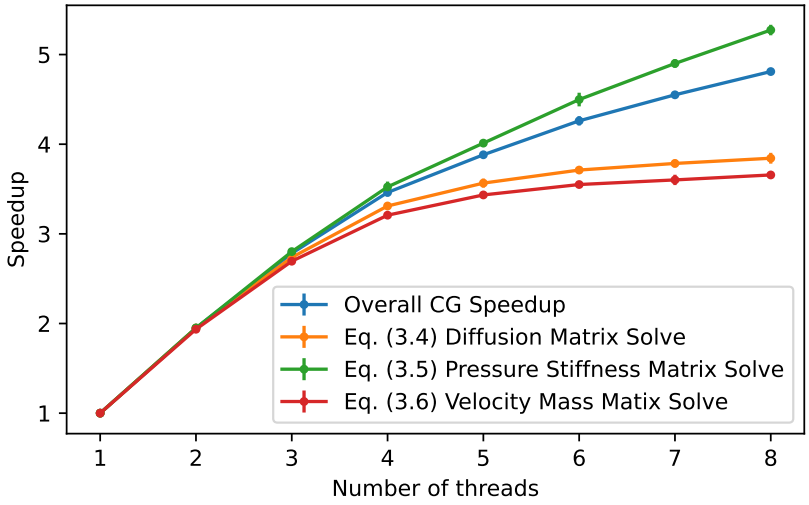
3.1.3 Conjugate Gradient

Contrary to the advection, Conjugate Gradient cannot be multithreaded as a whole. However we can multithread each step of it: the matrix vector product, the vector additions and the dot product. All of these are trivially parallelizable on their own. TBB was used for this part as well. For the vector matrix product the matrix is split into blocks, each block containing some set of rows. Thus each CPU thread gets to compute all dot products of the rows in its block of the matrix and the vector which multiplies the matrix. For the vector additions we use a similar technique, the vectors are split in blocks and each CPU thread gets to add the parts of the two vectors in its block. For the dot product the built-in tbb::parallel_deterministic_reduce operation is used, it also uses the same idea, and splits the vectors into blocks, then each CPU thread computes the dot product of the subvectors in its block, this leaves us with an array $a_1, a_2, ...a_n$ of the computed (sub) dot products, which must be added together in order to compute the final result. What tbb::parallel_deterministic_reduce does is to provide specific order to all additions so that the result of the dot product is always deterministic. The downside of it is that TBB cannot dynamically split the workload and the sizes of each block must be defined by the programmer. We have empirically chosen blocks of size 8192. As we mentioned the Conjugate Gradient cannot

be multithreaded as a whole, and the reason is the dot product which acts as a barrier between the separate phases of the algorithm. Next in Fig. 3.5 and Fig. 3.6 we provide graphics and tables showing how the algorithm scales to multi-core systems and a commentary on the achieved results.

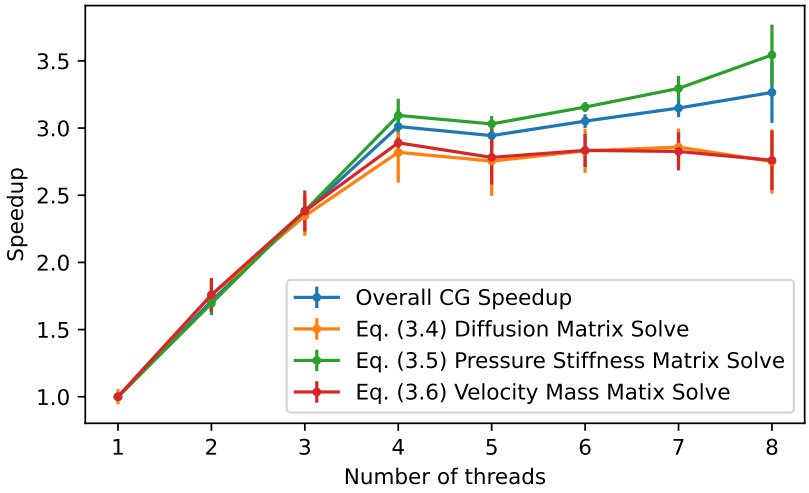


(a) Computer 1

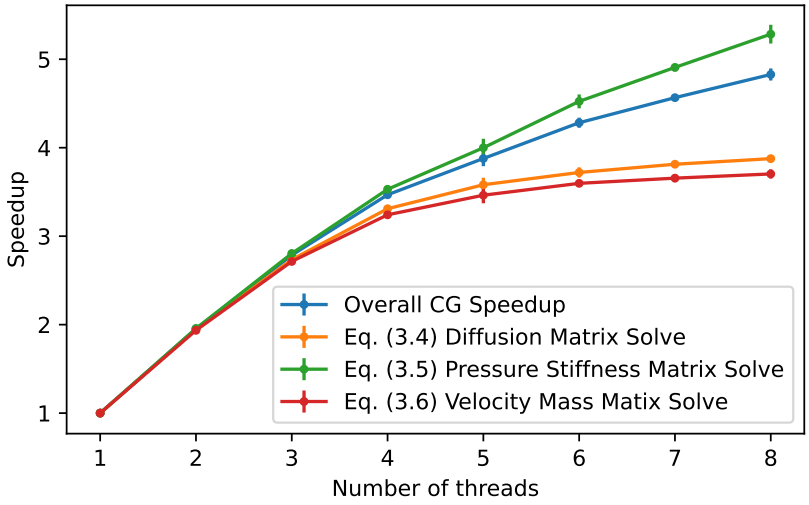


(b) Computer 2

Figure 3.5: Scaling of Eqs. (3.5) to (3.6) solved via Conjugate Gradient method with tolerance of 10^{-8} applied over the given mesh for laminar flow. Vertical bars represent 2 times the standard deviation of the measured speedup.



(a) Computer 1



(b) Computer 2

Figure 3.6: Scaling of Eqs. (3.5) to (3.6) solved via Conjugate Gradient method with tolerance of 10^{-8} applied over the given mesh for turbulent flow. Vertical bars represent 2 times the standard deviation of the measured speedup.

The plots show how each of Eqs. (3.5) to (3.6) scale and how the method scales overall[1]. From the plots we see that the method scales worse than the advection. Solving Eq. (3.6) and Eq. (3.4) scales up to about 4 cores and then almost stops scaling. The hyperthreading technology does not help much. We see that it improves the average speedup but the standard deviation is so large, hinting that most of the results are just noise. Looking at Computer 2 which has 8 physical cores we see that these equations indeed stop gaining any significant speedup after the 4 core mark.

On the other hand solving the Pressure Poisson equation gained speedup even with hyperthreading. There are two issues here. First, although each step of the CG method is parallelizable, the operations done at each step are simple (adding vectors together or computing dot products). Contemporary CPUs are very good at these things and increasing the core count does not

[1]Note, the overall scaling is not the mean scaling of Eqs. (3.5) to (3.6). It is the total time spent in the CG method on one thread divided by the total time spent in the CG method on multiple threads. It behaves more like a weighted average of Eqs. (3.5) to (3.6). The equations which are more time consuming have a higher weight.

help simply because there is not enough work to do. To make things worse the modern CPUs are so good at simple arithmetics, that they are faster than loading data from the memory. Second problem is concerned with Eq. (3.6) and Eq. (3.4). They do not scale because CG does a small amount of iterations with them, compared to the number of iterations done while solving the Pressure Poisson system, Table 3.5 shows the number of iterations done by the CG method for each equation.

One thing to note is that our implementation of the CG does not take advantage of superscalar instructions (SSE4.2, AVX, AVX2) [15]. The theoretical speed which they can provide is up to 4 (SEE4.2), 8 (AVX), 16 (AVX2). It is highly unlikely to reach the maximal theoretical speedup, but we are optimistic that they will provide considerable speedup. Implementing the CG method using superscalar instructions is left for further research.

The conclusion we can draw from this experiment is that the CG method scales to multiple processors only if there is enough data to saturate the CPU. There is always a point at which throwing more cores at the problem will not improve the performance. Thus it is better to aim at a CPU with higher clock rate.

3.1.3.1 Preconditioning

Given CG methods require a lot of computational time, we have tried to reduce the number of iterations done by it and therefore we have hoped that the run time would also be reduced. For that purpose we have decided to implement a preconditioned CG method. Since all matrices are symmetric and positive definite, the logical choice was to implement zero fill-in Incomplete Cholesky Preconditioner (IC0). As we can see from Table 3.5 the iterations done by the preconditioned method were significantly lowered. However we have faced two problems. Our implementation of the factorization is not multithreaded and did take a lot of time. Of course, when the simulation time is increased, this effect will become smaller. Since the matrices involved in solving Eq. (3.6) and Eq. (3.4) are larger than the one involved in Pressure Poisson equation, computing a preconditioner for them takes significantly more time. Given the fact that we spent significantly more time solving the Pressure Poisson system it is justifiable not to build preconditioners for Eq. (3.6) and Eq. (3.4). The second problem we have encountered is at the application of the preconditioner, which is also single threaded. Although the number of iterations is cut in half the performance was almost the same

for single threaded execution (see Tables 3.6 to 3.8). In order to apply the preconditioner two triangular systems must be solved. Multithreading of such tasks depends on the structure of the mesh and also on the numbering of the nodes and is left for further research (see Section 3.5).

Implementing multithreaded computation of IC0 preconditioner and multithreaded application of it is not a trivial task and is left for further research. It's worth noting that the multithreaded properties of the algorithms are limited and highly depend on the structure of the underlying matrices (and on the geometry of the simulation). Usually the recommended preconditioner in a multithreaded environment is the Block Jacobi preconditioner [5], [23]. Further research is needed in order to compare them.

Differential problem	Total iterations Eq. (3.5)	Total iterations Eq. (3.6)	Total iterations Eq. (3.4)
Laminar flow	195422	2763	10370
Turbulent flow	264394	3504	14820
Laminar flow (IC0)	72353	495	1855
Turbulent flow (IC0)	85047	693	2693

Table 3.5: Total number of iterations done by the CG method with the given mesh after 100 time-steps of Eqs. (3.3) to (3.6) with $\Delta t = 0.01$

	Mean Time	Median	SD	Min Time	Max Time
Compute IC0 for Pressure Stiffness Matrix	15.66	15.67	0.02	15.64	15.68
Compute IC0 for Velocity Mass Matrix	390.33	389.98	1.32	389.22	391.79
Compute IC0 for Diffusion Matrix	380.68	379.17	4.98	376.62	386.23

Table 3.6: Time needed to compute the IC0 preconditioner on a single thread on Computer 2 for all matrices.

	Mean Time	Median	SD	Min Time	Max Time
Solve Eq. (3.5)	393.30	393.50	0.29	393.09	393.50
Solve Eq. (3.6)	22.01	22.03	0.03	21.99	22.03
Solve Eq. (3.4)	54.76	54.76	0.00	74.75	54.76

Table 3.7: Time needed to solve Eqs. (3.5) to (3.6) by PCG with IC0 preconditioner for a laminar flow on a single thread on Computer 2

	Mean Time	Median	SD	Min Time	Max Time
Solve Eq. (3.5)	252.75	259.95	10.19	245.54	259.95
Solve Eq. (3.6)	15.54	15.55	0.02	15.52	15.55
Solve Eq. (3.4)	37.77	37.86	0.12	37.68	37.86

Table 3.8: Time needed to solve Eqs. (3.5) to (3.6) by PCG with IC0 preconditioner for a laminar flow on 8 threads on Computer 2. The preconditioner is applied on single thread, however, the other steps of the method (dot product, vector matrix multiplication and vector addition) are computed on 8 threads.

3.1.4 Conclusion

- Building the FEM matrices takes a small amount of time compared to solving the equations. If the geometry is not changing over time, it is acceptable to build all matrices upfront, each matrix can be built on a separate thread.
- The semi-Lagrangian approach scales almost linearly with the increase of the CPU cores.
- Building KD trees by choosing the midpoint of the splitting axis as a separation point works for small meshes. For large meshes it produces an unbalanced tree which is slow to traverse.
- CG method is composed of simple operations. It scales well with the increase of CPU cores only when there are enough operations to saturate the CPU.

- IC0 preconditioning reduces significantly the number of iterations needed for the CG method in order to converge.
- Single threaded computation and application of IC0 are impractical.

3.2 GPU Implementation

Our GPU implementation is written in CUDA, a general purpose language provided by NVidia. Applications written in CUDA can be run only on NVidia GPUs. However GPU architectures do not vary drastically between different vendors, thus the lessons learned from the experiments should be valid for all vendors.

3.2.1 A brief introduction to GPU architecture and programming model

Here we shall present the GPU architecture and introduce the terms which are used later in the research. For detailed explanation refer to the NVidia Programming Guide[2].

GPUs and CPUs have evolved to do different tasks, thus they have different architecture and excel at different problems. Widespread CPUs have relatively small core count, usually between 2 and 16. Only recently companies started producing mainstream CPUs with higher core count with AMDs 64 core Threadripper being one of the mainstream CPUs with highest core count[3]. A single CPU core has a higher clock rate (compared to a single GPU core) and it excels at doing sequential work. CPUs deal with latency (waiting for data to arrive from the RAM, waiting for other instructions to complete, etc.) by having a lot of supplementary hardware, besides the arithmetic units and registers. They have sophisticated memory caching systems (because waiting for data from RAM is slow), prefetching systems which try to predict from which part of the RAM data will be requested and fetch it in advance, branch prediction, instructions are decomposed to even smaller

[2]The programming guide is available at `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`. This page links to the programming guide for the latest version of the CUDA language

[3]The Threadripper has 64 physical cores which can effectively spawn 128 threads

instructions and executed on a pipeline, instructions can be executed out of order and the list goes on and on.

On the other hand NVidia GPUs have higher CUDA core count starting from a few hundred and reaching to a few thousand, depending on the GPU architecture. For example NVidia A5000 card has 8,192 CUDA cores. CUDA cores, however, have lower clock rate, there is no branch prediction, prefetching, out of order execution. We can compare a CUDA core to a single lane in a superscalar CPU. On an NVidia GPU, CUDA cores are distributed among several Streaming Multiprocessors (SM). For example NVidia A5000 has 64 SMs. The streaming multiprocessor is responsible for fetching data and instructions and execution scheduling. Each instruction is executed *simultaneously* by 32 CUDA cores (called a warp)[4]. When a warp reaches a conditional statement and some of the threads need to take one code path, while others need to take the other code we have thread (execution) divergence. The GPU will execute the first branch, with the threads which enter it, while the others are idle and then the second. Having thread divergence slows the program and should be avoided. At each cycle the SM selects warps which are active i.e. they are not waiting for memory to arrive from the VRAM, not waiting for operands to be computed, not waiting on some synchronization primitive, etc. and executes them. Thus the latency is being hidden by letting the GPU execute warps which are ready, instead by having sophisticated hardware which tries to prevent the latency (as the CPU does). A program executed on the GPU is called a *kernel*. When a kernel is called it is executed concurrently by $N = gridSize * blockSize$ separate threads. Each kernel is launched in a grid, divided in some, user-defined, number of (thread) blocks, where each block is limited to having at most 1024 threads in it. For convenience grids and blocks can be 1, 2 or 3 dimensional. Threads in a single block always reside on the same SM and they can share a small amount of memory called *shared memory*. Usually it is recommended for a kernel call to be launched with a grid size larger than the number of blocks all SMs can handle. This is done so that if a SM finishes earlier with its thread blocks it could take another block waiting to be executed, thus keeping the GPU occupied.

[4]Modern NVidia architectures sometimes can divide a warp into 2 half-warps

3.2.2 Building the FEM matrices

Building the FEM matrices is difficult to be implemented on a GPU. There are some dependencies between the data and some sort of sychronization primitive must be used. This is why, we have left it for a further research. The matrices which are used by the GPU implementation are built on the CPU using the algorithm which we have previously described.

3.2.3 Advection

The algorithm used for advection on GPU is the same as the one on the CPU. We have managed to write the source code so that it can be executed by CPU and GPU. There is one crucial detail for the GPU implementation and it is related to how the KD tree is traversed. Using recursion leads to a high amount of thread divergence which significantly slows down the GPU execution. In order to improve the performance the recursion must be emulated via a stack and a loop. This detail has an insignificant effect on the CPU implementation, however. The kernel is launched with threads equal to the number of velocity nodes in the mesh. Each GPU thread computes the new velocity for a single velocity node of the mesh. We have tested the code with different block sizes. While the best results were reached with a block size of 128 threads, we have found out that the size of the block does not change the performance significantly.

3.2.4 Conjugate Gradient

For the GPU implementation of the CG method only non preconditioned iterations were considered. We have two implementations: multi kernel and mega kernel implementation in order to compare which approach is better. Both implementations use the same implementation of the basic building blocks of the method (sparse matrix vector multiplication, vector addition and dot product) with the main difference being the way the kernels are launched. We shall elaborate on the differences between multi kernel and mega kernel approaches later. First we shall give some implementational details about the kernels we have implemented.

All matrices resulting from the use of triangular P_2P_1 Taylor-Hood element matrices have about 20 entries per row. For such matrices we implement the sparse matrix vector product by spawning kernels with threads equal to

the number of rows in the matrix. Each GPU thread computes the dot product between the $i-th$ row of the matrix and the vector multiplying it, which gives the $i-th$ element of the result vector. Should we choose elements with more degrees of freedom (producing matrices with more than 32 entries per row) it is better to use one warp per row to compute the dot product between the row and the vector. For our case changing the block size does not change the performance significantly. Blocks of 512 threads were used.

The dot product implementation is a bit more interesting because it is a reduction operation. Taking two vectors and "reducing" them to a number. Such operations usually require some form of thread communication and synchronization. In order to implement the dot product we need to use shared memory, block synchronization and atomic operations. The implementation is explained in detail in [17]. Because of the nature of the reduction algorithm the threads per block must be a power of 2. We have found out that the performance is significantly improved when more threads per block are used. Using the maximum allowed threads per block (1024) gives about 1.5 times improvement in run time compared to the previous power of 2 (512). The algorithm used is not deterministic.

3.2.4.1 Multi kernel vs. mega kernel implementation

The multi kernel approach is the one found usually in the literature. In it every operation (sparse matrix vector product, vector addition, dot product) is issued as a separate kernel call from the CPU side. It is easy to implement and it resembles the CPU version of the algorithm. The CG method has three implicit synchronization points (two after each dot product and one after the iteration ends) and it is easier to implement the synchronization using the multi kernel approach. Thus using the multi kernel approach at each CG iteration we will launch 6 kernels in total (one sparse matrix vector multiplication, two dot products and 3 vector additions). The downside of the multi kernel approach is that kernel calls could be expensive and when the kernel takes a relatively small amount of time the overhead of the kernel call could become larger than the performance gain of using a GPU for the task. Another disadvantage is that kernels submitted to the same CUDA stream are executed sequentially. The advantage of using a multi kernel approach is that smaller kernels are easier to analyze and optimize. There is less chance of having execution or data divergence among the threads.

On the other hand the mega kernel approach has the whole method exe-

cuted on the GPU. Thus there is one and only one kernel call when we need to solve a system using the CG method. The tricky part in implementing the CG method using the mega kernel approach is the synchronization. Our algorithm uses 3 *grid* wide synchronization points[5] per CG iteration: one after each dot product invocation and one at the end of each iteration. CUDA versions less than 9 and graphic cards with compute capability (CC) less than 6 do not support grid wide synchronizations natively. We have devised a custom thread barrier which can be used in case when native grid synchronization is not available and it can be found in Appendix B. However all programming guides and common wisdom strongly advise against using such custom barriers, so it must be used cautiously. No matter what synchronization primitive is used, native or the custom there is one limitation. The number of blocks in the grid must be equal to the total number of blocks that the GPU can handle. This is done by computing how much thread blocks a single SM can handle[6] and then multiplying it by the number of SMs in the GPU. Should a grid with more blocks be passed an error will occur. If native synchronizations are used the CUDA API will return an error (this is easy to spot and fix), however, if the custom barrier is used the kernel will (sometimes) be caught in a deadlock. On one hand using the mega kernel approach gives the GPU less blocks and warps which means that it has less chance to hide latencies. On the other hand this approach does not suffer the overhead of kernel launches[7]. Our experiments show that the CG method using the mega kernel approach is significantly faster than the multi kernel approach.

[5]By grid wide synchronization we mean a thread barrier. All threads in the grid must reach the synchronization point and only after that any thread is allowed to continue execution.

[6]CUDA Driver API provides the function *cuOccupancyMaxActiveBlocksPerMultiprocessor* to do so

[7]CUDA 11 provides an API called CUDA graphs which is supposed to lower the overheads of multi kernel approach by predefining the execution order of the kernels. In theory it should bring the best of both worlds. This approach needs further investigation.

3.2.5 Results

	GPU Multi Kernel approach						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	0.56s	14.67s	65.96s	4.71s	85.34s	4.02s	91.16s
SD	0.01s	0.07s	0.34s	0.02s	0.35s	0.02s	0.36s
Median	0.56s	14.69s	66.00s	4.71s	85.35s	4.02s	91.19s
Min	0.53s	14.56s	65.33s	4.69s	84.81s	3.99s	90.62s
Max	0.57s	14.75s	66.47s	4.74s	85.87s	4.03s	91.69s

Table 3.9: Execution time on of GPU 1 for solving for laminar flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and multi kernel Conjugate Gradient method used for solving the linear system.

	GPU Multi Kernel approach						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	0.84s	20.69s	78.12s	5.70s	104.50s	4.03s	110.63s
SD	0.01s	0.04s	0.25s	0.01s	0.25s	0.02s	0.23s
Median	0.85s	20.71s	78.12s	5.70s	104.56s	4.02s	110.66s
Min	0.83s	20.61s	77.75s	5.69s	104.15s	4.01s	110.33s
Max	0.86s	20.73s	78.55s	5.72s	104.99s	4.07s	111.11s

Table 3.10: Execution time on of GPU 1 for solving for turbulent flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and multi kernel Conjugate Gradient method used for solving the linear system.

	GPU Mega Kernel approach						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	0.54s	2.98s	8.19s	1.26s	12.44s	4.02s	18.24s
SD	0.01s	0.07s	0.09s	0.01s	0.06s	0.02s	0.06s
Median	0.54s	2.96s	8.20s	1.26s	12.43s	4.02s	18.22s
Min	0.53s	2.95s	7.95s	1.25s	12.36s	3.99s	18.13s
Max	0.55s	3.18s	8.29s	1.28s	12.54s	4.05s	18.32s

Table 3.11: Execution time on of GPU 1 for solving for laminar flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and mega kernel Conjugate Gradient method used for solving the linear system.

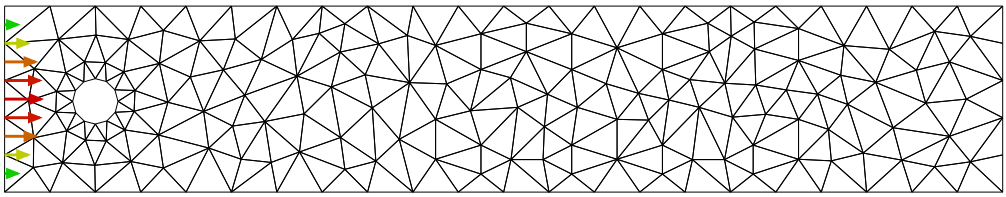
	GPU Mega Kernel approach						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	0.84s	4.01s	9.72s	1.45s	15.18s	4.00s	21.26s
SD	0.00s	0.01s	0.04s	0.00s	0.04s	0.01s	0.05s
Median	0.84s	4.01s	9.70s	1.45s	15.17s	4.00s	21.27s
Min	0.83s	3.99s	9.67s	1.44s	15.12s	3.98s	21.20s
Max	0.84s	4.02s	9.79s	1.46s	15.24s	4.02s	21.32s

Table 3.12: Execution time on of GPU 1 for solving for turbulent flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and mega kernel Conjugate Gradient method used for solving the linear system.

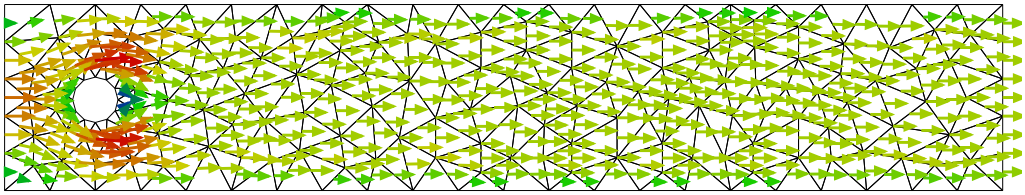
3.3 Numerical Experiment

Here we present the result produced by the GPU for both laminar and turbulent flows.

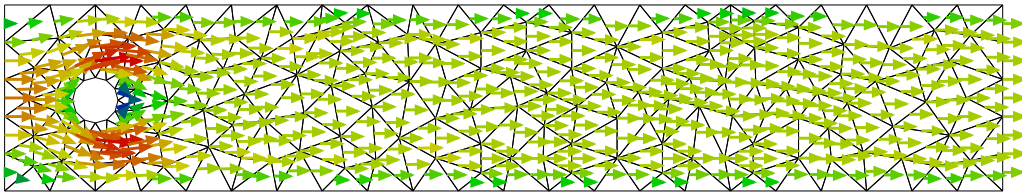
3.3.1 Laminar Flow



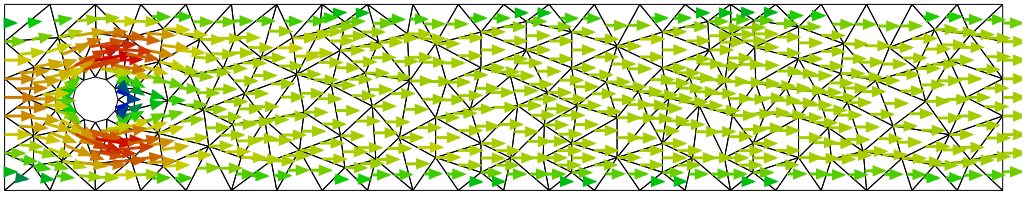
(a) Laminar flow at t=0s



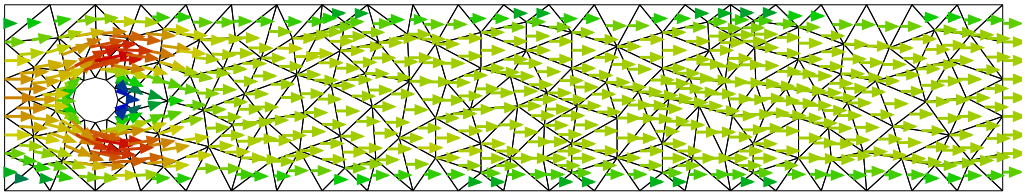
(b) Laminar flow at t=0.09s



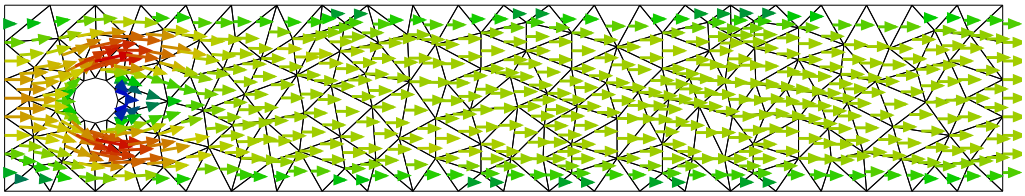
(c) Laminar flow at t=0.29s



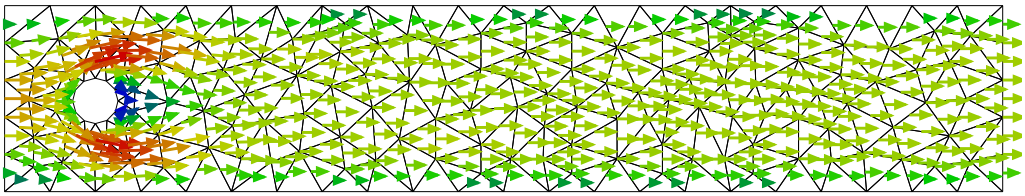
(d) Laminar flow at t=0.39s



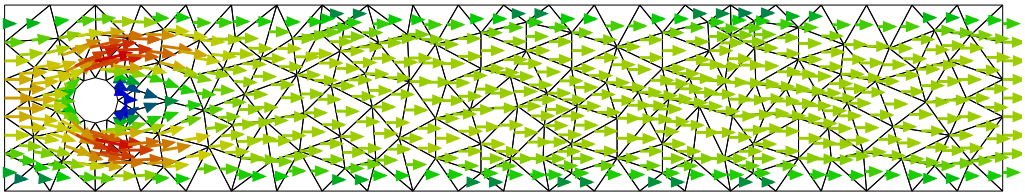
(e) Laminar flow at t=0.49s



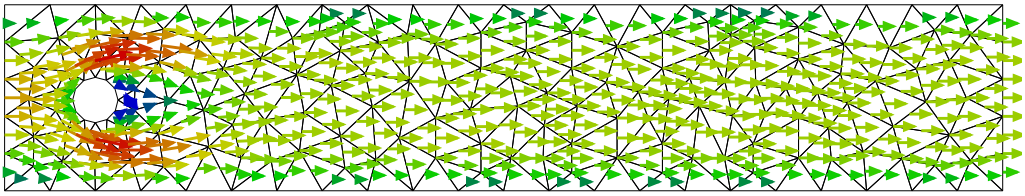
(f) Laminar flow at t=0.59s



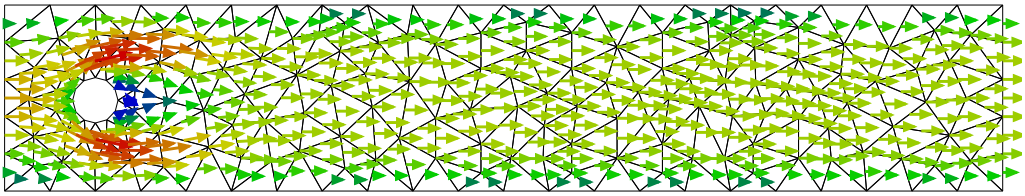
(g) Laminar flow at t=0.69s



(h) Laminar flow at t=0.79s



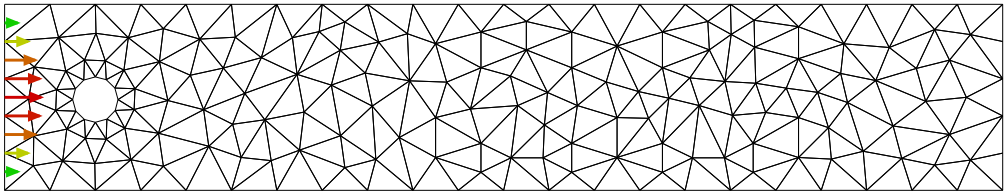
(i) Laminar flow at t=0.89s



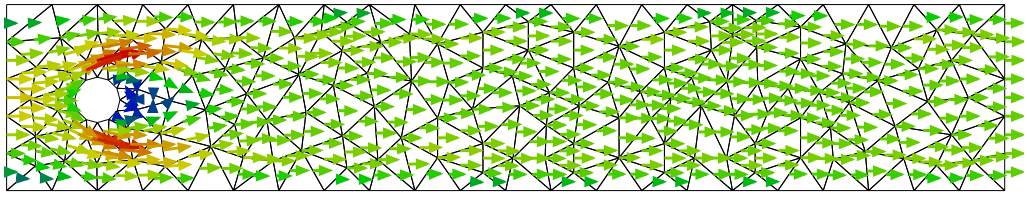
(j) Laminar flow at t=0.99s

Figure 3.7: Solution for laminar flow via Eqs. (3.3) to (3.4) on GPU with $\Delta t = 0.01$.

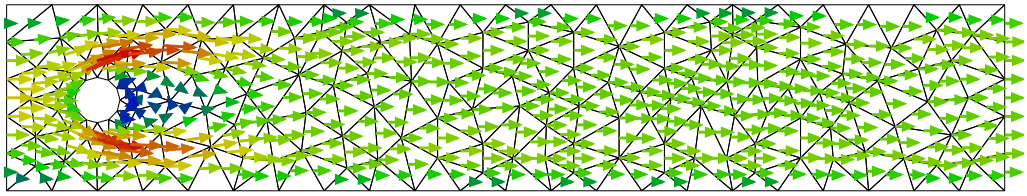
3.3.2 Turbulent Flow



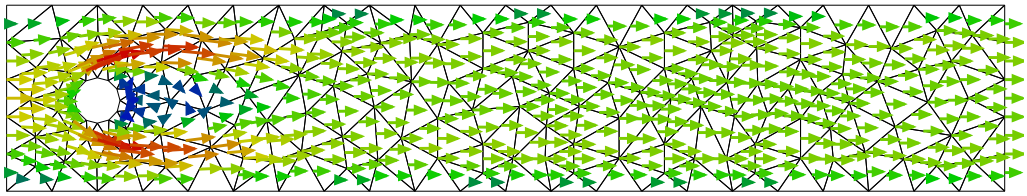
(a) Turbulent flow at t=0s



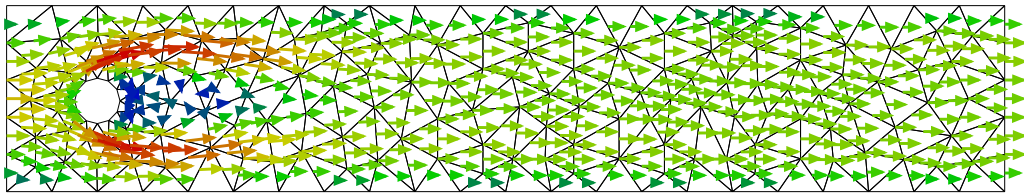
(b) Turbulent flow at t=0.09s



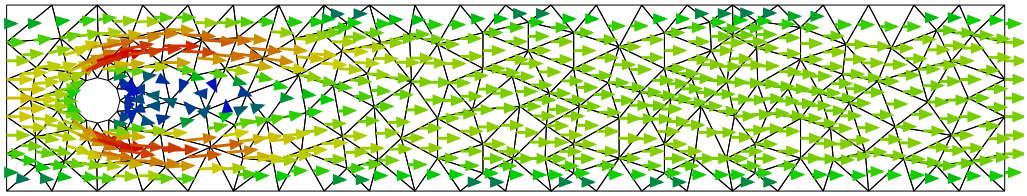
(c) Turbulent flow at t=0.29s



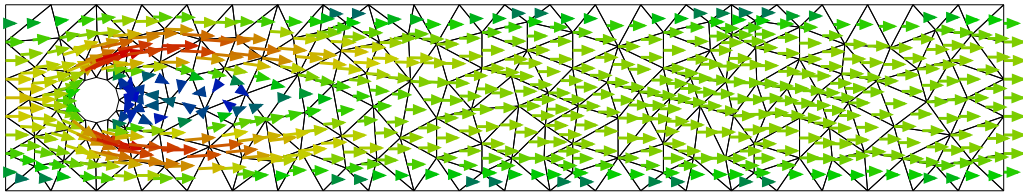
(d) Turbulent flow at t=0.39s



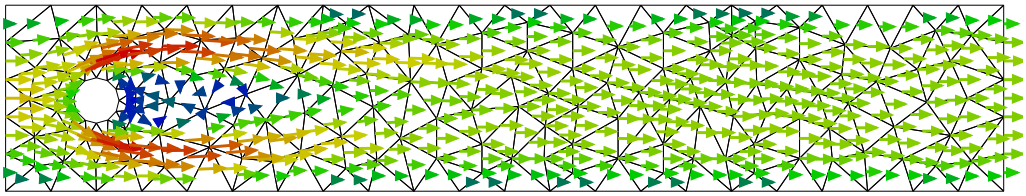
(e) Turbulent flow at t=0.49s



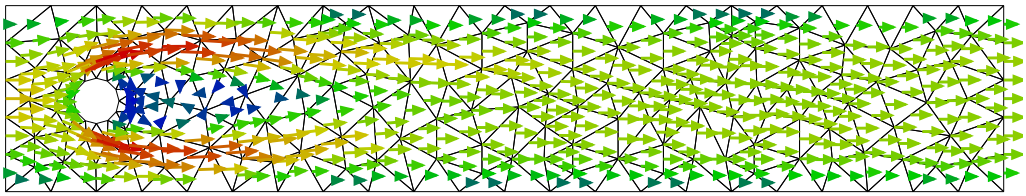
(f) Turbulent flow at t=0.59s



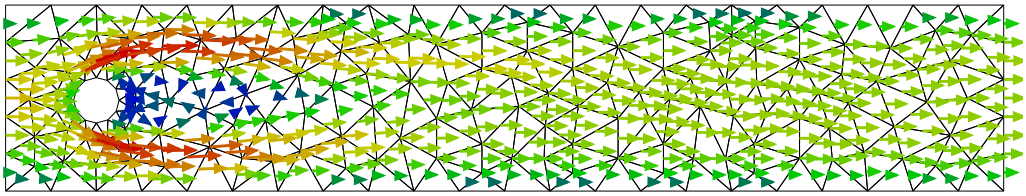
(g) Turbulent flow at t=0.69s



(h) Turbulent flow at t=0.79s



(i) Turbulent flow at t=0.89s



(j) Turbulent flow at t=0.99s

Figure 3.8: Solution for turbulent flow via Eqs. (3.3) to (3.4) on GPU with $\Delta t = 0.01$.

3.4 Comparison between GPU and CPU

From the statistical data we can see that the GPU does better than the CPU. This, however, does not mean that CPUs are outdated. In general GPUs have less memory than a CPU and there is no way to add additional memory to a GPU. It is, also, hard to compare such vastly different hardware architectures. There are a lot of factors leading to the presented result. The GPU used in the numerical experiment is a rather high-end GPU, while the CPUs are mid/low end CPUs. More tests with different hardware, different problems and different meshes are needed in order to give a conclusive answer on which is better. This work, however, shows that there is potential in implementing GPU based FEM fluid solvers. As it was shown the GPU performs well when there are a lot of simple, independent computations. There are some opened questions however:

- How suitable is the GPU for performing space discretization?
- How would the GPU perform in building the FEM matrices?
- How would the GPU perform in computing the IC0 preconditioner?
- How would the GPU perform in applying the IC0 preconditioner?
- How would the GPU perform in building the KD tree?

Most likely the GPU would not perform well in all of the above, thus a combination of CPU and GPU should be used.

3.5 Further development

Solving numerically the Navier-Stokes equations is a vast topic. We have not paid much attention to the type of the element we used. The P_2P_1 Taylor-Hood element is the simplest element which could be applied to this particular fractional time splitting algorithm. The semi-Lagrangian advection will fit nicely with quadrilateral elements in a structured grid. The accuracy and performance could be compared between the P_2P_1, Q_2P^{-1} isoparametric element. It would be also interesting to compare the accuracy and performance when a divergence free element is used. The algorithm has to be tested in 3D, a good starting point would be the Q_1Q_0 element over a structured grid,

which is very similar to the approach presented in [5]. The semi-Lagrangian method in 3D with unstructured meshes with tetrahedral elements, could become expensive due to the need to check whether a point lies in a tetrahedral. It should be evaluated if the algorithm is appropriate for such meshes.

The semi-Lagrangian method for unstructured meshes relies heavily on the efficient implementation of the KD Tree. We believe that the most important thing is to build a balanced KD Tree efficiently. The crucial part in building a balanced KD Tree is selecting the so-called splitting point. We have mentioned three possible ways to select the splitting point: using the PICK algorithm, Quickselect algorithm and using the approach presented in [6]. The latter seems to be the most promising algorithm.

The implementation of the semi-Lagrangian advection allows the advection quantities to travel only in straight lines, a multi staged semi-Lagrangian method could be implemented. This will allow the imaginary particles to have curved trajectories, which is more realistic. A good start would be [26].

It was noted that multithreading the assembling of the matrices in a sparse (CSR) format is not trivial and relies on graph coloring. It is worth trying to implement it, however. It has two advantages. First it could speed up the assembly phase. Second when such an algorithm is implemented the advection phase could be solved using the FEM and compared to the semi-Lagrangian method.

The performance of the CPU implementation of the CG and PCG methods will be improved if SIMD instructions are used. The computation and application of the IC0 preconditioner must be multithreaded. A good starting point for a GPU implementation would be [20]. Multithreaded Block Jacobi preconditioner should be implemented and compared to the multithreaded IC0 preconditioner.

Different algorithms, e.g. Runge-Kutta, can be used to improve the accuracy of the approximation of the time derivatives. However, it can be shown [5] that the splitting technique which was used provides $O(\Delta t)$ accuracy. This means a higher order splitting technique is required. For example the Strang splitting could be used.

Conclusion

Over the course of this work 3 different algorithms for solving the Navier-Stokes equations. One of them matched our goals (performance, scalability and accuracy) and was further developed both on CPU and GPU. The algorithm splits the differential operator into three parts: advection, diffusion and pressure projection. Semi-Lagrangian method accelerated by KD Tree data structure was presented for the advection part and it was shown that it scales well both on CPU and GPU. Both the diffusion and the pressure projection parts boil down to solving linear systems with SPD matrices. The CG method was used for solving the systems. For the GPU implementation two implementations were presented: a multi kernel and a mega kernel. It was shown that the mega kernel is about four times faster than the multi kernel implementation. Although the implementation of the mega kernel CG was trivial, we could not find another resource which presents it, nor a resource which provides comparison between the two approaches. The PCG method, using IC0 preconditioner, was also implemented and it was shown that it reduces the number of iterations about two times. Our implementation of the PCG method did not use multiple threads applying the preconditioner (line 3 of Algorithm 4) and thus performed worse than the CG method, despite the smaller number of iterations.

Appendices

Appendix A

Detailed stastical data of execution runtime

Here we shall present extended stastical data gatherd over several runs of the CPU version of the application.

A.1 Laminar Flow

A.1.1 Computer 1

	Computer 1						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	44.66s	77.43s	288.81s	24.98s	391.23s	11.13s	446.64s
SD	2.02s	3.62s	13.24s	1.03s	16.64s	0.29s	3.09s
Median	44.61s	77.13s	287.11s	25.10s	392.42s	11.05s	446.73s
Min	41.04s	70.91s	264.02s	22.98s	363.59s	10.90s	441.40s
Max	47.86s	84.15s	310.23s	26.83s	421.21s	11.86s	451.66s

Table A.1: Execution time on 2 threads of Computer 1 for solving for laminar flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and Conjugate Gradient method used for solving the linear system.

	Computer 1						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	31.49s	59.28s	210.51s	18.96s	288.75s	10.14s	333.54s
SD	0.90s	2.00s	7.57s	0.87s	8.75s	0.48s	9.79s
Median	31.68s	58.60s	211.02s	18.69s	289.06s	9.82s	333.92s
Min	28.85s	57.04s	191.83s	18.22s	270.39s	9.78s	312.16s
Max	32.02s	62.27s	224.64s	21.04s	307.48s	10.85s	353.45s

Table A.2: Execution time on 3 threads of Computer 1 for solving for laminar flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and Conjugate Gradient method used for solving the linear system.

	Computer 1						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	24.67s	49.45s	166.03s	15.90s	231.37s	5.68s	264.29s
SD	0.33s	1.91s	2.16s	0.50s	3.22s	0.26s	3.62s
Median	24.69s	48.69s	165.94s	15.74s	231.01s	5.50s	263.91s
Min	24.08s	48.03s	161.95s	15.53s	225.51s	5.48s	257.61s
Max	25.29s	53.87s	170.60s	17.35s	236.19s	6.01s	269.73s

Table A.3: Execution time on 4 threads of Computer 1 for solving for laminar flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and Conjugate Gradient method used for solving the linear system.

	Computer 1						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	21.93s	49.38s	158.97s	15.99s	224.33s	5.57s	254.39s
SD	0.23s	1.79s	2.06s	0.66s	3.41s	0.16s	3.60s
Median	21.83s	48.58s	158.33s	15.60s	222.56s	5.51s	252.39s
Min	21.74s	48.28s	157.28s	15.49s	221.14s	5.50s	250.93s
Max	22.38s	53.28s	163.74s	17.25s	230.87s	6.01s	261.32s

Table A.4: Execution time on 5 threads of Computer 1 for solving for laminar flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and Conjugate Gradient method used for solving the linear system.

	Computer 1						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	19.74s	48.47s	150.29s	15.63s	214.38s	5.72s	242.39s
SD	0.06s	1.09s	0.63s	0.44s	1.75s	0.26s	1.71s
Median	19.73s	47.93s	150.18s	15.44s	213.61s	5.57s	241.87s
Min	19.66s	47.64s	149.51s	15.37s	212.68s	5.50s	240.57s
Max	19.85s	51.05s	151.62s	16.81s	218.63s	6.11s	246.47s

Table A.5: Execution time on 6 threads of Computer 1 for solving for laminar flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and Conjugate Gradient method used for solving the linear system.

	Computer 1						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	18.63s	49.11s	147.45s	16.08s	212.64s	5.88s	239.75s
SD	0.24s	1.41s	2.40s	0.76s	3.57s	0.37s	3.89s
Median	18.57s	48.08s	146.54s	15.80s	212.69s	5.72s	240.11s
Min	18.27s	47.66s	144.55s	15.49s	207.75s	5.49s	234.01s
Max	19.09s	50.88s	152.90s	18.14s	219.34s	6.36s	246.59s

Table A.6: Execution time on 7 threads of Computer 1 for solving for laminar flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and Conjugate Gradient method used for solving the linear system.

	Computer 1						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	16.75s	50.02s	141.35s	15.98s	207.35s	5.82s	232.47s
SD	0.58s	2.63s	5.07s	0.53s	7.83s	0.22s	8.52s
Median	16.65s	49.34s	140.46s	15.94s	204.69s	5.77s	229.19s
Min	15.99s	47.90s	137.22s	15.37s	200.73s	5.50s	225.19s
Max	18.14s	56.16s	153.07s	17.22s	223.13s	6.27s	249.47s

Table A.7: Execution time on 8 threads of Computer 1 for solving for laminar flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and Conjugate Gradient method used for solving the linear system.

A.1.2 Computer 2

	Computer 2						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	30.29s	51.86s	204.38s	15.66s	271.89s	6.32s	310.44s
SD	0.03s	0.06s	1.52s	0.02s	1.50s	0.02s	1.53s
Median	30.29s	51.85s	203.95s	15.65s	271.46s	6.32s	309.98s
Min	30.25s	51.72s	203.03s	15.63s	270.46s	6.28s	309.00s
Max	30.36s	51.96s	208.08s	15.69s	275.46s	6.35s	314.09s

Table A.8: Execution time on 2 threads of Computer 2 for solving for laminar flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and Conjugate Gradient method used for solving the linear system.

	Computer 2						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	20.39s	36.94s	142.23s	11.25s	190.42s	5.90s	218.22s
SD	0.05s	0.08s	0.37s	0.02s	0.45s	0.04s	0.47s
Median	20.38s	36.93s	142.14s	11.25s	190.42s	5.90s	218.18s
Min	20.34s	36.85s	141.68s	11.21s	189.77s	5.83s	217.58s
Max	20.51s	37.11s	143.11s	11.29s	191.51s	5.97s	219.40s

Table A.9: Execution time on 3 threads of Computer 2 for solving for laminar flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and Conjugate Gradient method used for solving the linear system.

	Computer 2						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	15.61s	30.54s	113.10s	9.45s	153.09s	3.70s	173.75s
SD	0.02s	0.06s	0.97s	0.06s	0.95s	0.03s	0.96s
Median	15.61s	30.53s	112.82s	9.43s	152.78s	3.71s	173.43s
Min	15.59s	30.48s	112.27s	9.38s	152.19s	3.64s	172.80s
Max	15.66s	30.66s	115.30s	9.62s	155.16s	3.74s	175.82s

Table A.10: Execution time on 4 threads of Computer 2 for solving for laminar flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and Conjugate Gradient method used for solving the linear system.

	Computer 2						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	13.11s	28.36s	99.33s	8.83s	136.51s	3.70s	154.62s
SD	0.01s	0.20s	0.41s	0.05s	0.48s	0.02s	0.48s
Median	13.11s	28.30s	99.46s	8.82s	136.50s	3.70s	154.65s
Min	13.09s	28.15s	98.70s	8.77s	135.79s	3.68s	153.87s
Max	13.13s	28.92s	100.04s	8.97s	137.26s	3.74s	155.38s

Table A.11: Execution time on 5 threads of Computer 2 for solving for laminar flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and Conjugate Gradient method used for solving the linear system.

	Computer 2						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	11.42s	27.24s	88.60s	8.54s	124.38s	3.70s	140.78s
SD	0.02s	0.06s	0.76s	0.02s	0.77s	0.01s	0.78s
Median	11.42s	27.25s	88.54s	8.54s	124.35s	3.69s	140.76s
Min	11.39s	27.14s	88.00s	8.49s	123.78s	3.68s	140.19s
Max	11.46s	27.34s	90.67s	8.56s	126.48s	3.72s	142.94s

Table A.12: Execution time on 6 threads of Computer 2 for solving for laminar flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and Conjugate Gradient method used for solving the linear system.

	Computer 2						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	10.21s	26.71s	81.32s	8.42s	116.45s	3.71s	131.65s
SD	0.01s	0.16s	0.40s	0.07s	0.48s	0.03s	0.48s
Median	10.21s	26.67s	81.37s	8.38s	116.56s	3.71s	131.74s
Min	10.20s	26.51s	80.58s	8.33s	115.56s	3.67s	130.79s
Max	10.22s	27.14s	81.92s	8.52s	117.06s	3.78s	132.26s

Table A.13: Execution time on 7 threads of Computer 2 for solving for laminar flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and Conjugate Gradient method used for solving the linear system.

	Computer 2						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	9.21s	26.31s	75.57s	8.29s	110.17s	3.72s	124.40s
SD	0.01s	0.21s	0.42s	0.03s	0.43s	0.02s	0.41s
Median	9.21s	26.26s	75.56s	8.27s	110.16s	3.72s	124.38s
Min	9.19s	26.04s	74.91s	8.24s	109.43s	3.70s	123.68s
Max	9.23s	26.70s	76.31s	8.33s	110.83s	3.78s	125.09s

Table A.14: Execution time on 8 threads of Computer 2 for solving for laminar flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and Conjugate Gradient method used for solving the linear system.

A.2 Turbulent Flow

A.2.1 Computer 1

	Computer 1						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	46.53s	112.17s	352.72s	31.49s	496.37s	11.40s	557.75s
SD	1.07s	3.76s	8.92s	1.12s	11.90s	0.58s	13.20s
Median	46.36s	112.05s	355.49s	31.12s	497.35s	11.07s	558.70s
Min	45.11s	106.98s	338.90s	30.05s	478.38s	10.93s	538.00s
Max	48.16s	120.76s	366.01s	32.95s	519.71s	12.59s	582.47s

Table A.15: Execution time on 2 threads of Computer 1 for solving for turbulent flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and Conjugate Gradient method used for solving the linear system.

	Computer 1						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	32.29s	83.98s	250.74s	23.24s	357.96s	10.08s	403.49s
SD	0.07s	2.70s	0.88s	0.78s	3.48s	0.44s	3.55s
Median	32.27s	82.39s	250.97s	22.83s	356.22s	9.81s	402.37s
Min	32.20s	81.76s	249.48s	22.64s	354.02s	9.77s	399.16s
Max	32.47s	88.78s	252.46s	24.82s	364.39s	10.86s	409.77s

Table A.16: Execution time on 3 threads of Computer 1 for solving for turbulent flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and Conjugate Gradient method used for solving the linear system.

	Computer 1						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	24.43s	69.91s	193.16s	19.14s	282.20s	5.64s	314.81s
SD	0.22s	2.98s	4.01s	0.13s	6.18s	0.24s	6.50s
Median	24.38s	68.61s	191.43s	19.11s	279.26s	5.51s	312.16s
Min	24.24s	68.36s	190.55s	19.04s	277.98s	5.48s	310.21s
Max	25.05s	76.42s	202.63s	19.49s	298.54s	6.02s	332.32s

Table A.17: Execution time on 4 threads of Computer 1 for solving for turbulent flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and Conjugate Gradient method used for solving the linear system.

	Computer 1						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	22.87s	71.61s	197.13s	19.91s	288.64s	5.93s	320.06s
SD	0.53s	3.41s	1.88s	0.74s	4.26s	0.21s	4.77s
Median	22.63s	70.52s	197.44s	19.74s	289.31s	6.02s	320.74s
Min	22.18s	68.10s	192.96s	19.20s	282.66s	5.51s	313.77s
Max	23.57s	77.05s	199.30s	21.33s	296.29s	6.04s	328.55s

Table A.18: Execution time on 5 threads of Computer 1 for solving for turbulent flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and Conjugate Gradient method used for solving the linear system.

	Computer 1						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	20.38s	69.58s	189.29s	19.53s	278.40s	5.69s	307.02s
SD	0.14s	2.06s	1.13s	0.43s	2.34s	0.25s	2.33s
Median	20.39s	68.71s	189.20s	19.34s	278.29s	5.58s	306.79s
Min	20.18s	67.54s	187.79s	19.03s	275.14s	5.50s	303.68s
Max	20.61s	72.81s	191.35s	20.38s	282.15s	6.09s	311.11s

Table A.19: Execution time on 6 threads of Computer 1 for solving for turbulent flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and Conjugate Gradient method used for solving the linear system.

	Computer 1						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	18.93s	68.89s	181.35s	19.58s	269.82s	5.66s	296.95s
SD	0.16s	1.71s	2.57s	0.50s	2.90s	0.10s	2.88s
Median	18.99s	68.01s	181.21s	19.34s	269.61s	5.71s	296.51s
Min	18.59s	67.71s	176.57s	19.08s	265.77s	5.50s	293.02s
Max	19.13s	71.94s	185.47s	20.54s	275.75s	5.77s	303.12s

Table A.20: Execution time on 7 threads of Computer 1 for solving for turbulent flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and Conjugate Gradient method used for solving the linear system.

	Computer 1						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	17.02s	71.66s	168.72s	20.07s	260.45s	5.90s	285.95s
SD	0.35s	3.13s	5.33s	0.79s	8.97s	0.16s	9.45s
Median	17.05s	72.12s	168.73s	20.17s	260.42s	5.89s	285.96s
Min	16.34s	66.87s	159.27s	18.83s	244.96s	5.70s	269.42s
Max	17.58s	77.57s	176.36s	21.31s	272.75s	6.19s	299.16s

Table A.21: Execution time on 8 threads of Computer 1 for solving for turbulent flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and Conjugate Gradient method used for solving the linear system.

A.2.2 Computer 2

	Computer 2						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	30.73s	73.70s	276.62s	19.47s	369.79s	6.34s	408.82s
SD	0.12s	0.35s	1.67s	0.12s	1.96s	0.03s	2.08s
Median	30.75s	73.79s	276.37s	19.44s	369.23s	6.34s	408.23s
Min	30.56s	73.30s	274.41s	19.31s	367.16s	6.31s	406.14s
Max	30.96s	74.44s	278.94s	19.69s	372.95s	6.39s	412.20s

Table A.22: Execution time on 2 threads of Computer 2 for solving for turbulent flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and Conjugate Gradient method used for solving the linear system.

	Computer 2						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	20.67s	52.46s	192.93s	13.89s	259.28s	5.92s	287.39s
SD	0.02s	0.37s	0.78s	0.01s	0.88s	0.03s	0.89s
Median	20.67s	52.37s	192.61s	13.89s	258.81s	5.93s	286.93s
Min	20.64s	52.27s	192.01s	13.86s	258.23s	5.88s	286.33s
Max	20.72s	53.57s	194.29s	13.91s	260.58s	5.96s	288.69s

Table A.23: Execution time on 3 threads of Computer 2 for solving for turbulent flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and Conjugate Gradient method used for solving the linear system.

	Computer 2						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	15.87s	43.35s	153.28s	11.63s	208.26s	3.76s	229.24s
SD	0.06s	0.19s	0.88s	0.08s	1.00s	0.11s	1.10s
Median	15.85s	43.29s	153.37s	11.60s	208.17s	3.73s	229.16s
Min	15.81s	43.21s	151.94s	11.58s	206.89s	3.70s	227.84s
Max	16.04s	43.91s	154.73s	11.84s	210.11s	4.10s	231.23s

Table A.24: Execution time on 4 threads of Computer 2 for solving for turbulent flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and Conjugate Gradient method used for solving the linear system.

	Computer 2						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	13.32s	40.07s	135.31s	10.89s	186.26s	3.72s	204.60s
SD	0.12s	0.46s	1.74s	0.15s	2.10s	0.05s	2.19s
Median	13.29s	39.96s	134.45s	10.84s	185.37s	3.71s	203.67s
Min	13.25s	39.73s	133.63s	10.82s	184.23s	3.65s	202.53s
Max	13.65s	41.40s	138.59s	11.33s	190.66s	3.82s	209.42s

Table A.25: Execution time on 5 threads of Computer 2 for solving for turbulent flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and Conjugate Gradient method used for solving the linear system.

	Computer 2						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	11.51s	38.57s	119.61s	10.48s	168.66s	3.71s	185.17s
SD	0.02s	0.30s	1.05s	0.05s	1.13s	0.03s	1.12s
Median	11.51s	38.47s	119.46s	10.47s	168.42s	3.70s	184.93s
Min	11.49s	38.29s	118.50s	10.42s	167.30s	3.69s	183.83s
Max	11.54s	39.24s	122.38s	10.62s	171.39s	3.78s	187.87s

Table A.26: Execution time on 6 threads of Computer 2 for solving for turbulent flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and Conjugate Gradient method used for solving the linear system.

	Computer 2						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	10.30s	37.63s	110.25s	10.31s	158.19s	3.70s	173.48s
SD	0.01s	0.10s	0.50s	0.06s	0.54s	0.02s	0.53s
Median	10.29s	37.60s	110.12s	10.31s	158.10s	3.70s	173.38s
Min	10.28s	37.50s	109.50s	10.22s	157.37s	3.66s	172.70s
Max	10.31s	37.77s	111.18s	10.48s	159.10s	3.73s	174.36s

Table A.27: Execution time on 7 threads of Computer 2 for solving for turbulent flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and Conjugate Gradient method used for solving the linear system.

	Computer 2						
	Step 1	Step 2	Step 3	Step 4	CG (Total)	Matrix Assembling	Total Time
Mean Time	9.28s	37.02s	102.41s	10.18s	149.61s	3.69s	163.88s
SD	0.02s	0.14s	1.05s	0.07s	1.08s	0.02s	1.07s
Median	9.27s	37.04s	102.13s	10.17s	149.36s	3.69s	163.65s
Min	9.25s	36.75s	101.66s	10.09s	148.76s	3.67s	163.02s
Max	9.32s	37.18s	105.44s	10.36s	152.68s	3.73s	166.93s

Table A.28: Execution time on 8 threads of Computer 2 for solving for turbulent flow via Eqs. (3.3) to (3.6) over the given mesh with $\Delta t = 0.01$ and Conjugate Gradient method used for solving the linear system.

Appendix B

Implementation of CUDA grid sync

```
1  void __device__ syncGrid(unsigned int* barrier, unsigned
       int* generation) {
     if(threadIdx.x == 0) {
       volatile const unsigned int myGeneration = *generation;
       const unsigned int oldCount = atomicInc(barrier,
          gridDim.x - 1);
5      if(oldCount == gridDim.x - 1) {
         atomicAdd(generation, 1);
       }
       while(atomicCAS(generation, myGeneration, myGeneration)
           == myGeneration);
     }
10   __syncthreads();
   }
```

Listing B.1: Custom CUDA grid synchronization

The parameter barrier is a handle to the synchronization primitive, it must reside in the global memory. There can be multiple barriers. Parameter generation is used in order to distinguish threads which are waiting on the barrier and others which have left it and entered it again. It is safe if generation wraps around to 0. The *atomicCAS* operation is needed in order to prevent the compiler from optimizing the while loop away.

Bibliography

[1] M.H Aissa. "GPU-accelerated CFD Simulations for Turbomachinery Design Optimization". PhD thesis. Delft University of Technology, 2017. ISBN: 978-2-87516-123-9. DOI: https://doi.org/10.4233/uuid:1fcc6ab4-daf5-416d-819a-2a7b0594c369.

[2] Manuel Blum et al. "Time bounds for selection". In: *Journal of Computer and System Sciences* 7 (1973), pp. 448–461. DOI: 10.1016/S0022-0000(73)80033-9.

[3] Jeff Bolz, Ian Farmerand Eitan Grinspun, and Peter Schrőoder. "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrids". In: *SIGGRAPH '03 ACM SIGGRAPH 2003* (2003).

[4] Giuseppe Bonaccorso. *Mastering Machine Learning Algorithms Expert techniques for implementing popular machine learning algorithms, fine-tuning your models, and understanding how they work*. 2nd ed. Packt Publishing, 2020;2018. ISBN: 9781838821913; 1838821910.

[5] Robert Bridson. *Fluid Simulation for Computer Graphics*. A K Peters/CRC Press (September 18, 2008), 2016. ISBN: 1568813260.

[6] Russell A. Brown. "Building a Balanced k-d Tree in O(kn log n) Time". In: *ArXiv* abs/1410.5420 (2014).

[7] Cris Cecka, Adrian J. Lew, and E. Darve. "Assembly of finite element methods on graphics processors". In: *INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN ENGINEERING* (2010). DOI: 10.1002/nme.2989.

[8] A J Chorin. "THE NUMERICAL SOLUTION OF THE NAVIER–STOKES EQUATIONS FOR AN INCOMPRESSIBLE FLUID." In: *Mathematics of Computation* 22 (1968), pp. 745–762. URL: https://www.osti.gov/biblio/4568862.

[9] Thomas H. Cormen et al. *Introduction to Algorithms*. 3rd ed. MIT Press, 2009. ISBN: 0262033844; 9780262033848.

[10] J.P. D'Amato and M. Vénere. "A CPU–GPU framework for optimizing the quality of large meshes". In: *Journal of Parallel and Distributed Computing* 73 (2013), pp. 1127–1134.

[11] Dale R. Durran. *Numerical Methods for Wave Equations in Geophysical Fluid Dynamics*. Texts in Applied Mathematics. Springer New York, 1999. ISBN: 9781441931214; 144193121X; 9781475730814; 1475730810.

[12] J. Eddie Welch Francis H. Harlow. "Numerical calculation of time-dependent viscous incompressible flow of fluid with a free surface". In: *The Physics of Fluids* 8 (1965).

[13] *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. ISBN: 0321228324.

[14] Suito Hiroshi Huynh Quang Huy Viet. "A GPU Parallel Solver for 3D Incompressible Navier–Stokes Equations Discretized by Stabilized Finite Element". In: Computational Fluid Dynamics Symposium.

[15] *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 2021. URL: `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html`.

[16] Mike Giles István Reguly. "Efficient sparse matrix-vector multiplication on cache-based GPUs". In: *Innovative Parallel Computing (InPar)* (2012). DOI: `10.1109/InPar.2012.6339602`.

[17] Edward Kandrot Jason Sanders. *CUDA by example : an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010. ISBN: 0132180138.

[18] Fredrik Bengzon Mats G. Larson. *The Finite Element Method: Theory, Implementation, and Applications (Texts in Computational Science and Engineering, 10)*. Springer, 2013. ISBN: 3642332862.

[19] Greg Humphreys Matt Pharr Wenzel Jakob. *Physically Based Rendering. From Theory to Implementation*. 3rd ed. Morgan Kaufmann series in computer graphics and geometric modeling. Morgan Kaufmann, 2002. ISBN: 0128006455; 9780128006450; 9780080512846; 0080512844; 9780128007099; 0128007095; 9780585453767; 0585453764; 9781558607873; 1558607870.

[20] Maxim Naumov. "Parallel Incomplete-LU and Cholesky Factorization in the Preconditioned Iterative Methods on the GPU". In: (). URL: https://research.nvidia.com/sites/default/files/pubs/2012-05_Incomplete-LU-and-Cholesky/nvr-2012-003.pdf.

[21] G. Ortega et al. "The BiConjugate gradient method on GPUs". In: *The Journal of Supercomputing* 64 (2013), pp. 49–58. DOI: 10.1007/s11227-012-0761-2.

[22] R. L. Sani P. M. Gresho. *Incompressible Flow and the Finite Element Method*. Vol. 1. Wiley, 1999. ISBN: 0471967890.

[23] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2003. ISBN: 0898715342.

[24] Carlos Eduardo Scheidegger, Joao Luiz Dihl Comba, and Rudnei Dias da Cunha. "Navier-Stokes on Programmable Graphics Hardware Using SMAC". In: *In Proceedings of XVII SIBGRAPI – II SIACG*. IEEE Press, 2004, pp. 300–307.

[25] S. Tureka. "Benchmark Computations of Laminar Flow Around a Cylinder". In: vol. 48. 1996.

[26] Dongbin Xiu and George Karniadakis. "A Semi-Lagrangian High-Order Method for Navier–Stokes Equations". In: *Journal of Computational Physics* 172 (Sept. 2001), pp. 658–684. DOI: 10.1006/jcph.2001.6847.