DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

GPU-Accelerated Pressure Solver for Deep Learning

Martin Vincent Wepner

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

GPU-Accelerated Pressure Solver for Deep Learning

GPU-Beschleunigter Druck Löser für Deep Learning

Author:	Martin Vincent Wepner
Supervisor:	Prof. Dr. Nils Thürey
Advisor:	M. Sc. Philipp Holl
Submission Date:	15. March 2019

I confirm that this bachelor's thesis in informatics: games engineering is my own work and I have documented all sources and material used.

Munich, 15. March 2019   Martin Vincent Wepner

Acknowledgments

First and most importantly, I want to thank my advisor Philipp Holl. In the past four months, I could always count on his assistance and valuable feedback. The weekly meetings helped me a lot; if I had problems at home, I usually got a detailed answer via mail within minutes. Without him, this thesis would not have been possible. Again, thank you very much Philipp for your support, dedication and understanding in the last few months!

I would also like to thank my supervisor Prof. Dr. Nils Thürey who gave me the opportunity to write this thesis at the Chair of Computer Graphics and Visualization. Many thanks for the personal meetings and interesting conversations. I have learned an incredible amount about GPU programming in this short time, which I will continue to utilize beyond this thesis.

Before I started my studies at TUM, I already knew that my thesis was supposed to be in the field of physics simulations. Therefore, I would like to thank my math and physics teacher Dr. Clemens Koerdt, who introduced us to numerical n-body gravitation simulations in addition to his lessons, which aroused my interest in this topic.

I also want to thank my Swabian fellow students and friends Christian, Yannick, Tobias and Karl for making the last 4.5 years in Munich very enjoyable! *Thanks* you.

Last but not least I have to thank my family for the encouragement through my studies. None of this would have been possible without them. I especially want to thank my dad and brother who motivated and supported me in a very challenging phase of my life during my studies.

Abstract

Simulating incompressible fluids requires numerically solving the Navier-Stokes equations. An integral part of many commonly used methods achieving this is solving a system of linear equations to obtain the pressure.

In recent years, many machine learning and optimization methods involving fluids have been proposed, ranging from learned approximations of fluid behavior to optimal control. Φ_{Flow} is a fully differentiable fluid simulation toolkit based on the machine learning framework TensorFlow. This enables both classical optimization of simulation properties as well as the training of neural networks which predict inputs or control parameters of a simulation. Its pressure solver is implemented in TensorFlow and thus capable of running on CPUs and GPUs. However, TensorFlow's dataflow graph causes considerable computational overhead, which slows down the simulation and hence the training of neural networks, especially on the GPU.

This thesis addresses this performance problem by presenting a TensorFlow custom op that solves the pressure efficiently on GPUs. A tailored version of the widely used compressed sparse row matrix format is introduced, which efficiently stores the Laplace matrix of the pressure to minimize memory access.

My solution outperforms Φ_{Flow}'s native TensorFlow implementation by a factor of up to two orders of magnitudes compared to the CPU version and is up to 70 times faster than the GPU version on my test environment while producing the same results.

Contents

1 Introduction

The simulation of fluids is part of our daily life, even if many do not have this in mind. It facilitates the planning phase of vehicles, airplanes and boats by allowing the aerodynamic properties to be determined even before the first prototype has been built [Tsu+09]. A weather forecast as precise as we know it today would not be possible if we could not simulate the air flows of the earth [Kim02]. And the simulation of fluids also plays a role in medicine, for example, to better understand blood flow in the human heart [Pes77]. Many designers and artists are working to immerse the audience in games and movies as much as possible [Gil09]. However, compared to the design for instance of 3D objects, it is very complex and time-consuming to create realistic looking artificial waves with all the splashes and foam by hand [Gil09].

Existing solutions are often very computationally intensive and lead to compromises between speed and accuracy, resolution or size and thus to the believability of the simulation. Especially in games you only have a few milliseconds until the result has to be delivered. Only the immense parallel computing power of graphics cards made it possible to compute dense fluids in real-time.

Deep Learning has become state-of-the-art in many areas such as object recognition in recent years [LBH15]. Deep neural networks also appear to be capable of realistically mimicking physical phenomena with the advantage of doing so in a fraction of the time required by conventional solutions [Tom+17] [Thu+18].

1.1 Φ_{Flow}

Φ_{Flow} is a toolkit written in Python and currently under development at the TUM Chair of Computer Graphics and Visualization for solving and visualizing n-dimensional fluid simulations. It focuses on keeping every simulation step differentiable which makes it suitable for backpropagation. Backpropagation is required to calculate the gradient of the loss function of neural networks to adjust the weights of the network. Being developed on top of TensorFlow, it is not only capable of running on CPUs, but also completely on GPUs. TensorFlow simplifies the implementation of machine learning algorithms including the use of neural networks for predictions and classification. This makes Φ_{Flow} a powerful tool to train neural networks on fluid simulations and also visualize them by its interactive GUI running in the browser.

1.2 Definition of the problem

This work proposes an implementation to solve the pressure equation - the most computationally intensive part of Eulerian fluid simulations. Being as efficient as possible is crucial for fast simulations and thus most importantly for the training of neural networks. Φ_{Flow} already comes with its own pressure solver running directly in TensorFlow. However, TensorFlows dataflow graph causes considerable computational overhead, which slows down the simulation and hence the training of neural networks, especially on the GPU.

TensorFlow offers the possibility to write custom operations ("custom ops") in C++ to tackle this issue: Writing efficient code to solve a specific problem within the dataflow graph. Furthermore, the CUDA API by Nvidia makes it possible to write low-level CUDA code that runs natively on GPUs. Combining custom ops and CUDA kernels lead to a solution that is both efficient and transparent and also compliant to TensorFlow's Tensors which is required within Φ_{Flow}. It can then be compared to the built-in solution of Φ_{Flow}.

1.3 Outline

Chapter 2 discusses past research in this area.
Chapter 3 repeats the theoretical and mathematical background necessary for this work.
In chapter 4 I present how I improved the Pressure Solve on GPUs.
In chapter 5 I show the results and compare them with those from Φ_{Flow}.
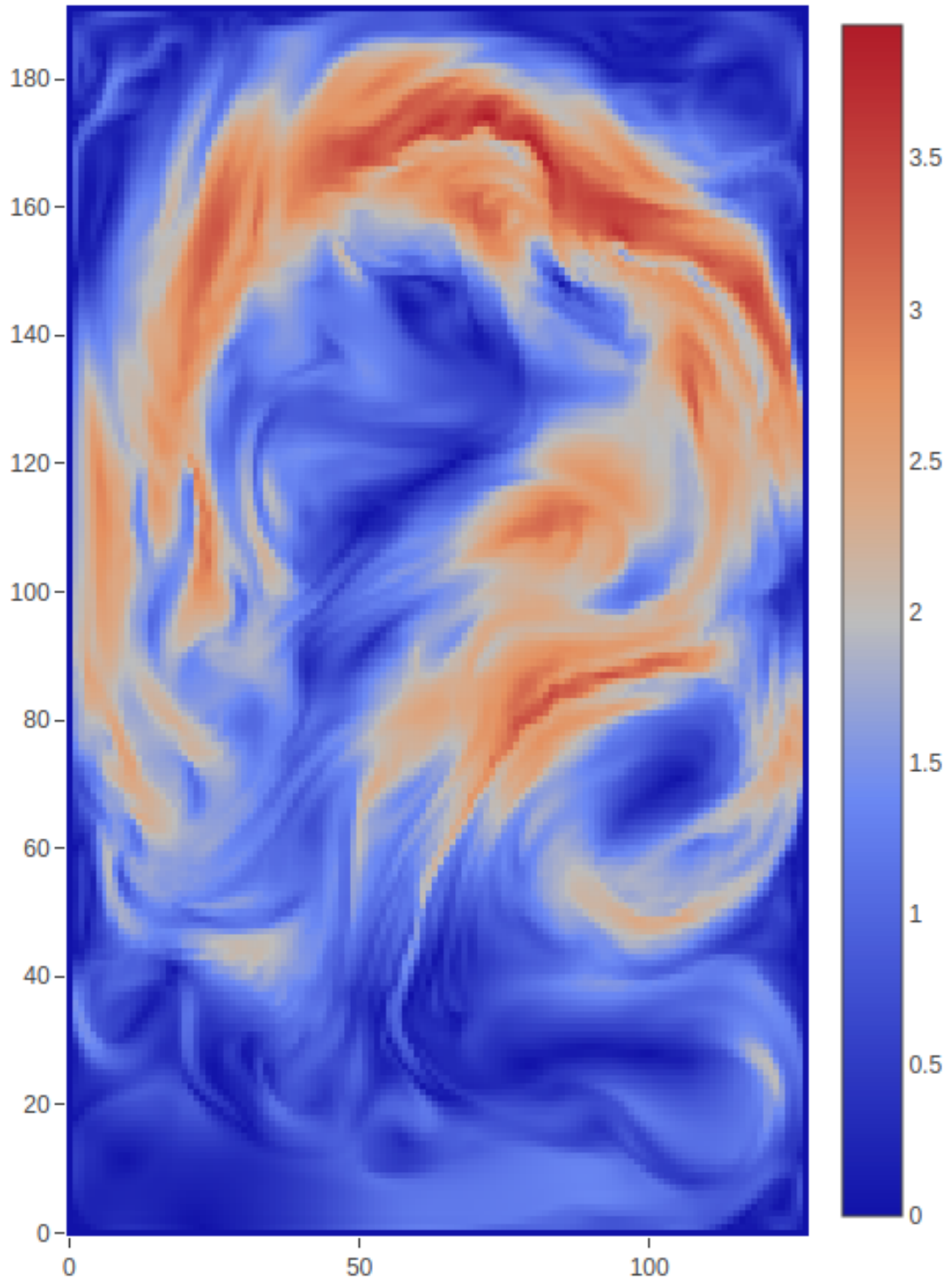Chapter 6 summarizes this work and gives suggestions for further work.

Figure 1.1: Grid-based fluid simulation in Φ_{flow}

2 Related Work

Computational Fluid Dynamics (CFD) has been a research topic long before the first computers appeared. Based on Newton's Second Law, Leonard Euler developed a mathematical model to describe the flow of frictionless inviscid fluids by a set of differential equations in 1757 [Eul57]. Claude Navier and George Stokes extended the Euler Equations by a viscosity term and formulated the Navier-Stokes-Equations [Nav27] [Sto80]:

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho} \nabla p = \vec{F} + \nu \nabla \cdot \nabla \vec{u}$$

$$\nabla \cdot u = 0$$

In computer graphics two distinctive approaches have established to approximate the Navier-Stokes-Equations numerically:

Lagrangian methods use particles to represent a fluid. The Smoothed Particle Hydrodynamics (SPH) method was presented by Lucy in 1977 to simulate astrophysical problems [Luc77]. In 1992 Monaghan applied the SPH method to fluid dynamics by approximating continuous velocity flow-fields of the Navier-Stokes-Equations using particles with discrete positions and velocities [Mon92]. SPH particles are advected by exerting pressure, viscosity and/or surface tension forces on each other using smoothing kernels [MCG03]. However, it did not fulfill the incompressibility constraint, which can cause undesired visual artifacts. Becker and Teschner presented the weakly compressible SPH system (WCSPH) to maintain incompressibility at the cost of limiting the time-step to impractical sizes [BT07]. Macklin and Mueller introduced Position Based Fluids (PBF), a method which first advects particles and then corrects its positions to satisfy the incompressibility conditions iteratively [MM13]. Several improvements had been made since then and it is still an active topic in research [Mor+].

The alternative to particle-based approaches is the Eulerian viewpoint that splits continuous quantities on a discrete, regular grid. First described by Harlow and Welch in 1965 [HW65], they presented the Marker-and-Cell (MAC) grid to prevent biased or null-spaced central differences when advecting velocities. Incompressibility in grid-based approaches is enforced by solving the pressure Poisson equation with a divergence-free velocity field. The pressure equation consists of a symmetric, semi-definite matrix so the Conjugate Gradient method can solve this linear system [HS52].
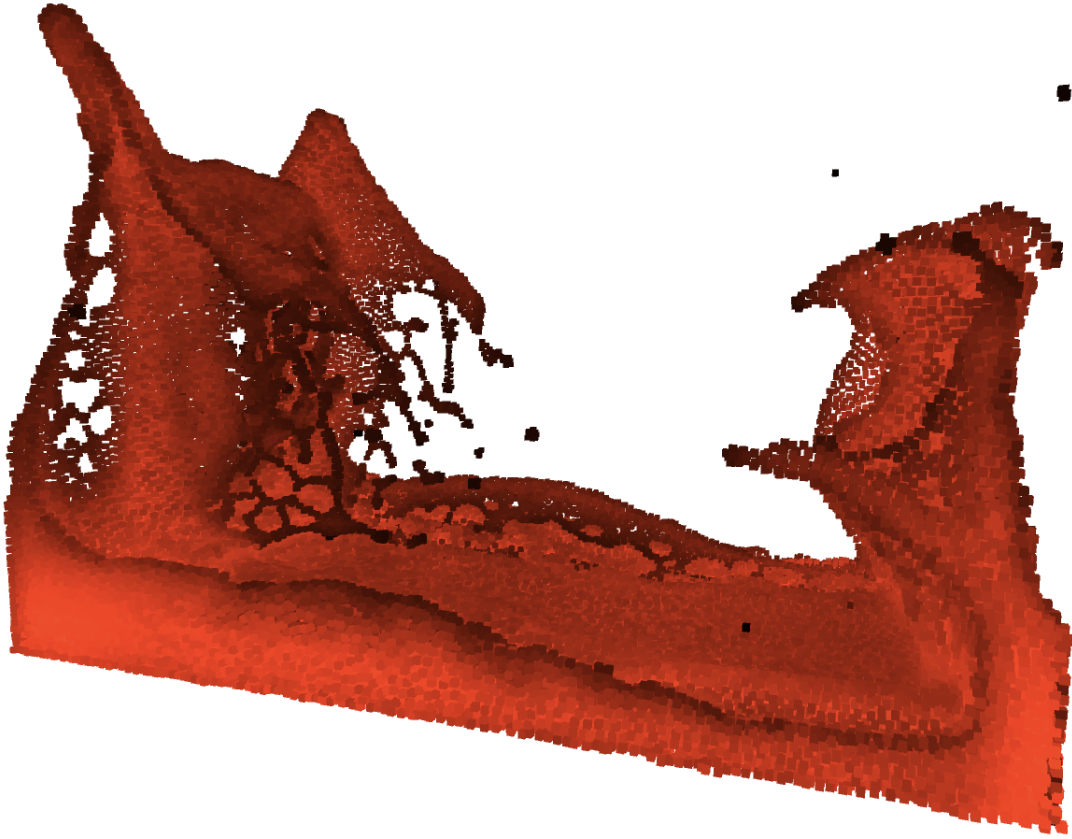
Figure 2.1: Particle-based fluid simulation in Unity3D [Tec19] [Wep17]

In recent years Deep Learning emerged in many fields of research because GPUs far surpassed the computational capabilities needed to train neural networks of multicore CPUs [RMN09]. NNs learn patterns in large data sets by using the backpropagation algorithm and dramatically improved the state-of-the-art in speech recognition, visual object recognition, object detection and many other domains [LBH15]. Furthermore several recent papers have targeted the use of NNs to mimic physical phenomena like rigid-body simulations [Cha+16] and also fluid dynamics [Tom+17][CT17][SF18]. Those approaches benefit from the fact, that the evaluation and thus simulation of those NNs are much faster than the conventional mathematical approaches.

The Adjoint method calculates the gradient of a function numerically by solving the transpose linear system for, inter alia, backpropagation [Lio71]. The pressure equation consists of a symmetric matrix vector multiplication. Therefore, the forward computations are identical to the adjoint of the pressure equation [McN+04].

3 Numerical approaches for simulating fluids

Incompressible fluids are described by the Navier Stokes equations [Nav27] [Sto80]:

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho} \nabla p = \vec{F} + \nu \nabla \cdot \nabla \vec{u} \qquad (3.1)$$

$$\nabla \cdot u = 0 \qquad (3.2)$$

Here, $\vec{u}$ denotes the velocity vector of the fluid.
ρ stands for the density of the fluid. E.g. $\rho_{Water} \approx 1000 kg/m^3$.
p, on the other hand, holds the pressure, which is the force per unit the fluid exerts on its surroundings.
$\vec{F}$ denotes external forces such as gravity and buoyancy.
ν is called kinematic viscosity. The higher the viscosity of a fluid is the higher its inherent inertia. For example, honey has a higher viscosity than water.

Equation 3.1 - the "momentum equation" - very briefly describes how the fluid accelerates due to the forces acting on it. Eq. 3.2 is called "incompressibility condition" and means that the density of the fluid does not change over time.

If real fluids were incompressible, we could not hear anything underwater [Uri67], so why do we assume them to be incompressible? Liquids are practically incompressible - it requires a lot of force to do so, and it is also only possible up to a small degree of compression [Kel75]. Gases like air are easier to compress, but only with the help of pumps [Bur78] or extreme situations like blast waves caused by for example explosions [Tay50a] [Tay50b]. Those kind of scenarios are called "compressible flow" and are expensive to simulate [PL92]. Additionally, they are only visible on a macroscopic level and hence irrelevant for the animation of fluids [Bri15].

Viscosity needs to be considered only in those cases where animating viscous fluids like honey is desired [Bri15]. For most cases, however, it plays a minor role and can be dropped. Those ideal fluids without viscosity are called "inviscid". The Navier Stokes

equations can then be simplified to the following, the so-called "Euler Equations":

$$\frac{\partial \vec{u}}{\partial t} + \frac{1}{\rho} \nabla p = \vec{F} \tag{3.3}$$

$$\nabla \cdot u = 0 \tag{3.4}$$

3.1 The role of the pressure solve in incompressible Navier Stokes equations

In this section, we briefly derive a numerical approximation for the pressure equation of the Navier Stokes equations with the help of a MAC-grid for two-dimensional simulations. Bridson [Bri15] [BM07] already explained this for both 2D and 3D in detail.

3.1.1 Discretization and derivation of the pressure equation

Discretizing eq. 3.3 and 3.4 in time and space gives for a discrete point in time n the following equations:

$$\vec{u}^{n+1} = \vec{u}^n - \Delta t \frac{1}{\rho} \nabla p \quad s.t. \quad \nabla \cdot \vec{u}^{n+1} = 0 \tag{3.5}$$

The divergence of a two dimensional vector $\vec{u} = (u, v)$ is defined as

$$\nabla \cdot \vec{u} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \tag{3.6}$$

So for a fluid grid cell (i, j):

$$(\nabla \cdot \vec{u})_{i,j} = \frac{u_{i+1/2,j}^{n+1} - u_{i-1/2,j}^{n+1}}{\Delta x} + \frac{v_{i,j+1/2}^{n+1} - v_{i,j-1/2}^{n+1}}{\Delta x} + \mathcal{O}(\Delta x^2) \tag{3.7}$$

Using the central difference for $\frac{\partial p}{\partial x}$ and $\frac{\partial p}{\partial y}$ the velocity update is:

$$u_{i+1/2,j}^{n+1} = u_{i+1/2,j}^n - \Delta t \frac{1}{\rho} \frac{p_{i+1,j} - p_{i,j}}{\Delta x} \tag{3.8}$$

$$v_{i+1/2,j}^{n+1} = v_{i,j+1/2}^n - \Delta t \frac{1}{\rho} \frac{p_{i,j+1} - p_{i,j}}{\Delta x} \tag{3.9}$$

Substituting 3.8 and 3.9 into 3.7 finally gives:

$$\frac{1}{\Delta x}\left[\left(u_{i+1/2,j}^n - \Delta t\frac{1}{\rho}\frac{p_{i+1,j}-p_{i,j}}{\Delta x}\right) - \left(u_{i-1/2,j}^n - \Delta t\frac{1}{\rho}\frac{p_{i-1,j}-p_{i,j}}{\Delta x}\right)\right.$$
$$\left. + \left(v_{i,j+1/2}^n - \Delta t\frac{1}{\rho}\frac{p_{i,j+1}-p_{i,j}}{\Delta x}\right) - \left(v_{i,j-1/2}^n - \Delta t\frac{1}{\rho}\frac{p_{i,j-1}-p_{i,j}}{\Delta x}\right)\right] = 0$$

$$-4p_{i,j} + p_{i+1,j} + p_{i,j+1} + p_{i-1,j} + p_{i,j-1} = \frac{\rho\Delta x}{\Delta t}\left(u_{i+1/2,j}^n - u_{i-1/2,j}^n + v_{i,j+1/2}^n - v_{i,j-1/2}^n\right)$$

(3.10)

Now we have a system of linear equations in the form of the Poisson problem:

$$\mathbf{A}\vec{p} = \frac{\rho\Delta x^2}{\Delta t}\nabla \cdot \vec{u}^n \tag{3.11}$$

The Laplace Matrix $\mathbf{A}$ is a symmetric and positive semi-definite matrix so eq. 3.11 can therefore be solved by the Conjugate Gradient method. Each row in $\mathbf{A}$ represents one cell (i,j) on the MAC-grid.

3.1.2 Boundary Conditions

Solids

If an inviscid fluid is in contact with a wall, it cannot move into or out of the boundary. To achieve this, the velocity components of the solids normal needs to match the speed of the fluid at this position:

$$\vec{u} \cdot \hat{n} = \vec{u}_{solid} \cdot \hat{n} \tag{3.12}$$

Concerning the pressure update, assume grid cell (i,j) to be fluid and its neighbor $(i+1,j)$ to be solid. Then $u_{i+1/2,j}^{n+1} = u_{solid}$ and 3.8 can be rearranged to

$$p_{i+1,j} = p_{i,j} + \frac{\rho\Delta x}{\Delta t}(u_{i+1/2,j} - u_{solid}) \tag{3.13}$$

This is called Neumann boundary condition because the derivative of the solution is specified along the boundary.

Open/Air

Compared to water, air is about 780 times lighter. When simulating liquids, the pressure p of air regions may be set to zero since it has only minimal impact and therefore no significant visual effect. Since we directly specify the value of the pressure at those air cells it is called a Dirichlet boundary condition.

Extended MAC-grid

From equation 3.10 we know that for one pressure update of one cell we need to consider all its neighbors. What about those grid cells lying on the edge of the grid? Technically those cells have fewer neighbors than the others. To solve this issue, we extend the grid that holds the information about the state (fluid, solid, air, open) on every dimension. So for a (n x m) sized simulation, we need a (n+2 x m+2) grid that holds the information. We differ between open boundaries where those extended cells are open cells and closed boundaries where solid cells surround the grid.

3.2 The Laplace Matrix

To implement a method that extracts the Laplace Matrix **A**, we need to take a look on the boundary conditions first. Assume a grid cell (i,j) of which one neighbor $(i-1,j)$ is an air cell ($p_{i-1,j} = 0$) and another $(i+1,j)$ is a solid cell ($p_{i+1,j} = p_{i,j} + \frac{\rho \Delta x}{\Delta t}(u_{i+1/2,j} - u_{solid})$). The remaining neighbors are fluid cells. Knowing that, we can rearrange 3.10 to the following:

$$-4p_{i,j} + \left[p_{i,j} + \frac{\rho \Delta x}{\Delta t}(u_{i+1/2,j} - u_{solid}) \right] + p_{i,j+1} + 0 + p_{i,j-1}$$
$$= \frac{\rho \Delta x}{\Delta t} \left(u^n_{i+1/2,j} - u^n_{i-1/2,j} + v^n_{i,j+1/2} - v^n_{i,j-1/2} \right) \tag{3.14}$$

which results in

$$-3p_{i,j} + p_{i,j+1} + p_{i,j-1} = \frac{\rho \Delta x}{\Delta t} \left(u^n_{solid} - u^n_{i-1/2,j} + v^n_{i,j+1/2} - v^n_{i,j-1/2} \right) \tag{3.15}$$

We can make some very important observations from 3.10, 3.14 and 3.15 for building the Laplace Matrix in code:

- **A** is similar to an adjacency matrix: It is symmetric and non adjacent cells are zero.

- Note that the coefficient of $p_{i,j}$ in 3.10 matches the negative number of neighbors of a cell (i,j). Let's call this coefficient $k_{diagonal} = -dim(grid) * 2$, since it is always on the diagonal of **A**'s entries.

- Every neighbor of cell (i,j) is 1 in the corresponding row of **A**.

- For every neighbor of cell (i,j) that is a solid cell increment $k_{diagonal}$ by 1 and set the corresponding column to 0 (Neumann condition).

- For every neighbor of cell (i,j) that is an open cell set the corresponding column to 0 (Dirichlet condition).

- In liquid simulations: For every neighbor of cell (i,j) that is an air cell set the corresponding column to 0 (Dirichlet condition).

With that in mind, we can derive an algorithm which builds **A**. Note that most of **A** is zero so it is a sparse matrix.

Figure 3.1: Example grid with fluid (blue), air (white) and solid (brown) cells and a closed boundary.

	00	01	02	03	10	11	12	13	20	21	22	23	30	31	32	33
00	-2	1			1											
01	0	-3	0			1										
02		1	-3	0			1									
03			0	-2				0								
10	0				-3	1			0							
11		1			1	-4	1			1						
12			0			1	-4	0			0					
13				0			1	-3				0				
20					0				-3	1			0			
21						1			0	-4	0			0		
22							1			1	-4	0			0	
23								0			0	-3				0
30									0				-2	0		
31										1			0	-3	0	
32											0			0	-3	0
33												0			0	-2

$$\cdot \begin{pmatrix} p_{0,0} \\ p_{0,1} \\ p_{0,2} \\ p_{0,3} \\ p_{1,0} \\ p_{1,1} \\ p_{1,2} \\ p_{1,3} \\ p_{2,0} \\ p_{2,1} \\ p_{2,2} \\ p_{2,3} \\ p_{3,0} \\ p_{3,1} \\ p_{3,2} \\ p_{3,3} \end{pmatrix}$$

Figure 3.2: Laplace Matrix **A** derived from Fig. 3.1 with corresponding pressure vector. Diagonal are marked as red, neighbors as blue and non neighbors are left out.

4 CUDA Pressure Solver

In this chapter, we find an efficient numerical approach to solve the system of linear equations $\mathbf{A}\vec{p} = \frac{\rho \Delta x^2}{\Delta t} \nabla \cdot \vec{u}^n$ (3.11) using TensorFlow's custom ops and Nvidia CUDA kernels. First, we discuss how $\mathbf{A}$ is built and stored in memory and then how the Conjugate Gradient method can be used to solve it.

We will use some recurring variables which I clarify here:

- *dimensions* is an array which holds the size of each dimension of the grid.
- *dimSize* holds the number of dimensions.
- *dimProduct* is the product $\prod_{i=0}^{dimSize} dimensions[i]$ of all dimensions.
- *mask* holds the information of fluid, open and solid cells including the boundary cells.

4.1 GPU Laplace Matrix generation

The goal of this section is to find a parallelizable algorithm that creates the Laplace Matrix $\mathbf{A}$, so that a GPU can compute exactly one cell (i, j) or one row of $\mathbf{A}$ per thread.

GPU memory is very limited. For a simulation of the size 64 x 64 x 64 the dimension of the Laplace Matrix is 262.144 x 262.144. This would require 256 GB of GPU-memory if we tried to cache the matrix with 32 bit numbers. At the time of this writing this is only possible with special GPUs that have additional GPU-SSD memory.

Take a look back on the example Laplace Matrix in Fig. 3.2. Notice, that most of it is zero. In fact, there are only at most as many non-zero entries as the number of neighbors of a cell plus one for the cell itself:

$$maxNonZerosPerRow = dimSize * 2 + 1 \tag{4.1}$$

For 2D this means at most five and for 3D at most seven entries per row.

$$\begin{pmatrix} 4 & 0 & 0 & 0 & 0 & 0 \\ 1 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 \\ 3 & 0 & 0 & 4 & 0 & 0 \end{pmatrix} \quad \begin{aligned} data &= [4, 1, 4, 4, 3, 4] \\ columnptr &= [0, 0, 1, 2, 0, 3] \\ rowcnt &= [0, 1, 3, 4, 6] \end{aligned}$$

Figure 4.1: Example Matrix in CSR format

The *compressed sparse row* (CSR) format tackles this memory problem by only storing information of non-zero entries of a sparse matrix [TW67]. It consists of three arrays:

data: values of non-zero entries in row-major order.
columnptr: the column indices of all non-zero data entries in row-major order.
rowcnt: The ith element points to the index in *data* of the first non-zero entry of the ith row.

Because **A** is sparse to a vast extent, the CSR format reduces memory consumption dramatically. However, it requires three memory accesses to get all data of one entry. Compared to pure computation, memory accesses are very costly on GPUs and should be kept as low as possible [Fan+18][Fuj+13]. To improve speed, reducing those memory accesses is crucial.

4.1.1 Tailoring the CSV format for the GPU pressure solve

For every row i we know that all non-zero entries of this row lay in the range $[rowcnt[i] \; ; \; rowcnt[i+1][$ in the *data* array.

The maximum number of non-zero entries per row in the Laplace Matrix is $maxNonZerosPerRow$ as defined above. It is only the maximum possible number of non-zeros because neighbor cells of (i, j) might be open or solid cells, thus zero in the (i, j)th row of the Laplace Matrix. For cells at the edge of the grid, we set the corresponding value for neighbor cells that lay outside the grid to zero in the *data* array as well. The CSR format would have removed those zero entries but if we keep them, we have a regular arrangement of *data* ($data_{regular}$)-entries per row. As a trade-of this will require slightly more memory. However, we are now able to calculate the index of the row i_{row} of any $data_{regular}$ entry by its index i_{data}:

$$i_{row} = \left\lfloor \frac{i_{data}}{maxNonZerosPerRow} \right\rfloor \tag{4.2}$$

For efficient parallelizing, this step is indispensable. If the GPU executes two cells for example $(i + 1, j)$ and (i, j) in two threads at the same time and every *data* entry does not have a predefined place in the array, the threads would mix up their insertions and thus the row-major structure. Since $data_{regular}$ is now regular, each thread knows the exact position in $data_{regular}$ for every entry to comply the row-major order.

$$data = \left[\underbrace{0, 0, \textcolor{red}{-2}, 1, 1}_{i_{row}=0}, \underbrace{0, 0, \textcolor{red}{-3}, 0, 1}_{i_{row}=1}, \underbrace{0, 0, \textcolor{red}{-3}, 0, 1}_{i_{row}=2}, \underbrace{0, 0, \textcolor{red}{-2}, 0, 0}_{i_{row}=3}, \underbrace{0, 0, \textcolor{red}{-3}, 1, 0}_{i_{row}=4}, \dots \right]$$

Figure 4.2: Vizualization of $data_{regular}$ array for Fig. 3.2. Diagonal entries are printed in red. Neighbor cells are printed in blue if they are within the grid and brown otherwise.

$$diagonalOffsets = [\ \text{-4, -1, } \textcolor{red}{0} \text{, 1, 4 }]$$

Figure 4.3: $diagonalOffsets$ array that corresponds to Fig. 3.2.

More in-depth inspection of Fig. 3.2 reveals that neighbor-cell entries always appear by the same distances to the diagonal of each row. Of course, this is due to the fact that the respective neighbors on the linearized grid lie at the same distance from each other for every cell accordingly. This is also applicable to the $data_{regular}$ array: in respect to the diagonal (red) entries the neighbors always appear in the same order (Fig. 4.3). With this knowledge we can extract a method to infer from each $data_{regular}$ entry the respective column index i_{column} via the data index i_{data}.

To do this, we define a new helper array $diagonalOffsets$ which stores the column-index distance to the diagonal of every entry of a row. Here is an algorithm to create the $diagonalOffsets$ array:

Algorithm 1 Creates an array containing the column offset for every neighbor of a row in **A** in respect to the diagonal

---

1: **function** GET DIAGONALOFFSETS($maxDataPerRow$, $dimensions$)
2:     $diagonalOffsets = [\cdots]$ ▷ Declare array of length $maxDataPerRow$
3:     $factor = 1$
4:     **const** $diagonal = \left\lfloor \frac{maxNonZerosPerRow}{2} \right\rfloor$
5:
6:     **for** $i = 0$, $offset = 1$; $i < diagonal$; i++, $offset$++ **do**
7:         $diagonalOffsets[diagonal - offset] = -factor$
8:         $diagonalOffsets[diagonal + offset] = factor$
9:         $factor = factor * dimensions[i]$
10:     $diagonalOffsets[diagonal] = 0$
11:     **return** $diagonalOffsets$

Now getting the column index i_{column} by an arbitrary $data_{regular}$ index i_{data} is trivial:

$$i_{column} = \left\lfloor \frac{i_{data}}{maxNonZerosPerRow} \right\rfloor + columnOffsets\left[i_{data} \mod maxNonZerosPerRow\right]$$

$$= i_{row} + columnOffsets\left[i_{data} \mod maxNonZerosPerRow\right]$$

$$(4.3)$$

4.1.2 Laplace Matrix in the Tailored CSV format generation on the GPU

We now derive an algorithm, that creates the Laplace Matrix in parallel on the GPU. We apply the Tailored CSV format that was introduced in the previous section.

The goal is, that one cell or one row in **A** is computed per GPU thread. To do that, we start *dimProduct* threads each providing the *rowID* by its thread id. Using the *rowID* we have to check the state in *mask* of all neighbors and insert the data accordingly in row-major format. To do that, we first review the neighbors that appear before the current cell in the linearized grid and push their values, add the diagonal entry of the current cell itself and then check the remaining neighbors and add their data:

Algorithm 2 Create Laplace Matrix

1: **procedure** GET LAPLACE MATRIX(*rowID*, *mask*, *data$_{regular}$*)
2:      $i_{data} = rowID \cdot (dimSize \cdot 2 + 1)$
3:      $diagonalValue = -dimSize * 2$
4:
5:      ▷ Insert data of neighbors that appear before the current cell in linearized grid
6:      **for** *neighbor* of *beforeNeighbors* in row-major **do**
7:          **if** *neighbor* is Fluid in *mask* AND NOT *currentCell* is Solid in *mask* **then**
8:              $data_{regular}[i_{data}] = 1$
9:          **else if** *neighbor* is Solid in *mask* **then**
10:              $data_{regular}[i_{data}] = 0$
11:              $diagonalValue$++
12:          **else if** *neighbor* is Open in *mask* **then**
13:              $data_{regular}[i_{data}] = 0$
14:          i_{data}++
15:
16:                                  ▷ Modify *diagonalValue* according to the *afterNeighbors*
17:      **for** *neighbor* of *afterNeighbors* in row-major **do**
18:          **if** *neighbor* is Solid Cell in *mask* **then**
19:              $diagonalValue$++
20:
21:   ▷ Insert diagonal entry of the current cell
22:      $data_{regular}[i_{data}] = diagonalValue$
23:      i_{data}++
24:
25:      ▷ Insert data of neighbors that appear after the current cell in linearized grid
26:      **for** *neighbor* of *afterNeighbors* in row-major **do**
27:          **if** *neighbor* is Fluid in *mask* AND NOT *currentCell* is Solid in *mask* **then**
28:              $data_{regular}[i_{data}] = 1$
29:          **else if** *neighbor* is Solid in *mask* **then**
30:              $data_{regular}[i_{data}] = 0$
31:          **else if** *neighbor* is Open in *mask* **then**
32:              $data_{regular}[i_{data}] = 0$
33:          i_{data}++

4.2 GPU Pressure Solve

In the previous section we have found an algorithm that creates and stores the Laplace Matrix $\mathbf{A}$ efficiently on GPUs. In this section I present how the system of linear equations $\mathbf{A}\vec{p} = \frac{\rho \Delta x^2}{\Delta t} \nabla \cdot \vec{u}^n$ (3.11) can be solved on GPUs using the Conjugate Gradient method. First, we simplify equation 3.11:

$$\mathbf{A}\vec{p} = \vec{d} \qquad (4.4)$$

Now solving this system of linear equations with the cg-method works as follows [HS52]:

Algorithm 3 Pressure Solve with cg-method

1: **function** Solve Pressure($\mathbf{A}$, $\vec{d}$, *maxIterations*, *accuracy*, $\vec{p}_0$)
2:     $\vec{m} = \vec{r} = \vec{d} - \mathbf{A}\vec{p}_0$ ▷ Init helper variables
3:     $\vec{p} = \vec{p}_0$
4:     **for** $k = 1$ to *maxIterations* **do**
5:         $\vec{z} = \mathbf{A}\vec{m}$ ▷ calcZ
6:         $alpha = \frac{\vec{m} \bullet \vec{r}}{\vec{m} \bullet \vec{z}}$
7:         $\vec{p} = \vec{p} + alpha \cdot \vec{m}$
8:         $\vec{r} = \vec{r} - alpha \cdot \vec{z}$
9:         **if** $max_i(|\vec{r}_i|) < accuracy$ **then**   ▷ checkResiduum
10:             **return** $\vec{p}$
11:         $beta = -\frac{\vec{r} \bullet \vec{z}}{\vec{m} \bullet \vec{z}}$
12:         $\vec{m} = \vec{r} + beta \cdot \vec{m}$
13:     **return** $\vec{p}$

All mathematical operations except for operation 5 and 9 can be executed by the highly optimized cuBLAS library which is part of CUDAs standard libraries.
Obviously operation 5 is the most demanding because it involves a matrix-vector multiplication. Now we can take advantage of the findings and optimization steps in the previous section.

calcZ

Without further ado, we can derive an algorithm to calculate $\vec{z}$ using $\mathbf{A}$ in the Tailored CSR format. Every thread computes one entry of the resulting vector $\vec{z}$. The row *rowID* of this vector is given by the CUDA thread and block id:

Algorithm 4 Multiplies $\mathbf{A}$ with a vector

1: **function** CALCZ(*rowID*, $\mathbf{A}$, $\vec{m}$, $\vec{z}$, *columnOffsets*)
2:     $rowStartIndex = rowID * maxNonZerosPerRow$
3:     $tmp = 0$
4:     **for** $i = rowStartIndex$; $i < rowStartIndex + maxNonZerosPerRow$; i++ **do**
5:         $tmp = tmp + \mathbf{A}[i] \cdot \vec{m}[rowID + columnOffsets[i - rowStartIndex]]$
6:     $\vec{z}[rowID] = tmp$

---

checkResiduum

The cg-method approximates the solution of a system of linear equations iteratively. It should terminate if all entries of the residuum vector $\vec{r}$ is smaller than the given *accuracy*. We introduce a new variable *finished* and set it to true at the beginning of every cg-iteration. The checkResiduum method searches for an entry in $\vec{r}$ to be greater than *accuracy* in parallel. Every match sets *finished* to false. When all checks have been done we can break the cg-iteration if *finished* is still true.

5 Results

In this chapter I present how my CUDA solution compares against the TensorFlow implementations of Φ_{Flow}.

All benchmarks were performed in an open boundary environment with random divergence vectors in the $[-1, 1]$ range. This does not represent real fluid simulations. However, as a benchmark it is well suited to compare the individual implementations.

The targeted residual accuracy was 1^{-5}. Before starting the measurement, a warm-up of 5 runs was performed. The data shows the mean and standard deviation of 25 runs. The test system consists of a GeForce GTX 980, a Quad-Core Intel Core i7 3779K @3.5GHz CPU and 8GB of RAM. Because of memory limitations on the test system, some benchmarks could not be done in the Φ_{Flow} implementations.

Performance Pressure Solve

The following table shows the execution time per solve and the average number of cg-iterations for quadratic grids and compares the TensorFlow solutions with my CUDA implementation. In contrast to the Φ_{Flow} variants, my implementation additionally calculates the Laplace matrix in each step:

| Dimension	TF CPU	TF GPU	CUDA	Iterations	Speedup CPU	Speedup GPU
64^2	30,11 ms	108,2 ms	25,36 ms	160	1.2 x	4.2 x
128^2	139,8 ms	396,3 ms	39,49 ms	310	3.5 x	10.0 x
256^2	1,161 s	1,839 s	77,19 ms	600	15.1 x	23.8 x
512^2	11,58 s	12,90 s	240,9 ms	1175	48.3 x	53.6 x
1024^2	114,6 s	87,47 s	1,412 s	2300	81.2 x	61.9 x
2048^2	n/a	n/a	9,742 s	4450	n/a	n/a
64^3	1,966 s	2,473 s	42,18 ms	170	46.6 x	58.6 x
128^3	37,03 s	29,35 s	397,1 ms	320	93,3 x	73.9 x
256^3	573,7 s	n/a	5,814 s	610	98,7 x	n/a
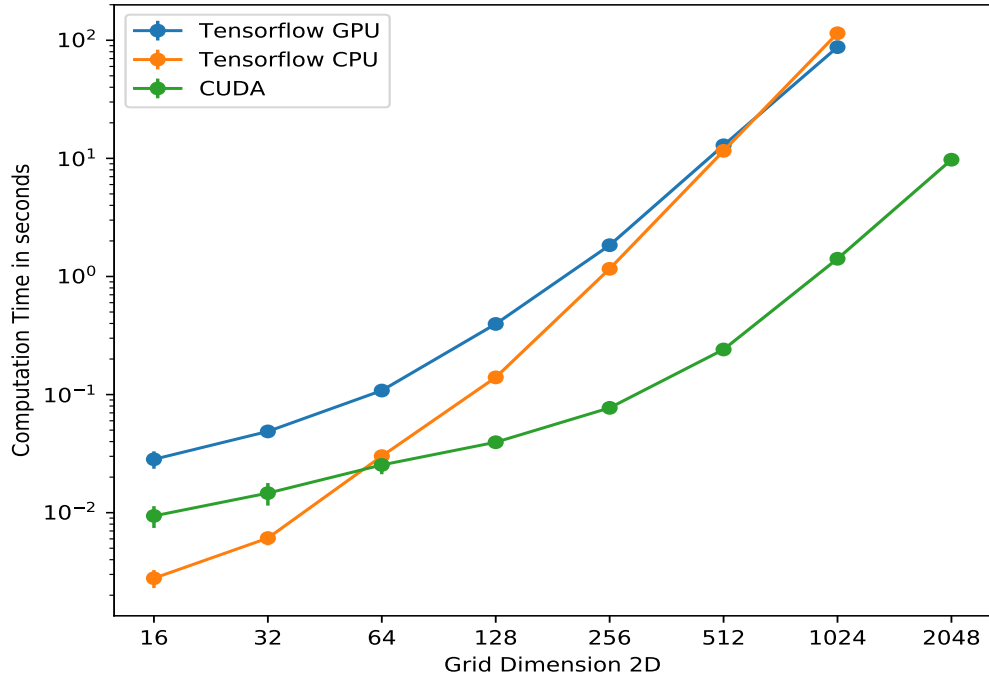
(a) Pressure Solve in 2D
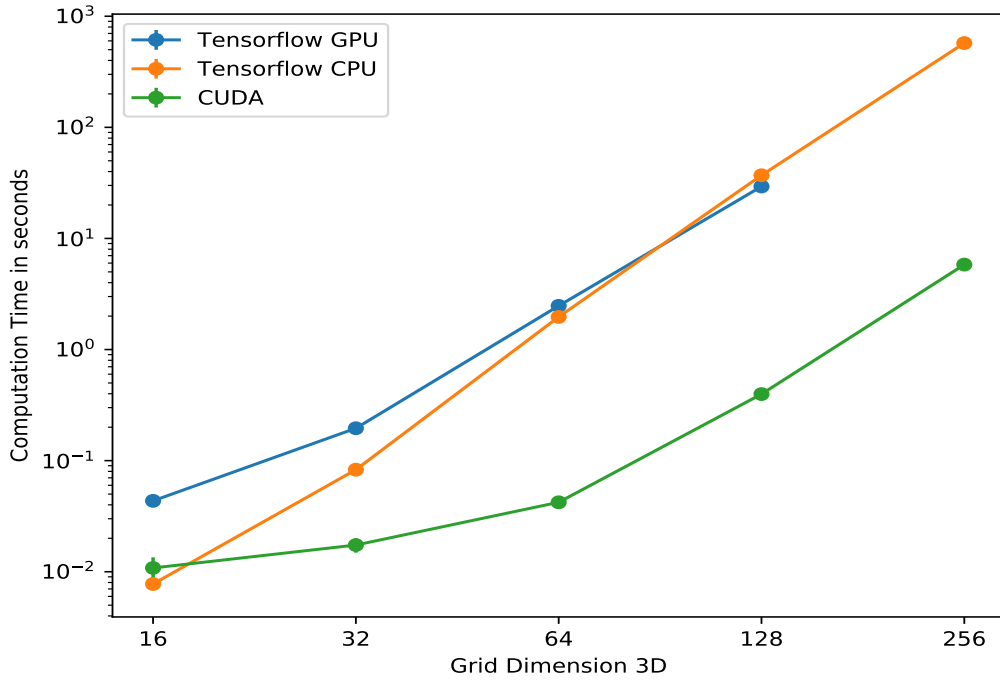
(b) Pressure Solve in 3D

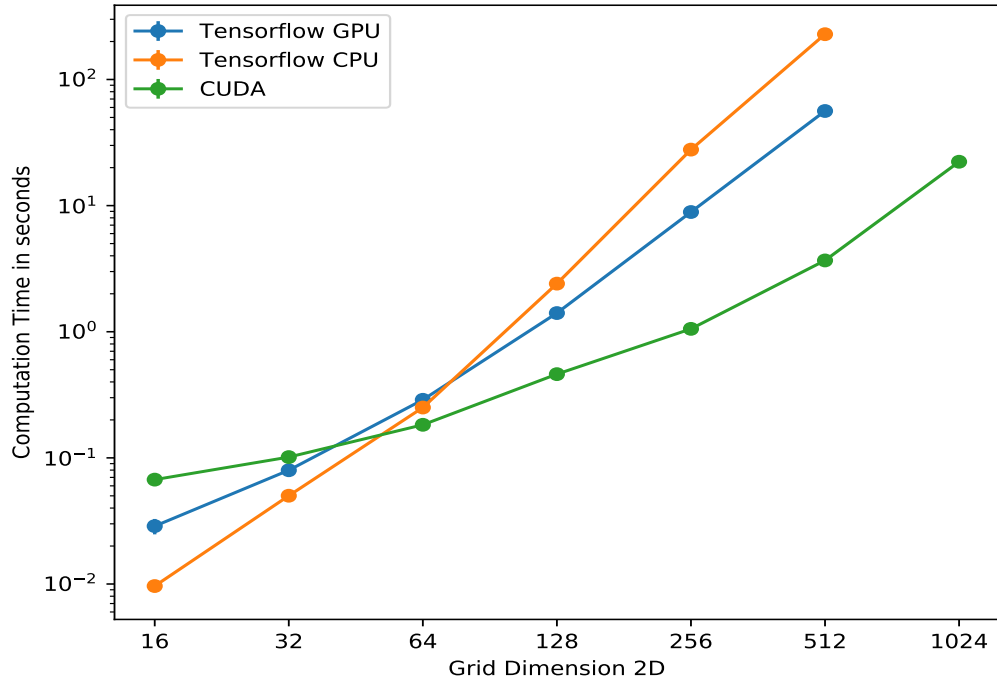Figure 5.1: Pressure Solve execution time with batch size 1.
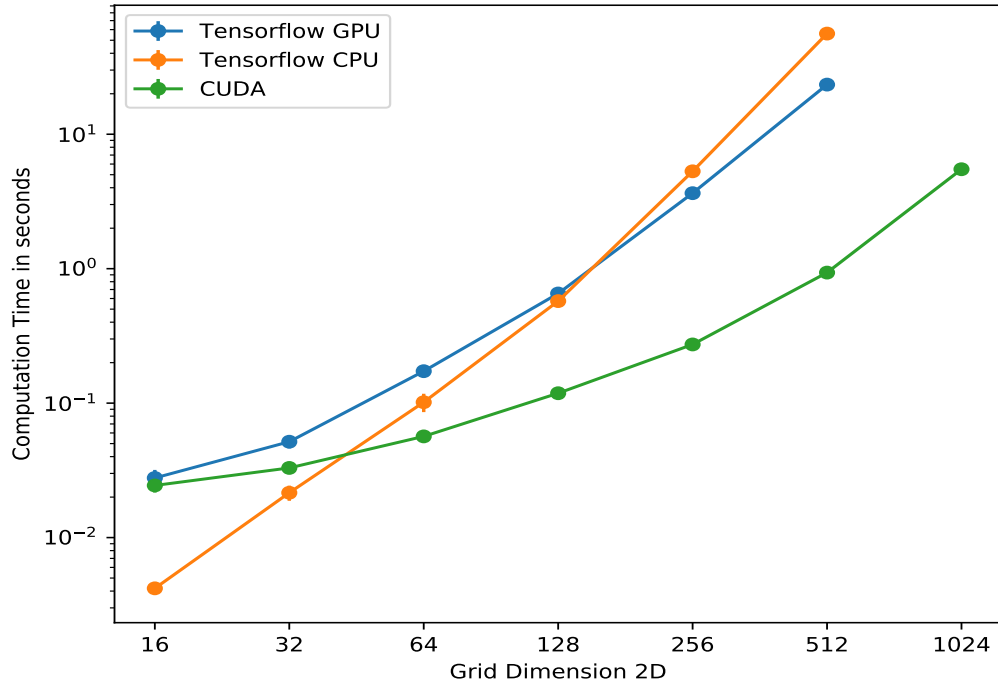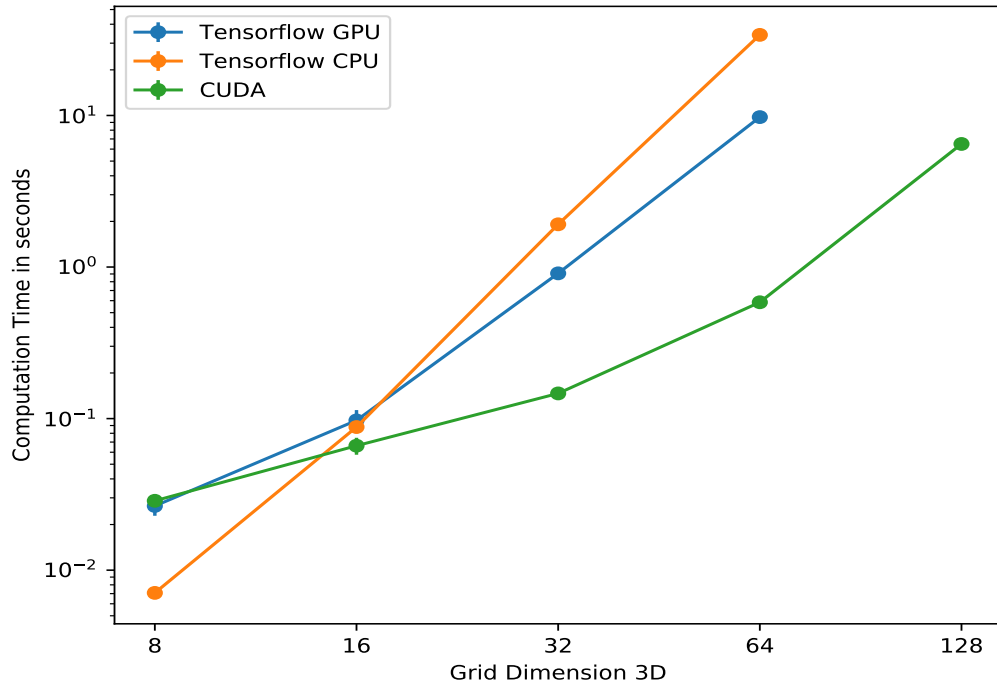
(a) Pressure Solve in 2D
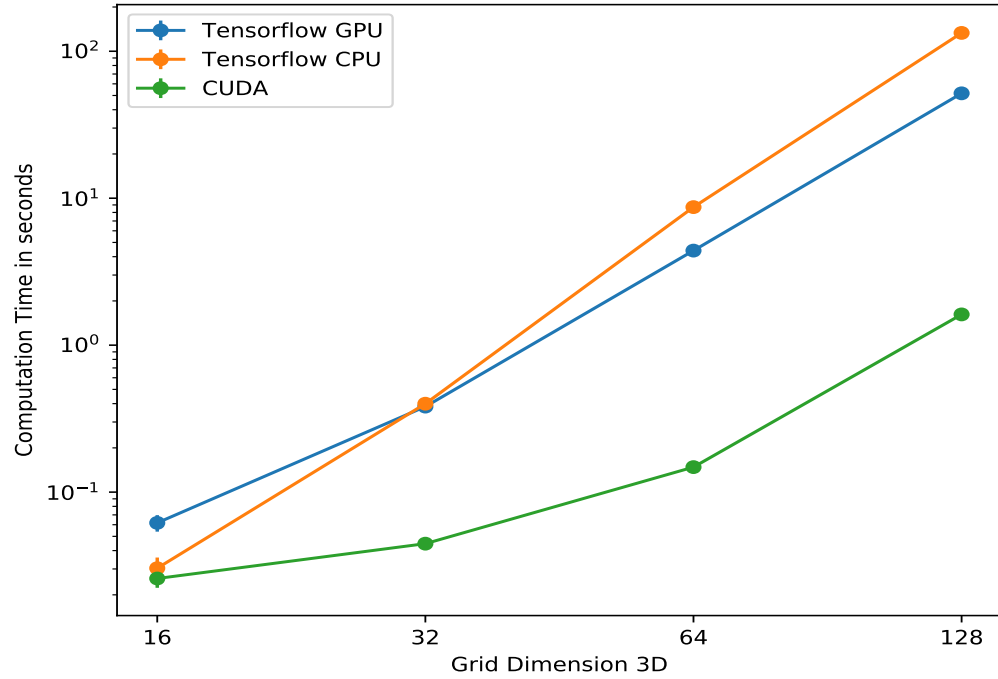
(b) Pressure Solve in 3D

Figure 5.2: Pressure Solve execution time with batch size 4.

(a) Pressure Solve in 2D

(b) Pressure Solve in 3D

Figure 5.3: Pressure Solve execution time with batch size 16.
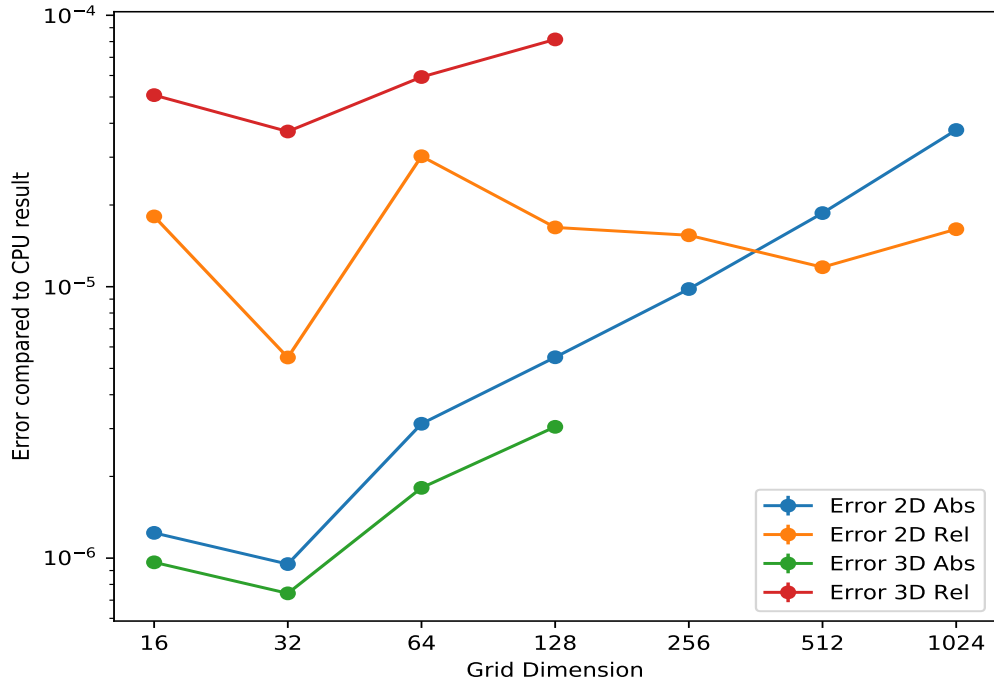
Figure 5.4: Absolute and relative error compared to the CPU version

Absolute and relative error

In this test, I compare how much my GPU solution differs from the CPU version of Φ_{Flow}. I have calculated the absolute and relative difference between the pressure vectors component-wise. As stated in Fig. 5.4, the error is sufficiently small.

Performance Laplace Matrix generation

The speed of the Laplace Matrix generation is crucial for simulation with moving boundary conditions. Φ_{Flow} calculates the Laplace Matrix once at startup on the CPU, so it has no effect on the actual fluid simulation. Compared to the CPU implementation I could shorten the generation time up to 800 times. While the CPU version scales with the simulation size, my parallel CUDA implementation remains constant for significantly longer (Fig. 5.5).
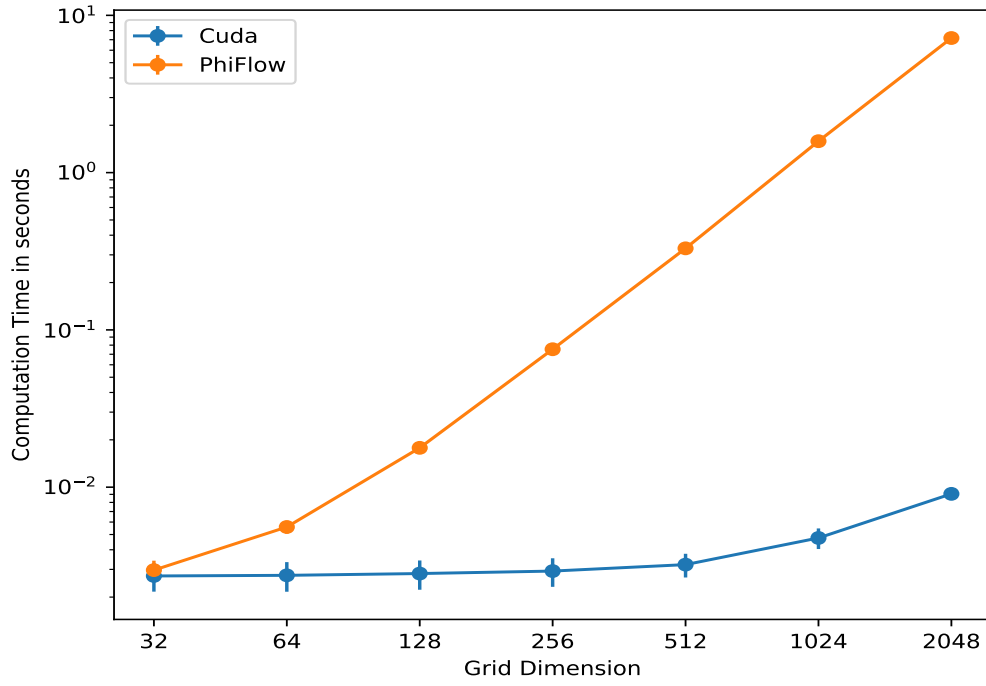
(a) Laplace Matrix generation in 2D
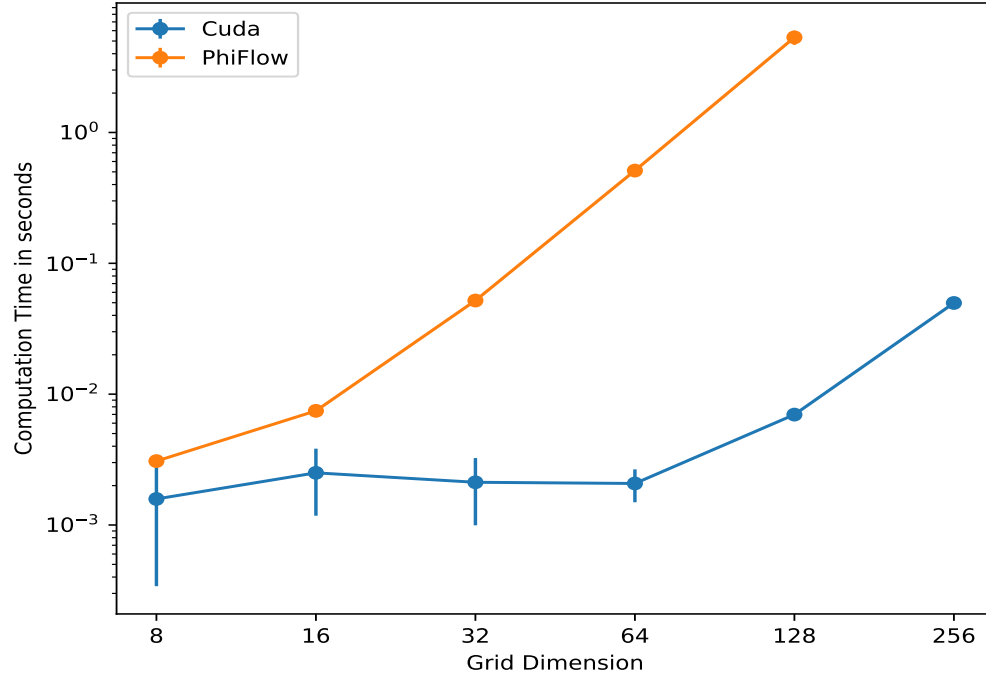
(b) Laplace Matrix generation in 3D

Figure 5.5: Laplace Matrix generation execution time

6 Conclusion and Future Work

In this thesis, I presented an efficient implementation for the pressure solver. I introduced the Tailored CSR format that allows the Laplace Matrix to be created efficiently on the GPU with little memory usage. This allows the matrix multiplication in the cg-method to be calculated faster than conventional methods. The highly optimized cuBLAS library can execute all other mathematical operations efficiently on the GPU. The source of this work is available in the mantaflow git repository including the build and benchmark scripts.

I was able to improve the speed of the pressure solve compared to the multicore CPU version of Φ_{Flow} by a factor of up to 99 and by a factor of up to 74 compared to its GPU implementation. This will help to speed up the training process of the neural networks considerably.

Because the main focus of this work was the acceleration of the pressure solver, there is still potential to improve the Laplace Matrix generation kernel. A fast method is crucial when objects move through space during the simulation. Furthermore, there is certainly room for further optimizations to the pressure solve kernel. At the moment it scales pretty much with the number of batches. That can be addressed in future work and with the help of CUDA experts, who can utilize the resources of GPUs more effectively than myself.

Bibliography

[BM07]   R. Bridson and M. Müller-Fischer. "Fluid simulation: SIGGRAPH 2007 course notes Video files associated with this course are available from the citation page." In: *ACM SIGGRAPH 2007 courses*. ACM. 2007, pp. 1–81.

[Bri15]   R. Bridson. *Fluid simulation for computer graphics*. AK Peters/CRC Press, 2015.

[BT07]   M. Becker and M. Teschner. "Weakly compressible SPH for free surface flows." In: *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*. Eurographics Association. 2007, pp. 209–217.

[Bur78]   T. I. Burenga. *Portable air compressor*. US Patent 4,077,747. Mar. 1978.

[Cha+16]   M. B. Chang, T. Ullman, A. Torralba, and J. B. Tenenbaum. "A compositional object-based approach to learning physical dynamics." In: *arXiv preprint arXiv:1612.00341* (2016).

[CT17]   M. Chu and N. Thuerey. "Data-driven synthesis of smoke flows with CNN-based feature descriptors." In: *ACM Transactions on Graphics (TOG)* 36.4 (2017), p. 69.

[Eul57]   L. Euler. "Principes généraux du mouvement des fluides." In: *Mémoires de l'Académie des Sciences de Berlin* (1757), pp. 274–315.

[Fan+18]   M. Fang, J. Fang, W. Zhang, H. Zhou, J. Liao, and Y. Wang. "Benchmarking the GPU memory at the warp level." In: *Parallel Computing* 71 (2018), pp. 23–41.

[Fuj+13]   Y. Fujii, T. Azumi, N. Nishio, S. Kato, and M. Edahiro. "Data transfer matters for GPU computing." In: *2013 International Conference on Parallel and Distributed Systems*. IEEE. 2013, pp. 275–282.

[Gil09]   J. Gilland. *Elemental magic: the art of special effects animation*. 2009.

[HS52]   M. R. Hestenes and E. Stiefel. *Methods of conjugate gradients for solving linear systems*. Vol. 49. 1. NBS Washington, DC, 1952.

[HW65]   F. H. Harlow and J. E. Welch. "Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface." In: *The physics of fluids* 8.12 (1965), pp. 2182–2189.

[Kel75]   G. S. Kell. "Density, thermal expansivity, and compressibility of liquid water from 0. deg. to 150. deg.. Correlations and tables for atmospheric pressure and saturation reviewed and expressed on 1968 temperature scale." In: *Journal of Chemical and Engineering Data* 20.1 (1975), pp. 97–105.

[Kim02]   R. Kimura. "Numerical weather prediction." In: *Journal of Wind Engineering and Industrial Aerodynamics* 90.12-15 (2002), pp. 1403–1414.

[LBH15]   Y. LeCun, Y. Bengio, and G. Hinton. "Deep learning." In: *nature* 521.7553 (2015), p. 436.

[Lio71]   J. L. Lions. *Optimal Control of Systems Governed by Partial Differential Equations (Grundlehren der Mathematischen Wissenschaften)*. Vol. 170. Springer Berlin, 1971.

[Luc77]   L. B. Lucy. "A numerical approach to the testing of the fission hypothesis." In: *The astronomical journal* 82 (1977), pp. 1013–1024.

[MCG03]   M. Müller, D. Charypar, and M. Gross. "Particle-based fluid simulation for interactive applications." In: *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*. Eurographics Association. 2003, pp. 154–159.

[McN+04]   A. McNamara, A. Treuille, Z. Popović, and J. Stam. "Fluid control using the adjoint method." In: *ACM Transactions On Graphics (TOG)*. Vol. 23. 3. ACM. 2004, pp. 449–456.

[MM13]   M. Macklin and M. Müller. "Position based fluids." In: *ACM Transactions on Graphics (TOG)* 32.4 (2013), p. 104.

[Mon92]   J. J. Monaghan. "Smoothed particle hydrodynamics." In: *Annual review of astronomy and astrophysics* 30.1 (1992), pp. 543–574.

[Mor+]   D. Morikawa, M. Asai, Y. Imoto, M. Isshiki, et al. "Improvements in highly viscous fluid simulation using a fully implicit SPH method." In: *Computational Particle Mechanics* (), pp. 1–16.

[Nav27]   C. L. Navier. *Mémoire sur les lois de l'équilibre et du mouvement des corps solides élastiques*. 1827.

[Pes77]   C. S. Peskin. "Numerical analysis of blood flow in the heart." In: *Journal of computational physics* 25.3 (1977), pp. 220–252.

[PL92]   T. J. Poinsot and S. Lelef. "Boundary conditions for direct simulations of compressible viscous flows." In: *Journal of computational physics* 101.1 (1992), pp. 104–129.

[RMN09]   R. Raina, A. Madhavan, and A. Y. Ng. "Large-scale deep unsupervised learning using graphics processors." In: *Proceedings of the 26th annual international conference on machine learning*. ACM. 2009, pp. 873–880.

[SF18]   C. Schenck and D. Fox. "Spnets: Differentiable fluid dynamics for deep neural networks." In: *arXiv preprint arXiv:1806.06094* (2018).

[Sto80]   G. G. Stokes. "On the theories of the internal friction of fluids in motion, and of the equilibrium and motion of elastic solids." In: *Transactions of the Cambridge Philosophical Society* 8 (1880).

[Tay50a]   G. I. Taylor. "The formation of a blast wave by a very intense explosion I. Theoretical discussion." In: *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences* 201.1065 (1950), pp. 159–174.

[Tay50b]   G. I. Taylor. "The formation of a blast wave by a very intense explosion.-II. The atomic explosion of 1945." In: *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences* 201.1065 (1950), pp. 175–186.

[Tec19]   U. Technologies. *Unity3D*. `https://unity3d.com/`. 2019.

[Thu+18]   N. Thuerey, K. Weissenow, H. Mehrotra, N. Mainali, L. Prantl, and X. Hu. "Well, how accurate is it? a study of deep learning methods for reynolds-averaged navier-stokes simulations." In: *arXiv preprint arXiv:1810.08217* (2018).

[Tom+17]   J. Tompson, K. Schlachter, P. Sprechmann, and K. Perlin. "Accelerating eulerian fluid simulation with convolutional networks." In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 3424–3433.

[Tsu+09]   M. Tsubokura, T. Kobayashi, T. Nakashima, T. Nouzawa, T. Nakamura, H. Zhang, K. Onishi, and N. Oshima. "Computational visualization of unsteady flow around vehicles using high performance computing." In: *Computers & Fluids* 38.5 (2009), pp. 981–990.

[TW67]   W. F. Tinney and J. W. Walker. "Direct solutions of sparse network equations by optimally ordered triangular factorization." In: *Proceedings of the IEEE* 55.11 (1967), pp. 1801–1809.

[Uri67]   R. J. Urick. *Principles of underwater sound for engineers*. Tata McGraw-Hill Education, 1967.

[Wep17]   M. Wepner. *SPH in Unity3D*. `https://martin-wepner.de/fluid-simulation-with-compute-shader/`. 2017.