



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

todo

# Abstract

TODO: Abstract

# Contents

<b>Acknowledgments</b>	<b>iii</b>
<b>Abstract</b>	<b>iv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 $\Phi_{Flow}$ . . . . .	1
1.2 Description of the problem . . . . .	2
1.3 Outline . . . . .	2
<b>2 Related Work</b>	<b>4</b>
<b>3 Numerical approaches for simulating fluids</b>	<b>6</b>
3.1 The role of the pressure solve in incompressible Navier Stokes equations	7
3.1.1 Discretization and derivation of the pressure equation . . . . .	7
3.1.2 Boundary Conditions . . . . .	8
3.2 The Laplace Matrix . . . . .	10
<b>4 CUDA Pressure Solver</b>	<b>12</b>
4.1 GPU Laplace Matrix generation . . . . .	12
4.1.1 Tailoring the CSV format for the GPU pressure solve . . . . .	13
4.2 GPU Pressure Solve . . . . .	15
<b>5 Results</b>	<b>17</b>
5.1 Section . . . . .	17
5.1.1 Subsection . . . . .	17
<b>List of Figures</b>	<b>19</b>
<b>List of Tables</b>	<b>20</b>
<b>Bibliography</b>	<b>21</b>

# 1 Introduction

The simulation of fluids is part of our daily lives, 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. 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. And the simulation of fluids also plays a role in medicine, for example to better understand blood flow and how nutrients get from A to B in the human body. Many designers and artists are working to deliver the greatest possible immersion to the viewer in games and films. 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 or smoke that rises into the sky after an explosion.

Existing solutions are often very computationally intensive and lead to compromises between speed and accuracy, density 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 taken state-of-the-art in many areas such as object recognition to a new level in recent years. 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].

## 1.1 $\Phi_{Flow}$

$\Phi_{Flow}$  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 to keep every simulation step differentiable which makes it suitable for backpropagation. Backpropagation, on the other hand, is required to calculate the gradient of the loss function of Neural Networks to adjust the weight of the network. Being developed on top of TensorFlow, it is not only capable of running on CPU, but also completely on GPU. TensorFlow simplifies the implementation of Machine Learning algorithms including the use of Neural Networks for predictions

and classification. This makes  $\Phi_{Flow}$  a powerful tool to train Neural Networks on fluid simulations and also visualize them by its interactive GUI running in the browser.

## 1.2 Description 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.  $\Phi_{Flow}$  already comes with its own pressure solver running directly in TensorFlow. However, TensorFlow's dataflow graph is unnecessarily complex which brings overhead to the computation.

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  $\Phi_{Flow}$ . It can then be compared to  $\Phi_{Flows}$  built-in solution.

## 1.3 Outline

This chapter introduces the topic and gives an overview of what fluid simulations are and what they are useful for.

Chapter 2 discusses past research in this area. Chapter 3 repeats the theoretical background necessary for this work.

Chapter 4 describes my solution.

In chapter 5 I present the results and compare them with those from Phiflow.

Chapter 6 summarizes and gives suggestions for further work.



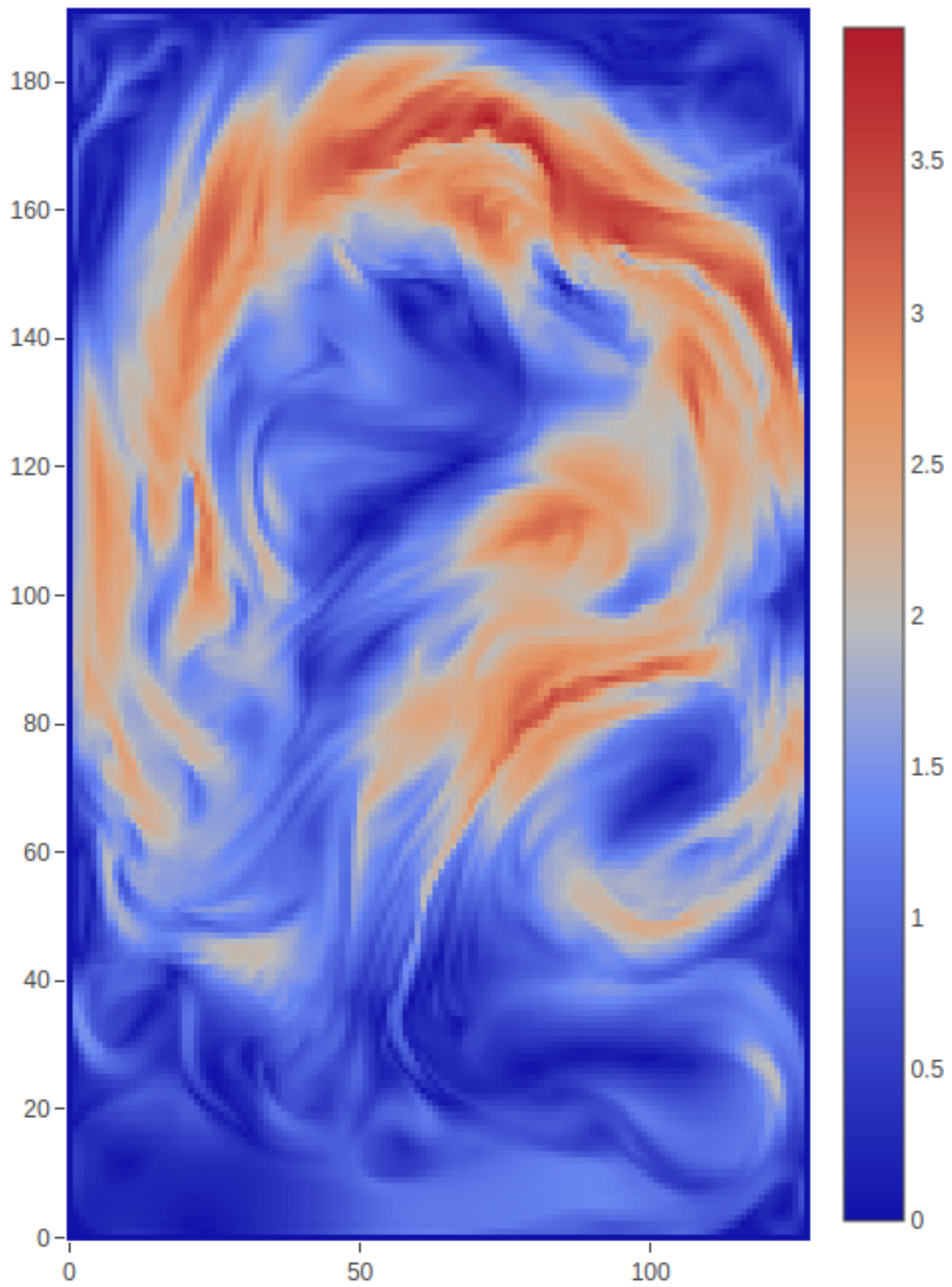


Figure 1.1: Grid-based fluid simulation in  $\Phi_{flow}$

## 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.

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

The 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 does not fulfill the incompressibility constraint, which can cause undesired visual artifacts. Becker and Teschner presented the weakly compressible SPH system (WCSPH) to maintain incompressibility with the cost of limiting the time-step to impractical sizes [BT07]. Macklin and Mueller introduced Position Based Fluids (PBF) which first advects particles and then corrects its positions to satisfy the incompressibility conditions iteratively [MM13]. Several improvements had been done since then and it is still an active topic in research (TODO papers) [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. [TODO: Smth. about CG]

Compared to particle-based approaches the grid-based approach achieve higher accuracy. Particle-based methods only needs computation where particles are; the grid-based method always considers to whole grid. This makes the particle-based viewpoint more suitable for simulations in which the fluid moves through an open

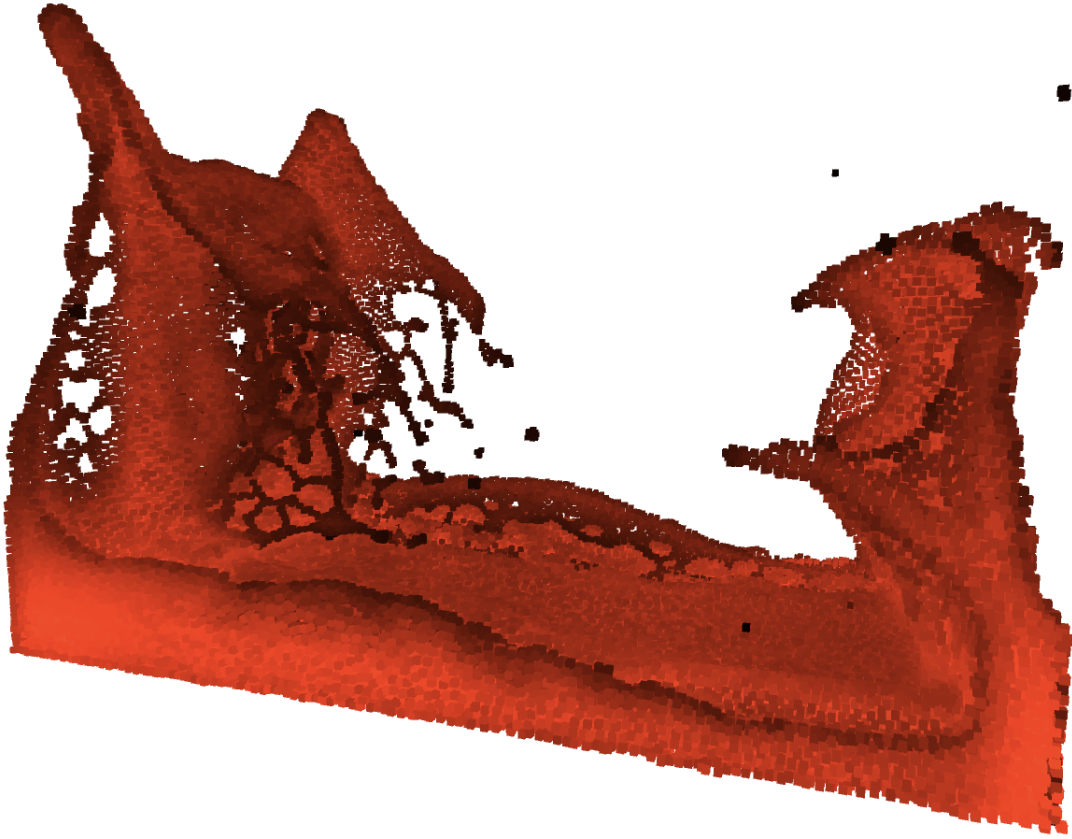


Figure 2.1: Particle-based fluid simulation in Unity3D [Tec19] [Wep17]

domain environment like for example in games.

In the 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]. Deep Neural Networks 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 to use 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.

### 3 Numerical approaches for simulating fluids

Incompressible fluids are described by Navier Stokes equations:

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

$$\nabla \cdot \vec{u} = 0 \quad (3.2)$$

Here,  $\vec{u}$  denotes the velocity vector of the fluid.

$\rho$  stands for the density of the fluid. E.g.  $\rho_{\text{Water}} \approx 1000 \text{kg/m}^3$ .

$p$ , 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.

$\nu$  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 "volume?" of the fluid doesn't change.

If real fluids were incompressible, we couldn't 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 until 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":

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

$$\nabla \cdot \vec{u} = 0 \quad (3.4)$$

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

The pressure solve is part of the Navier Stokes equations:

$$\frac{\partial \vec{u}}{\partial t} + \frac{1}{\rho} \nabla p = 0 \quad \text{s.t.} \quad \nabla \cdot \vec{u} = 0 \quad (3.5)$$

This means, that the pressure always needs to be so, that it advects the velocity field  $\vec{u}$  divergence-free and thus makes the fluid incompressible. [TODO: Maybe write *what exactly* the pressure is and what it means for the simulation]

#### 3.1.1 Discretization and derivation of the pressure equation

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.

Discretizing 3.5 in time and space gives for a discrete point in time  $n$  the following equations:

$$\vec{u}^{n+1} = \vec{u}^n - \Delta t \frac{1}{\rho} \nabla p \quad \text{s.t.} \quad \nabla \cdot \vec{u}^{n+1} = 0 \quad (3.6)$$

The divergence of a 2 dimensional vector  $\vec{u} = (u, v)$  is defined as

$$\nabla \cdot \vec{u} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \quad (3.7)$$

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

$$(\nabla \cdot \vec{u})_{i,j} = \frac{u_{i+1/2,j}^{n+1} - u_{i-1/2,j}^{n+1}}{\Delta x} + \frac{v_{i,j+1/2}^{n+1} - v_{i,j-1/2}^{n+1}}{\Delta x} + \mathcal{O}(\Delta x^2) \quad (3.8)$$

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

$$u_{i+1/2,j}^{n+1} = u_{i+1/2,j}^n - \Delta t \frac{1}{\rho} \frac{p_{i+1,j} - p_{i,j}}{\Delta x} \quad (3.9)$$

$$v_{i,j+1/2}^{n+1} = v_{i,j+1/2}^n - \Delta t \frac{1}{\rho} \frac{p_{i,j+1/2} - p_{i,j}}{\Delta x} \quad (3.10)$$

Substituting 3.9 and 3.10 into 3.8 finally gives:

$$\begin{aligned} & \frac{1}{\Delta x} \left[ \left( u_{i+1/2,j}^n - \Delta t \frac{1}{\rho} \frac{p_{i+1,j} - p_{i,j}}{\Delta x} \right) - \left( u_{i-1/2,j}^n - \Delta t \frac{1}{\rho} \frac{p_{i-1,j} - p_{i,j}}{\Delta x} \right) \right. \\ & \left. + \left( v_{i,j+1/2}^n - \Delta t \frac{1}{\rho} \frac{p_{i,j+1} - p_{i,j}}{\Delta x} \right) - \left( v_{i,j-1/2}^n - \Delta t \frac{1}{\rho} \frac{p_{i,j-1} - p_{i,j}}{\Delta x} \right) \right] = 0 \\ & -4p_{i,j} + p_{i+1,j} + p_{i,j+1} + p_{i-1,j} + p_{i,j-1} = \frac{\rho \Delta x}{\Delta t} \left( u_{i+1/2,j}^n - u_{i-1/2,j}^n + v_{i,j+1/2}^n - v_{i,j-1/2}^n \right) \end{aligned} \quad (3.11)$$

Now we have a system of linear equations in the form of

$$\mathbf{A} \vec{p} = \frac{\rho \Delta x^2}{\Delta t} \nabla \cdot \vec{u}^n \quad (3.12)$$

$\mathbf{A}$  is symmetric and each row represents one cell  $(i, j)$  on the 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 solid normal needs to match the speed of the fluid at this position:

$$\vec{u} \cdot \hat{n} = \vec{u}_{solid} \cdot \hat{n} \quad (3.13)$$

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 3.9 can be rearranged to

$$p_{i+1,j} = p_{i,j} + \frac{\rho \Delta x}{\Delta t} (u_{i+1/2,j} - u_{solid}) \quad (3.14)$$

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. For simplicity, 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 3.11 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) on every dimension. So for a  $(n \times m)$  sized simulation, we need an  $(n+2 \times m+2)$  grid that holds the information. We differ between open boundaries where those extended cells are just air cells and closed boundaries where solid cells surround the grid.

## 3.2 The Laplace Matrix

Taking a closer look and with some rearrangements on equation 3.11 shows that it is an approximation of the Poisson problem  $\frac{-\Delta t}{\rho} \nabla \cdot \nabla \vec{p} = -\nabla \cdot \vec{u}$ . Because of that I call **A** *Laplace Matrix* in the following.

To implement a method that extracts the Laplace Matrix, 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 3.11 to the following:

$$\begin{aligned} & -4p_{i,j} + \left[ p_{i,j} + \frac{\rho \Delta x}{\Delta t} (u_{i+1/2,j} - u_{solid}) \right] + p_{i,j+1} + 0 + p_{i,j-1} \\ & = \frac{\rho \Delta x^2}{\Delta t} \left( \frac{u_{i+1/2,j}^n - u_{i-1/2,j}^n}{\Delta x} + \frac{v_{i,j+1/2}^n - v_{i,j-1/2}^n}{\Delta x} \right) \end{aligned} \quad (3.15)$$

which results in

$$-3p_{i,j} + p_{i,j+1} + p_{i,j-1} = \frac{\rho \Delta x^2}{\Delta t} \left( \frac{u_{solid}^n - u_{i-1/2,j}^n}{\Delta x} + \frac{v_{i,j+1/2}^n - v_{i,j-1/2}^n}{\Delta x} \right) \quad (3.16)$$

We can make some very important observations from 3.11, 3.15 and 3.16 for building the Laplace Matrix in code:

- **A** is similar to an adjacency matrix. Every row represents one cell  $(i, j)$  and every column all cells on the MAC-grid. Non-neighbor entries of **A** are 0.
- Note that the coefficient of  $p_{i,j}$  in 3.11 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  $k_{diagonal}$  and set the corresponding column to 0.
- For every neighbor of cell  $(i, j)$  that is an air cell set the corresponding column to 0.

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. Also, because of the symmetry of neighboring cells, **A** is symmetric.



s	s	s	s	s	s
s	30	31	32	33	s
s	20	21	22	23	s
s	10	11	12	13	s
s	00	01	02	03	s
s	s	s	s	s	s

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

$$\begin{matrix}
 & 00 & 01 & 02 & 03 & 10 & 11 & 12 & 13 & 20 & 21 & 22 & 23 & 30 & 31 & 32 & 33 \\
 \begin{matrix} 00 \\ 01 \\ 02 \\ 03 \\ 10 \\ 11 \\ 12 \\ 13 \\ 20 \\ 21 \\ 22 \\ 23 \\ 30 \\ 31 \\ 32 \\ 33 \end{matrix} & \begin{pmatrix} -2 & 1 & & & 1 & & & & & & & & & & & \\ 0 & -3 & 0 & & & 1 & & & & & & & & & & & \\ & 1 & -3 & 0 & & & 1 & & & & & & & & & & \\ & & 0 & -2 & & & & 0 & & & & & & & & & \\ 0 & & & & -3 & 1 & & & 0 & & & & & & & & \\ & 1 & & & 1 & -4 & 1 & & & 1 & & & & & & & \\ & & 0 & & 1 & 1 & -4 & 0 & & & 0 & & & & & & \\ & & & 0 & & 1 & 1 & -3 & & & & 0 & & & & & \\ & & & & 0 & & & & -3 & 1 & & & 0 & & & & \\ & & & & & 1 & & & 0 & -4 & 0 & & & 0 & & & \\ & & & & & & 1 & & & 1 & -4 & 0 & & & 0 & & \\ & & & & & & & 0 & & & 0 & -3 & & & & 0 & \\ & & & & & & & & 0 & & & & -2 & 0 & & & \\ & & & & & & & & & 1 & & & 0 & -3 & 0 & & \\ & & & & & & & & & & 0 & & & 0 & -3 & 0 & \\ & & & & & & & & & & & 0 & & & 0 & -2 & \end{pmatrix} \cdot \begin{pmatrix} p_{0,0} \\ p_{0,1} \\ p_{0,2} \\ p_{0,3} \\ p_{1,0} \\ p_{1,1} \\ p_{1,2} \\ p_{1,3} \\ p_{2,0} \\ p_{2,1} \\ p_{2,2} \\ p_{2,3} \\ p_{3,0} \\ p_{3,1} \\ p_{3,2} \\ p_{3,3} \end{pmatrix}
 \end{matrix}$$

Figure 3.2: Laplace Matrix  $\mathbf{A}$  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  $\mathbf{A}\vec{p} = \frac{\rho\Delta x^2}{\Delta t}\nabla \cdot \vec{u}^n$  (3.12) using TensorFlows 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.
- *maskDimensions* is an array that holds the dimensions of the mask.

### 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  $64 \times 64 \times 64$  the dimension of the Laplace Matrix is  $262.144 \times 262.144$ . This would require 256 gigabyte 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:

$$maxNonZerosPerRow = dimSize * 2 + 1 \quad (4.1)$$

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

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

Figure 4.1: Example Matrix in CSR format

The *compressed sparse row* (CSR) [TW67] format tackles this memory problem by only storing information of non-zero entries of a sparse matrix. 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  $i$ th element points to the index in *data* of the first non-zero entry of the  $i$ th row.

Because  $\mathbf{A}$  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<sub>regular</sub>*)-entries per row. As a trade-off this will require slightly more memory. However we are now able to calculate the index of the row  $i_{row}$  of any *data* entry by its index  $i_{data}$ :

$$i_{row} = \left\lfloor \frac{i_{data}}{maxNonZerosPerRow} \right\rfloor \quad (4.2)$$

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<sub>regular</sub>* is now regular, each thread knows the exact position in *data<sub>regular</sub>* for every entry to comply the row-major order.

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

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.

$$diagonalOffset = [-4, -1, 0, 1, 4]$$

Figure 4.3:  $diagonalOffset$  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:    $offsets = [i_0, i_1, \dots, i_{maxDataPerRow}]$   $\triangleright$  Declare array of length  $maxDataPerRow$ 
3:    $factor = 1$ 
4:   const  $diagonal = \lfloor \frac{maxNonZerosPerRow}{2} \rfloor$ 
5:
6:   for  $i = 0, offset = 1; i < diagonal; i++, offset++$  do
7:      $offsets[diagonal - offset] = -factor$ 
8:      $offsets[diagonal + offset] = factor$ 
9:      $factor = factor * dimensions[i]$ 
10:   $offsets[diagonal] = 0$ 
11:  return  $offsets$ 

```

---

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

$$\begin{aligned}
 i_{column} &= \left\lfloor \frac{i_{data}}{maxNonZerosPerRow} \right\rfloor + columnOffsets[i_{data} \bmod maxNonZerosPerRow] \\
 &= i_{row} + columnOffsets[i_{data} \bmod maxNonZerosPerRow]
 \end{aligned} \tag{4.3}$$

## 4.2 GPU Pressure Solve

In the previous section we found an algorithm that creates and stores the Laplace Matrix efficiently on GPUs. In this section I present how the system of linear equations  $\mathbf{A}\vec{p} = \frac{\rho\Delta x^2}{\Delta t} \nabla \cdot \vec{u}^n$  (3.12) can be solved on GPUs using the Conjugate Gradient method. First, we simplify equation 3.12:

$$\mathbf{A}\vec{p} = \vec{d} \tag{4.4}$$

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

---

**Algorithm 2** Pressure Solve with cg-method

---

```

1: function SOLVE PRESSURE( $\mathbf{A}, \vec{d}, maxIterations, accuracy$ )
2:    $\vec{m} = \vec{d}$  ▷ Init helper variables
3:    $\vec{r} = \vec{d}$ 
4:    $\vec{p} = 0$  ▷ TODO initial guess?
5:   for  $k = 1$  to  $maxIterations$  do
6:      $\vec{z} = \mathbf{A}\vec{m}$  ▷ calcZ
7:      $alpha = \frac{\vec{m} \bullet \vec{r}}{\vec{m} \bullet \vec{z}}$ 
8:      $\vec{p} = \vec{p} + alpha \cdot \vec{m}$ 
9:      $\vec{r} = \vec{r} - alpha \cdot \vec{z}$ 
10:    if  $max_i(|\vec{r}_i|) < accuracy$  then ▷ checkResiduum
11:      return  $\vec{p}$ 
12:     $beta = -\frac{\vec{r} \bullet \vec{z}}{\vec{m} \bullet \vec{z}}$ 
13:     $\vec{m} = \vec{r} + beta \cdot \vec{m}$ 
14:  return  $\vec{p}$ 

```

---

All mathematical operations but operation 6 and 10 can be executed by the highly optimized cuBLAS library which is part of CUDAs standard libraries. Obviously operation 6 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. The row *rowID* of this vector is given by the CUDA thread and block id:

---

**Algorithm 3** Multiplies  $\mathbf{A}$  with an vector

---

```

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

```

---

### 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

## 5.1 Section

Citation test [**latex**].

### 5.1.1 Subsection

See Table 5.1, Figure 5.1, Figure 5.2, Figure 5.3.

Table 5.1: An example for a simple table.

A	B	C	D
1	2	1	2
2	3	2	3

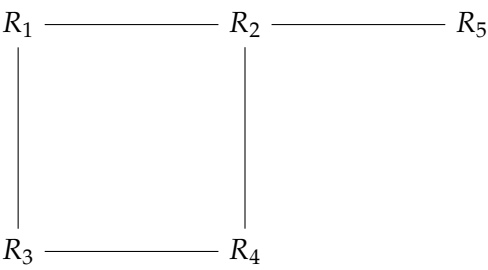


Figure 5.1: An example for a simple drawing.

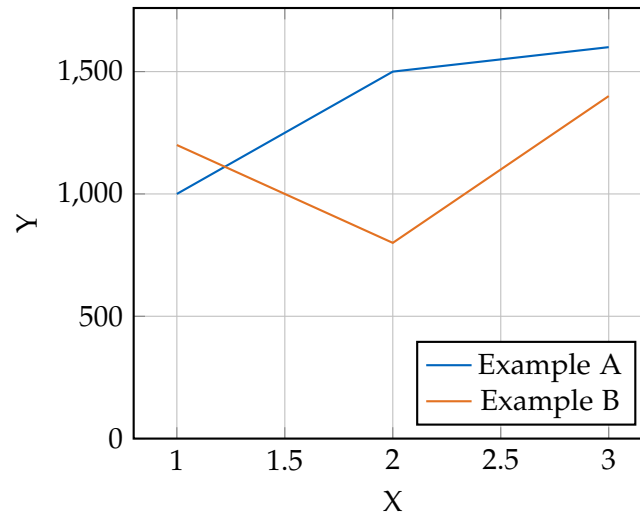


Figure 5.2: An example for a simple plot.

```
SELECT * FROM tbl WHERE tbl.str = "str"
```

Figure 5.3: An example for a source code listing.



## List of Figures

1.1	Grid-based fluid simulation in $\Phi_{flow}$ . . . . .	3
2.1	Particle-based fluid simulation in Unity3D [Tec19] [Wep17] . . . . .	5
3.1	Example grid with fluid (blue), air (white) and solid (brown) cells and a closed boundary. . . . .	11
3.2	Laplace Matrix $\mathbf{A}$ derived from Fig. 3.1 with corresponding pressure vector. Diagonal are marked as red, neighbors as blue and non neighbors are left out. . . . .	11
4.1	Example Matrix in CSR format . . . . .	13
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. . . . .	14
4.3	$diagonalOffset$ array that corresponds to Fig. 3.2. . . . .	14
5.1	Example drawing . . . . .	17
5.2	Example plot . . . . .	18
5.3	Example listing . . . . .	18

# List of Tables

5.1	Example table . . . . .	17
-----	-------------------------	----

# 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.
- [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.
- [LBH15] Y. LeCun, Y. Bengio, and G. Hinton. "Deep learning." In: *nature* 521.7553 (2015), p. 436.
- [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.
- [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.
- [PL92] T. J. Poinso and S. Lele. "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).
- [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.

- [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.
- [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.