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:	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

1 Introduction

The simulation of fluids like the behavior of water in a waterfall or smoke ascending from fire has become a key element of visual effects in the past years increasing the immersion of movies and computer games. Two main approaches have arisen to achieve realistic results: The Lagrangian method, which computes the dynamics of individual particles in respect to each other and the Eulerian method, that uses a fixed grid and computes how mass is moved within the grid. Existing solutions are often highly computational expensive leading to trade-offs between speed and accuracy, density or size and thus believability of the simulation.

Some recent approaches tackling this issue by leveraging the power of Deep Neural Networks and transforming it into a data-driven unsupervised learning problem. Until recently only Eulerian methods were fully differentiable making them suitable for deep-learning. Using simulation data such as velocity and density fields of each timestep, Deep Neural Networks can be trained to mimic the concepts of the Navier-Stokes equation [Paper of Jonathan Tompson, Ken Perlin].

For larger simulations, the amount of physical memory needed to store the training data gets huge quickly. To avoid this, running data generation and the training process itself can be done in parallel: After every time-step is calculated the network's weights can be adjusted.

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 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 to run 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 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 Using TensorFlow Custom Operations for efficient computation

This work proposes an implementation to solve the pressure equation - the most computational 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 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 leads to a solution that is both efficient and transparent and also compliant to TensorFlow's Tensors which is required to be used within Φ_{Flow}. It can then be compared to Φ_{Flows} built-in solution.

1.3 Pressure Solve and Conjugate Gradient

The Pressure Solve Problem can be expressed as a system of linear equations whose solution is the exact pressure value of each cell on a grid in order to be compliant with the incompressibility constraint of fluids. The matrix of the linear system is symmetric and positive-definite which makes it suitable to be solved by the Conjugate Gradient method. GPUs are well suited for parallel computations, especially linear algebra. The cg-method consists of only some vector operations and matrix-vector multiplications and can therefore be run on GPUs. Being a sparse matrix gives additional room for improvement, by not only keeping the required memory low but also decreasing the amount of data, that needs to be sent between the GPU's global memory to the on-chip local memory and vice versa. Compared to computation, memory operations are very costly. By not changing the boundary conditions of the simulation the matrix stays constant. Very expensive transfers between CPU and GPU can therefore kept low.

2 Related Work

TODO: Related Work

3 Navier Stokes Equations [TODO title]

The fundamentals of any fluid simulation lay in the Navier Stokes equations[TODO]:

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

$$\nabla \cdot u = 0 \tag{3.2}$$

The letter $\vec{u}$ is the velocity vector of the fluid.
The Greek letter Rho ρ stands for the density of the fluid. E.g. $\rho_{Water} \approx 1000kg/m^3$.
The arabic p on the other hand holds the pressure, which is the force per unit the fluid exerts on its surroundings. Note $\rho \neq p$.
External forces like gravity and buoyancy are collected in the letter $\vec{F}$.
The Greek letter Nu ν is called kinematic viscosity. The higher the viscosity of a fluid is the higher its inner inertia. For example honey has a higher viscosity than water.

The first of those equations - the "momentum equation" - very briefly describes how the fluid accelerates due to the forces acting on it. The second is called "incompressibility condition" and means that the volume of the fluid doesn't change.

However, real fluids are compressible, so why do we assume them to be incompressible? Liquid fluids are in theory compressible, but it requires a lot of force to do so and it is also only possible until a tiny degree of compression. Gases like air are easier to compress, but only with the help of pumps or extreme situations like blast waves caused by explosions. Those kind of scenarios are called "compressible flow" and are expensive to simulate. Additionally they are only visible on macroscopic level and hence irrelevant for the animation of fluids.

Only in those cases where animating viscous fluids like honey is desired, viscosity needs to be considered. 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

The pressure solve is part of the Navier Stokes equations:

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

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 Staggered MAC Grids

Grid based fluid simulation have been discussed in this work already, but not yet clarified what's exactly meant by "grid". To numerically approximate the Navier Stokes equations, quantities like pressure, velocity and fluid density needs to be computed through space. As the name suggest those quantities are layed out on some kind of regular grid. [Harlow and Welch 1965] have developed the Marker-and-Cell (MAC) grid technique which is still used in many simulators. Instead of tracking all quantities on the same grid, they are staggered on the grid. This avoids biased or null spaced central differences for the pressure gradient.

TODO Figures of grids

3.1.2 Discretization and derivation of the pressure equation

In this section we will 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] already explained this for both 2D and 3D in detail.
Discretizing 3.5 gives the following equations:

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

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

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

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

$$(\nabla \cdot \vec{u})_{i,j} \approx \frac{u^{n+1}_{i+1/2,j} - u^{n+1}_{i-1/2,j}}{\Delta x} + \frac{v^{n+1}_{i,j+1/2} - v^{n+1}_{i,j-1/2}}{\Delta x} \tag{3.8}$$

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

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

$$v^{n+1}_{i+1/2,j} = v^{n}_{i,j+1/2} - \Delta t \frac{1}{\rho} \frac{p_{i,j+1/2} - p_{i,j}}{\Delta x} \tag{3.10}$$

Substituting 3.9 and 3.10 into 3.8 finally gives:

$$\begin{aligned}\frac{1}{\Delta x}\Bigg[&\left(u^{n}_{i+1/2,j} - \Delta t \frac{1}{\rho} \frac{p_{i+1,j} - p_{i,j}}{\Delta x}\right) - \left(u^{n}_{i-1/2,j} - \Delta t \frac{1}{\rho} \frac{p_{i-1,j} - p_{i,j}}{\Delta x}\right)\\ &+ \left(v^{n}_{i,j+1/2} - \Delta t \frac{1}{\rho} \frac{p_{i,j+1} - p_{i,j}}{\Delta x}\right) - \left(v^{n}_{i,j-1/2} - \Delta t \frac{1}{\rho} \frac{p_{i,j-1} - p_{i,j}}{\Delta x}\right)\Bigg] = 0\end{aligned}$$

$$-4p_{i,j} + p_{i+1,j} + p_{i,j+1} + p_{i-1,j} + p_{i,j-1} = \frac{\rho \Delta x^2}{\Delta t}\left(\frac{u^{n}_{i+1/2,j} - u^{n}_{i-1/2,j}}{\Delta x} + \frac{v^{n}_{i,j+1/2} - v^{n}_{i,j-1/2}}{\Delta x}\right) \tag{3.11}$$

Now we have a bunch of linear equations in the form of $\mathbf{A}p = \frac{\rho \Delta x^2}{\Delta t} \nabla \cdot \vec{u}^n$. $\mathbf{A}$ is symmetric and each row represents one cell (i, j) on the grid.

3.1.3 Boundary Conditions

Solids

If an inviscid fluid is in contact with a wall respective solid, it should move in the tangential direction of the solids normal. To achieve this the velocity components of the solid normal needs to match the speed of the fluid at this position:

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

Concerning the pressure update, assume grid cell (i, j) to be fluid and its neighbor $(i + 1, j)$ to be solid. Then $u^{n+1}_{i+1/2,j} = u_{solid}$ and (3.10) can be rearranged to

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

Air

Compared to water, air is 700 times lighter. For simplicity the pressure p of air regions may be set to zero since it has only very little impact and therefore no significant visual effect.

4 CUDA Pressure Solver

In this chapter we will find an efficient numerical approach to solve the system of linear equations $\mathbf{A}p = -\frac{\rho \Delta x^2}{\Delta t} \nabla \cdot \vec{u}^n$. $\mathbf{A}$ to find p using TensorFlows Custom Kernels and Nvidia CUDA. 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.

4.1 The Laplace Matrix

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

$$\begin{aligned} &-4p_{i,j} + \left[p_{i,j} + \frac{\rho \Delta x}{\Delta t}(u_{i+1/2,j} - u_{solid})\right] + p_{i,j+1} + 0 + p_{i,j-1} \\ &= \frac{\rho \Delta x^2}{\Delta t}\left(\frac{u^n_{i+1/2,j} - u^n_{i-1/2,j}}{\Delta x} + \frac{v^n_{i,j+1/2} - v^n_{i,j-1/2}}{\Delta x}\right) \end{aligned} \tag{4.1}$$

which results in

$$-3p_{i,j} + p_{i,j+1} + p_{i,j-1} = \frac{\rho \Delta x^2}{\Delta t}\left(\frac{u^n_{solid} - u^n_{i-1/2,j}}{\Delta x} + \frac{v^n_{i,j+1/2} - v^n_{i,j-1/2}}{\Delta x}\right) \tag{4.2}$$

We can make some very important observations from 3.11, 4.1 and 4.2 for building the Laplace Matrix in code:

- $\mathbf{A}$ is similar to an adjacency matrix. Every row represents one cell (i, j) and every column all cells on the MAC grid.

- 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 row (i, j) is 1 in **A**.
- Every other cell which is no neighbor of row (i, j) is zero
- For every neighbor of row (i, j) that is a solid cell decrement $k_{diagonal}$ and set the corresponding column to zero.
- For every neighbor of row (i, j) that is an air cell set the corresponding column to zero.

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.

5 Results

TODO: Results

List of Figures

List of Tables