Utrecht University

MSc thesis in Game and Media Technology

Utrecht University - Faculty of Beta Sciences

Improved methods on GPU based versatile and efficient hydrodynamics code for scientific applications

First supervisor:
Dr. D. Panja

Second supervisor:
Dr. J. de Graaf

Author:
Florian Gaeremynck
7006705
f.k.r.gaeremynck@students.uu.nl

12/12/2022

Contents

Abstract

Fluid simulations are very expensive computational tasks. There is a balancing act involved between the complexity of the system and its relative scale. Being able to optimise these algorithms at scale makes this trade-off more forgiving and allows researchers to perform more detailed simulations. As such this is a topic of ongoing research and development. There is a general trend towards integration with techniques and hardware used in the field of high performance computing (HPC) [WJ17]. We investigate and expand the use of heterogeneous systems taking advantage of the parallel architecture of a graphics processing unit (GPU) to optimise parts of the fluid simulation algorithm. In this thesis we expand on work done by Stam *et al.* [Sta21]. In this preceding work an algorithm was developed to offload parts of the task to the GPU. We use this algorithm to simulate more complex fluid systems. We also improve on the serial logic of the algorithm and discuss how this ties into parallel performance down the line. Additionally, we expose the resulting simulation data to scripting languages allowing for easier analysis.

1 Introduction

1.1 Mathematical background

To describe the behavior of a fluid we make use of the Navier Stokes equations. This set of two equations is used by most computational fluid dynamics (CFD) solvers to predict a flow for the fluid that is in line with reality. The equations are the following:

$$\vec{\nabla} \cdot \vec{u} = 0 \tag{1}$$

$$\rho \left(\frac{\partial}{\partial t} + \vec{u} \cdot \vec{\nabla} \right) \vec{u} = -\vec{\nabla} p + \mu \nabla^2 \vec{u} + \vec{F} \tag{2}$$

where:

- ρ is the density of the fluid (kg/m^3)
- u is the flow velocity (m/s)
- μ is the dynamic viscosity of the fluid (kg/ms)
- t is time (s)

The first of the two equations states that the divergence of the velocity is 0 at every point, meaning that the fluid cannot be compressed. The second, more complex looking equation, expresses the conservation of momentum. It can be read as applying Newton's second law ($\vec{F} = m\vec{a}$) to the context of fluids. Fortunately for us, when dealing with fluids at the microscopic scale, the second equation is simplified significantly.
To understand why, we need to look at the non-linear terms in the fluid velocity $\vec{u}$ in equation 2. More specifically, it is the material derivative term that causes this non-linearity:

$$\frac{\partial}{\partial t} + \vec{u} \cdot \vec{\nabla} \tag{3}$$

The importance of this non-linear term is linked to two different properties of a fluid called the Reynolds- (Re) and Strouhal- (St) numbers. Both serve as a form of dimensional analysis [MS15]. They are defined as follows:

$$Re = \frac{\rho u L}{\mu} \tag{4}$$

$$St = \frac{L}{ut} \tag{5}$$

One additional term is introduced here, L which is analogous to the linear dimension (m).
The Reynolds number expresses the importance of inertia in the system relative to viscous dissipation. For high Reynolds numbers ($Re \to \text{inf}$) the viscous terms vanish, while for low Reynolds numbers ($Re \ll 1$) the inertial terms vanish. Similarly, the Strouhal number expresses the importance of convective transport relative to temporal perturbations.
It is important to note that both Re and St are dimensionless, and the exact weighting they apply is dependent on the characteristics of the body that is applying a force on the liquid, as well as the characteristics of the liquid itself.

To bring this all back to why the Navier-Stokes equations are simplified in the context of microscopic environments. It is well understood ([Rey83] [Sta21]) that for sufficiently small values of Re, and a combination of $ReSt \gtrsim 1$ which holds for fluid flow in very small environments, equation 2 may be expressed as follows:

$$\mu \nabla^2 \vec{u} = \vec{\nabla} p - \vec{F} \tag{6}$$

This is what is known as the linear Stokes equation, sometimes also referred to as "laminar flow", which describes the tendency of a fluid to flow in parallel layers. That means the fluid mixes primarily through dissipation, and not through lateral motion [Rey83]. What this means for us is that the complex non-linear term is gone.
As the name "linear Stokes equations" suggests, our math problem just converted from a non-linear to a linear one. This means that we may take advantage of a range of linear differential equation techniques to solve various fluid flows.

1.2 Trends towards parallel compute

As is expressed in various publications on the state of the art and prospects of CFD [WJ17][XXP+18]. There is a need for and a trend towards increasing use of high performance computing (HPC) techniques in CFD applications. As Witherden *et al.* [WJ17] explain in their article on the future directions of computational fluid dynamics there is an increasing heterogeneity within HPC. Massively parallel accelerators, including GPUs from AMD and NVIDIA, and co-processors from Intel are now a mainstay. These are used in tandem with more traditional CPU architectures. Relegating compute-intensive tasks of CFD simulations to a parallel architecture is an avenue which the field wants to embrace and move towards.

1.3 Previous work

In the previous thesis by Stam *et al.* [Sta21], on which this work has built, techniques for solving linear equations as described above have already been experimented with. Specifically, they took advantage of the Fourier transform. The Fourier transform is an operation which transforms a space or time-dependent function domain into a set of frequencies. These frequencies exactly describe the spatial domain while not losing any data in the process. This method is widely used to solve partial differential equations (PDEs). It works by first deconstructing a function into frequencies, solving the equation for each of those frequencies separately, and lastly, transforming the result back into the original domain. This is done to calculate a fluid velocity field from an input pressure field. Stam *et al.* derived the velocity and pressure fields as a function of the external force field from the Stokes equations:

$$\check{p}(\vec{k}) = \frac{i\vec{k} \cdot \check{\vec{f}}(\vec{k})}{k^2} \tag{7}$$

$$\check{\vec{u}}(\vec{k}) = \frac{1}{\mu k^2}(I_3 - \check{k} \otimes \check{k})\check{\vec{f}}(\vec{k}) \tag{8}$$

Here $\check{p}$ and $\check{\vec{u}}$ are the Fourier transformed pressure and velocity fields respectively. I_3 is the 3×3 identity matrix. The frequencies making up the signal of the flow field are all periodic in the domain of the simulated system. The only frequency mode that introduces net flow is the 0^{th} frequency mode. This is problematic because its $\vec{k}$ is as follows:

$$\vec{k} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \implies k^2 = 0 \tag{9}$$

in equations 7 and 8 this creates a situation where a denominator becomes 0. The algorithm at hand opts to solve this by omitting the 0^{th} frequency so that it does not contribute to the flow. This introduces a limitation because the 0^{th} frequency mode expresses the net flow of the system. Because it is omitted, the volume of fluid flowing out of the system must always be equal to the volume of fluid flowing into the system at all times.

Stam *et al.* delivered a code base to perform such laminar flow simulations using Fourier transformations and focused on porting their algorithm to run on modern graphics processing units (GPUs) to decrease runtime. The algorithm employed by the program performs all Fourier transformations through library functions that utilize CUDA, a general-purpose GPU (GPGPU) language developed by NVIDIA used to take advantage

of massively parallel compute capabilities on GPUs manufactured by the same company. This program was then used to simulate a system exhibiting Couette flow between two boundaries positioned parallel to one another on either side of the system illustrated in figure 1a. As can be seen from the image the system also adheres to a periodic boundary condition, meaning that if the fluid reaches the set limit of any of the 3 spatial dimensions, it will continue on the other side as if the system were continuous.

In order to know if the velocity field is correct the steps of transforming and solving the derived stokes equations are done continuously within an overarching gradient descent algorithm. The gradient descent aims to solve a no-slip boundary condition, meaning that at every point where the fluid comes into contact with a solid boundary of the system the fluid velocity must be 0. It does this by calculating virtual forces in the solid sections of the system which counteract the real fluid forces, this way the relative velocity on the boundaries remains 0. The program relegates one more step to the graphics card pertaining to this gradient descent algorithm, namely, the calculation of the gradients of velocity with respect to force defined as:

$$G_{x_i x_j} = \frac{\delta u_{x_i}}{\delta f_{x_j}} \tag{10}$$

While, as we will demonstrate later, this is a factor for the performance of larger simulations. It is still a pre-processing step, not part of the main gradient descent loop and as such its impact on the runtime of the previous code-base is minimal. A more in-depth technical explanation will be provided in the next chapter.

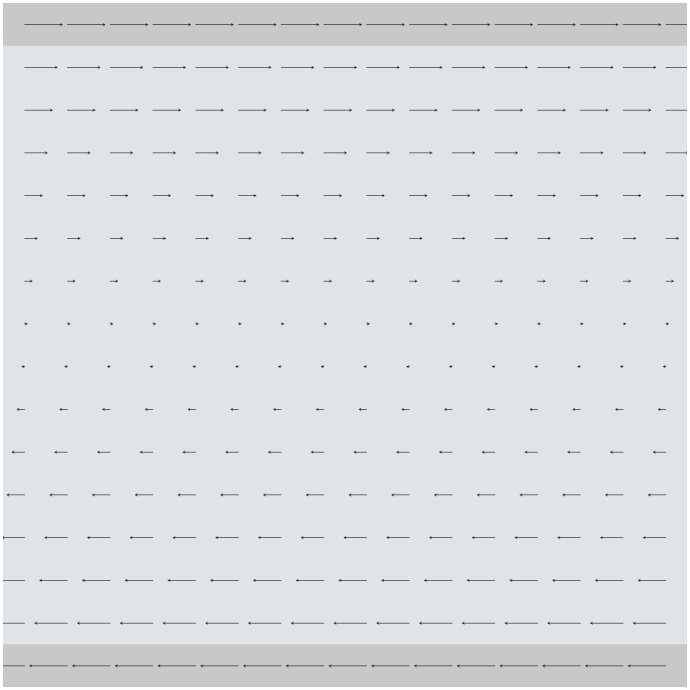
1.4 Taking the next steps

The previously described research leaves plenty of avenues for expansion and improvement which are further developed in this paper. Specifically, the following two aspects which serve as our initial research questions:

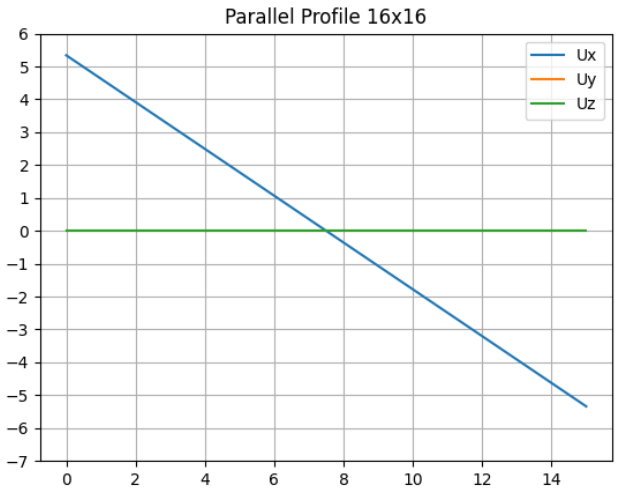
- We examine the quality of the code pertaining to flexibility and by extension maintainability with the goal of adding a number of new configurations to be simulated.
- We also explore better ways of interacting with the program. Mainly interaction with the result data, but also invoking the executable.

As will be explained later in this thesis, we did this by thoroughly stress testing the system and overhauling a lot of the underlying structure in order to solve the new

configurations. We have also reworked the gradient descent part of the algorithm, all Leading to great improvements in single core performance. Following this, we speculate on taking these improvements to a concurrent system which harnesses the power of the GPU in much more aspects of the program. Lastly, we have also exposed the result data to be readable in scripting languages in order to make a more detailed analysis of results aimed at users that are not necessarily familiar with C++ and CUDA.



(a) a 16x16 system exhibiting Couette flow between two solid moving plates. The darker cells represent solid regions where virtual forces are applied. The lighter cells are the open regions of the system where the fluid flows, the arrows in these cells are the velocity vectors of the actual fluid flow.



(b) The flow profile of figure 1a, this graph represents the x, y and z velocity of each cell in the first column.

Figure 1: Figures 1a and 1b illustrate both the resulting vector field of a simulation and its corresponding flow profile.

2 Methodology

2.1 Technical details of the code-base

A brief overview was already given in the previous chapter but here we will give a detailed explanation of how the program solves the no-slip boundary condition and by extension the fluid flow from the initial input configuration and boundary condition. Here we will explain the pipeline that produces the results of the fluid simulations from the various configurations of solids input by the user. We will also delve deeper into some of its steps. Throughout its runtime the code goes through four distinct steps:

1. Setting up the configuration
2. Calculating the velocity field from the pressure field
3. Solving the no-slip boundary conditions across all boundaries in the system
4. Displaying the solved information

These four steps are compiled together into a main program loop that looks roughly as follows:

Algorithm 1 Main program loop

$u \leftarrow DesiredVelocity$
function SET CONFIGURATION(u) ... ▷ Setting configuration can take many forms
end function
while $Cost > \epsilon$ **do**
  $Cost \leftarrow \sum_{n=0}^{boundarypairs} (u_n - u)^2$
  function CALCULATE ... ▷ Calculate Velocity field
  end function
  function SOLVE ... ▷ Perform a gradient descent step towards solving the no-slip boundary condition
  end function
end while
function DISPLAY
end function

2.1.1 Initialization of configuration

In the first step the configuration of solid and empty cells that will be the stage, if you will, of the simulation is initiated. To give an example: in the aforementioned parallel plate configuration, the top and bottom rows of the grid are defined as solid cells and their desired fluid velocity at the boundary is set. This velocity coincides with the idea that the plate moves relative to the fluid, this way the no-slip boundary condition can be

satisfied while still simulating interesting fluid flows. If the plates do not move the fluid needs to be static at all times. This is because the program yields undefined behaviour when the fluid exhibits a net flow. A more in-depth explanation of this will be given in chapter 3.

2.1.2 Calculating and solving the velocity field

The second and third steps are the core of the simulation and run repeatedly in an alternating fashion until the no-slip boundary condition is solved. As seen earlier, this means that the relative velocity where the fluid is tangent to the boundary must be 0. A solution is accepted when the cost is lower than a small percentage ϵ of the initial cost. Up until now, part of the *Calculate* step is the only one that is completely handled by the GPU. The solver invokes the GPU in one instance, to calculate the gradients (11) but is otherwise, a function ran on the CPU.

$$\frac{\partial u}{\partial f} \tag{11}$$

The calculate step transforms the pressure field into Fourier space. At this point the Stokes equations become linear problems and the correlating velocities can be calculated easily before transforming them back to real space. This is the one step that makes use of the GPU when transforming to and from Fourier space. The solving of the equations once they are transformed into Fourier space also happens on the CPU side.

Next, the solver stage takes the newly calculated velocity field and tests the velocities against the no-slip boundary condition using a cost function. If the cost remains outside the acceptable margins, the solver will attempt to nudge the pressure field to manipulate the velocities in the right direction using a gradient descent algorithm. This algorithm makes use of the gradient describing the change in the cost function relative to the change in pressure.

$$\frac{\partial C}{\partial f} \tag{12}$$

It then uses this to nudge the result to meet a minimum in the cost function. This way convergence can be reached so the no-slip boundary condition is satisfied. We will revisit this when we explain how we improved the gradient descent algorithm in chapter 4.

2.1.3 Displaying results

Lastly, the display step renders the velocity field to the screen. This is done from within the core code-base by passing a reference to the "Stokes" solver object, in such way that that all information can be read directly from the solved configuration.

2.2 Methodology of this project

We started off by attempting to implement the configurations we wanted using the existing code and altered it where needed as we went along. Several configurations were implemented and subsequently used as a baseline to see how far the size of the simulation could be pushed. Additionally, some new techniques for visualizing the results were developed in order to get more information and spot errors faster. This will also serve as a basis upon which future work can build to improve the interaction between the user and the system.

2.3 Tooling

For completeness, we will also be continuing the same tool use as Stam *et al.* No alterations were made to the use of FFTW for Fourier transforms and CUDA for general purpose GPU code. Stam *et al.* [Sta21] again contains a more in-depth breakdown of why these were used. Specifically, an analysis of why to use CUDA over other general purpose GPU (GPGPU) languages like OpenCL. As well as a breakdown of FFTW and its planner step to optimize between small- and large-scale Fourier transforms. Lastly, we make use of python to do scripting for both interacting with the code base and analyzing the results of the various simulations.

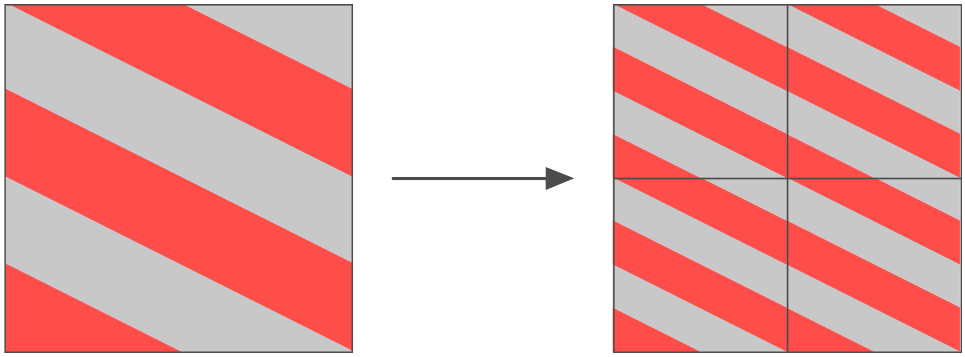


Figure 2: A working implementation of tilted channels. As the figure illustrates, this specific inclination angle of the channels satisfies the periodic boundary condition.

3 Adding configurations

3.1 Tilted channels

The first configuration that is added to the program aims at simulating Couette flow between two oppositely moving tilted channels. This means that the boundaries will be tilted at an angle θ with the fluid running through it at the same angle in opposite directions.

3.1.1 Implementation

Implementation of this system has a catch and brings some limitations with it. Naively implementing it leads to a configuration in which two boundaries run parallel at an arbitrary angle θ, analogous to the previous configuration with $\theta = 0$. However, as is described in a thesis by Rempfer *et al.* [Rem10], this violates the periodic boundary condition (PBC). This condition is important to the algorithm in order to simulate the infinite expansion of the channel in all spatial dimensions and thus can not be violated.

Suppose we want to have tilted channels at an angle θ we need to construct a way such that the PBC is satisfied. For this we can start with a square of length 1 and draw a line at angle θ from one corner to the boundary of the square. The construction only works if the intersection of the line and the boundary is at a rational point. Assuming that the intersection coordinates are rational, now repeat the square into a 2D grid. Since the intersection point is rational if we continue the line we will eventually encounter a grid point. Now repeat the process width a subsequent corner of the square until another grid point is met. Now take the minimal rectangle encompassing the two lines and copy the rectangle into a square configuration. Now we have a square which we can use as our simulation playground.

3.1.2 Extending control over the creation of configurations

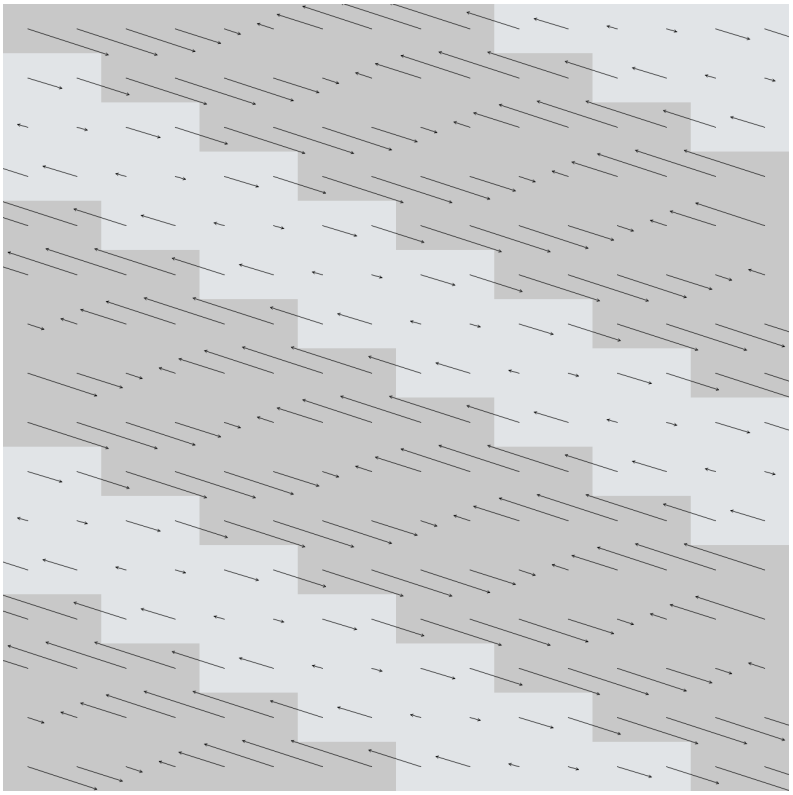
During this implementation it was clear that at the end of the previous project, the code-base creates and manipulates properties of the boundaries in a way that is specific to a horizontal Couette flow between two boundaries but cannot be scaled or skewed to meet more complex layouts. For example, it looks at the position of a certain solid cell along the y axis to determine whether it is part of the top or bottom boundary in order to assign it the proper desired velocity. These conditions of course assume the system to always be a parallel moving plates configuration. Now that more complex scenarios will be added, a general approach is needed.

The initialization of boundaries was altered in two key ways which allowed for much more control and more intuitive interaction. First, a rasterization algorithm based on Bresenham's technique [Bre65] was added to the code-base in order to make drawing arbitrary lines through the scene easy. Do note that at the time of writing the implemented version of Bresenham's algorithm only supports 2D lines to be drawn. It is not able to deal with depth in the configuration yet. This can be used for both drawing arbitrary boundaries as well as extracting information on a line of cells that don't necessarily have to be solids. The function merely returns an array with the cells it protrudes to. This is also handy for drawing up velocity graphs of the simulation in order to verify its accuracy. However, as will be shown later this responsibility was pushed on to python scripts rather than remaining in the main code-base.

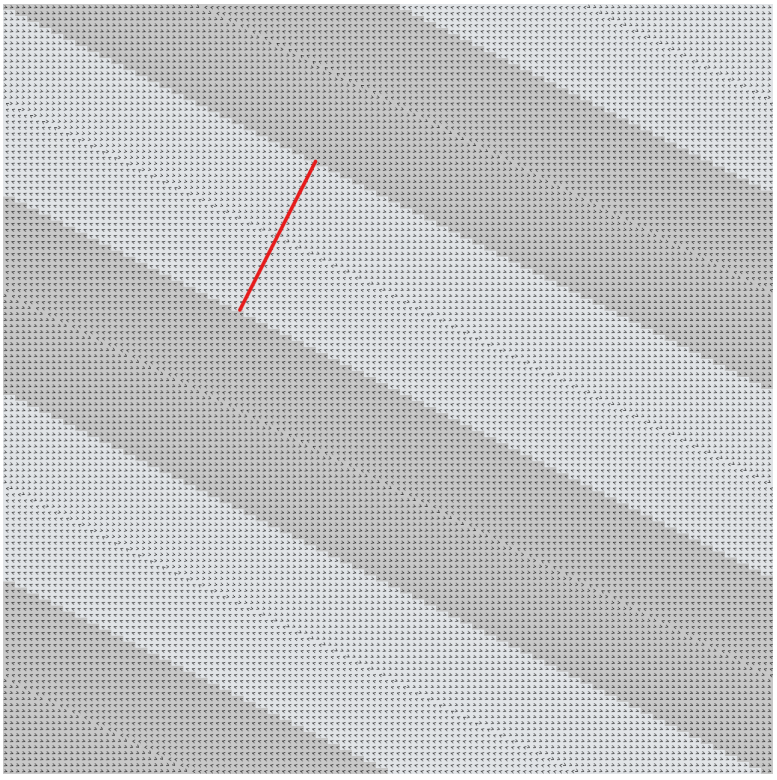
Secondly, the data structure in which the sets of boundary pairs are stored was changed to an unordered map. With this, a simple hashing algorithm is added which assigns a unique key to every pair. This is useful for assigning types to the boundaries using an enumeration system such that the correct desired velocity can be assigned later in the initialization stage.

3.1.3 Results

Figure 3a displays an example of a resulting velocity field of the tilted channel configuration. This one is a 16x16x1 field for the purpose of readability. The challenges of going to higher resolution levels are explained in section 4. To verify the tilted channel configuration, figure 5 displays a progression of the parallel and perpendicular velocity components along a line running orthogonal through one of the channels. The data is gathered using the aforementioned rasterization method and corrected using bi-linear interpolation to get the exact point along the orthogonal line through the channel. A visualization of the line along which velocities are sampled can be found in figure 3b where a 128x128x1 example is used to illustrate. The line shows the velocity profile progressing from 5 to -5 with some margin above and below because the line slightly runs into the boundary since the discretized grid does not perfectly overlap with the straight lines along which the



(a) 16x16 velocity field resulting from a simulation with tilted channels.



(b) Illustration of the orthogonal line along which velocities are sampled to get the orthogonal velocity profile of a tilted channel system. In this example the dimensions of the system are 128x128x1

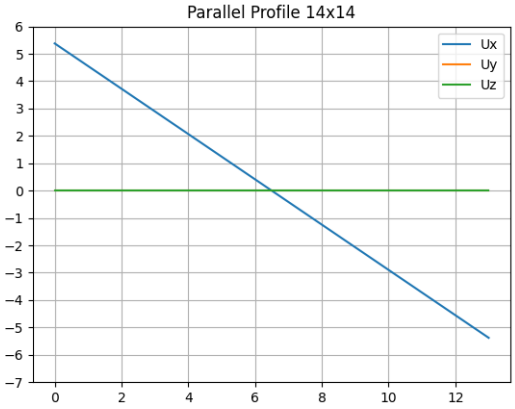


Figure 4: x, y and z velocities extracted along a vertical column in a parallel plates configuration.

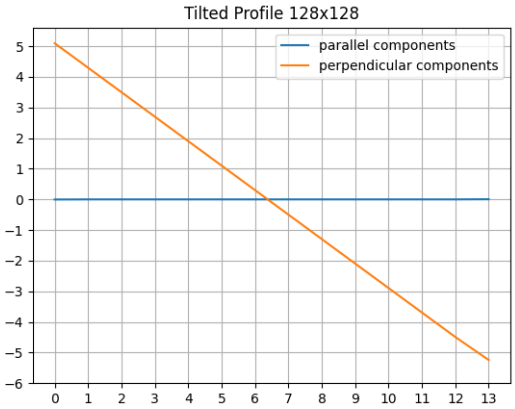


Figure 5: Orthogonal velocity profile of a tilted channel configuration. This was taken from a 128x128 configuration, the line along which velocities are extracted is illustrated in figure 3b

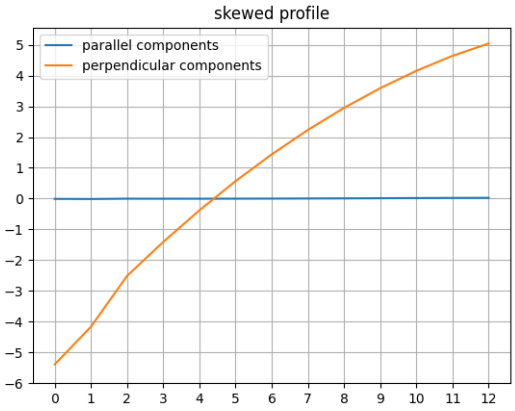


Figure 6: Orthogonal profile of a 128 x 128 simulation revealing the undesirable results in the velocity profile, the desired graph would exhibit a straight line. Do note that the perpendicular components run in the opposite direction (-5 to 5) compared to figure 5. This is simply because the desired velocities were swapped on the upper and lower boundaries in this simulation.

boundaries are constructed. Comparing this graph to the one shown in figure 4 depicting the x, y and z velocities along a vertical line in a 128x128 Couette flow example, the similarity is clearly visible. Note that at this size (128x128) the gradient descent solver already needs to be tweaked to utilize a smaller initial learning rate. Otherwise, proper convergence isn't reached which is displayed by the resulting orthogonal velocity profile (figure 6). This indicated that extensive further optimization of the code base would be needed in order to reach the larger simulation sizes we wanted to achieve. All technical details about this are discussed in section 4.

3.2 Arbitrary boundaries

In order to fully stress test the overhaul (which will be discussed in section 4) and show off its improvement over the previous iteration of the code-base, complex boundary shapes were implemented and tested. This was done in the form of sinusoidal boundaries in both 2- and 3 dimensional configurations. which yielded velocity profiles as shown in the figures 7a and 7b.

Sinusoidal boundaries were used for ease of implementation simply for this proof of concept. In the long term a proper rasterization algorithm for drawing and describing curves should be implemented [Zin16] [LWZ+11]. This would allow for truly arbitrary boundaries and a much greater diversity in the amount of configurations available to users.

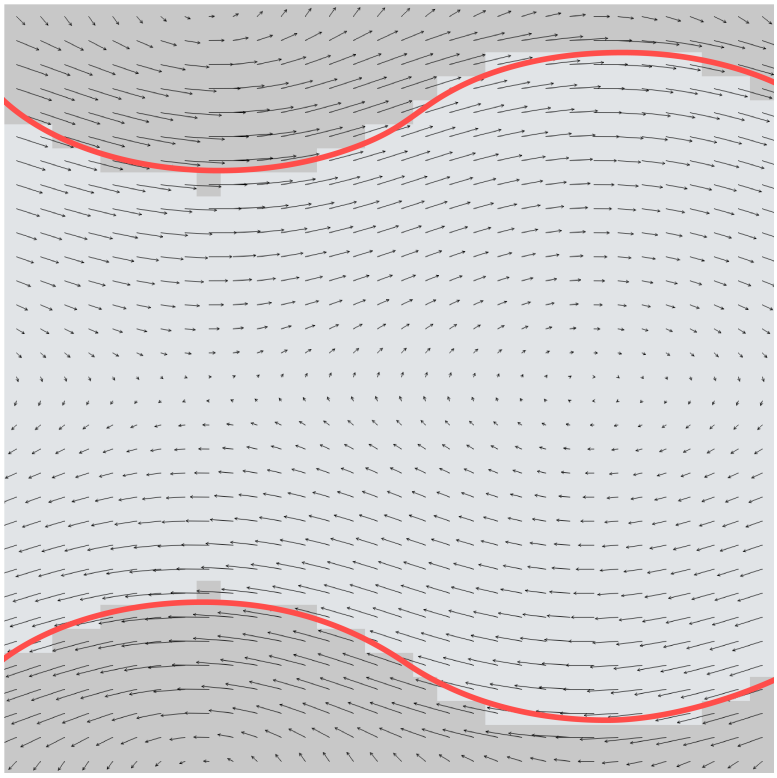
The sinusoidal boundaries were also run in 3 dimensions to further emphasize the complexity that the new solver is able to deal with. A detailed performance comparison

between this and the previous iteration of the code base will be outlined in section 4. Shown in tables in the mentioned section.

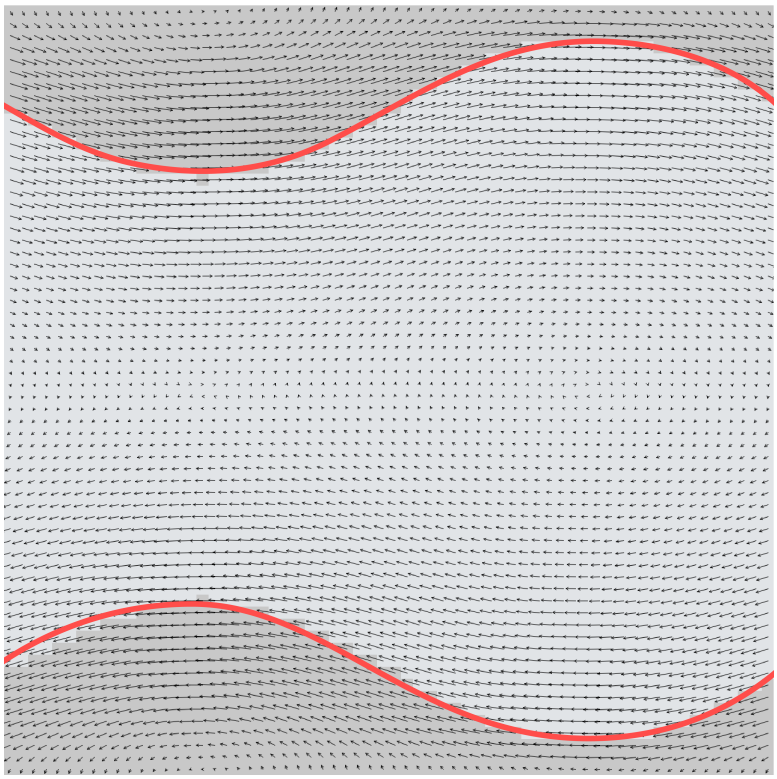
3.3 Cylindrical obstacle

Moving on from the arbitrary boundaries we can start playing around with complex boundaries which could serve other uses in the future. In preparation of having a moving object run through the configuration we also ran a test with a cylindrical obstacle in the center of the system. For now, the obstacle is kept static and the flow around it remains of Couette nature. Visually the flow around the object looks plausible. This is a welcome testament to the flexibility of the solver. The result is shown in figure 8

Taking a good look at figure 8 reveals that the point at which the arrows flip from pointing one side to the other is located slightly below the vertical center of the system. This is simply due to the velocity values here being incredibly small as they approach 0. As this is a gradient descent algorithm, the convergence is not 100% perfect so small asymmetries like this are to be expected. Forcing the solver to run for longer thus enforcing tighter convergence balances this symmetry out slightly more.



(a) Velocity field of the first layer of a 32 x 32 x 16 configuration with sinusoidal boundaries



(b) Velocity field of the first layer of a 64 x 64 x 4 configuration with sinusoidal boundaries

Figure 7: Demonstration of arbitrary boundaries at varying levels of details and depth complexity.

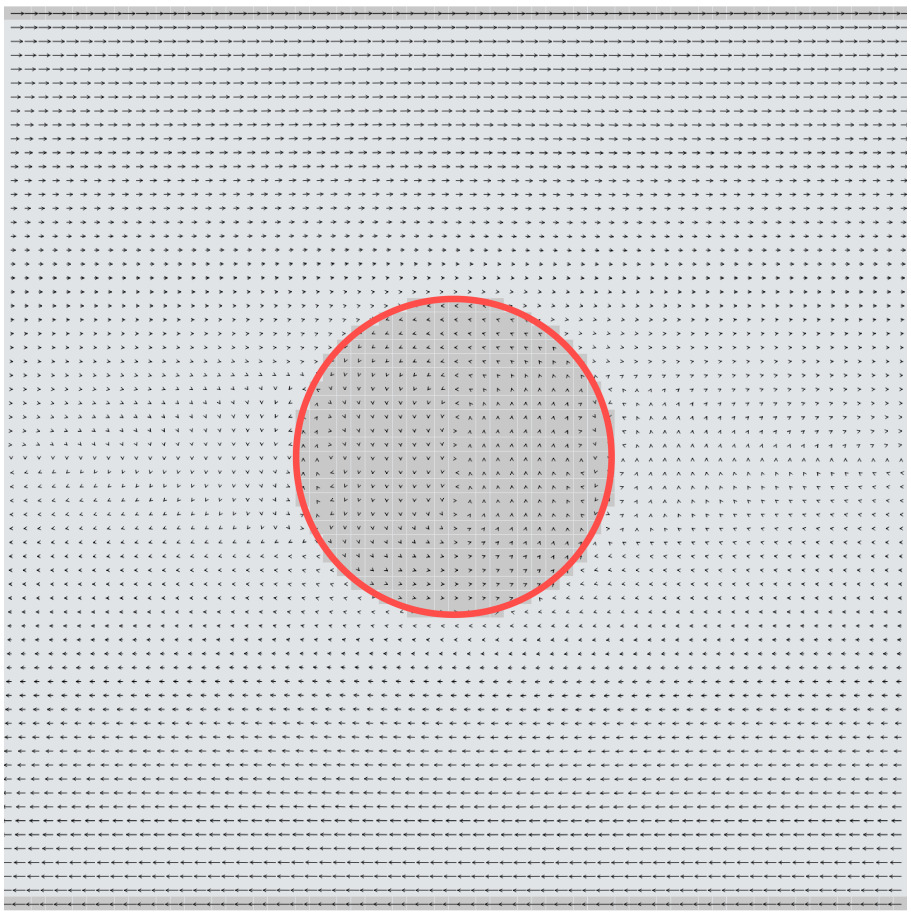


Figure 8: 65x65 Couette flow between two parallel plates with a cylindrical obstacle introduced into the system

4 Single core optimization of the solving stage

As was mentioned in the introduction, in stress testing and adding configurations we were forced to re-evaluate the performance of the program. Especially that of the gradient descent stage. These optimizations would then enable the code-base to support configurations of higher complexity and a much greater number of both solid and fluid cells. As the key Fourier transformation steps were off loaded to the GPU, all the most impactful performance bottlenecks could be found on the CPU side. This is where we focused our optimization efforts. Most of these pertain to the single core logic of the algorithm. However, as we will discuss in more detail when speculating on future work, the entire solving function lends itself very well to being parallelized. However, we did not yet go through this development as we have first taken steps to speed up the solving solution itself. Note that when we talk about single core optimizations, we also do not mean SIMD or other vectorization approaches. SIMD is short for "single instruction multiple data" and is a form of hardware accelerated parallelism present on most modern CPUs in the form of vector registers. While these would be useful, the eventual goal is to port the function to a graphics card architecture where such registers are not found on individual logical cores. Nevertheless, we will still touch on these in more detail in the chapter on future work, but they are not a part of this project.

Apart from the general CPU optimizations there were also alterations made to the gradient descent algorithm that was implemented. Specifically, the way it calculates gradients as well as how it determines and takes steps towards convergence. Lastly, we also briefly experimented with multi-threading. This was done as an experiment to probe the viability of further parallelization of the main loop in the program.

At the time of finishing this project the run time of the 128x128 tilted profile simulation was brought from 744 seconds down to 1,6 second. Parallel plates simulations have been ran all the way up to system dimensions of 1024x1024. In the following sections we will break down the technical details of the optimizations we applied to key steps of the solver stage.

4.1 Analytical gradient calculation

In the introductory chapter it was explained that the solver makes use of a gradient descent algorithm in order to bring the pressure- and consequently the velocity fields in line with the no-slip boundary condition. The gradient used in the solver looks at the difference to the outcome of the cost function against the change in force defined as:

$$\frac{\partial C}{\partial f} = \frac{\partial C}{\partial \vec{v}} * \frac{\partial \vec{v}}{\partial f}$$

in the pre-existing code-base the latter half of this gradient, $\frac{\partial \vec{v}}{\partial f}$, is pre-calculated on the GPU while first half, $\frac{\partial C}{\partial \vec{v}}$, was retrieved in a numerical fashion using the CPU. This entailed adjusting the velocity field with incremental steps before analysing the change in cost function. This was done for all 3 dimensions with respect to 3 dimensions individually. This construction resulted in a rather stark discrepancy between the two processes and slow overall computation time despite the use of the graphics card. All the while the components of $\frac{\partial C}{\partial \vec{v}}$ are known at the time of computation. Using these decreased the computational load significantly. In the cases which, after implementing analytical gradient calculation, yield stable results a speed up of a factor of ~ 9 is observed. The catch being that this implementation reveals a further shortcoming of the solver function namely the rudimentary adaptive learning rate is unable to deal with the newly calculated gradients and frequently leads to divergent results.

A note on implementation details: Analysis of the numerical against the analytical results reveals that the results differ by roughly a factor of 10 so a simple corrective multiplication suffices to stabilize some configurations and resolutions. However, a more robust method is needed which will be discussed in the next section.

4.2 Adaptive learning rate using Polyak's length

It is no secret that the gradient descent algorithm is one of tedious tuning and adjusting of the different parameters which influence it. Though this reality usually applies to complex multidimensional dependencies such as those found in machine learning methods, our algorithm has one important parameter who's value is imiportant to the result of the simulation. This parameter is the learning rate or the "step size" with which each incremental nudge toward the optimum is taken. Make the step size too large and the algorithm will overshoot the minimum and diverge into increasingly bigger and nonsensical values. Make it too small and the optimum will eventually be found but the simulation will take longer than it could to find such a result. Since optimization is the name of the game in this project, it is important that a good learning rate is found based on the optimization function that is being solved.

Before, the gradient descent algorithm took an initial learning rate (η_t) that was defined by the user and made use of the following logic to adapt this step size when the function overshot its target:

$$C_t > C_{t-1} \rightarrow \eta_{t+1} = \frac{\eta_t}{2}$$

The cost function which is the subject of this solver is however convex, meaning there exists only one global minimum. Note that this does not mean the retrieving of such a minimum is trivial. The function may still exhibit a steep, multidimensional topology

which is challenging not to diverge from. Regardless, in 1987 Boris T. Polyak published a book [Pol87] containing a method for dealing with convex optimization problems that should theoretically always find the global minimum. The technique adapts the learning rate according to the local topology of the function or the "slope" so to speak.

If the distance to the optimum from a certain iteration t is notated as $h(\mathbf{x}_t) = h_t = f(\mathbf{x}_t) - f(\mathbf{x}^*)$ where $\mathbf{x}^*$ is the optimum distance. Polyak's method describes the learning rate η_t as follows:

$$\eta_t = \frac{h_t}{||\nabla_t||^2}$$

Implementing this way of determining the learning rate at each step of the algorithm has a much higher success rate at finding a stable solution to the optimization problem. Especially in complex cases where a great number of both solid and fluid cells are used and where the previous method fails. Keep in mind that the initial learning rate still must be adjusted by the user but divergence is much rarer when using the Polyak length while maintaining over 99% convergence. The previous method is however not removed from the code-base entirely and the option should still be available to the user. When doing different tests and comparing them to each other, the learning rate halving technique shows occasional improvements over the Polyak's method. That said, the instances are rare, and it is better for the user to consider which one is needed for their use case. Tables 1 and 2 show the different results obtained from using different techniques. The specifications of the computer used to run these tests are listed in table 3. It must be noted that in these cases the initial learning rate was set to 0.1 for the parallel plates and 0.01 for the tilted plates. Highlighting the need for user intervention and tuning according to the needs of the simulation.

Table 1: Couette flow between horizontal parallel plates

32x32	LR halving	Runtime	Optimization	Polyak LR	Runtime	Optimization
Numeric Gradients		3s	99.99%		¡1s	Divergence
Analytic Gradients		183ms	100.00%		13ms	100.00%
64x64	LR halving	Runtime	Optimization	Polyak LR	Runtime	Optimization
Numeric Gradients		13s	99.99%		¡1s	Divergence
Analytic Gradients		305ms	100.00%		118ms	100.00%
128x128	LR halving	Runtime	Optimization	Polyak LR	Runtime	Optimization
Numeric Gradients		81s	99.95%		¡0s	Divergence
Analytic Gradients		3s	28.49%		1s	100.00%

Across the board however, it is clear that analytical gradients yield much better results by orders of magnitude much greater than 9. This is also due to additional optimizations

Table 2: Couette flow between tilted parallel plates

32x32	LR halving	Runtime	Optimization	Polyak LR	Runtime	Optimization
Numeric Gradients		89s	99.88%		¡1s	Divergence
Analytic Gradients		419ms	99.57%		41s	99.93%
64x64	LR halving	Runtime	Optimization	Polyak LR	Runtime	Optimization
Numeric Gradients		345s	99.95%		¡1s	Divergence
Analytic Gradients		6s	99.95%		197s	99.97%
128x128	LR halving	Runtime	Optimization	Polyak LR	Runtime	Optimization
Numeric Gradients		744s	99.98%		2s	Divergence
Analytic Gradients		118s	99.37%		803s	99.99%

Table 3: PC specifications:

CPU	Intel i7-9750H
GPU	NVIDIA GeForce GTX 1660 Ti
Memory	16 GB DDR4
PCIe gen	3
Year	2020

happened in between the initial tests and these results where the code was optimised for CPU cache hits which further yielded shorter simulation times.

Another thing which was discovered during this process is that the solver spends about $80\% - 90\%$ of its computation time solving the last 0.1% of the convergence. Because of this it is useful to also expose the tolerance threshold to the user for them to tune to the accuracy needs of their simulations. Because lowering the solvers tolerance to 99.9% in stead of 99.99% increased simulation speed by another significant factor. We also developed a method we called "Dynamic Polyak Length" which is a much better solution to this problem that also works in cases where very close convergence is needed. The specifics of this will be discussed in the next section.

The performance increases in the table are quite varied but on average the speed up is roughly of a factor of 60. This is an incredible upgrade over the previous work. However, it can most likely be improved even further. Note that these results are only tested on 2D configurations. When the 3rd dimension was included (in the example of arbitrary boundaries) the performance suffered once again.

4.3 Dynamic Polyak length

As was observed, near complete divergence ($99.9\% - 99.99\%$) the step size gets so small that the algorithm on average spends $80\% - 90\%$ of its runtime on this tiny convergence improvement. We reckoned we could speed up these steps by looking for sequences of gradient descent steps in which the cost consistently drops and carefully trying to take

bigger steps. When the cost consistently drops over a great number of steps, we may assume that the algorithm is walking in the right direction but because the topology of the function is so shallow, the step size calculated by the Polyak length is so small that it takes many of them to go where we are heading anyway. In short, we count the number of successful steps, when they reach a predetermined threshold the step size gets multiplied by a factor of 10. This is kept up until either the success threshold is reached again, at which point we multiply the step size once again because it indicates we can go even faster. Alternatively, a number of missteps is taken which also exceeds a predetermined critical point, at which point we decrease the custom learning rate multiplier with a factor of 0.1. After a bit of trial and error the thresholds that we landed on were at 100 successful cycles and 2 consecutive missteps.

Algorithm 2 Dynamic Polyak Length

$Learningrate \leftarrow 1$
$FailStreak \leftarrow 0$
$SuccessStreak \leftarrow 0$
if $NewCost > PrevCost$ **then**
    $FailStreak \leftarrow FailStreak + 1$
    if $FailStreak \geq 2$ **then**
        $FailStreak \leftarrow 0$
        $LearningrateMultiplier \leftarrow LearningrateMultiplayer * 0.1$
    end if
else
    $SuccessStreak \leftarrow SuccessStreak + 1$
    if $SuccessStreak \geq 100$ **then**
        $SuccessStreak \leftarrow 0$
        $LearningrateMultiplier \leftarrow LearningrateMultiplayer * 10$
    end if
end if
$Learningrate = Learningrate * LearningrateMultiplier$

The addition of this modifier to the Polyak length algorithm to spur on the solver when it is too cautious with its steps towards convergence yields significant speedups. Especially in bigger and/or more complex amalgamations of solid and fluid cells in the simulated system. The runtimes (in seconds) are further displayed in table 4

Table 4: Tilted channels runtime (s)

	Previous Method	Polyak Length	Dynamic Polyak Length
32x32	89	41	0.765
64x64	345	197	0.870

Do note that the use of the "Dynamic Polyak length" is no silver bullet and we have found some cases where disabling it yielded better faster runtimes for the simulations in

question. An example of this is the 128x128x1 tilted channel configuration which sees significant slowdowns when using the method. Whether or not the use of this algorithm to speed up the Polyak length is needed is up to the researcher to determine through trial and error.

As was mentioned multiple times already, all of these alterations pertain to algorithms that are currently only executed on single CPU cores. Use of the graphics card in improving these solver algorithms are key to this project in the long term. In the next sections some experimentation with parallel computing principles gives insights into avenues for further bettering the run-time of the simulations.

4.4 Multi-threading

Calculation of the gradients, both numerically as well as analytically is a process which lends itself well to parallelization. Different components of the gradient with respect to the 3 spatial dimensions do not depend on each others outcome to obtain their results. Furthermore, the process is flow independent, meaning that at no point a decision needs to be made based on a given condition (if-else statements). The combination of these two characteristics, in theory, make the function an ideal fit to make use of multi-threaded architectures present on modern CPU's and GPU's.

Initially some experiments were done as a proof of concept to show that in further work on the solver can take these concepts even further. The work was split between two CPU threads. This did require some abstraction of the function and creation of additional chunks of memory to bypass restrictions concerning read and write violations. Fortunately, after the fact a nice 2x speedup was observed on the 32 x 32 tilted channels configuration using Polyak's length from $41s$ to $20s$ run-time. This optimization is however not present in the final code-base produced by this project. This is because the implementation was very aggressive and severely under tested. We fear that it might destabilize other parts of the pipeline and since a full implementation would take a not insignificant amount of time, we have left it as a proof of concept for future work. There we will also mention the step of vectorization which is usually advised before moving into multi-threading. In general, writing good serial code is a prerequisite for achieving a good parallel counterpart. Stretching the single core performance as far as possible ensures the best starting point for parallel algorithms.

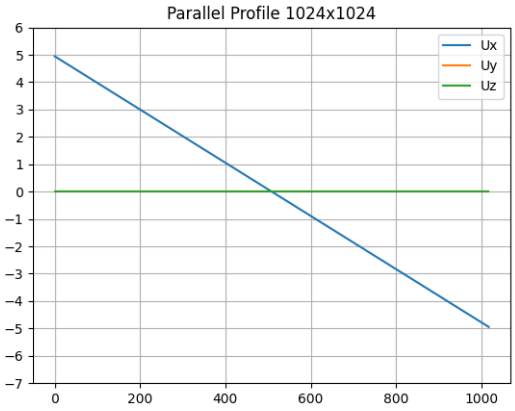


Figure 9: Flow profile of the system with dimension 1024

5 Expanding Simulation Size

The optimization techniques previously discussed that were initially developed to help the code base deal with the increasing complexity of the added configurations also allows us to expand the simulation sizes that were previously not possible. The new code is able to simulate a Couette flow configuration with a system dimension of 1024 where the previous method went no further than 64.

5.1 Forcing convergence

The code does start to struggle in converging to a good solution once sizes become very large. On the one hand this is caused by the threshold of 99.99% being too lenient relative to the error in large configurations. On the other hand, the numerical precision of floating point numbers starts to show its limit as step sizes become too small for the computer to handle. The program often exits when it detects the cost to be equal between iterations while it is still a ways away from convergence within the desired threshold. Simply because the computer is unable to see a difference between the two costs. To remedy this, we created a second executable where the solving stage makes use of double precision floating point numbers. This again allows for scale up to the point of reaching the size limits of the double. Next logical steps from here are to make use of long doubles or even program custom data types which encode floating points in even more bytes. The risk associated with this becomes then the memory footprint of the program. This becomes a particularly tricky problem when transferring data to and from the GPU as video RAM is often smaller in size compared to overall system working memory. Not only this, when the system gets eventually scaled up to make use of the entire graphics card, it will need a way of intelligently streaming/sharing data with the GPU which becomes a more challenging task when the amount of data is much bigger.

A second way of tackling this problem would be to, instead of breaking the gradient descent loop when equal costs are detected, to artificially bump up the step size similar to what we did with the dynamic Polyak length technique. This way the solver will more likely come up with a smaller cost which is further from the previous one. This artificial process does run the risk of creating divergence, especially when the solution gets incredibly close to its solution. This technique was used to force better convergence on the cylindrical boundary configuration. Here we found a sweet spot of multiplying the step size by 3 whenever equal cost was detected. However, we did not experiment with it enough to say this technique is flawless. We speculate that a better solution to forcing better convergence will probably be found in a combination of better precision and slightly forcing the solver past cases of equal cost. Right now, when we attempted to run this technique to its limit, only exiting when convergence was found rather than an iteration limit. The simulation was run for over 10 hours and still did not yield a result.

6 Scripting

Lastly, as was mentioned earlier in the introduction, there is a need for improvement of user interaction with the output of the simulation. Everything from visualization, to verification, to analysis of the data should ideally be done outside the realm of the C++ code base. Reasons for this being that C++ is a rather difficult language to pick up an learn for new users, especially those who are not primarily interested in using it other than for this application. Additionally, scripted and interpreted languages like Python, R or Julia are more often used in computational science and engineering and as such are more likely to be familiar by the users of this program. Unfortunately, they sacrifice significant performance when compared to a low-level language such as C++ [ZJKP20].

Ideally, all C++ code is left under the hood where it performs the bulk of the simulations and can be called and manipulated from Python scripts. Also, the output should be dumped into a readable file which the user can use as an input to their own scripts in order to analyse and/or visualise the data.

Previously, when the simulation was finished, a separate class within the C++ code would read in the results and draw up the final velocity field to the screen. Unfortunately, this was only really useful in case of low-resolution simulations as bigger ones made the cells so small the vectors where not readable anymore. It also did not allow for much control from the user to tackle these issues. Moving forward, the solution of the simulation should be written to an output file to be parsed by scripts writing by the users. This allows for great flexibility in analysis and display of the results. It is also further in line with the idea of keeping the code-base as user friendly as possible.

6.1 Examples

During this project this concept was worked out with an output file in plain text format. Resulting velocity and pressure components were written to the file when the simulation was done as well as information of whether or not the cell in question was solid. A number of scripts were made using Python in order to analyse the data for different purposes which included:

1. Extracting x, y and z velocities along columns in the simulation (for example, figure 4). This also allowed for averaging over columns and other manipulations of the arrays.

2. Extracting perpendicular and parallel velocity components along line parallel to the tilted channels (figure 5). This script was kitted out with the same Bresenham rasterization algorithm as well as the bi-linear interpolation used to better the accuracy of the extracted values.

3. Interactive visualisation of the velocity field which showed insights into the fluid velocities in each cells, during the project this was primarily used for debugging.

4. Velocity field plotter, which provided a lot of the figures in this thesis (such as figures 3a and 7b)

These various examples are a showcase of the flexibility and ease of use of the scripting approach compared to implementing them directly into the C++ framework. These are applications which have on need for raw performance and as such a low-level language such as C++ present more obstructions than benefits. Different scripting languages as well as storage methods could be explored in the future, but this will be discussed further in the future work section.

6.2 Interface

We also created a small python executable which sports a GUI, and which allows the user to specify some variables before executing the code base. Through this platform people with no familiarity with C++ or CUDA are able to utilize the simulations nevertheless. Of course, it must be noted that this implementation requires a small amount of overhead which slightly affects performance. This is however only noticeable in very large simulation setups.

7 Future work

7.1 Optimisation

Even though the performance of the program has bettered substantially over the course of this project. As was shown by the multi-threading experiments, hypothetically a lot of improvement is still possible. The solving stage still makes use of a lot of nested loops and separate calculations on a data set that do not depend on each others result. All these things make the function a very suited one for high levels of parallelization which, for example, a graphics card allows. With these improvements made to the algorithmic performance of the solver, these should allow us to significantly accelerate on multiple cores.

7.1.1 Vectorization

Still in line with single core optimizations, making use of vectorization capabilities present on modern CPU's will allow for further reductions in single core run times. This approach makes sure that every bit of the CPU performance capability is utilized. Modern CPU's provide specialized instructions to take advantage of what are called "SIMD registers". SIMD is short for "single instruction multiple data" and is a protocol that was conceptualized in 1966 for supercomputers at the time making use of parallel architectures. It was widely incorporated in consumer CPUs in 1996 before the advent of graphics cards to deal with things like rudimentary rendering at the time. Most CPU's have special registers which allow for a single instruction to be applied to 4 (SSE protocol) or 8 (AVX protocol) pieces of data at the same time. In the case the CPU is used for solving the no-slip boundary conditions this would allow for optimal use of a single thread before moving into multiple ones as mentioned in section 4. [Bik17]

At the end of the thesis by Stam et al SIMD is also proposed as a possible future consideration for runtime optimization. However, this approach only makes sense when it is determined that the algorithm is to run independently from graphics cards and utilize the central processing unit to its fullest. From the moment there is access to a graphics card it makes much more sense to simply focus on achieving maximum saturation of the many threads these systems offer.

7.1.2 Complete use of GPU for calculation and solving

As was briefly mentioned in section 4, two properties of the analytical gradient calculations make it very well suited to parallel computing, be it using vectorization registers or the use of multiple threads. Also shifting from a system of halving gradients to one where step size depends on the geometry of the function eliminates some flow control out of the

system which is something that normally hurts the saturation of GPU threads but is now a non-factor. The key part to the overarching project is the use of the GPU to enhance Navier-Stokes based fluid solvers. Graphics cards knock central processors right out of the park when it comes to raw core count and compute power. It is not hard to imagine they would take the performance benefits gained from multi-threading to another level entirely. The prospect of doing the gradient descent solving on the GPU does not seem like an illogical choice for the program in the long term. In addition, the solving stage is located squarely adjacent to the calculation stage in the pipeline and the two go back and forth until a result is found. This means that every time the algorithm switches between the two, data needs to be transferred from one device to another through hardware buses which in itself is also a significant performance hit. Transferring data, be it between the CPU and RAM unit or GPU is a bigger concern for performance these days compared to the raw capabilities of the central processor.

Hypothetically, the entirety of the two vital stages of the simulation could be handled by the graphics card. Leveraging the power and parallel nature of the GPU in more stages of the simulation would not only further the goal of this project. Like mentioned earlier it would also reduce the number of data transfers dramatically. Right now, for every iteration of the solver data is transferred back and forth between CPU and GPU twice for calculation of the $\frac{\partial \vec{v}}{\partial f}$ gradient and 12 times the Fourier transforms. Once for every spatial dimension and twice for both transformations to and from Fourier space. Avoiding these transfers could see the program unlock a far greater potential from the graphics unit compared to employing it in separated stages curated by the GPU. This is only speculation, and time and care are needed to develop the set of GPU kernels to make this happen.

Memory streaming to and from the device also needs to be considered carefully as this will be needed when system sizes start exceeding the amount of video memory present in a given system. We suspect that memory will be the primary development bottleneck going into the future of this code base. That said we also see an opportunity for taking advantage of the layout of the data to increase memory capacity and cache hits on modern GPUs. Since these vector fields can easily be understood as images, we think they can be streamed to the device in the form of textures. Since modern consumer graphics cards are primarily aimed at enabling greater graphical fidelity for video games, they hold special registers and memory blocks to stream and store textures. Encoding vector information as textures is a technique that is used in this industry as well, an example of this is what are called normal maps which store normal vector information on a surface in order to manipulate light behaviour and create the illusion of an embossed surface. We believe that experimenting with processing for example, the velocity and pressure fields as texture data could lead to noticeable speedups.

7.1.3 Symmetry

Another avenue for performance improvements, especially in rather simple systems such as the parallel plates configuration is to abuse the inherent symmetry of its layout. Because we know that all channels must look identical, we can refrain from solving all but one vertical channel and reusing them to fill the rest of the system dimensions. An intelligent system able to detect symmetries in any configuration would theoretically be able to skip a lot of the computing load and as such speedup the solver significantly.

7.2 Convergence

In chapter 5 we extensively explained the problem of insufficient convergence on the largest and most complex systems. As well as the initial rudimentary ways we dealt with the it. However, more time can be committed to this problem as it is very likely that a better method can be devised which yields consistent closer convergence while sacrificing less of the simulations run-time. When we forced better convergence through brute forcing the learning rate past the points of equal cost. Getting those extra fractions of percentages of convergence took significantly more time than the rest of the simulation.

7.3 Output

During this project the parsing of output data was done from a plain text file in function of ease of implementation for the sake of experimentation rather than long term considerations. In the future however XML might be a better format option for storing the data. Both because most languages come with a built in or extendable XML parser as well as XML's ability to enumerate element types much more easily.

Similar to what was speculated on earlier with regards to processing the field information as textures, in the context of Python they can also be seen as images. The nature of the vector fields lends itself well to outputting the result as an image file. An RGBA image format can store information about the 3 spatial dimensions of either velocity or pressure for their respective fields and has another member in the alpha channel to store some meta data such as solid cells. We have already made use of techniques from the field of image processing such as bi-linear interpolation. On top of that, Python is a widely used programming language in this field.

Lastly, with regards to Python itself. The scripting language itself is also not limited to Python exclusively. Other languages aimed at scientific applications and data analysis such as Julia [BKSE12] have sprung up in the last few years. Julia prides itself on its performance and the speed at which it can parse data compared to Python. If this is true

and performance for script analysis were ever to become an issue, other options could be explored.

References

[Bik17] Jacco Bikker. practical simd programming. 2017.

[BKSE12] Jeff Bezanson, Stefan Karpinski, Viral B Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145*, 2012.

[Bre65] Jack Bresenham. Algorithm for computer control of a digital plotter. *IBM Syst. J.*, 4:25–30, 1965.

[LWZ+11] Yong Kui Liu, Peng Jie Wang, Dan Dan Zhao, Denis Spelic, Domen Mongus, and Borut Zalik. Pixel-Level Algorithms for Drawing Curves. In Ian Grimstead and Hamish Carr, editors, *Theory and Practice of Computer Graphics*. The Eurographics Association, 2011.

[MS15] Alexander Morozov and Saverio E. Spagnolie. *Introduction to Complex Fluids*, pages 3–52. Springer New York, New York, NY, 2015.

[Pol87] Boris T Polyak. *Introduction to optimization*. New York: Optimization Software, Publications Division, New York, 1987.

[Rem10] Georg Rempfer. Lattice-boltzmann simulations in complex geometries, October 2010.

[Rey83] Osborne Reynolds. An experimental investigation of the circumstances which determine whether the motion of water shall be direct or sinuous, and of the law of resistance in parallel channels. *Philosophical Transactions of the Royal Society of London*, 174:935–982, 1883.

[Sta21] Bart Stam. A gpu-based versatile and efficient hydrodynamics code for scientific applications, July 2021.

[WJ17] Freddie Witherden and Antony Jameson. Future directions in computational fluid dynamics. 06 2017.

[XXP+18] Qingang Xiong, Fei Xu, Yaoyu Pan, Yang Yang, Zhiming Gao, Shuli Shu, Kun Hong, Francois Bertrand, and Jamal Chaouki. Major trends and roadblocks in cfd-aided process intensification of biomass pyrolysis. *Chemical Engineering and Processing - Process Intensification*, 127:206–212, 2018.

[Zin16] Alois Zingl. A rasterizing algorithm for drawing curves. page 106, 2016.

[ZJKP20] Farzeen Zehra, Maha Javed, Darakhshan Khan, and Maria Pasha. Comparative analysis of c++ and python in terms of memory and time. 12 2020.