seit 1558

Parallel Numerical Simulation of Navier-Stokes-Equations on GPUs

DIPLOMARBEIT

zur Erlangung des akademischen Grades
Diplom-Mathematiker

FRIEDRICH-SCHILLER-UNIVERSITÄT JENA
Fakultät für Mathematik und Informatik

eingereicht von Marcus Fritzsche
geb. am 19.12.1982 in Werdau

Betreuer: Prof. Dr. G. Zumbusch

Jena, 09.04.2009

Zusammenfassung

Die vorliegende Diplomarbeit behandelt die numerische Lösung der Navier-Stokes Gleichungen mit dem Schwerpunkt Parallelisierung auf Grafikkarten.
Jeder Abschnitt der Arbeit kann weitgehend isoliert behandelt werden, Ausnahmen sind im Folgenden ausdrücklich erwähnt.
Im Abschnitt Introduction wird das Studium der Navier-Stokes Gleichungen motiviert und deren Bezug zu Parallelrechnern diskutiert. Derivation of Navier-Stokes equations zeigt die Grundzüge der Kontinuum-Theorie und ihre Verbindung mit der Hydrodynamik von einem physikalischen Standpunkt und liefert die Navier-Stokes Gleichungen für inkompressible Fluide. Anschließend wird in Numerical Approach ein numerischer Algorithmus hergeleitet, der auf diesen Gleichungen basiert. Dieser Algorithmus wird in CPU Implementation verwendet um Fluide auf einem einzelnen Computer-Prozessor zu simulieren. Dieser Abschnitt beschreibt hauptsächlich die Umwandlung des Algorithmus aus dem vorigen Abschnitt in Programmcode und dessen Ausführungsergebnisse.[1] Daraufhin beschreibt Domain Decomposition einen allgemeinen Weg um einen Fluid-Algorithmus zu erhalten, der auf Parallelrechnern mit beliebiger Hardware Architektur implementiert werden kann. Gpu Implementation beschränkt diese Architektur auf NVIDA-CUDA-fähige Grafikkarten und beschreibt (analoge Gliederung wie im Abschnitt für CPU's) die Implementierung auf Solchen in Bezug auf den im vorangegangenen Abschnitt entwickelten Algorithmus.[2] Schließlich werden in Conclusion die Ergebnisse zusammengefasst und Schlussfolgerungen aus der Diplomarbeit gezogen.[3]

[1] Abhängigkeitsgrad von vorangegangenen Kapiteln ist hier sehr hoch.
[2] Abschnitt bezieht sich wieder auf vorangegangenen.
[3] Hängt von allen Abschnitten ab.

Acknowledgments

I would like to thank my diploma-thesis supervisor Prof. Dr. Gerhard Zumbusch, professorship for Scientific Computing . He recognized problems in its early stages and provides me with a red thread in order to straightforward development of the work.
I would also express thank to Dipl.-Phys. Frank Peuker. He provides me at any time with his well-fundamental knowledge-base about numerical computing and gives great advise in optimizing source code. It was often useful to speak with him about a problem and the problem gets solved of its own.
Finally, I am grateful for many helpful comments and suggestions of my friends.

Notations

$\mathbf{a}$	Boldface letter represents a column vector $\mathbf{a}$
$\mathbf{a} \cdot \mathbf{b}$	Dot product of vector $\mathbf{a}$ and $\mathbf{b}$
$\frac{\partial}{\partial x}$	Partial differential operator with respect to x
∂_x	Shorthand for $\frac{\partial}{\partial x}$
f_x	Shorthand for $\frac{\partial f}{\partial x}$
∇	Del vector differential operator (Nabla); in case $\mathbb{R}^3$: $\nabla := \mathbf{i}\frac{\partial}{\partial x} + \mathbf{j}\frac{\partial}{\partial y} + \mathbf{j}\frac{\partial}{\partial z}$ with $\{\mathbf{i}, \mathbf{j}, \mathbf{k}\}$ standard basis in $\mathbb{R}^3$
Δ	Laplacian differential operator, $\Delta f := \nabla \cdot (\nabla f)$
$\mathbf{n}$	Symbol of the normal vector

Contents

1 Introduction

This diploma thesis deals with **c**omputational **f**luid **d**ynamics (CFD) which is an intersection of applied physics, mathematics and computer science. Physics is the science of exploring and describing principles of nature by observation[4]. The role of mathematics in CFD is obvious on the one hand since it is used to express observed physical laws and on the other hand given by needs of an algorithm which is reproducing these laws as accurate as possible. This chain is called computational simulation: producing data whose observation would lead to the same model as observation from nature. The production of such data is the task of an algorithm whose output should be nearly equivalent to a series of measurements underlying the original model to simulate reality most accurately. Ensuring this similarity is done by choosing appropriate numerical methods. Therefore one can say computer science meets physics linked by mathematics in simulations.
The mathematical model of fluid motion can be derived in two alternative ways. First is obtained from the kinetic theory which treats the fluid as consisting of molecules whose motion is governed by the laws of dynamics. With this theory it is attempted to derive macroscopic behavior from laws of mechanics and probability theory.
To interpret the fluid as a continuum which means to omit discrete particles leads to the second approach. This is subject of this diploma thesis.
Both theories lead to the Navier-Stokes equations which is a partial differential equation and describes the motion of fluids.
The continuum method holds as long as the microscopic scale is negligible compared to the smallest physical length scale of the flow field. Therefore kinetic theory on the other hand is at least useful if mean free path cannot be ignored like in rarefied gases.
Before going further, a short excursion to computational computer science is done in order to understand the implementation differences of these two theories. Computing power[5] continuously arises and is still following Moore's law. Currently, reaching this goal is done by adding computing cores to processors instead of arising the clock frequency. This rings in a new age of computational computer science because the programming paradigm has to change in order to use computing power of massively parallel machines more effectively[6]. Changing the programming paradigm implies the necessity of introducing new industry standards for parallel programming purposes which are suitable for different hardware architectures. This has to be done because future computing converges from two directions: CPU cores per chip increase while GPUs become more versatile in general purpose computing. Such general purpose GPU (GPGPU) units are manufactured from NVIDIA and ATI, their software models are called CUDA (subject of this work) and Stream, respectively. Uniformed convergence is hopefully achieved by the OpenCL standard which is an open computing language framework for writing programs with focus on parallelizing that executes across heterogeneous platforms consisting of CPUs,

[4] In this case it is hydrodynamic and the derived theory is called fluid mechanics.
[5] In the sense of floating point operations per second (FLOPS).
[6] Clusters of single chip machines exist long ago but mainstream multicore chips are new.

GPUs and other processing units.
The kinetic theory approach seems to be ideal for parallelizing since each molecule can be mapped to a processing unit which calculates the motion of its own. This kind is called particle simulation and is an application of a cellular automaton, the lattice gas automaton. Its continuous counterpart is the lattice Boltzmann automaton which is an implementation of the lattice Boltzmann equation (a molecular dynamic model).
Continuum method on the other hand is parallelized in a less natural way, domain decomposition methods are applied to obtain areas which can be mapped to a single processing unit.
The following work describes the continuum method, provides a numerical method for solving Navier-Stokes equations, its CPU implementation and discusses its parallelizing on NVIDIA GPUs implemented by the CUDA technology.

2 Derivation of Navier-Stokes equations

This section reviews the essentials of the continuum theory and its link to hydrodynamics from a physical point of view. It results in the incompressible Navier-Stokes equations.

2.1 Continuum mechanics

The aim of this subsection is to express a physical fluid as a mathematical model. A physical fluid consists of a set of matter particles in an area at defined positions in space $\mathbb{R}^3$. Such an area can be infinite large or have different boundary conditions. Therefore it is useful to consider only a small volume inside the fluid to describe its dynamics. In other words: the volume in which fluid flows is controllable. The particles in such an controllable volume are thought to be infinitesimal volumetric elements and hence points which union is named control volume $\mathbf{\Omega}_t$. The index $t \in [0, \infty)$ is used to express the time dependence of the fluid area $\mathbf{\Omega}_t$ which never leaves the domain[7] $\mathbb{R}^n$.
To describe the movement in time of a single particle $\mathbf{c} \in \mathbf{\Omega}_0$ the function[8]

$$\mathbf{x}_c : [0, \infty) \to \mathbb{R}^n$$

$$t \mapsto \mathbf{x}_c(t)$$

is introduced. In other words $\mathbf{x}_c(t)$ describes the path of particle $\mathbf{c}$ in fluid. Especially particle position at beginning of the path is $\mathbf{x}_c(0) = \mathbf{c}$ and at time t it is somewhere in $\mathbf{\Omega}_t$.
The movement in time of all particles is described by the function Φ:

$$\Phi : \mathbf{\Omega}_0 \times [0, \infty) \to \mathbb{R}^n$$

$$(\mathbf{c}, t) \mapsto \Phi(\mathbf{c}, t).$$

Expressing Φ in words of $\mathbf{x}_c$ leads to the definition $\Phi(\mathbf{c}, t) := \mathbf{x}_c(t)$.
Especially again - fluid position at beginning is $\Phi(\mathbf{\Omega}_0, 0) = \mathbf{\Omega}_0$ and at time t it is $\Phi(\mathbf{\Omega}_0, t) = \mathbf{\Omega}_t$.
These definitions enable to express the speed of any fluid particle at fixed time. Let $\mathbf{x} \in \mathbf{\Omega}_t$ be the position of particle $\mathbf{c}$ at time t. Speed at this point $\mathbf{x}$ in fluid $\mathbf{x}$ is given through:

$$\mathbf{v}(\mathbf{x}, t) := \frac{\partial}{\partial t}\Phi(\mathbf{c}, t).$$

Before proceeding it is important to be aware of two kinds of reference systems.
The first is called Eulerian reference system. It is an absolute reference system hence properties are measured from a fixed point in space. For example the speed $\mathbf{v}(\mathbf{x}, t)$ of a particle at point $(\mathbf{x}, t)$ is measured in an Eulerian reference system.
The second reference system is the Lagrangian. It is a relative reference system

[7] In progress, general n-dimensional fluid dynamics is considered and the names of physical quantities defined in space $\mathbb{R}^3$ are used to represent more general quantities in space $\mathbb{R}^n$.

[8] Here the index $\mathbf{c}$ of $\mathbf{x}_c$ links to a fluid particle and is not the partial derivative with respect to some vector variable $\mathbf{c}$.

and properties are measured from a moving point in space. For example the rate of change of temperature along a path of a moving particle.
In conclusion, the aim of deduce the mathematical model of an physical fluid is to obtain a system of differential equations for the velocity of an arbitrary fluid with properly chosen boundary conditions.

2.2 Convection and Advection

Practically convection is a mechanism of heat transfer from one place to another. A physical motivated definition is transportation of a physical property $\mathbf{f}(\mathbf{x}, t)$ in time within fluid. The rate of change of this property bounded at a particle on path $\mathbf{x}(t)$ is called convective derivative or material derivative and is mathematically no more than the total derivative:[9]

$$\frac{D\mathbf{f}}{Dt} := \frac{\partial \mathbf{f}}{\partial t} + \frac{d\mathbf{x}}{dt} \cdot \nabla \mathbf{f}. \tag{1}$$

The term $d\mathbf{x}/dt$ equals exactly to the previous defined speed $\mathbf{v}$. Equation (1) links the Lagrangian rate of change in time of property $\mathbf{f}$ with the Eulerian and an additional term generally called convection. This time independent term $\mathbf{v} \cdot \nabla \mathbf{f}$ is called convection if $\mathbf{f}$ is a vector field and advection if $f := \mathbf{f}$ is a scalar field. It is interesting to note that no rate of change in time of the property $\mathbf{f}$ occurs in the convective term.
Now practically one can clearly speak of heat convection in a fluid if the convective term doesn't vanish. Therefore particles have to be in movement and the temperature needs a gradient different to zero.

2.3 Continuity equations

Physical Conservation Laws in fluid dynamics are used to express the conservation of a property in a control volume. Such properties are for example the mass density or the temperature whose conservation laws are known as conservation of mass respective conservation of thermal energy.
While conservation laws are expressed in integral form their stronger differential analogue are called continuity equations. These are differential equations of the general form

$$\frac{\partial f}{\partial t} + \nabla \cdot (f\mathbf{v}) = s \tag{2}$$

where $\mathbf{f}$ is again some property of the fluid, $\mathbf{v}$ is the speed describing the flux of f and s describes the generation (or removal) rate of f.
Navier-Stokes equations are a special case of such a general continuity equation, derived by developing conservation laws. Before proceeding it is necessary to talk about a tool which is used for derivation: it is called Reynolds transport theorem.

[9] The equation make use of the tensor derivative $\nabla \mathbf{f}$ because $\mathbf{f}$ may be a vector.

Reynolds transport theorem This theorem is used to express the rate of change in time of the amount of some property $\mathbf{f}$ in a time dependent control volume.[10]

$$\frac{d}{dt}\int_{\mathbf{\Omega}_t} f(\mathbf{x},t)d\mathbf{x} = \int_{\mathbf{\Omega}_t} \Big(\frac{\partial}{\partial t}f + \nabla\cdot(f\mathbf{v})\Big)(\mathbf{x},t)d\mathbf{x}. \tag{3}$$

Like equation (1), equation (3) gives a link between the Langrangian and Eulerian reference system of a control Volume.
The intuitive character of this equation unfolds by applying Divergence theorem:

$$\frac{d}{dt}\int_{\mathbf{\Omega}_t} f(\mathbf{x},t)d\mathbf{x} = \int_{\mathbf{\Omega}_t} \frac{\partial}{\partial t}f d\mathbf{x} + \int_{\partial\mathbf{\Omega}_t} f\mathbf{v}\cdot\mathbf{n}d\mathbf{S}.$$

Changing the amount of some property in a moving volume is the same as changing it in a fixed volume plus the amount of the property flowing with speed through the boundaries of this volume.

Conservation of mass A fundamental physical law is that mass of an isolated physical system does not change as the system evolves. Since mass of a domain is calculated by the integral of the density over this domain, the following equation holds for all $t \geq 0$:

$$\int_{\mathbf{\Omega}_0} \rho(\mathbf{x},t)d\mathbf{x} = \int_{\mathbf{\Omega}_t} \rho(\mathbf{x},t)d\mathbf{x}.$$

Therefore the rate of change in time of mass has to be zero. Applying transport theorem (3) leads to:

$$\frac{d}{dt}\int_{\mathbf{\Omega}_t} \rho(\mathbf{x},t)d\mathbf{x} = \int_{\mathbf{\Omega}_t} \Big(\frac{\partial}{\partial t}\rho + \nabla\cdot(\rho\mathbf{v})\Big)(\mathbf{x},t)d\mathbf{x} := 0 \quad \forall t \geq 0.$$

This is true for all domains $\mathbf{\Omega}_t$, therefore the rightmost integrand have to be null itself and leads to the equation

$$\frac{\partial}{\partial t}\rho + \nabla\cdot(\rho\mathbf{v}) = 0, \tag{4}$$

which is obviously a continuity equation in sense of the definition (2).
Important to note that (4) is the general continuity equation for compressible fluids whereas

$$\nabla\cdot\mathbf{v} = 0, \tag{5}$$

is the special case of incompressible fluids. This is because here the density depends neither on place nor time.

[10] The proof works as follows: Applying the rule of Integration by substitution, the integration domain is transformed with help of the mapping $\mathbf{\Omega}_0 \ni c \mapsto \mathbf{x}_c(t) \in \mathbf{\Omega}_t$ and by multiplying the integrand with the absolute Jacobian determinant. Now the Leibniz integral rule works since integration domain is no more time independent and yields a convective derivative term as integrand. Back substitution leads to Reynolds transport theorem.

Conservation of Momentum Conservation of momentum is also a fundamental physical law which states that total momentum of objects in an isolated physical system is constant in time. Considering two point masses and their momenta $\mathbf{m}_1(t), \mathbf{m}_2(t)$ this law reduces to $d/dt(\mathbf{m}_1 + \mathbf{m}_2) = 0$. Assuming that a force[11] $-\mathbf{F} := d/dt\mathbf{m}_2$ acting on the first point mass, one gets

$$\frac{d}{dt}\mathbf{m}_1 - \mathbf{F} = 0. \tag{6}$$

This is also known as a special case of Newtons second law: the sum of all forces of a mass point vanishes.
In classical mechanics momentum is a quantity related to an object. If this object is a point mass, momentum is calculated by the product of its velocity and mass. To compute the momentum of an object like a control volume in a fluid, one has to sum up its point masses multiplied by their velocity. Spoken in infinitesimal language, this statement produces the equation

$$\mathbf{m}(t) := \int_{\mathbf{\Omega}_t} (\rho\mathbf{v})(\mathbf{x}, t)d\mathbf{x}.$$

Analogue, point mass forces $\mathbf{F}$ are replaced by the integral over the body forces $\mathbf{b}(\mathbf{x}, t)$ of the control volume. Therefore equation (6) leads to vector equation

$$\frac{d}{dt}\int_{\mathbf{\Omega}_t} (\rho\mathbf{v})(\mathbf{x}, t)d\mathbf{x} - \int_{\mathbf{\Omega}_t} \mathbf{b}(\mathbf{x}, t)d\mathbf{x} = \mathbf{0}.$$

Applying Reynolds transport theorem component-wise to left-hand side and connecting the integrands results in[12]

$$\int_{\mathbf{\Omega}_t} \Big(\frac{\partial}{\partial t}(\rho v_i) + \nabla \cdot (\rho v_i \mathbf{v}) - \mathbf{b}_i\Big)(\mathbf{x}, t)d\mathbf{x} = \mathbf{0} \quad \forall\Big(i \in \{1, \ldots, n\} \wedge t \geq 0\Big)$$

which applies for n-dimensional fluids.
In this position of the derivation, the non-linearity enters. It can be considered as momentum-convection in the sense of the convection section above.
The last equation is true for all $t \geq 0$ and hence leads to vector continuity equation[13]

$$\frac{\partial}{\partial t}(\rho\mathbf{v}) + \nabla \cdot (\rho\mathbf{v}\mathbf{v}) = \mathbf{b}. \tag{7}$$

Merging In conclusion, in a n-dimensional fluid[14] conservation of mass holds if the continuity equation (4) for ρ (density) vanish and conservation of momentum holds if the system of the n continuity equations (7) for $\rho\mathbf{v}$ (momentum) equals with the body forces:

$$\begin{aligned} &\partial_t\rho + \nabla \cdot (\rho\mathbf{v}) = 0, \\ &\partial_t(\rho v_i) + \nabla \cdot (\rho v_i \mathbf{v}) = b_i \quad \forall i \in \{1, \ldots, n\}. \end{aligned} \tag{8}$$

[11] The minus sign of $-\mathbf{F}$ is used to avoid another minus sign in progress.
[12] Vector function v splits up into scalar function components $v_1, \ldots, v_n$.
[13] The term $\mathbf{vv}$ make use of the dyad product.
[14] Again, a physical quantity (mass) in space $\mathbb{R}^3$ is representative for the n-dimensional case.

This $n+1$ system is the most general form of the Navier-Stokes equations in continuity form, hence they are a special case of a system of very general continuity equations.
Left-hand term of equation (7) simplifies with help of product rule and rearranging:

$$\begin{aligned}\partial_t(\rho\mathbf{v}) + \nabla\cdot(\rho\mathbf{v}\mathbf{v}) &= \Big(\mathbf{v}\partial_t\rho + \rho\partial_t\mathbf{v}\Big) + \Big((\rho\mathbf{v})\nabla\cdot\mathbf{v} + \mathbf{v}\nabla\cdot(\rho\mathbf{v})\Big)\\ &= \mathbf{v}\Big(\underbrace{\partial_t\rho + \nabla\cdot(\rho\mathbf{v})}_{=0}\Big) + \rho\Big(\underbrace{\partial_t\mathbf{v} + \mathbf{v}\nabla\cdot\mathbf{v}}_{=\frac{D}{Dt}\mathbf{v}}\Big).\end{aligned}$$

The term over the first curly bracket vanishes because conservation of mass and the second term is no more than a convective derivative of velocity in vector form.
Surprisingly the system of Navier-Stokes continuity equations can be written as a system of n convective derivatives and the mass continuity equation:

$$\rho\frac{D}{Dt}v_i = b_i \quad \forall i \in \{1,\dots,n\} \quad \Longleftrightarrow \quad \rho\frac{D}{Dt}\mathbf{v} = \mathbf{b}, \tag{9}$$

$$\frac{\partial}{\partial t}\rho + \nabla\cdot(\rho\mathbf{v}) = 0. \tag{10}$$

The First equation suggests to be Newtons second law in terms of body forces instead point forces. Acceleration is modeled by the convective derivative of velocity, in other words: change of velocity along particle path.

2.4 Body forces

For modeling body forces in a fluid it is necessary to fix the dimension n to three because assumptions are made which originates from experimental physics.
The general body force $\mathbf{b}$ is broken up in two parts: volume forces $\mathbf{f}$ and surface forces σ. Surface forces of a small cube C can be derived via divergence theorem from volume forces:

$$\int_C \rho\frac{D}{Dt}\mathbf{v}d\mathbf{x} = \int_C (\underbrace{\mathbf{f} + \nabla\sigma}_{=:\mathbf{b}})d\mathbf{x} = \int_C \mathbf{f}d\mathbf{x} + \int_{\partial C}\sigma\mathbf{n}d\mathbf{S}$$

where σ is a second order stress tensor and hence represented as a 3×3 matrix

$$\sigma = \begin{pmatrix}\sigma_{11} & \tau_{12} & \tau_{13}\\ \tau_{21} & \sigma_{22} & \tau_{23}\\ \tau_{31} & \tau_{32} & \sigma_{33}\end{pmatrix}$$

with σ_{ii} normal stresses and τ_{ij} shear stress components with respect to direction i,j on the cube surface. Decomposing matrix σ in two terms of the sum[15]

$$\sigma = -\begin{pmatrix}p & 0 & 0\\ 0 & p & 0\\ 0 & 0 & p\end{pmatrix} + \begin{pmatrix}\sigma_{11}-p & \tau_{12} & \tau_{13}\\ \tau_{21} & \sigma_{22}-p & \tau_{23}\\ \tau_{31} & \tau_{32} & \sigma_{33}-p\end{pmatrix} = -p\mathbf{I} + \mathbb{T}$$

[15]Symbol $\mathbf{I}$ is the identity matrix.

is motivated by introducing pressure p because it is generally a variable of interest. The first summand tends to force body in changing volume and therefore models pressure. It is negative mean normal stress and given by $p := 1/3(\sigma_{11} + \sigma_{22} + \sigma_{33})$. The second summand tends to distort the body: $\mathbb{T}$ is called deviatoric stress Tensor which is a traceless matrix.
This leads to the differential equation for fluid modeling[16]

$$\rho \frac{D}{Dt}\mathbf{v} = -\nabla p + \nabla \mathbb{T} + \mathbf{f}.$$

In order to describe real fluids and test correctness of this equation additional hypothesis on form of $\mathbb{T}$ is needed to model different fluid families. For example if tensor $\mathbb{T}$ vanishes which means modeling non-viscous fluids[17]. The resulting formulas are called Eulerian equations, often applied to compressible inviscid fluids

$$\rho \frac{D}{Dt}\mathbf{v} = -\nabla p + \mathbf{f}.$$

The following parts of this work deal with viscid fluids where friction is payed attention and therefore $\mathbb{T} \neq 0$. Such fluid families are called Newtonian fluids.
Physical observation of such fluids obtains that stress τ is proportional to velocity gradient and applying stokes postulates leads to[18] [19]

$$\mathbb{T}_{ij} := \mu\Big(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i}\Big) + \delta_{ij}\lambda \nabla \cdot \mathbf{v}$$

with μ and λ viscosity coefficients. Substitution in the momentum continuity equation (7) while preserving vector form results in

$$\rho \frac{D}{Dt}\mathbf{v} = -\nabla p + (\mu + \lambda)\nabla^T(\nabla \cdot \mathbf{v}) + \mu \Delta \mathbf{v} + \mathbf{f}$$

where μ is called dynamical viscosity. Assuming incompressible flow $\rho(\mathbf{x}, t) := \rho := const$ and using (5) leads to

$$\rho \frac{D}{Dt}\mathbf{v} = -\nabla p + \mu \Delta \mathbf{v} + \mathbf{f}. \tag{11}$$

These are the equations for momentum conservation of the Navier-Stokes equations which are used in progress. These holds for incompressible flow of Newtonian fluids in $\mathbb{R}^3$.

2.5 Similarity of flows

The point of interest in this section is to compare a large scale problem with a small scale problem, in other words: what are the conditions to obtain similar flows with

[16] The pressure is determined up to an undefined additive constant because only the gradient of the pressure is of interest.
[17] These are fluids without friction.
[18] Symbol δ_{ij} is the Kronecker delta.
[19] Vector x splits up into scalar components $x_1, \ldots, x_n$.

different fluid geometries.
To achieve the comparison, dimensionless values are introduced by dividing dimensioned values with comparable dimensioned values. Such comparable dimensioned values for a given example of a flow are l_∞ which is the scale value of the flow, v_∞ is the basic velocity of the flow, p_∞ is the basic pressure of the flow and ρ_∞ is the basic density of the flow[20]:

$$\hat{\mathbf{x}} := \frac{\mathbf{x}}{l_\infty}, \quad \hat{t} := \frac{v_\infty t}{l_\infty}, \quad \hat{\mathbf{v}} := \frac{\mathbf{v}}{v_\infty}, \quad \hat{p} := \frac{p - p_\infty}{\rho_\infty v_\infty^2}.$$

Substitution into incompressible momentum equation (11) and dividing by ρ_∞ leads to[21]

$$\frac{D}{D\hat{t}}\hat{\mathbf{v}} = -\hat{\nabla}\hat{p} + \frac{\mu}{\rho_\infty v_\infty l_\infty}\hat{\Delta}\hat{\mathbf{v}} + \frac{l_\infty}{v_\infty^2 \rho_\infty}\mathbf{f}.$$

This equation is dimensionless with respect to their variables and their solution only depends on the two fractions. Therefore two flows are similar if these two values are equal in both flows. This motivates to introduce the dimensionless values[22]

$$Re := \frac{\rho_\infty v_\infty l_\infty}{\mu} \quad \text{and} \quad Fr := \frac{v_\infty}{\sqrt{l_\infty \|\mathbf{f}\|}}.$$

These numbers are called Reynolds number and Froude number, respectively. While Reynolds number specifies the ratio of inertial forces to viscous forces, Froude number specifies the ratio of inertial forces to gravitational forces.
Here it comes to a first practical operational area for the Navier-Stokes equations: with their help one can express criteria (namely the Reynolds and the Froude number) to build a model of a small scale flow which is similar to an original.
Another operational area which shares the same idea is the content of the following work: the simulation of a flow which is similar to a possibly unknown original flow.
The following equation system shows the variant of the Navier-Stokes equations which is used in progress:

$$\frac{D}{Dt}\mathbf{v} = -\nabla p + \frac{1}{Re}\Delta\mathbf{v} + \mathbf{g}, \tag{12}$$

$$\nabla \cdot \mathbf{v} = 0. \tag{13}$$

Gravity forces are substituted with

$$\mathbf{g} := \frac{1}{Fr^2 \rho_\infty}\frac{\mathbf{f}}{\|f\|}.$$

It deals with dimension $n := 2$ and works for incompressible Newtonian fluid families. In component-wise form vector $\mathbf{v}$ is split up into components u and v, $\mathbf{g}$ into[23] g_x and g_y and $\mathbf{x}$ into x and y. Thus the system simplifies to[24]:

$$\frac{\partial}{\partial t}u + \frac{\partial}{\partial x}(u^2) + \frac{\partial}{\partial y}(uv) - \frac{1}{Re}\Big(\frac{\partial^2}{\partial x^2}u + \frac{\partial^2}{\partial y^2}u\Big) + \frac{\partial}{\partial x}p = g_x, \tag{14}$$

[20] Pressure comparison differs because of the undefined additive pressure constant.
[21] The operators $\hat{\nabla}$ and $\hat{\Delta}$ work with respect to $\hat{\mathbf{x}}$.
[22] The Euclidean norm $\|\mathbf{f}\|$ is the length of the vector $\mathbf{f}$.
[23] Here the index of the variables g_x respectively g_y do not present a partial derivative.
[24] Simplifying make use of mass continuity equation to convert momentum continuity equations.

$$\frac{\partial}{\partial t}v + \frac{\partial}{\partial y}(v^2) + \frac{\partial}{\partial x}(uv) - \frac{1}{Re}\Big(\frac{\partial^2}{\partial x^2}v + \frac{\partial^2}{\partial y^2}v\Big) + \frac{\partial}{\partial y}p = g_y, \tag{15}$$

$$\frac{\partial}{\partial x}u + \frac{\partial}{\partial y}v = 0. \tag{16}$$

This is a system of three differential equations in the unknown variables u, v and p over a bounded domain of place and time: $\mathbf{\Omega} \times [0, T]$, $\mathbf{\Omega} \subset \mathbb{R}^2$ and $[0, T] \subset \mathbb{R}$. First and second equations are nonlinear in u and v and variable p is determined up to an additive constant as mentioned in subsection 2.4 Body forces.
These equations are in the following used to develop a numerical algorithm which gives an approximation of the exact solution.

3 Numerical Approach

This section provides a detailed derivation of the numerical scheme (presented as an algorithm) to approximate the solution of the Navier-Stokes equations.

3.1 Finite difference method

The typical goal of numerical mathematics is to approximate the solution of differential equations with methods which are controllable, by meaning to be aware of the error size with respect to the exact solution. The Finite difference method for example is a numerical method, where finite differences are used to approximate derivatives. In progress the notation for partial derivatives f_x is omitted in order to use sub-indexes either as discretization points or domain variables to distinguish vector components[25].

Taylor Theorem Finite differences can be derived from the Taylor theorem applied to a function of which the derivation is in demand. At First, finite differences are demonstrated in a one-dimensional domain which is given by $[0, x_{\text{end}}] \subset \mathbb{R}$. This domain is discretized by division in $i_{\max}$ parts of length h, bounded by the points

$$x_i := ih, \quad i \in \{0, \dots, i_{\max}\}.$$

A Function f over this domain is given by[26]

$$f : [0, x_{\text{end}}] \to \mathbb{R}, \quad f \in \mathcal{C}^m([0, x_{\text{end}}]).$$

The Taylor theorem gives a sequence of approximations for this function f at a special point. Because the error of one of these approximation is given by Lagrange or Cauchy form, it becomes clear that the remainder term of this sequence, chopped at some member m, can be written in Big-$\mathcal{O}$-Notation. The following equations evaluate f via Taylor at grid point x_{i+1} and x_{i-1}:

$$f(x_{i+1}) = \sum_{k=0}^{m} \frac{h^k}{k!} \frac{d^k}{dx^k} f(x_i) + \mathcal{O}(h^{m+1}) \tag{T1}$$

$$f(x_{i-1}) = \sum_{k=0}^{m} \frac{(-h)^k}{k!} \frac{d^k}{dx^k} f(x_i) + \mathcal{O}(h^{m+1}). \tag{T2}$$

Equation (T1) leads to forward difference quotient in point x_i

$$\frac{d}{dx} f(x_i) = \underbrace{\frac{f(x_{i+1}) - f(x_i)}{h}}_{=: \left[\frac{df}{dx}\right]_i^{\mathrm{f}}} + \mathcal{O}(h).$$

[25] This is the case for $g_x, g_y, n_x, n_y, h_x, h_y$, and h_t. Partial derivative operator ∂_x is not affected and used as usual.

[26] The set $\mathcal{C}^m(D)$ consists of functions which are m-times continuously differentiable on D.

Equation (T2) leads to backward difference quotient in point x_i

$$\frac{d}{dx}f(x_i) = \underbrace{\frac{f(x_i) - f(x_{i-1})}{h}}_{=:\left[\frac{df}{dx}\right]_i^{\mathrm{b}}} + \mathcal{O}(h).$$

Subtraction of equation (T2) from (T1) leads to central difference quotient in point x_i

$$\frac{d}{dx}f(x_i) = \underbrace{\frac{f(x_{i+1}) - f(x_{i-1})}{2h}}_{=:\left[\frac{df}{dx}\right]_i^{\mathrm{c}}} + \mathcal{O}(h^2).$$

Adding equation (T1) and (T2) leads to second order central difference quotient in point x_i

$$\frac{d^2}{dx^2}f(x_i) = \underbrace{\frac{f(x_{i+1}) - 2f(x_i) + f(x_{i-1})}{h^2}}_{=:\left[\frac{d^2f}{dx^2}\right]_i^{\mathrm{c}}} + \mathcal{O}(h^2).$$

Now, a two-dimensional domain is considered: domain $[0, x_{\mathrm{end}}] \times [0, y_{\mathrm{end}}]$ divided by a grid in $i_{\max}$ parts of length h_x in x-direction and in $j_{\max}$ parts of length h_y in y-direction, bounded by the mesh points (x_i, y_j):

$$x_i := ih_x, \quad i \in \{0, \ldots, i_{\max}\} \quad \text{and} \quad y_j := jh_y, \quad j \in \{0, \ldots, j_{\max}\}.$$

The function f is now defined on this two-dimensional domain $[0, x_{\mathrm{end}}] \times [0, y_{\mathrm{end}}] \subset \mathbb{R}$

$$f : [0, x_{\mathrm{end}}] \times [0, y_{\mathrm{end}}] \to \mathbb{R}, \quad f \in \mathcal{C}^m([0, x_{\mathrm{end}}] \times [0, y_{\mathrm{end}}]).$$

In order to apply finite difference method to Navier-Stokes equations over a two-dimensional domain it is sufficient to consider two points: Firstly substitute the differential operator d in the Taylor series with ∂ and take care to multiple variables. Secondly discretize the Laplacian derivative $\Delta f = f_{xx} + f_{yy}$ which is no more than the sum of two partial second order finite differences:

$$\begin{aligned}[\Delta f]_{i,j} :=& \frac{f(x_{i+1}, y_i) - 2f(x_i, y_i) + f(x_{i-1}, y_i)}{h_x^2} \\ &+ \frac{f(x_i, y_{i+1}) - 2f(x_i, y_i) + f(x_i, y_{i-1})}{h_y^2}.\end{aligned}$$

Convection terms Finite difference discretization of convection-terms of the form[27] $k\partial f/\partial x$ potentially leads to stability problems if the lattice is too rough. Therefore many methods exists to handle oscillations caused by these stability problems. For example the Upwind-Discretization where the central difference quotient is replaced by either forward or backward difference quotient - depending on the sign of f. Another possibility is the Donor-Cell scheme often used with convection terms of the form[28] $\partial(kf)/\partial x$. This scheme works in the way that u is given at grid points and

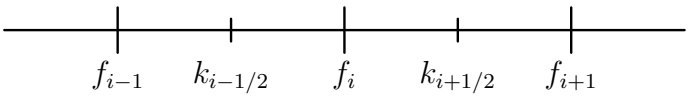


Figure 1: Discretization of Donor-Cell scheme.

values of k in the middle of these grid points (Figure 1).
Donor-Cell scheme discretization as finite differences of convective term $\partial(kf)/\partial x$ at point x_i:

$$\Big[\frac{\partial(kf)}{\partial x}\Big]_i^{\mathrm{dc}} := \frac{1}{2h_x}\Big(k_{i+1/2}(f_i + f_{i+1}) - k_{i-1/2}(f_{i-1} + f_i) + |k_{i+1/2}|(f_i + f_{i+1}) - |k_{i-1/2}|(f_{i-1} - f_i)\Big).$$

Approximation error of Donor-Cell is larger than central difference quotient therefore mixing both:

$$\Big[\frac{\partial(kf)}{\partial x}\Big]_i^{\mathrm{m}} := (1-\gamma)\Big[\frac{\partial(kf)}{\partial x}\Big]_i^{\mathrm{c}} + \gamma\Big[\frac{\partial(kf)}{\partial x}\Big]_i^{\mathrm{dc}}.$$

Parameter $\gamma \in [0,1]$ determines the weight-proportions of the mix.

3.2 Numerical Solution of Navier-Stokes equations

This chapter explains the details of a numeric algorithm to solve the two-dimensional Navier-Stokes equations derived in the last chapter. This numerical solution should sufficiently approximate u, v and p which are the unknowns over a bounded domain of place and time: $\mathbf{\Omega} \times [0,T]$, $\mathbf{\Omega} \subset \mathbb{R}^2$ and $[0,T] \subset \mathbb{R}$. To achieve a proper approximation an algorithm is derived which make use of discretization of this domain and applies the finite difference method to the analytic derivatives.
The structure of this chapter equals with the structure of the algorithm, with additional derivation information of the steps. A first overview of the derivation is given by the following sequence:

- Discretization in time of equations (14), (15) and (16) yields to recursive equation for velocity.
- Pressure is required in order to calculate velocity field in next time step: convert resulting equations into a Poisson equation for pressure.
- Discretization in place of Poisson equation to obtain system of linear equations for pressure.
- Calculate new velocity field.

The non-linearity remains in the time discretization.

[27] Variable k is a function of x.
[28] Applying product rule gives two convective terms.

3.2.1 Discretization in time

Time loop The time domain $[0,T] \subset \mathbb{R}$ is divided in N equidistant parts of length h_t which define the discretization grid:

$$t_n = nh_t, \quad n \in \{0, \dots, Nh_t\}.$$

Evaluating equation (14) and (15) in these time points requires to express the time derivation as difference quotient which is done with forward finite differences[29]:

$$\left[\frac{\partial u}{\partial t}\right]^n = \frac{u^{(n+1)} - u^{(n)}}{h_t}, \quad \left[\frac{\partial v}{\partial t}\right]^n = \frac{v^{(n+1)} - v^{(n)}}{h_t}.$$

Substitution in the Navier-Stokes equations with attention evaluating all terms at time t_n and rearranging gives

$$\begin{aligned}
u^{(n+1)} =& u^{(n)} + h_t\left(\frac{1}{Re}\Big(\partial_{xx}u^{(n)} + \partial_{yy}u^{(n)}\Big) - \partial_x(u^{(n)}u^{(n)}) - \partial_y(u^{(n)}v^{(n)}) + g_x\right) \\
&+ h_t\Big(\partial_x p^{(n)} + \mathcal{O}(h_t)\Big), \\
v^{(n+1)} =& v^{(n)} + h_t\left(\frac{1}{Re}\Big(\partial_{xx}v^{(n)} + \partial_{yy}v^{(n)}\Big) - \partial_y(v^{(n)}v^{(n)}) - \partial_x(u^{(n)}v^{(n)}) + g_y\right) \\
&+ h_t\Big(\partial_y p^{(n)} + \mathcal{O}(h_t)\Big).
\end{aligned}$$

This is an explicit scheme in time but implicit schemes allow bigger time steps. For this reason the last equations are modified to be time implicit in pressure. This is done by evaluate the pressure at time t_{n+1}. It is allowed because developing the partial derivative of pressure via the Taylor theorem gives an expression as

$$\partial_x p^{(n)} = \partial_x p^{(n+1)} + \mathcal{O}(h_t) \quad \text{and} \quad \partial_y p^{(n)} = \partial_y p^{(n+1)} + \mathcal{O}(h_t).$$

To deal subsequently with short expressions, the definition follows:

$$\begin{aligned}
F^{(n)} &:= u^{(n)} + h_t\left(\frac{1}{Re}\Big(\partial_{xx}u^{(n)} + \partial_{yy}u^{(n)}\Big) - \partial_x\Big(u^{(n)}u^{(n)}\Big) - \partial_y\Big(u^{(n)}v^{(n)}\Big) + g_x\right), \\
G^{(n)} &:= v^{(n)} + h_t\left(\frac{1}{Re}\Big(\partial_{xx}v^{(n)} + \partial_{yy}v^{(n)}\Big) - \partial_y\Big(v^{(n)}v^{(n)}\Big) - \partial_x\Big(u^{(n)}v^{(n)}\Big) + g_y\right).
\end{aligned}$$

This reduces time discretization of momentum continuity equations to

$$u^{(n+1)} = F^{(n)} - h_t\partial_x p^{(n+1)} + \mathcal{O}(h_t^2), \tag{17}$$
$$v^{(n+1)} = G^{(n)} - h_t\partial_y p^{(n+1)} + \mathcal{O}(h_t^2). \tag{18}$$

Discretization in time can considered as explicit in velocity and implicit in pressure. To calculate new velocity field $\mathbf{v}$ with this recursion the pressure at time t_{n+1} must be known. This is done via a Poisson equation for pressure which is derived in the next subsection.

[29] Upper index in round brackets at some function like u means in the progress: $u^{(n)} := u(x, t_n)$.

3.2.2 Discretization in place

In progress the bounded domain $\mathbf{\Omega}$ is restricted to be a rectangle. Hence it can be described as $\mathbf{\Omega} = [0, x_{\text{end}}] \times [0, y_{\text{end}}]$.
To avoid oscillations in pressure u, v and p are evaluated at different discretization points in the bounded domain $\mathbf{\Omega}$. In other words: three grids are in use, one per u, v and p. The grids are equal-sized and half-mesh-size parallel shifted to each other. The first grid is for pressure p. With respect to the first grid, the second grid is parallel shifted to east and the third grid is parallel shifted to north. This is called a staggered Grid (Figure 3).
Yet another explanation is expressed in mathematical syntax: rectangle $[0, x_{\text{end}}] \times [0, y_{\text{end}}]$ is decomposed equidistant in $i_{\max}$ columns of length h_x and $j_{\max}$ rows of height h_y. The intersection of column i with row j is named cell (i, j), for $i \in \{1, \ldots, i_{\max}\}$ and $j \in \{1, \ldots, j_{\max}\}$. This defines the grid points (Figure 2)

$$x_i := ih_x, \quad i \in \{1, \ldots, i_{\max}\} \quad \text{and} \quad y_j := jh_y, \quad j \in \{1, \ldots, j_{\max}\}.$$

Now pressure p is evaluated in the middle of the cell (i, j) which corresponds to

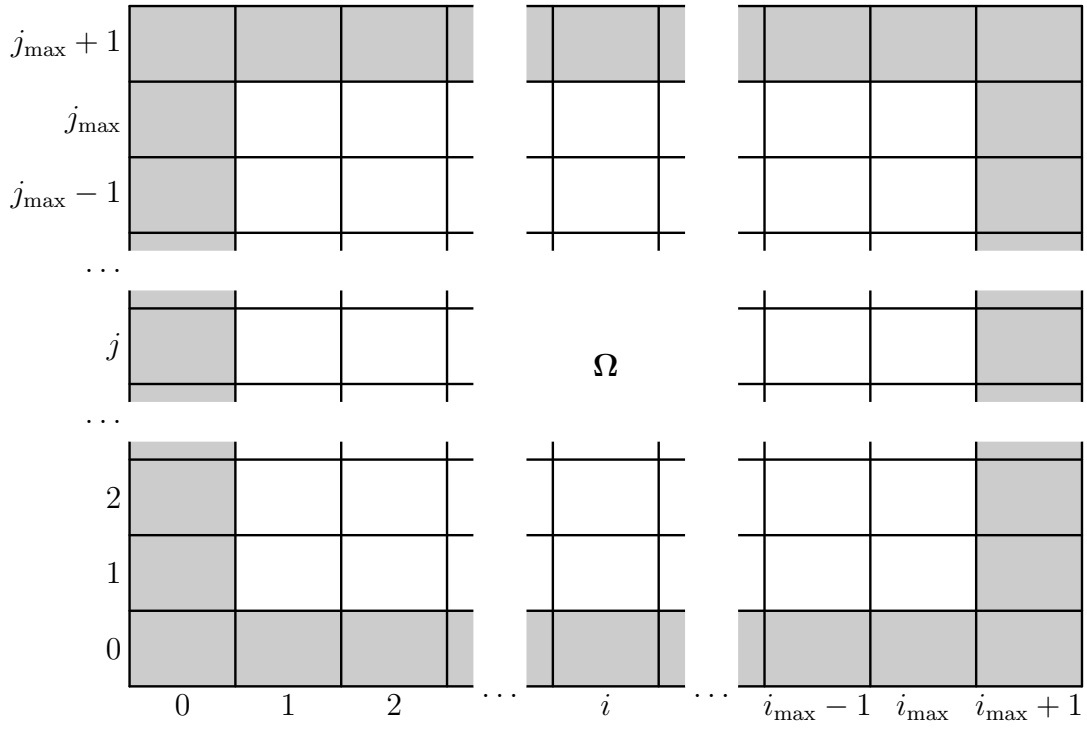


Figure 2: Domain as cells and boundary cells.

point (x_i, y_j) and is written as $p_{i,j}$. Velocity u is evaluated in middle of the east corner of the cell (i, j) written as $u_{i+1/2,j}$ and v is evaluated in middle of the north corner of the cell (i, j) written as $v_{i,j+1/2}$ (Figure 3). Because the velocity is used in progress mostly at the boundaries of the cells, the variables U and V are introduced:

$$U_{i,j} := u_{i+1/2,j} \quad \text{and} \quad V_{i,j} := v_{i,j+1/2}.$$

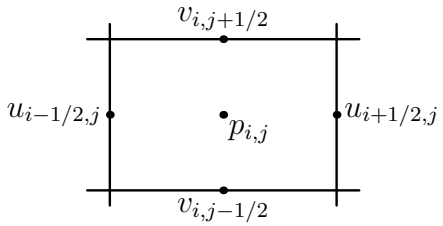


Figure 3: Cell (i, j) and its associated values.

Hence, expressing $u_{i,j}$ in words of U is done with arithmetic middle

$$u_{i,j} = \frac{u_{i-1/2,j} + u_{i+1/2,j}}{2} = \frac{U_{i-1,j} + U_{i,j}}{2}$$
$$v_{i,j} = \frac{v_{i,j-1/2} + v_{i,j+1/2}}{2} = \frac{V_{i,j-1} + V_{i,j}}{2}.$$

These averaging is also done at the boundaries, for example values like $U_{0,j} = u_{1/2,j}$ are needed at the left boundary of the domain because of $u_{0,j} = (U_{0,j} + U_{1,j})/2$. Therefore a boundary-layer of cells is added around the domain $\mathbf{\Omega}$ (Figure 2).
Equation (14) and (15) show the derivatives which have to be discretized. Four derivatives are convective terms which means that they have to be treat separate to avoid stability problems. This is done via the Donor-Cell scheme. An example is given below. The other convective terms are treated analogue.

Example: The finite difference quotient of the convective term $\partial_y(uv)$ at discretization point $(i + 1/2, j)$ is expressed in words of (u, v) and (U, V), latter with help of arithmetic middle:

$$\left[\frac{\partial}{\partial y}(uv)\right]_{i+1/2,j} = \frac{u_{i+1/2,j+1/2}v_{i+1/2,j+1/2} - u_{i+1/2,j-1/2}v_{i+1/2,j-1/2}}{h_x}$$
$$= \frac{1}{h_y}\left(\frac{U_{i,j+1} + U_{i,j}}{2}\frac{V_{i,j} + V_{i+1,j}}{2} - \frac{U_{i,j-1} + U_{i,j}}{2}\frac{V_{i,j-1} + V_{i+1,j-1}}{2}\right).$$

Figure 4 show all values in the grid which are used to calculate the finite differential quotient. Applying the Donor-Cell scheme to the convective differential quotient $\partial_y(u, v)$:

$$\left[\partial_y(UV)\right]_{i,j} := \frac{1}{h_y}\left(\frac{U_{i,j+1} + U_{i,j}}{2}\frac{V_{i,j} + V_{i+1,j}}{2} - \frac{U_{i,j-1} + U_{i,j}}{2}\frac{V_{i,j-1} + V_{i+1,j-1}}{2}\right)$$
$$+ \gamma\frac{1}{h_y}\left(\frac{U_{i,j+1} - U_{i,j}}{2}\frac{|V_{i,j} + V_{i+1,j}|}{2} - \frac{U_{i,j-1} - U_{i,j}}{2}\frac{|V_{i,j-1} + V_{i+1,j-1}|}{2}\right),$$

with parameter $\gamma \in [0, 1]$. If $\gamma = 0$ is chosen, the formula equals with central difference quotient and $\gamma = 1$ gives pure Donor-Cell discretization.
It follows the complete list of discretized convective partial derivatives which have

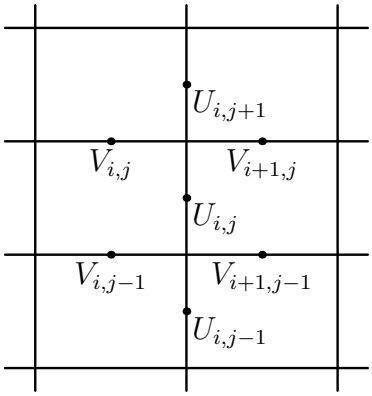


Figure 4: Values needed for discretization of $\partial_y(uv)$ at point $(i+1/2, j)$.

to be discretized in place in the same manner:

$$\begin{aligned}
&\forall\Big(i \in \{1,\dots,i_{\max}-1\} \wedge j \in \{1,\dots,j_{\max}\}\Big) : \\
&\Big[\partial_x(U^2)\Big]_{i,j} := \frac{1}{h_x}\left(\left(\frac{U_{i,j}+U_{i+1,j}}{2}\right)^2 - \left(\frac{U_{i-1,j}+U_{i,j}}{2}\right)^2\right) \\
&\qquad\qquad + \gamma\frac{1}{h_x}\left(\frac{|U_{i,j}+U_{i+1,j}|}{2}\frac{U_{i,j}+U_{i+1,j}}{2} - \frac{|U_{i-1,j}+U_{i,j}|}{2}\frac{U_{i-1,j}+U_{i,j}}{2}\right), \\
&\Big[\partial_y(UV)\Big]_{i,j} := \frac{1}{h_y}\left(\frac{U_{i,j+1}+U_{i,j}}{2}\frac{V_{i,j}+V_{i+1,j}}{2} - \frac{U_{i,j-1}+U_{i,j}}{2}\frac{V_{i,j-1}+V_{i+1,j-1}}{2}\right) \\
&\qquad\qquad + \gamma\frac{1}{h_y}\left(\frac{U_{i,j+1}-U_{i,j}}{2}\frac{|V_{i,j}+V_{i+1,j}|}{2} - \frac{U_{i,j-1}-U_{i,j}}{2}\frac{|V_{i,j-1}+V_{i+1,j-1}|}{2}\right),
\end{aligned}$$

$$\begin{aligned}
&\forall\Big(i \in \{1,\dots,i_{\max}\} \wedge j \in \{1,\dots,j_{\max}-1\}\Big) : \\
&\Big[\partial_y(V^2)\Big]_{i,j} := \frac{1}{h_x}\left(\left(\frac{V_{i,j}+V_{i,j+1}}{2}\right)^2 - \left(\frac{V_{i,j-1}+V_{i,j}}{2}\right)^2\right) \\
&\qquad\qquad + \gamma\frac{1}{h_x}\left(\frac{|V_{i,j}+V_{i,j+1}|}{2}\frac{V_{i,j}+V_{i,j+1}}{2} - \frac{|V_{i,j-1}+V_{i,j}|}{2}\frac{V_{i,j-1}+V_{i,j}}{2}\right), \\
&\Big[\partial_x(UV)\Big]_{i,j} := \frac{1}{h_x}\left(\frac{U_{i,j}+U_{i,j+1}}{2}\frac{V_{i,j}+V_{i+1,j}}{2} - \frac{U_{i-1,j}+U_{i-1,j+1}}{2}\frac{V_{i-1,j}+V_{i,j}}{2}\right) \\
&\qquad\qquad + \gamma\frac{1}{h_x}\left(\frac{|U_{i,j}+U_{i,j+1}|}{2}\frac{V_{i,j}-V_{i+1,j}}{2} - \frac{|U_{i-1,j}+U_{i-1,j+1}|}{2}\frac{V_{i-1,j}-V_{i,j}}{2}\right).
\end{aligned}$$

In [Hirt 1975] it is recommend to choose

$$\gamma \geq \max_{i,j}\left(\left|\frac{U_{i,j}h_t}{h_x}\right|, \left|\frac{V_{i,j}h_t}{h_y}\right|\right).$$

The error term of Donor-Cell discretization is greater than pure central difference. A worst case is given with $\mathcal{O}(h_x)$ or $\mathcal{O}(h_y)$. The second order derivatives of equation

(14) and (15) is also done with central differences. It has to be expressed in terms of U and V, too. For $i \in \{1, \dots, i_{\max} - 1\}$ and $j \in \{1, \dots, i_{\max}\}$ holds:

$$\left[\partial_{xx} U\right]_{i,j} := \frac{U_{i+1,j} - 2U_{i,j} + U_{i-1,j}}{h_x^2}, \quad \left[\partial_{yy} U\right]_{i,j} := \frac{U_{i,j+1} - 2U_{i,j} + U_{i,j-1}}{h_x^2}.$$

For $i \in \{1, \dots, i_{\max}\}$ and $j \in \{1, \dots, i_{\max} - 1\}$ holds:

$$\left[\partial_{xx} V\right]_{i,j} := \frac{V_{i+1,j} - 2V_{i,j} + V_{i-1,j}}{h_x^2}, \quad \left[\partial_{yy} V\right]_{i,j} := \frac{V_{i,j+1} - 2V_{i,j} + V_{i,j-1}}{h_x^2}.$$

Now only the pressure is missing which is discretized in the same way: to discretize the pressure gradient via central differences in the upper or right cell corner, one has to define for all $i \in \{1, \dots, i_{\max}\}$ and $j \in \{1, \dots, i_{\max} - 1\}$:

$$\left[\partial_x p\right]_{i,j} := \frac{p_{i+1,j} - p_{i,j}}{h_x} \quad \text{and} \quad \left[\partial_y p\right]_{i,j} := \frac{p_{i,j+1} - p_{i,j}}{h_y}.$$

Boundary conditions The discretization in place of velocity needs boundary values

$$\begin{aligned} &V_{i,0}, \quad && V_{i,j_{\max}}, \quad && i \in \{1, \dots, i_{\max}\}, \\ &U_{0,j}, \quad && U_{i_{\max},j}, \quad && j \in \{1, \dots, j_{\max}\} \end{aligned}$$

and values outside of the domain

$$\begin{aligned} &U_{i,0}, \quad && U_{i,j_{\max}+1}, \quad && i \in \{1, \dots, i_{\max}\}, \\ &V_{0,j}, \quad && V_{i_{\max}+1,j}, \quad && j \in \{1, \dots, j_{\max}\}. \end{aligned}$$

To obtain these velocities, boundary conditions are necessary. Boundary values vanish for no-slip, free-slip and moving-boundary conditions:

$$\begin{aligned} &V_{i,0} := 0, \quad && V_{i,j_{\max}} := 0, \quad && i \in \{1, \dots, i_{\max}\}, \\ &U_{0,j} := 0, \quad && U_{i_{\max},j} := 0, \quad && j \in \{1, \dots, j_{\max}\}. \end{aligned}$$

The values outside the domain, for example $V_{0,j}$ are obtained via averaging:

$$V_{1/2,j} = \frac{V_{0,j} + V_{1,j}}{2} \Leftrightarrow V_{0,j} = 2V_{1/2,j} - V_{1,j}.$$

In the case of **no-slip** conditions $V_{1/2,j}$ has to vanish because velocity at the boundary is zero. This gives $V_{0,j} := -V_{1,j}$ and hence

$$\begin{aligned} &U_{i,0} := -U_{i,1}, \quad && U_{i,j_{\max}+1} := -U_{i,j_{\max}}, \quad && i \in \{1, \dots, i_{\max}\}, \\ &V_{0,j} := -V_{1,j}, \quad && V_{i_{\max}+1,j} := -V_{i_{\max},j}, \quad && j \in \{1, \dots, j_{\max}\}. \end{aligned}$$

In the case of **free-slip** conditions $V_{1/2,j}$ must be equal to $V_{1,j}$ because velocity at the boundary does not change. This gives $V_{0,j} := V_{1,j}$ and hence

$$\begin{aligned} &U_{i,0} := U_{i,1}, \quad && U_{i,j_{\max}+1} := U_{i,j_{\max}}, \quad && i \in \{1, \dots, i_{\max}\}, \\ &V_{0,j} := V_{1,j}, \quad && V_{i_{\max}+1,j} := V_{i_{\max},j}, \quad && j \in \{1, \dots, j_{\max}\}. \end{aligned}$$

In case of **moving boundaries** $V_{1/2,j}$ has to be constant $\bar{V}$ because fluid velocity at the boundary has to equal moving boundary velocity. This gives $V_{0,j} := 2\bar{V} - V_{1,j}$ and hence

$$\begin{aligned} U_{i,0} &:= 2\bar{U} - U_{i,1}, \qquad U_{i,j_{\max}+1} := 2\bar{U} - U_{i,j_{\max}}, \quad i \in \{1,\dots,i_{\max}\}, \\ V_{0,j} &:= 2\bar{V} - V_{1,j}, \qquad V_{i_{\max}+1,j} := 2\bar{V} - V_{i_{\max},j}, \quad j \in \{1,\dots,j_{\max}\}. \end{aligned}$$

Velocity does not change in **outflow** region. Therefore normal derivation of u and v vanish which is done in discrete case by

$$\begin{aligned} U_{0,j} &:= U_{1,j}, \qquad U_{i_{\max},j} := U_{i_{\max}-1,j}, \quad j \in \{1,\dots,j_{\max}\}, \\ V_{0,j} &:= V_{1,j}, \qquad V_{i_{\max}+1,j} := V_{i_{\max},j}, \quad j \in \{1,\dots,j_{\max}\}, \end{aligned}$$

$$\begin{aligned} U_{i,0} &:= U_{i,1}, \qquad U_{i,j_{\max}+1} := U_{i,j_{\max}}, \quad i \in \{1,\dots,i_{\max}\}, \\ V_{i,0} &:= V_{i,1}, \qquad V_{i,j_{\max}} := V_{i,j_{\max}-1}, \quad i \in \{1,\dots,i_{\max}\}. \end{aligned}$$

Both velocity components at the boundary are given in case of **inflow** conditions. Calculation of values out of domain like $V_{0,j}$ can be done by averaging as before.

Discretization of momentum continuity Last step is the substitution of these derivatives in equation (17) and (18). Giving attention to the staggered grid and using the advantage of the notation U and V, one obtains:

$$\begin{aligned} U_{i,j}^{(n+1)} &= F_{i,j}^{(n)} - \frac{h_t}{h_x}\left(p_{i+1,j}^{(n+1)} - p_{i,j}^{(n+1)}\right) + \mathcal{O}(h_x) + \mathcal{O}(h_y) + \mathcal{O}(h_t^2), \\ &\quad \text{for} \quad i \in \{1,\dots,i_{\max}-1\} \quad \text{and} \quad j \in \{1,\dots,i_{\max}\}, \qquad (19) \\ V_{i,j}^{(n+1)} &= G_{i,j}^{(n)} - \frac{h_t}{h_y}\left(p_{i,j+1}^{(n+1)} - p_{i,j}^{(n+1)}\right) + \mathcal{O}(h_x) + \mathcal{O}(h_y) + \mathcal{O}(h_t^2), \\ &\quad \text{for} \quad i \in \{1,\dots,i_{\max}\} \quad \text{and} \quad j \in \{1,\dots,i_{\max}-1\}, \qquad (20) \end{aligned}$$

$$\begin{aligned} F_{i,j}^{(n)} &:= u_{i,j}^{(n)} + h_t\left(\frac{1}{Re}\left(\left[\partial_{xx}U\right]_{i,j}^{(n)} + \left[\partial_{yy}U\right]_{i,j}^{(n)}\right) - \left[\partial_x(U^2)\right]_{i,j}^{(n)} - \left[\partial_y(UV)\right]_{i,j}^{(n)} + g_x\right), \\ &\quad \text{for} \quad i \in \{1,\dots,i_{\max}-1\} \quad \text{and} \quad j \in \{1,\dots,i_{\max}\}, \qquad (21) \\ G_{i,j}^{(n)} &:= v_{i,j}^{(n)} + h_t\left(\frac{1}{Re}\left(\left[\partial_{xx}V\right]_{i,j}^{(n)} + \left[\partial_{yy}V\right]_{i,j}^{(n)}\right) - \left[\partial_y(V^2)\right]_{i,j}^{(n)} - \left[\partial_x(UV)\right]_{i,j}^{(n)} + g_y\right), \\ &\quad \text{for} \quad i \in \{1,\dots,i_{\max}\} \quad \text{and} \quad j \in \{1,\dots,i_{\max}-1\}, \qquad (22) \end{aligned}$$

where every variable gets its assignment to cell (i,j) and the discrete derivations gets their assignment to time t_n. Equations (21) and (22) are full determined if the boundary conditions are set. The discretization error with respect to place is $\mathcal{O}(h_x)$ + $\mathcal{O}(h_y)$ in worst case scenario because of Donor-Cell.

Deriving Poisson equation Substitute equation (17) and (18) in momentum continuity equation[30]:

$$0 = \partial_x u^{(n+1)} + \partial_y v^{(n+1)} = \partial_x F^{(n)} - h_t \partial_{xx} p^{(n+1)} + \partial_y G^{(n)} - h_t \partial_{yy} p^{(n+1)} + \mathcal{O}(h_t^2). \quad (23)$$

Conversion shows that this equation is a Poisson equation for pressure:

$$\Delta p^{(n+1)} = \frac{1}{h_t}\Big(\partial_x F^{(n)} + \partial_y G^{(n)}\Big) + \mathcal{O}(h_t), \quad \text{onto} \quad \mathbf{\Omega}.$$

The question of the kind of the boundary conditions is answered as follows: looking at equation (17) and (18) raise the idea to use Neumann boundary conditions. Characteristic of these conditions is that the normal derivative (of the function one is looking for) is given at the boundary of the domain. This approach leads to equation[31]

$$\begin{aligned}\nabla p^{(n+1)}\mathbf{n} &= \partial_x p^{(n+1)} n_1 + \partial_y p^{(n+1)} n_2 \\ &= -\frac{1}{h_t}\left(\Big(u^{(n+1)} - F^{(n)}\Big)n_1 + \Big(v^{(n+1)} - G^{(n)}\Big)n_2\right) + \mathcal{O}(h_t), \quad \text{onto} \quad \partial\mathbf{\Omega}\end{aligned}$$

which has to be true on the boundary of $\mathbf{\Omega}$, usually written as $\partial\mathbf{\Omega}$. This Method is called Chorin-Projection-method and described in [Chorin 1968] and [Temam 1969]. Summarizing the results to express the algorithm with the new knowledge:

1. Calculate $F^{(n)}$ and $G^{(n)}$ from $u^{(n+1)}$ and $v^{(n+1)}$.
2. Solve Poisson equation to get $p^{(n+1)}$.
3. Calculate new velocity field $(u^{(n+1)}, v^{(n+1)})^T$.

Discrete Poisson equation The discrete sizes of the last paragraph are now used to discretize the Poisson equation for pressure. Discrete Laplacian derivative for pressure is obtained in the way which is shown in subsection 3.1 Finite difference method. The discrete equation is evaluated in a mix at time t_n or t_{n+1} (see equation (23)) and at place (x_i, y_j) which means in the middle of the cell (i, j):[32]

$$\begin{aligned}&\frac{p_{i+1,j}^{(n+1)} - 2p_{i,j}^{(n+1)} + p_{i-1,j}^{(n+1)}}{h_x^2} + \frac{p_{i,j+1}^{(n+1)} - 2p_{i,j}^{(n+1)} + p_{i,j-1}^{(n+1)}}{h_y^2} \\ &= \underbrace{\frac{1}{h_t}\left(\frac{F_{i,j}^{(n)} - F_{i-1,j}^{(n)}}{h_x} + \frac{G_{i,j}^{(n)} - G_{i,j-1}^{(n)}}{h_y}\right)}_{=:RHS_{i,j}} + \mathcal{O}(h_x) + \mathcal{O}(h_y) + \mathcal{O}(h_t^2), \\ &i \in \{1, \ldots, i_{\max}\} \quad \text{and} \quad j \in \{1, \ldots, j_{\max}\}. \end{aligned} \quad (24)$$

[30] Big-$\mathcal{O}$-term does not change after partial derivation because developing u in Taylor series and applying partial derivative with respect to t to all terms shows the error size is still $\mathcal{O}(h_t^2)$.
[31] Normal vector $\mathbf{n}$ split up into scalar components n_x and n_y.
[32] The definition of $RHS_{i,j}$ is used subsequently.

Omitting the error term $\mathcal{O}(h_x)+\mathcal{O}(h_y)+\mathcal{O}(h_t^2)$ leads to a system of linear equations for pressure in $i_{\max}j_{\max}$ unknowns. The system uses the pressure boundary-values

$$\begin{aligned} &p_{i,0}, \qquad p_{i,j_{\max}+1}, \quad i \in \{1,\dots,i_{\max}\},\\ &p_{0,j}, \qquad p_{i_{\max}+1,j}, \quad j \in \{1,\dots,j_{\max}\}. \end{aligned}$$

Furthermore the following boundary values for $F_{i,j}$ and $G_{i,j}$ are needed:

$$\begin{aligned} &G_{i,0}, \qquad G_{i,j_{\max}}, \quad i \in \{1,\dots,i_{\max}\},\\ &F_{0,j}, \qquad F_{i_{\max},j}, \quad j \in \{1,\dots,j_{\max}\}. \end{aligned}$$

Determining these values is done via the discrete boundary condition of poisson equation:

$$\begin{aligned} &\frac{p_{i+1,j}^{(n+1)}-p_{i,j}^{(n+1)}}{h_x}n_1+\frac{p_{i,j+1}^{(n+1)}-p_{i,j}^{(n+1)}}{h_y}n_2\\ &=-\frac{1}{h_t}\left(\left(u_{i,j}^{(n+1)}-F_{i,j}^{(n)}\right)n_1+\left(v_{i,j}^{(n+1)}-G_{i,j}^{(n)}\right)n_2\right)+\mathcal{O}(h_x)+\mathcal{O}(h_y)+\mathcal{O}(h_t),\\ &\forall i\in\{1,\dots,i_{\max}\}, j\in\{1,\dots,j_{\max}\}:\left(i=1\vee j=1\right). \end{aligned}$$

For example the discretization on the left boundary (formalized as $i=1$) where $\mathbf{n}=(-1,0)^T$ gives:

$$\frac{-p_{1,j}^{(n+1)}+p_{0,j}^{(n+1)}}{h_x}n_1=-\frac{1}{h_t}\Big(-u_{0,j}^{(n+1)}+F_{0,j}^{(n)}\Big)+\mathcal{O}(h_x)+\mathcal{O}(h_y)+\mathcal{O}(h_t).$$

Substitution in (24) for $i=1$ leads to:

$$\frac{p_{2,j}^{(n+1)}-p_{1,j}^{(n+1)}}{h_x^2}+\frac{p_{1,j+1}^{(n+1)}-2p_{1,j}^{(n+1)}+p_{1,j-1}^{(n+1)}}{h_y^2}=\frac{1}{h_t}\left(\frac{F_{1,j}^{(n)}-u_{0,j}^{(n+1)}}{h_x}+\frac{G_{1,j}^{(n)}-G_{1,j-1}^{(n)}}{h_y}\right).$$

This equation shows a degree of freedom because it is independent of $F_{0,j}^{(n)}$. Therefore the choice of $F_{0,j}^{(n)}$ is arbitrary. If it is defined as $F_{0,j}^{(n)}:=u_{0,j}^{(n+1)}$ it leads to $p_{0,j}^{(n+1)}=p_{1,j}^{(n+1)}$ which defines the left boundary value of the pressure. This is done analogous for the right, top and bottom boundary and hence gives the values:

$$\begin{aligned} &p_{i,0}:=p_{i,1}, \qquad p_{i,j_{\max}+1}:=p_{i,j_{\max}}, \quad i\in\{1,\dots,i_{\max}\},\\ &p_{0,j}:=p_{1,j}, \qquad p_{i_{\max}+1,j}:=p_{i_{\max},j}, \quad j\in\{1,\dots,j_{\max}\}, \end{aligned}$$

$$\begin{aligned} &G_{i,0}:=v_{i,0}, \qquad G_{i,j_{\max}}:=v_{i,j_{\max}}, \quad i\in\{1,\dots,i_{\max}\}, &(25)\\ &F_{0,j}:=u_{0,j}, \qquad F_{i_{\max},j}:=u_{i_{\max},j}, \quad j\in\{1,\dots,j_{\max}\}. &(26) \end{aligned}$$

Hence, after installation of these boundary conditions in discretization of Poisson system for pressure, the modified system is:

$$\begin{aligned} &\frac{\epsilon_i^E(p_{i+1,j}^{(n+1)}-p_{i,j}^{(n+1)})-\epsilon_i^W(p_{i,j}^{(n+1)}-p_{i-1,j}^{(n+1)})}{h_x^2}+\frac{\epsilon_j^N(p_{i,j+1}^{(n+1)}-p_{i,j}^{(n+1)})-\epsilon_j^S(p_{i,j}^{(n+1)}-p_{i,j-1}^{(n+1)})}{h_y^2}\\ &=\frac{1}{h_t}\left(\frac{F_{i,j}^{(n)}-F_{i-1,j}^{(n)}}{h_x}+\frac{G_{i,j}^{(n)}-G_{i,j-1}^{(n)}}{h_y}\right),\\ &i\in\{1,\dots,i_{\max}\} \quad \text{and} \quad j\in\{1,\dots,j_{\max}\}. \end{aligned}$$

Parameter ϵ is defined as

$$\epsilon_i^W = \begin{cases} 0 & i = 1 \\ 1 & i > 1 \end{cases}, \quad \epsilon_i^E = \begin{cases} 1 & i < i_{\max} \\ 0 & i = i_{\max} \end{cases},$$
$$\epsilon_j^S = \begin{cases} 0 & j = 1 \\ 1 & j > 1 \end{cases}, \quad \epsilon_j^N = \begin{cases} 1 & j < j_{\max} \\ 0 & j = j_{\max} \end{cases},$$

and implements the property if the pressure values has to be eliminated because of boundary or not. Indexes W, E, S and N are in relation to compass points and refers to the direction of the boundary.

3.2.3 SOR

Solving a system of linear equations $Ap = b$ of size $i_{\max}j_{\max}$ is based on the Gauss-Seidel method which is an improvement of the Jacobi method. It is an iterative algorithm to approximate the exact solution p of the linear equation system with $p^{(k)}$ at iteration step k. An initial value $p^{(k)}$ has to be chosen. See Algorithms 1 and 2 for Jacobi and Gauss-Seidel method, respectively. A special Gauss-Seidel enhancement is called SOR method. This stands for successive over-relaxation and is used to speed up convergence of Gauss-Seidel method, see Algorithm 3.[33] Parameter ω is

Algorithm 1 Jacobi

1: **for** $k = 0$ to k_{end} **do**
2: **for** $n = 1$ to $i_{\max}j_{\max}$ **do**
3: $p_n^{[k+1]} := \frac{1}{a_{n,n}}\left(b_n - \sum_{m \neq n} a_{n,m} p_m^{[k]}\right)$
4: **end for**
5: **end for**

Algorithm 2 Gauss-Seidel

1: **for** $k = 0$ to k_{end} **do**
2: **for** $n = 1$ to $i_{\max}j_{\max}$ **do**
3: $p_n^{[k+1]} = \frac{1}{a_{n,n}}\left(b_n - \sum_{m<n} a_{n,m} p_m^{[k+1]} - \sum_{m>n} a_{n,m} p_m^{[k]}\right)$
4: **end for**
5: **end for**

Algorithm 3 SOR

1: **for** $k = 0$ to k_{end} **do**
2: **for** $n = 1$ to $i_{\max}j_{\max}$ **do**
3: $p_n^{[k+1]} = (1-\omega)p_n^{[k]} + \frac{\omega}{a_{n,n}}\left(b_n - \sum_{m<n} a_{n,m} p_m^{[k+1]} - \sum_{m>n} a_{n,m} p_m^{[k]}\right)$
4: **end for**
5: **end for**

[33] The upper index of p written in square brackets refers to iteration step, not time (which is written in round brackets).

called relaxation factor. This factor has to be in the interval $[0, 2]$ and influences the convergence speed. If $\omega = 1$ is chosen, the SOR method is the same as Gauss-Seidel. To choose a value smaller than one leads to slower convergence but helps establishing convergence of divergent iterations. Values greater than one speeds up the convergence but may lead sometimes to divergence.

Applying the SOR method to the discrete pressure equation leads to Algorithm 4

Algorithm 4 Poisson SOR

1: **for** $k = 0$ to $k_{\max}$ **do**
2: **for** $i = 1$ to $i_{\max}$ **do**
3: **for** $j = 1$ to $j_{\max}$ **do**
4: $p_{i,j}^{[k+1]} = (1-\omega)p_{i,j}^{[k]} + \frac{\omega}{\frac{\epsilon_i^O+\epsilon_i^W}{h_x^2}+\frac{\epsilon_j^N+\epsilon_j^S}{h_y^2}}\left(\frac{\epsilon_i^O p_{i+1,j}^{[k]}+\epsilon_i^W p_{i-1,j}^{[k+1]}}{h_x^2} + \frac{\epsilon_j^N p_{i,j+1}^{[k]}+\epsilon_j^S p_{i,j-1}^{[k+1]}}{h_y^2} - RHS_{i,j}\right)$
5: **end for**
6: **end for**
7: **end for**

where[34] $RHS_{i,j}$ stands for the right-hand-side of the discrete Poisson equation for pressure.

The iteration stops either when $k_{\max}$ is reached or when the L^2-norm of the residuum

$$r_{i,j}^{[k]} := \frac{\epsilon_i^E\left(p_{i+1,j}^{[k]} - p_{i,j}^{[k]}\right) - \epsilon_i^W\left(p_{i,j}^{[k]} - p_{i-1,j}^{[k]}\right)}{h_x^2} + \frac{\epsilon_j^N\left(p_{i,j+1}^{[k]} - p_{i,j}^{[k]}\right) - \epsilon_j^S\left(p_{i,j}^{[k]} - p_{i,j-1}^{[k]}\right)}{h_y^2} - RHS_{i,j},$$
$$i \in \{1, \ldots, i_{\max}\}, \quad j \in \{1, \ldots, i_{\max}\}$$

drops below an absolute tolerance limit *eps*. The L^2-norm is defined as

$$||r^{[k]}||_2 := \left(\frac{1}{i_{\max} j_{\max}} \sum_{i=1}^{i_{\max}} \sum_{j=1}^{j_{\max}} \left(r_{i,j}^{[k]}\right)^2\right)^{1/2}. \tag{27}$$

For calculating $p^{(n+1)}$ the SOR algorithm needs initial conditions for $k = 0$. These are taken from the pressure calculation at previously time step n. Therefore initial conditions for pressure $p^{(0)} := p_{i,j}^{(0)}$ are needed[35]. Initial conditions $U^{(0)} := U_{i,j}^{(0)}$ and $V^{(0)} := V_{i,j}^{(0)}$ are also needed[36] because of the term $RHS^{(0)}$.

This may lead to a problem: the matrix of the system (24) is singular because of Neumann boundary conditions for pressure. To obtain a solution, the right-hand side of the system has to be in the image of the matrix. This solution has a

[34] This was defined in System (24).
[35] Identical initial conditions are chosen for all $i \in \{1, \ldots, i_{\max}\}, \quad j \in \{1, \ldots, i_{\max}\}$.
[36] Again, identical initial conditions are chosen for all $i \in \{1, \ldots, i_{\max}\}, \quad j \in \{1, \ldots, i_{\max}\}$.

degree of freedom (corresponding to pressure) which is determined up to an additive constant[37]. If the velocity field at time t_n does not satisfy the continuity equation[38] the pressure values became unphysical.
In [Griebel 1995] it is written about numerical experiments which show that the problem is manageable if following is done before each iteration step in the SOR:[39]

$$p_{i,0}^{[k+1]} = p_{i,1}^{(k)}, \quad p_{i,j_{\max}+1}^{[k]} = p_{i,j_{\max}}^{[k]}, \quad i \in \{1,\dots,i_{\max}\},$$
$$p_{0,j}^{[k+1]} = p_{1,j}^{(k)}, \quad p_{i_{\max}+1,j}^{[k]} = p_{i_{\max},j}^{[k]}, \quad j \in \{1,\dots,j_{\max}\}.$$

Concurrently the SOR has to be modified by setting all ϵ to one. This defines the Aalgorithm 5. The residuum has to be adapted too: all ϵ are simply set to one.

Algorithm 5 Modified Poisson SOR

for $k = 0$ to $k_{\max}$ **do**
  for $i = 1$ to $i_{\max}$ **do**
    $p_{i,0}^{[k+1]} = p_{i,1}^{[k]}$
    $p_{i,j_{\max}+1}^{[k]} = p_{i,j_{\max}}^{[k]}$
  end for
  for $j = 1$ to $j_{\max}$ **do**
    $p_{0,j}^{[k+1]} = p_{1,j}^{[k]}$
    $p_{i_{\max}+1,j}^{[k]} = p_{i_{\max},j}^{[k]}$
  end for
  for $i = 1$ to $i_{\max}$ **do**
    for $j = 1$ to $j_{\max}$ **do**
      $p_{i,j}^{[k+1]} = (1-\omega)p_{i,j}^{[k]} + \frac{\omega}{\frac{2}{h_x^2}+\frac{2}{h_y^2}}\left(\frac{p_{i+1,j}^{[k]}+p_{i-1,j}^{[k+1]}}{h_x^2} + \frac{p_{i,j+1}^{[k]}+p_{i,j-1}^{[k+1]}}{h_y^2} - RHS_{i,j}\right)$
    end for
  end for
end for

3.2.4 Stability conditions

To guarantee the stability of the algorithm it is necessary to keep the Courant-Friedrichs-Lewy conditions. They describe how to choose time step with given step sizes in place:

$$\frac{2h_t}{Re} < \left(\frac{1}{h_x^2}+\frac{1}{h_y^2}\right)^{-1}, \quad |u_{\max}|h_t < h_x, \quad |v_{\max}|h_t < h_y.$$

Based on this, the length of the time step at t_n is adapted to the velocity[40]:

$$h_t^{(n)} := \tau \min\left(\frac{Re}{2}\left(\frac{1}{h_x^2}+\frac{1}{h_y^2}\right)^{-1}, \frac{h_x}{|u_{\max}^{(n)}|}, \frac{h_y}{|v_{\max}^{(n)}|}\right). \tag{28}$$

[37] Shown in subsection 2.4 Body forces.
[38] Especially at the beginning if choosing initial conditions for a given domain is difficult.
[39] For pressure values $p^{(k)}$ out of the domain SOR is asking only for east respectively north values.
[40] Because of its time character $h_t^{(n)}$ is simply called amplitude at time n in progress.

Parameter τ is a safety factor out of the interval $(0, 1]$ and $u_{\max}^{(n)}$ or $v_{\max}^{(n)}$ is defined as the maximum of all $u_{i,j}^{(n)}$ or $v_{i,j}^{(n)}$ in the domain.

3.2.5 Summary

In conclusion, the numerical scheme to solve the Navier-Stokes equations is shown in Algorithm 6.

Algorithm 6 Computational fluid dynamics

1: $t := 0, n := 0$, set initial conditions $U_{i,j}^{(0)}, V_{i,j}^{(0)}, p_{i,j}^{(0)}$
2: **while** $t < t_{\text{end}}$ **do**
3: set amplitude $h_t^{(n)}$ {equation (28)}
4: set boundary conditions for u and v
5: set $F_{i,j}^{(n)}, G_{i,j}^{(n)}$ {equations (21),(22) and boundary conditions (26), (25)}
6: set $RHS_{i,j}^{(n)}$ {defined in system (24)}
7: $k := 0$
8: **while** $k < k_{\max} \wedge ||r^{[k]}|| > eps$ **do**
9: apply kernel of SOR {algorithm 5}
10: set $||r^{[k]}||$ {equation (27)}
11: $k := k + 1$
12: **end while**
13: set $U_{i,j}^{(n+1)}, V_{i,j}^{(n+1)}$ {equations (17),(18)}
14: $t := t + h_t^{(n)}, n := n + 1$
15: **end while**

4 CPU Implementation

To code a numerical algorithm, one is well-advised keeping following programming-paradigm in mind: *Make run, make fast, make good!* The meaning is nearly self explaining. First challenge is to make the algorithm run in code whereas less attention is given to speed and form while run-time. After achieving functionality the aim is to provide the code with pure functionality without too much overhead: making fast - speed up in run-time. Next field of attention is given to the look: making code readable - inserting comments and making sure the output has an attractive form. This paradigm can also be interpreted as *form follows function*, a popular principle in modern architecture and industrial design.
While following this paradigm it makes sense to structure the following sections likewise: First is **Correctness** where the relationship is shown between source code and Algorithm 6 and therefore the correctness of the implementation. This corresponds to make run. Second section is **Profiling** which analyze the performance and measures the FLOPS of the implementation. Corresponding section of make fast. Third section is devoted to the output and the graphical representation: **Visualization** according to make good.
Last but not least a section **Validation** exists where the code is demonstrated at some suitable test cases and compared with the results of existing CFD implementations to *proof* the correctness by experiments.
The source code is appended on a mini CD which also includes a compiled version for x86_64 PC architectures[41].

4.1 Correctness

This section describes the implementation of the Algorithm 6 in the programming language C. To proof the functionality of the implementation with respect to what it should do, a one-to-one mapping of the source code and the Algorithm 6 is presented.
Writing in C implies that the algorithm is coded in a procedural and serial language. The source code is split into several procedures providing clarity and module-like flexibility. The implementation of each line of Algorithm 6 is explained in detail below. Numbering refers to the line numbering of the algorithm and therefore provides a mapping-like unique correspondence of implementation and algorithm.

[Line 1] Declaration and initialization of all variables: Following values are read while run-time from an external text file with procedure `READ_PARAMETER`:

$$\texttt{x\_end} := x_{\text{end}}, \quad \texttt{y\_end} := y_{\text{end}}, \quad \texttt{imax} := i_{\max}, \quad \texttt{jmax} := j_{\max},$$
$$\texttt{t\_end} := t_{\text{end}}, \quad \texttt{tau} := \tau,$$
$$\texttt{itermax} := k_{\max}, \quad \texttt{eps} := eps, \quad \texttt{omg} := \omega,$$
$$\texttt{gamma} := \gamma, \quad \texttt{Re} := Re, \quad \texttt{GX} := g_x, \quad \texttt{GY} := g_y,$$
$$\texttt{UI} := U^{(0)}, \quad \texttt{VI} := V^{(0)}, \quad \texttt{PI} := p^{(0)}.$$

[41] A NVIDIA G200 chip is also required in order to fulfill purposes in subsequent section GPU Implementation.

The velocity U and V, the pressure P, F, the variables G and RHS are implemented via two-dimensional arrays. These are dynamically allocated with respect to domain size.

$$\texttt{U[i][j]} := U_{i,j}, \quad \texttt{V[i][j]} := V_{i,j}, \quad \texttt{P[i][j]} := p_{i,j},$$
$$\texttt{RHS[i][j]} := RHS_{i,j}, \quad \texttt{F[i][j]} := F_{i,j}, \quad \texttt{G[i][j]} := G_{i,j}.$$

The following values are declared while run-time with values:

$$\texttt{t} := t := 0, \quad \texttt{n} := n := 0, \quad \texttt{delt} := h_t, \quad \texttt{res} := ||r^{[k]}||.$$

Procedure `INIT_UVP` fills arrays U, V and P with the initial values `UI`, `VI` and `PI`.

[Line 3] Procedure `COMP_DELT` is responsible to compute the amplitude h_t respective `h_t` in source code. It calculates the maximum of each U and V and determines the minimum of equation (28).

[Line 4] As mentioned in the section about boundary conditions, the domain has to be rectangular. Furthermore the implementation is restricted to have only one kind of boundary conditions at each boundary. Therefore the variables `b_W`, `b_O`, `b_S` and `b_N` are used to store information about the kind of the boundary condition. The variables are arrays with two fields. The information stored in field one is an integer out of $\{1, 2, 3, 4, 5\}$. One means no-slip, two is free-slip, three is moving-boundary, four is outflow condition and five is inflow condition. The second field is used to store additional information needed by moving-boundary and inflow condition[42]. Procedure `SETBCOND` distinguishes between these cases and assigns the velocity on their boundaries appropriate values.

[Line 5] Computation of auxiliary variables $F^{(n)}$ and $G^{(n)}$ concerning to `F` and `G`. All difference quotients of u and v with respect to place are inserted in these fields. This is done by the procedure `COMP_FG`.

[Line 6] Computation of right-hand side `RHS` of Poisson pressure equation. This is done by the procedure `COMP_RHS`.

[Line 7-12] The implementation of the Poisson pressure equation solver is the procedure `POISSON`. Every iteration the pressure boundary conditions are adapted. New pressure values are calculated and the residual is computed.

[Line 13] Computation of the new velocity field `U` and `V`. This is done by `ADAP_UV`.

[42] This additional information is the velocity of moving-boundary respective inflow stream.

4.2 Profiling

Optimizing source code with respect to speed up execution at run-time is mainly done in two ways: At first introducing auxiliary variables per procedure to avoid repeatedly calculation of the same arithmetic terms and to provide the compiler using the cache of the processor. This leads to less calculation time with the cost of more memory usage.[43] Since the auxiliary variables are no big arrays[44] it does not affect the memory usage noticeably.
The second approach is avoidance of calling math-functions from the C library. This is done because it is an cost intensive operation to call the mathematics function `pow()` from the `math.h` library for example.
Consequence of these optimizations is that the source code only consists of a minimal set of native arithmetic operations. Because of the mapping given in the subsection 4.1 Correctness, the arithmetic operations of the implementation corresponds to the arithmetic operations in the Algorithm 6 but differs in general because of the described saving of operations.
To determine the performance of the implementation, the amount of the **fl**oating point **op**erations (FLOP) in the code is related to time. This gives a criterion which enables to bench the code with other implementations. Another point of view is to determine the performance of the system where the code is executed. In this sense the code acts as a special benchmark tool similar to general ones like LINPACK.
Table 1 gives an overview of how much FLOP are used in each procedure per time step. Variable K_n refers to the number of iterations of the SOR method at time t_n. The table shows that the major part of the FLOP depends on the SOR iterations because of the factor K_n. Therefore the FLOP estimation $\mathcal{O}(K_n i_{\max} j_{\max})$ holds. To ensure the performance of the implementation is as expected, the GNU

Table 1: FLOP per procedure

Procedure	**FLOP**	**Cost**
`COMP_DELT`	$2(i_{\max}+2)(j_{\max}+2)+12$	$\mathcal{O}(i_{\max} j_{\max})$
`SETBCOND`	$4(i_{\max}+j_{\max})$	$\mathcal{O}(i_{\max}+j_{\max})$
`COMP_FG`	$7+53\Big((i_{\max}-1)j_{\max}+i_{\max}(j_{\max}-1)\Big)$	$\mathcal{O}(i_{\max} j_{\max})$
`COMP_RHS`	$3+6i_{\max} j_{\max}$	$\mathcal{O}(i_{\max} j_{\max})$
`POISSON`	$11+K_n(3+22i_{\max} j_{\max})$	$\mathcal{O}(K_n i_{\max} j_{\max})$
`ADAP_UV`	$2+3\Big((i_{\max}-1)j_{\max}+i_{\max}(j_{\max}-1)\Big)$	$\mathcal{O}(i_{\max} j_{\max})$

profiler `gprof` is used. Table 2 shows profile results of the lid-driven cavity problem. In standard case the fluid is contained in a square domain with no-slip boundary conditions on three sides and moving-boundary conditions on one side. Fluid parameters are $Re = 1000$ and velocity of moving boundary $\bar{u} = 1$. The discretization grid size is 128×128 and $t_{end} = 8$. As expected from Table 1, procedure `POISSON`

[43] This method *saves* arithmetic operations in both senses: it saves the value of the operations in memory and leads to a lesser amount of operations.
[44] Every value of the stencil is mapped to an auxiliary variable.

Table 2: Profiler results.

Procedure	CPU Seconds	Percentage
POISSON	251.03	96.51
COMP_FG	6.65	2.56
ADAP_UV	1.14	0.44
COMP_RHS	0.80	0.31
COMP_DELT	0.55	0.21
SETBCOND	0.03	0.01

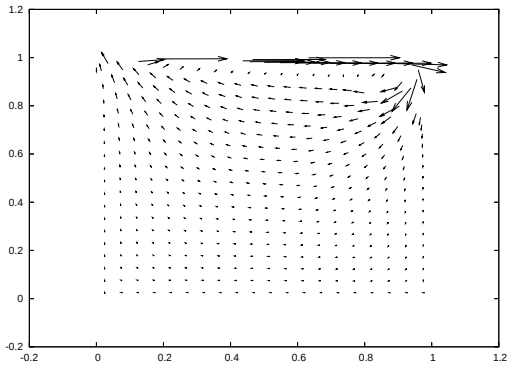
takes most of time[45] followed by COMP_FG and ADAP_UV. The CPU time proportions of the most busiest procedures also indicate the correctness of the implementation due to its performance.
At last Table 3 shows the real FLOPS performance at one core of an Intel Quad-core Xeon 2.0GHz processor with respect to different grid granularity.

Table 3: Intel Xeon 2.0GHz FLOPS.

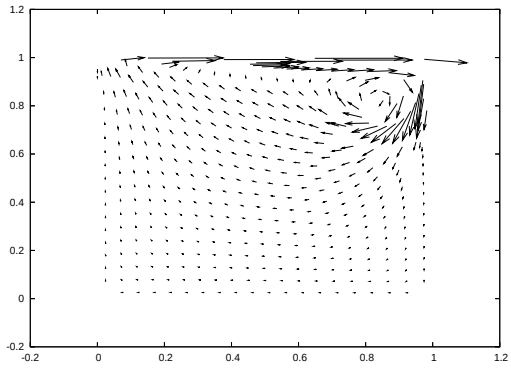
Grid Size	Mega FLOPS
$\mathbf{64 \times 64}$	380
$\mathbf{128 \times 128}$	389
$\mathbf{256 \times 256}$	383
$\mathbf{512 \times 512}$	386

4.3 Visualization

The command-line output during run-time shows some facts about the state of the execution. That is, the number of time iterations, Poisson iterations, elapsed time is displayed and the current MegaFLOPs. The resulting approximation of $\mathbf{v}$ is

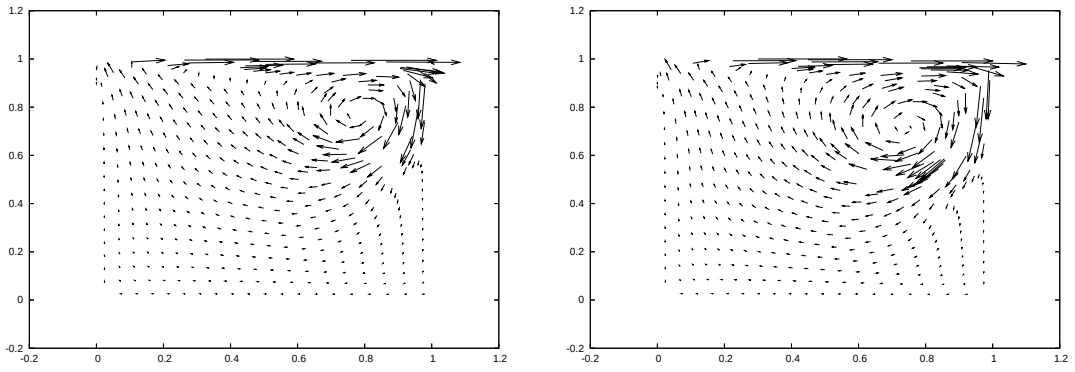


$t = 1$

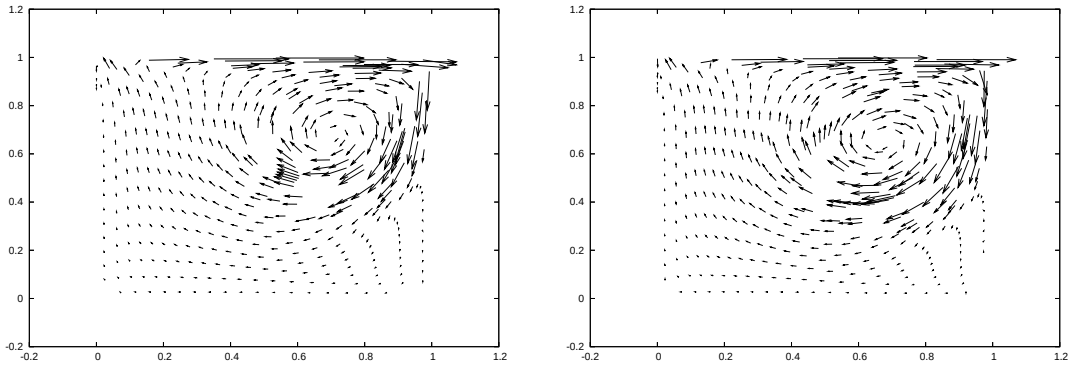


$t = 2$

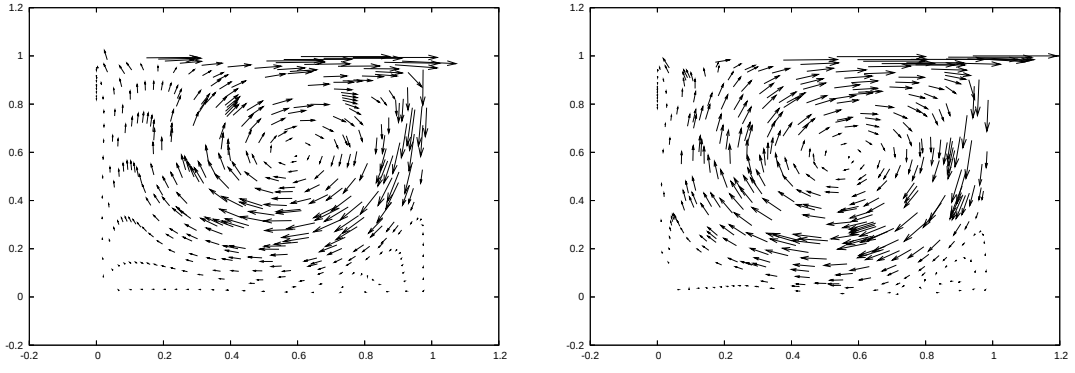
[45] According to Amdahl's law parallelizing of POISSON would lead to a maximal speedup of about 24.



$t = 3$ $t = 4$



$t = 5$ $t = 6$



$t = 10$ $t = 15$

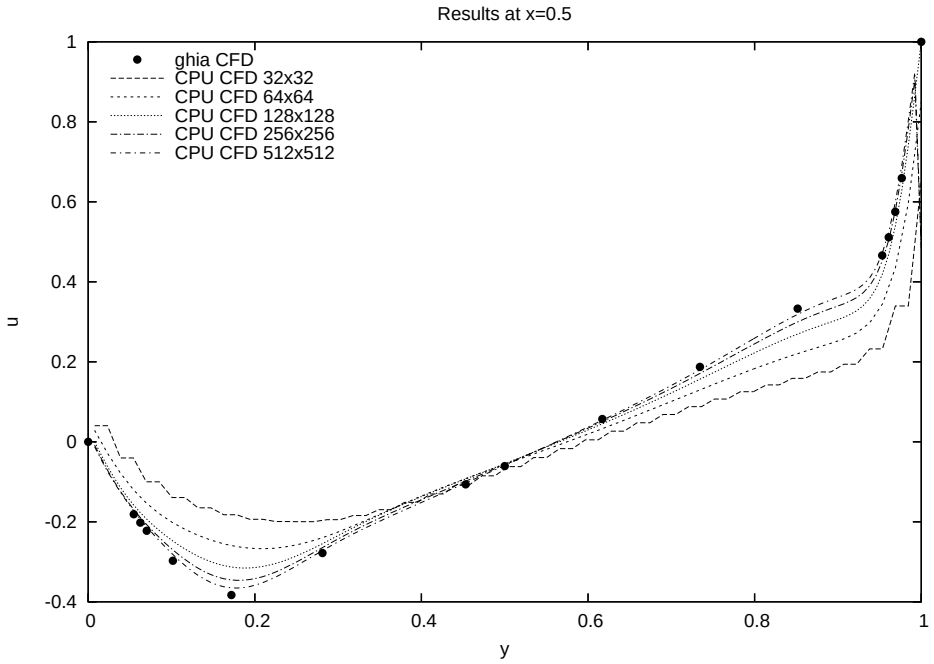
written in an output file. To view the velocity field in the GNU program `gnuplot` is used. The pictures above are also produced with this tool. They present the lid-driven cavity problem. The fluid parameters are $Re = 1000$, velocity of moving boundary $\bar{u} = 1$ and grid size of the area is 128×128. The results of this simulation coincide with the results of [Griebel 1995].

4.4 Validation

Up to here the CFD code works as expected. To proof that an implementation of fluid simulation behaves like a real fluid the CFD code is compared to a variety of popular test cases. These test cases are results of simple physical experiments

where the fluid behavior is measured as exact as possible. Comparison to these practical experiments gives information about the authentic nature of the CFD implementation and therefore is used to validate the code. This validation is done with many CFD codes and therefore a vast number of extended test cases (with respect to simple physical experiments) has grown which have a statistical degree of truth (*Wisdom of Crowds*). A good set of data for comparison is the data of

Figure 5: Validation of CFD CPU implementation



[Ghia 1982] since it includes tabulated results for various of Reynolds numbers. The only test case presented here is the very popular lid-driven cavity. Figure 5 shows the comparison between the results in [Ghia 1982] and the implementation of Algorithm 6. The fluid parameters are $Re = 1000$ and velocity of moving boundary $\bar{u} = 1$. The result shows the velocity u along the vertical line at $x = 0.5$. More grid granularity (of the implementation of Algorithm 8) leads to better approximation (of the results of [Ghia 1982]).

5 Domain Decomposition

Domain Decomposition methods provide a very natural way deriving parallel algorithms for the solution of linear systems. The idea is to decompose the domain Ω of a partial differential equation and after discretization one obtain a linear system which is also decomposed. It is solved on each domain which leads to computational advantage because solving many small systems is more cost effective than solving one large[46].
Depending of the decomposition method, the computations of the domains may does not depend on each other. This enables the according implementation to be parallelized. In other words: to solve a linear system of equations, an appropriate domain decomposition method has to be chosen.
The simplest form of such a method is the alternating Schwarz-method which is called multiplicative Schwarz-method if the grid points of their decomposed discrete domains match together in the overlapping regions.
Object of demonstration is the linear partial differential equation $Lp = f$ on the domain Ω with boundary conditions $u = b$ on $\partial\Omega$. For simplicity the domain Ω is decomposed in two overlapping sub-domains: Ω_1 and Ω_2. Their artificial boundaries Γ_1, Γ_2 are defined as follows: $\Gamma_1 := \partial\Omega_1 \cap \Omega_2$, $\Gamma_2 := \partial\Omega_2 \cap \Omega_1$. Solution p is also divided in parts p_1 and p_2 with respect of the domains. Same with f_1, f_2, b_1 and b_2.
The multiplicative Schwarz-method now works as follows: make an initial guess for values p_2^0 on Γ_1. Solve the system

$$\begin{aligned} Lp_1^n &= f_1 && \text{in} \quad \Omega_1, \\ p_1^n &= b_1 && \text{on} \quad \partial\Omega_1 \setminus \Gamma_1, \\ p_1^n &= p_2^{n-1} && \text{on} \quad \Gamma_1. \end{aligned}$$

Use p_1^n to solve

$$\begin{aligned} Lp_2^n &= f_2 && \text{in} \quad \Omega_2, \\ p_2^n &= b_2 && \text{on} \quad \partial\Omega_2 \setminus \Gamma_2, \\ p_2^n &= p_1^n && \text{on} \quad \Gamma_2. \end{aligned}$$

Writing the linear system for the discrete problem as $Au = f$, the two iteration steps are:

$$\begin{aligned} u^{n+1/2} &= u^n + \begin{pmatrix} A_{\Omega_1}^{-1} & 0 \\ 0 & 0 \end{pmatrix} (f - Au^n), \\ u^{n+1} &= u^{n+1/2} + \begin{pmatrix} 0 & 0 \\ 0 & A_{\Omega_2}^{-1} \end{pmatrix} (f - Au^{n+1/2}), \end{aligned}$$

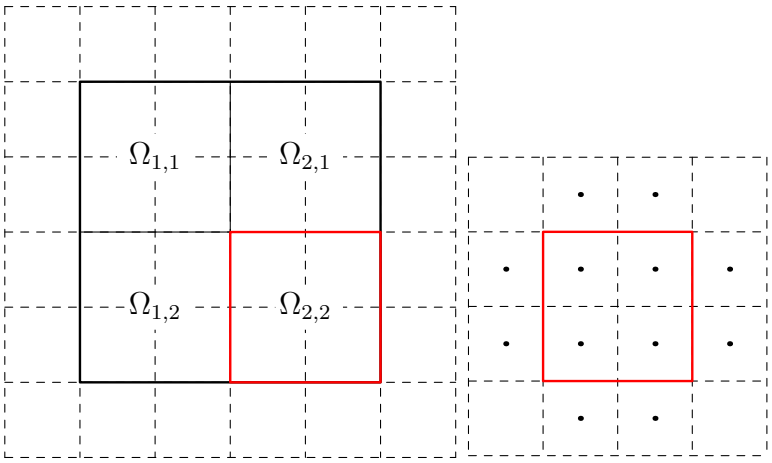
where A_{Ω_1} and A_{Ω_2} are discrete forms of the operator L restricted to Ω_1 and Ω_2, respective. This method can be considered as generalization of block Gauss-Seidel method. Since each step needs values from the other, this method is not usable for parallelizing.

[46] At least due to option of parallelizing.

A slightly modification gives the ability to compute concurrently: evaluation of the second step at u^n instead of $u^{n+1/2}$ and adding the two steps leads to

$$u^{n+1} = u^n + \begin{pmatrix} A_{\Omega_1}^{-1} & 0 \\ 0 & 0 \end{pmatrix} (f - Au^n) + \begin{pmatrix} 0 & 0 \\ 0 & A_{\Omega_2}^{-1} \end{pmatrix} (f - Au^n).$$

Calculation of the last two terms of the sum does not depend on each other and therefore can be done concurrently. As in the multiplicative algorithm, on the boundary Γ_1 the values of u_2 are used to compute the system. Same for Γ_2. This method can be considered as generalization of block Jacobi method.
The generalization to an arbitrary number of sub-domains works analogue.
For the purpose of solving the poisson equation over a rectangular domain of equidistant grid points it is sufficient to limit the following considerations to such conditions. Decomposition is done by dividing Ω with a equidistant grid with sub-domains called $\Omega_{l,m}$. This means minimal overlapping at the boundaries of each sub-domain. Figure 6 illustrates the partition (Figure 6a) and the pressure values which are needed to calculate SOR for a sub-domain (Figure 6b).



(a) Decomposition of four sub-domains. Dashed lines represent underlying discretization grid.

(b) Pressure values needed for SOR step in a sub-domain $\Omega_{l,m}$ represented as points.

Figure 6: Decomposition of Ω.

At each domain $\Omega_{l,m}$ the Poisson equation for pressure has to be solved in time step t_n. If a boundary of such a sub-domain touches the border of Ω, the corresponding boundary conditions are treated as usual. At borders which adjoins with another sub-domain, the boundary values are taken from that domain, according to the Schwarz-method. This means in time step t_n boundary values from time step t_{n-1} are used.
The discretization described in the subsection 3.2.3 SOR shows which values at the inner borders are needed for a domain $\Omega_{l,m}$ to compute solution of the linear subsystem. The pressure values at the borders has to be exchanged with their neighbors of the neighbored sub-domain.
Another strategy to obtain parallelizing is: Domain Ω is decomposed as above but

the pressure-boundary values of each sub-domain are exchanged in every iteration step of the SOR algorithm. Therefore in every time step the SOR only hast to be applied once at every sub-domain.
This method works because it can be developed from the Jacobi method of solving systems. If Jacobi is used to solve discrete Poisson on Ω, each element $p_{i,j}^{[k]}$ can be computed in parallel at iteration step k. Especially every sub-domain $\Omega_{l,m}$ can be computed parallel, no matter if parallelizing is done inside[47]. Applying Gauss-Seidel or SOR inside each sub-domain at iteration step k does not improve convergence rate in worst case but makes clear that this could be done without loosing convergence in general. Obviously convergence should be faster by applying SOR in sub-domains than Jacobi.
After each iteration step of SOR, the pressure values at the borders has to be exchanged with their neighbors of the neighbored sub-domain according to the Jacobian block method. The residuum is computed in each sub-domain and the norm of these residuum parts gives the main residuum as break-off criterion of SOR.
Algorithm 7 is based on Algorithm 6 and shows the summary of this approach.

Algorithm 7 Parallelizing Computational fluid dynamics

1: $t := 0, n := 0$, set initial conditions $U_{i,j}^{(0)}, V_{i,j}^{(0)}, p_{i,j}^{(0)}$
2: **while** $t < t_{\text{end}}$ **do**
3: set amplitude $h_t^{(n)}$ {equation (28)}
4: set boundary conditions for u and v
5: set $F_{i,j}^{(n)}, G_{i,j}^{(n)}$ {equations (21),(22) and boundary conditions (26), (25)}
6: set $RHS_{i,j}^{(n)}$ {defined in system (24)}
7: $k := 0$
8: **while** $k < k_{\max} \wedge ||r^{[k]}|| > eps$ **do**
9: apply kernel of SOR on every sub-domain $\Omega_{l,m}$ {algorithm 5}
10: set residuum part $r^{[k]}$ on every sub-domain $\Omega_{l,m}$ {equation (27)}
11: compute $||r^{[k]}||$ from residuum parts
12: $k := k + 1$
13: **end while**
14: set $U_{i,j}^{(n+1)}, V_{i,j}^{(n+1)}$ {equations (17),(18)}
15: $t := t + h_t^{(n)}, n := n + 1$
16: **end while**

[47] Again, Jacobi block method is derived.

6 GPU Implementation

This section modifies the Algorithm 7 to an algorithm which is implement-able on NVIDIA CUDA capable GPUs. Afterwards the conversion of the obtained algorithm into programming code is shown and its performance results are presented.

6.1 NVIDIA CUDA

The **C**ompute **U**nified **D**evice **A**rchitecture is a computing architecture developed by NVIDIA. This architecture enables **G**raphics **P**rocessing **U**nits to execute programming code.
Abstractly spoken GPUs are naturally designed to display many pixels on a screen and to change their colors very fast. To compute the color change an operation is performed which maps, for example, a three-dimensional environment in a game to the two-dimensional screen. To do so very fast the GPU tries to compute some pixel domains at the same time. Therefore GPUs are capable to perform many operations concurrently. CUDA allows to program these operations with a minimal set of extensions to C and hence abusing graphic processing capabilities to perform arbitrary computations on the GPU device. In [NVIDIA CUDA Programming Guide 2.0] this fact is described as: *"Driven by the insatiable market demand for realtime, high-definition 3D graphics, the programmable GPU has evolved into a highly parallel, multithreaded, manycore processor with tremendous computational horsepower and very high memory bandwidth..."*. A consequence is that the GPU is specialized in data processing instead of data caching and flow control as it is done in CPUs. This predestines GPUs for number crunching tasks.

GPU Hardware In order to write efficient code for GPUs it is necessary to understand the hardware architecture. A GPU consists of an array of multithreaded Streaming Multiprocessors. Each Multiprocessor has eight scalar processor cores, two special function units for transcendentals, a multithreaded instruction unit and on-chip **shared memory**. The Multiprocessor is responsible for scheduling many threads to its scalar processors. This is done with **warps** which are groups of 32 threads.
Figure 7 shows the hardware structure of a CUDA GPU device. All Multiprocessors have global memory access which is the slowest but largest form of memory on such GPUs. This kind of memory is accessible by all multiprocessors (and their threads) and by the host. The cost of this universal usability is bad performance compared to the following types of memory. Each Multiprocessor consist of a parallel data cache called **shared memory** which is shared by all scalar processors but is only accessible by threads within a block[48]. This kind of cache is much faster accessible by threads than global memory. By developing performance applications, one is well-advised to use this type of memory excessive. There are also registers available. Each scalar processor has a set of local 32-bit **registers**. Two memory type worth

[48]Blocks and Thread are keywords which are anticipated here and explained in detail in next paragraph.

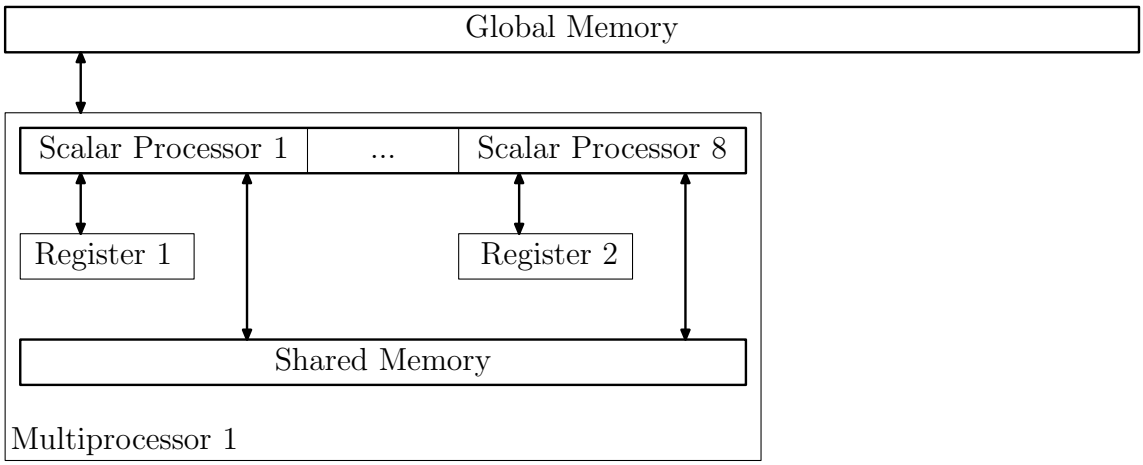
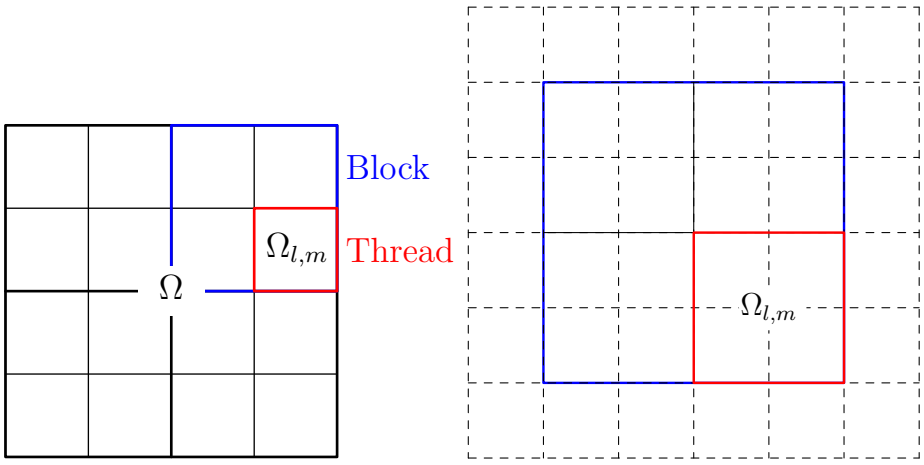


Figure 7: Hardware architecture of CUDA.

mentioning but not used further is constant and texture cache.

CUDA Software Model Programming procedures which has to be executed on the GPU are defined with a C extension and called **kernel**. Many of such kernels are executed concurrently. Each running instance is called **thread**. Threads are grouped into equal sized three-dimensional **blocks** which union is called **grid**.
The grid may be also understood as follows: threads are represented as cubes. Build a cuboid of threads - this is a block. Build a cuboid of blocks - this is the grid.
This structure of threads, blocks and grid is an abstraction model of the GPU hardware. Each block is assigned a multiprocessor which control unit divides the blocks into groups of 32 threads (called **warps**) and executes them.



(a) Ω decomposed in blocks and threads.

(b) Relation of grid and domain decomposition.

Figure 8: Grid, Blocks and Threads.

CUDA and SOR Each thread is responsible to calculate one step of the SOR iteration for a sub-domain $\Omega_{l,m}$.[49] The grid of the domain Ω is decomposed in equal sized rectangles and every rectangle consists of the same number of sub-domains. Each of these rectangles build up a CUDA block. A block (consisting of its threads) is therefore a collection of neighboring sub-domains which form a rectangular area. Figure 8a shows the decomposition of the grid of domain Ω in blocks and threads and Figure 8b clarify the relation to domain decomposition shown in Figure 6.
To improve efficiency the pressure values $P_{i,j}$ and $RHS_{i,j}$ needed for the blocks and its values out of block-borders are copied to shared memory (Figure 9). Since

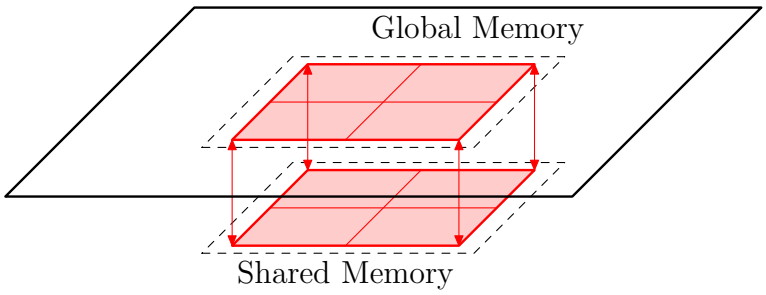


Figure 9: Copy between shared and global memory.

each block is executed (in general) at different multiprocessors and shared memory cannot be transferred between them the results of the block-borders have to be copied back to global memory to be readable by other blocks. Figure 10 shows the memory transfer from the border values of a block (rectangular area of sub-domains) to global memory. Neighbored sub-domains in neighbored blocks need these values

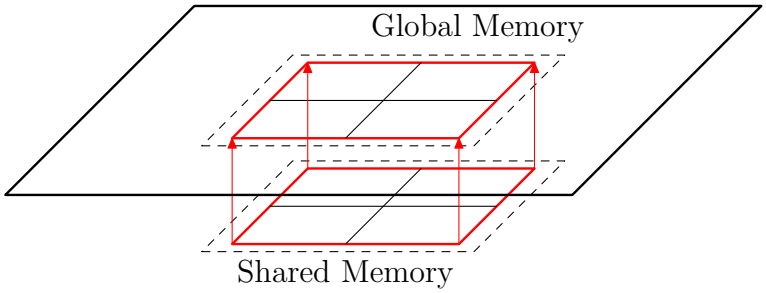


Figure 10: Copy block-borders from shared to global memory.

for next step of iteration[50], therefore the transfer of values out of block-borders from global to shared memory as illustrated in Figure 11 is also done for every block.
After completion of SOR iterations pressure values $P_{i,j}$ are copied back to global memory (Figure 9).
The calculation of the residuum is done by parallel computation of the residuum's in every sub-domain and is summed up in its blocks. These local residuum's are again added together to obtain the global residuum.
This leads to Algorithm 8.

[49] Sub-domains introduced in section 5 Domain Decomposition.
[50] Explained in section Domain Decomposition.

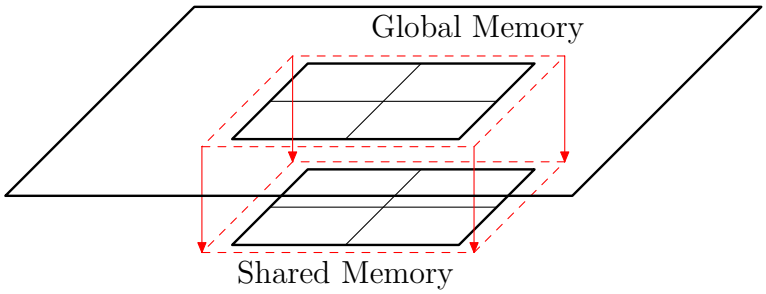


Figure 11: Copy out-of-border-values from global to shared memory.

Algorithm 8 CUDA Computational fluid dynamics.

1: $t := 0, n := 0$, set initial conditions $U_{i,j}^{(0)}, V_{i,j}^{(0)}, p_{i,j}^{(0)}$
2: **while** $t < t_{\text{end}}$ **do**
3: set amplitude $h_t^{(n)}$ {equation (28)}
4: set boundary conditions for u and v
5: set $F_{i,j}^{(n)}, G_{i,j}^{(n)}$ {equations (21),(22) and boundary conditions (26), (25)}
6: set $RHS_{i,j}^{(n)}$ {defined in system (24)}
7: copy $P_{i,j}, RHS_{i,j}$ from host memory to device global memory
8: copy $P_{i,j}, RHS_{i,j}$ from global to shared memory {figure 9}
9: copy values out of block-borders from global to shared memory {figure 11}
10: $k := 0$
11: **while** $k < k_{\max} \wedge ||r^{[k]}|| > eps$ **do**
12: apply kernel of SOR on every thread {algorithm 5}
13: copy block-borders from shared to global memory {figure 10}
14: copy values out of block-borders from global to shared memory {figure 11}
15: set residuum part $r^{[k]}$ on every thread {equation (27)}
16: compute $||r^{[k]}||$ from residuum parts
17: $k := k + 1$
18: **end while**
19: copy $P_{i,j}$ from shared to global memory {figure 9}
20: copy $P_{i,j}$ from device global memory to host memory
21: set $U_{i,j}^{(n+1)}, V_{i,j}^{(n+1)}$ {equations (17),(18)}
22: $t := t + h_t^{(n)}, n := n + 1$
23: **end while**

6.2 Implementation

As in the corresponding section for the CPU code the paradigm *Make good, make run, make fast!* is used to structure the following subsections into Correctness, Profiling, Visualization and Validation.
The source code is appended on a mini CD which also includes a compiled version for x86_64 PC architectures with NVIDIA G200 chip.

6.2.1 Correctness

Again a one-to-one mapping of Algorithm 8 to the source code is given to show the correctness of the implementation. The implementation of each line of Algorithm 8 is explained in detail below. Numbering refers to the line numbering of the algorithm and therefore provides a mapping-like unique correspondence of implementation and algorithm.

[Line 1-6] No difference to Algorithm 6 and therefore the correctness holds as it is shown in the CPU version.

[Line 7-20] The implementation of the Poisson pressure equation is the procedure `POISSON_CUDA`. It contains the definition of the CUDA kernel `SOR` which is a procedure that runs on the GPU. This kernel launches the blocks and threads on the GPU according to execution configuration, which is determined from an external text file. The `SOR` kernel contains [Line 8-19] of Algorithm 8. Copy instructions from host to device of array `P` and `RHS` correspond to [Line 7] and vice versa for `P` to [Line 20]. These instructions are initiated before and after kernel execution.
The shared-memory parts of `P` and `RHS` (corresponding to their block) are stored in an coherent array[51] `shared[]`.

[Line 21] No difference to [Line 13] in Algorithm 6 and therefore the correctness holds as it is shown in the CPU version.

6.2.2 Profiling

Because parallelizing affects only the SOR algorithm, code around the `POISSON_CUDA` procedure is equivalent to the serial implementation. The host CPU performance optimizations are therefore the same and this subsection is focused on the performance of `POISSON_CUDA` procedure and `SOR` kernel. Therefore the program was expanded to a benchmark mode in which it prints the FLOPS of the pure kernel execution to standard output.
To achieve high performance in CUDA GPU computing it is necessary to follow the performance guidelines in [NVIDIA CUDA Programming Guide 2.0]. This turned out to be the real challenge in programming with CUDA.
Results were achieved using a NVIDIA GeForce GTX260 GPU which is of compute capability 1.3.

Unrolling is a very effective kind of optimization and means the unrolling of loops. This is done by explicitly serializing code without using loops. Because every subdomain consists of a number of grid elements the SOR algorithm has to iterate (loop) about every single element. After unrolling, every thread iterates about its grid elements.
In the source code non-unrolling is done in the following CUDA kernels: SOR_00

[51] According to [NVIDIA CUDA Programming Guide 2.0] exactly one array can be allocated dynamically as shared memory on the GPU.

and SOR_0. All the other kernels are unrolled and they differ in their optimization goals (introduced in next paragraphs).

Example: Non-unrolling kernel for a sub-domain of 2×2:[52]

```
//copy global to shared memory
//south
for (int i=0; i<2; i++) {
  shared[i+vars]=*((float *)((char *)P_d+vary)+i+varx);
}
```

Unrolling for the same sub-domain gives:

```
//copy global to shared memory
//south
shared[0+vars]=*((float *)((char *)P_d+vary)+0+varx);
shared[1+vars]=*((float *)((char *)P_d+vary)+1+varx);
```

In the source code unrolling is done for subdomains up to 4×4.

Results of unrolling all loops is shown in table 4. The simulation results were ob-

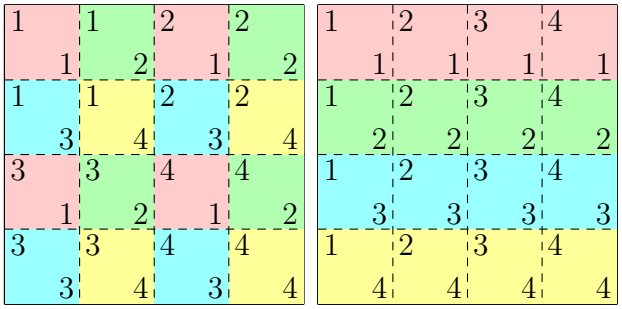
Table 4: Mega FLOPS unrolling.

Sub-domain size	**FLOPS standard**	**FLOPS unrolling**	**Speedup**
$\mathbf{1 \times 1}$	$4,902$	$19,969$	4.074
$\mathbf{2 \times 2}$	$11,746$	$25,428$	2.165
$\mathbf{4 \times 4}$	$12,136$	$19,630$	1.618

tained by suppressing residuum calculation. This is done because interrupting the `while`-loop before iteration `itermax` leads to significant bad performance. Probably the CUDA compiler make large optimizations by unrolling the loop if reaching the end is certain. Ending all `while`-loops in the algorithm at `itermax` does trivially not effect correctness of calculation results but takes more time. This is in the case of GPU implementation not very weighty because the GPU calculated part is much faster than CPU part of the program but is respected later in the comparison of both version.

Coalescing describes coordinated global memory access which is done by warps. Technically spoken, optimization by coalescing is achieved if a warp reads a contiguous global memory region. In case of floating point data types this memory region is 128 bytes which is the product of float size and warp length: $4\text{byte} \times \ 32 = 128\text{byte}$. This also implies that the first read of a thread in memory region has to start at a multiple of 128. Because the fluid discretization grid is a power of two but the border adds to this number, an offset of 32 is introduced to the number of columns from `P` to achieve coalescing.
Figure 12 shows the sub-array[53] of a block with four sub-domains. The Upper left

[52] The `var` variables are minor with respect to unrolling.
[53] The sub-array of the array `P` is meant, the value of which the algorithm is *looking for*.



(a) Reads from default sub-array P (b) Reads from rearranged sub-array P

Figure 12: Coalesced read from Sub-domains of a block.

indexing refers to thread number (equal to sub-domain) while lower right indexing refers to successive read access within threads. Same colors belong to concurrently running threads. Because succeeding threads do not read from succeeding memory addresses in figure 12a this access pattern is not coalesced. By rearranging the array elements like shown in figure 12b, memory reads are coalesced. Same colors mean concurrent execution.
In the source code coalescing (and unrolling) is done in the following CUDA kernels: SOR_22_COAL and SOR_44_COAL.

Example: Non-coalescing kernel for a sub-domain of 2×2:[54]

```
//copy global to shared memory
//south
shared[0+vars]=*((float *)((char *)P_d+vary)+0+idx*2);
shared[1+vars]=*((float *)((char *)P_d+vary)+1+idx*2);
```

The global thread-index `idx` is multiplied with 2 and therefore memory reads cannot be successive addresses. Coalescing for the same sub-domain gives:

```
//copy global to shared memory
//south
shared[0+vars]=*((float *)((char *)P_d+vary)+\
               (blockIdx.x*2+0)*blockDim.x+threadIdx.x);
shared[1+vars]=*((float *)((char *)P_d+vary)+\
               (blockIdx.x*2+1)*blockDim.x+threadIdx.x);
```

The rearranging gives successive read access to global memory per block through thread Index `threadIdx.x`.

Table 5 shows the benchmark results of a non-coalesced kernel and a coalesced one. Like expected, in the 1×1-case coalescing does not have an effect because trivial rearranging.

Occupancy is a dimensionless number which gives information about usage rate of a multiprocessor. It is calculated by the number of warps running concurrently

[54]The `var` variables are minor with respect to coalescing.

Table 5: Mega FLOPS coalescing.

Sub-domain size	FLOPS unrolling	FLOPS coalescing	Speedup	Total speedup
$\mathbf{1 \times 1}$	19,969	19,969	1.000	4.074
$\mathbf{2 \times 2}$	25,428	28,270	1.112	2.407
$\mathbf{4 \times 4}$	19,630	23,831	1.214	1.964

on a multiprocessor divided by maximum number of warps that can run concurrently. While high latency thread instructions executed sequentially, executing other warps is the only way to hide latencies and keep the hardware busy. This means low-occupancy multiprocessors cannot adequately hide latency on memory-bound kernels.
Achieving maximum occupancy is done by following: Firstly, all multiprocessors have at least one block to execute (better is many blocks run in a multiprocessor). Secondly, source's of a block is at most half of available. This affects mainly shared memory and registers.

Example: First goal is done by a granular discretization in place: grid size 512×512. The sub-domain size 1×1 is chosen. Shared memory is needed for array `P` and `RHS` limited on the sub-domains of the block:

$$2 \cdot \texttt{BLOCKSIZE\_X} \cdot \texttt{BLOCKSIZE\_Y} \cdot \texttt{sizeof(float)} \leq \texttt{sharedmemory\_size}/2 := 8\text{KB}/2$$

The test-system consists of an NVIDIA GTX260 GPU which has 24 multiprocessors with 8KB shared memory each. Because `BLOCKSIZE_X` have to be a multiple of 16 (for coalescing reasons) `BLOCKSIZE_Y` is given by 16 since lower equal is respected and only power-of-two values are considered.

For sub-domains greater than 1×1 `sharedmemory_size` hast to multiplied with sub-domain height and width.
Table 6 shows benchmark results for varying sub-domain and block sizes. In conclu-

Table 6: Mega FLOPS occupancy.

		Block size						
		$\mathbf{8 \times 4}$	$\mathbf{8 \times 8}$	$\mathbf{16 \times 1}$	$\mathbf{16 \times 2}$	$\mathbf{16 \times 4}$	$\mathbf{32 \times 1}$	$\mathbf{32 \times 2}$
Sub-domain size	$\mathbf{1 \times 1}$	19,109	23,753	11,567	19,791	28,243	19,901	28,807
	$\mathbf{2 \times 2}$	29,666	39,939	16,440	28,050	42,035	28,243	41,913
	$\mathbf{4 \times 4}$	24,966	19,969	19,084	23,871	18,896	18,749	18,896

sion best performance is obtained by choosing the sub-domain size 2×2 and block size 16×4 or 32×2.
At last, table 7 shows the real FLOPS performance of combined GPU and CPU power with respect to different grid granularity on types of GPUs. Test case of this table is the lid driven cavity with fluid parameters $Re = 1000$ and velocity of moving boundary $\bar{u} = 1$.

Table 7: CUDA accelerated CFD FLOPS.

GeForce GTX260	
Grid size	**Mega FLOPS**
$\mathbf{64 \times 64}$	$5,035$
$\mathbf{128 \times 128}$	$7,021$
$\mathbf{256 \times 256}$	$7,190$
$\mathbf{512 \times 512}$	$7,002$
$\mathbf{1024 \times 1024}$	$6,847$

GeForce 9800GX2	
Grid size	**Mega FLOPS**
$\mathbf{64 \times 64}$	$2,693$
$\mathbf{128 \times 128}$	$4,093$
$\mathbf{256 \times 256}$	$4,201$
$\mathbf{512 \times 512}$	$4,150$

GeForce 8600GTS	
Grid size	**Mega FLOPS**
$\mathbf{64 \times 64}$	$2,737$
$\mathbf{128 \times 128}$	$3,365$
$\mathbf{256 \times 256}$	$3,496$
$\mathbf{512 \times 512}$	$3,364$

Though theoretically peak FLOPS power of GeForce GTX260 and 9800GX2 are nearly equivalent, GTX260 wins the challenge because optimizations above are done for devices of compute capability 1.3 whereas 9800GX2 is of capability 1.1.
The peak performance of the implementation running on GPU is about 7 Giga FLOPS.

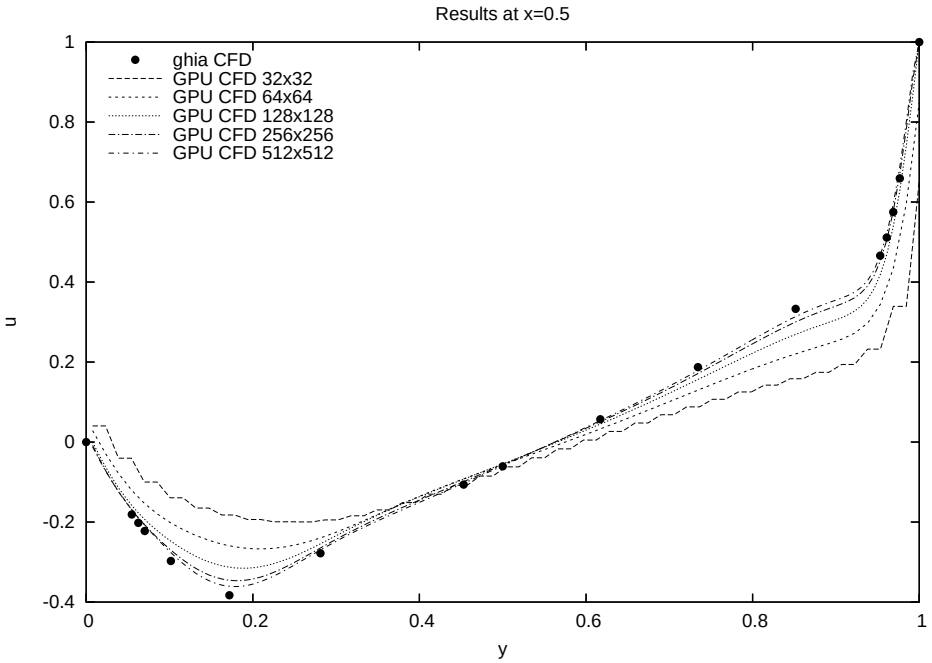
6.2.3 Visualization

No visual differences to CPU version since correctness has been proven. GPU implementation is faster in time which is proportional to speedup factor of CPU versus GPU version.

6.2.4 Validation

Again the results are compared with [Ghia 1982] to have a statistical degree of truth of the implementation. Figure 13 shows the comparison between the results in [Ghia 1982] and the implementation of algorithm 8 with different grid resolutions. Fluid parameters are $Re = 1000$ and velocity of moving boundary $\bar{u} = 1$. The results show the velocity u along the vertical line at $x = 0.5$. As in Section 4 CPU Implementation more grid granularity (of the implementation of Algorithm 8) leads to better approximation (of the results of [Ghia 1982]). Comparison with Figure 5 in subsection 4.4 Validation shows that the CPU approximation is better than GPU at the same grid resolution. This is due to the fact that the GPU version is a mix of Jacobi and Gauss-Seidel while CPU is Gauss-Seidel-only (see section 5 Domain Decomposition).

Figure 13: Validation of CFD GPU implementation.



6.3 Comparison of CPU and GPU

Since the GPU version does not make use of the residuum calculation due to CUDA optimization restrictions, comparison of CPU and GPU CFD implementation firstly is done by FLOPS. Subsection 4.2 Profiling gives 400 MegaFLOPS for CPU and above measurements 7,000 MegaFLOPS for GPU. This leads to a speedup of up to 18 which seems to be realistic with respect to Amdahl's Law which postulates a maximal speedup of 24.
Second comparison is done by measurement of the time until the residuum drops below a fixed limit[55] depending on the grid-size. Table 8 shows this comparison and the speedup. Low speedup at the rough grid resolution 64×64 is due to

Table 8: CFD comparison.

		Time in seconds		**Total**
		CPU	**GPU**	**speedup**
Grid size	$\mathbf{64 \times 64}$	74	18	4.1
	$\mathbf{128 \times 128}$	905	119	7.6
	$\mathbf{256 \times 256}$	10,507	1,812	5.8

less arithmetic density in the kernel which means memory operations cannot be hidden and became weigthy (cache latency overwhelms). The table shows also that speedup shrinks if the grid resolution gets higher. The CPU version with grid size 128×128 does only 37% of the SOR iterations with respect to GPU thanks to residuum calculation. With grid size 256×256 the CPU even needs 28% of the SOR

[55] This limit is detected with the CPU version since the GPU is not able to determine the residuum.

iterations than GPU. This explains the shrink of speedup while increasing grid size. In this constellation the GPU advantage is the best if grid size is 128×128 which leads to a speedup of almost 8 of the lid-driven cavity problem.

7 Conclusion

Reflecting the summary from where this work started is done in the following while highlighting its results.
Section Introduction has localized the scope of the diploma thesis and motivated the study of continuum theory. This was done in section Derivation of Navier-Stokes equations for the incompressible two dimensional case, resulting in conservation of momentum (Equations (14), (15)) and conservation of mass (Equation (16)) which are used subsequently.
The Numerical Approach has shown the discretization of these equations in time (Equations (17), (18)) in order to obtain the velocity field in the next time step. The time discretization keeps the nonlinear character of Navier-Stokes whereas discretization in place leads to linear Poisson-equation for pressure whose corresponding linear system (24) is solved with the SOR method. This has produced the CFD Algorithm 6.
The section CPU Implementation has proved the correctness of the CPU implementation with respect to the derived CFD algorithm and gives a statistical degree of truth of the simulation results in comparison to existing CFD implementations. Benchmark results attained an average performance of 400 Mega FLOPS.
Domain Decomposition methods have given the theoretical background and the section derived an Algorithm (7) which is parallelized and based on the algorithm used before.
The central section GPU Implementation has introduced the CUDA concept of programming NVIDIA GPUs and presented an algorithm (8) which benefits of this concept. Again, the correctness has been proven and validation has been done. Optimization methods were explained in detail and results in a peak performance of more than 40 Giga FLOPS for the parallelized SOR kernel. The total performance of the algorithm which makes use of both CPU and GPU is measured with 7 Giga FLOPS. Comparison of CPU and GPU CFD implementations shows an effective speedup of almost 20 with ideal configuration options.
The present work has shown that the computation of the Navier-stokes equations can be accelerated significantly by using GPUs. The speedup factor is at least 20 which has been shown by the simulations. It is the result of parallelizing the SOR kernel. More speedup could certainly be achieved if more parts of the CFD algorithm get parallelized.
Because CUDA is a very young project and is at present in an early stage of development, the adaption of CFD algorithms to use efficiently with CUDA is not yet easy to implement. This is because many restrictions have to be paid attention to in order to get a high-performance code. One would say: At present CUDA is rather looking for suitable applications than applications are suitable to CUDA.

References

[Griebel 1995] *M.Griebel, T.Dornseifer, T.Neunhoeffer 1995: Numerische Simulation in der Strömungsmechanik, Vieweg.*

[Chorin 1968] *A.Chorin 1968: Numerical solution of the Navier-Stokes-equations, Math.Comput.22 745-762.*

[Temam 1969] *R.Temam 1969: Sur l'approximation de la solution des equations de Navier-Stokes par la methode des pas fractionnaires, Arch.Ration.Mech.Anal.32 135-153.*

[Hirt 1975] *C.Hirt, B.Nichols, N.Romero 1975: SOLA - A Numerical Solution Algorithm for Transient Fluid Flows, Technical report Los Alamos Scientific Lab.Rep.LA-5852.*

[Grossmann Roos 2005] *C.Großmann, H.G.Roos 2005: Numerische Behandlung partieller Differentialgleichungen, Teubner.*

[Temam 1993] *R.Temam 1983: Navier-Stokes Equations and Nonlinear Functional Analysis, Society for Industrial and Applied Mathematics.*

[Sohr 2001] *H.Sohr 2001: The Navier-Stokes Equations An Elementary Analytical Approach, Birkhäuser.*

[Tveito 2002] *A.Tveito, R.Winther 2002: Einführung in partielle Differentialgleichungen, Springer.*

[Griebel Zumbusch 2004] *M.Griebel, S.Knapek, G.Zumbusch, A.Caglar 2004: Numerische Simulation in der Moleküldynamik, Springer.*

[Wolf-Gladrow 2000] *D.Wolf-Gladrow 2000: Lattice-Gas Cellular Automata and Lattice Boltzmann Models, Springer.*

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

[NVIDIA CUDA Programming Guide 2.0] *NVIDIA CUDA Programming Guide 2.0.*

Selbstständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.