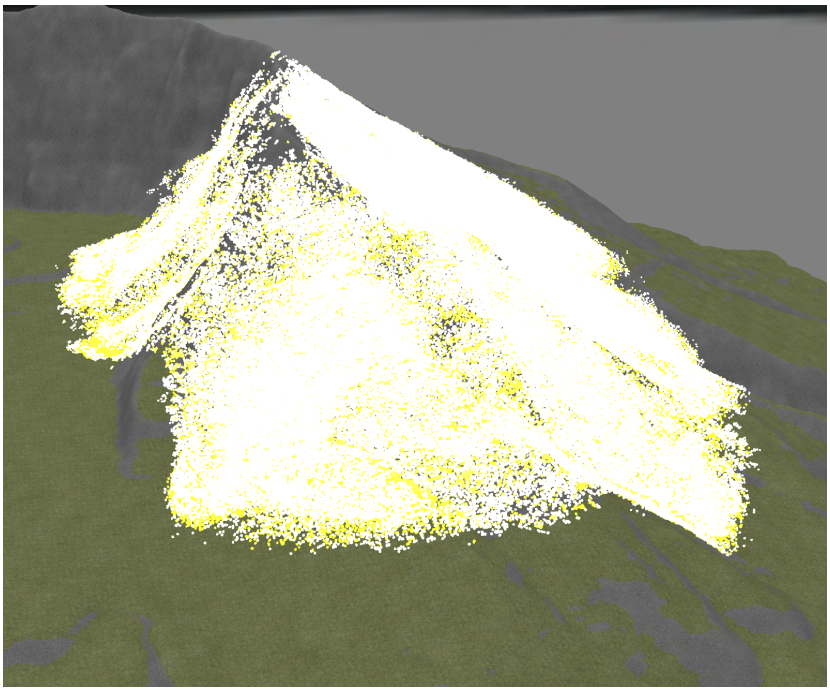
Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Ivar Andreas Helgestad Sandvik

Adding GPU-accelerated Real-time SPH-based Avalanche Simulations to the NTNU HPC-Lab Snow Simulator

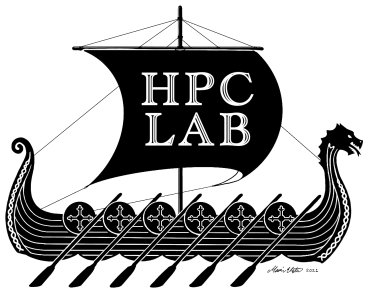
Master's thesis in Computer Science
Supervisor: Prof. Anne C. Elster
July 2021



NTNU
Norwegian University of
Science and Technology

Ivar Andreas Helgestad Sandvik

Adding GPU-accelerated Real-time SPH-based Avalanche Simulations to the NTNU HPC-Lab Snow Simulator



Master's thesis in Computer Science
Supervisor: Prof. Anne C. Elster
July 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Adding GPU-accelerated Real-time SPH-based Avalanche Simulations to the NTNU HPC-Lab Snow Simulator

Ivar Andreas Helgestad Sandvik

July 2021

Problem description

In this project, we will look at not only in-compressible flows, like we did on the fall project, but also do a literature search for compressible flows. This can be used to model snow that compacts. Assuming the task is reasonable, we will select one or more features/techniques such as adding “particle based” 3D models interacting with the fluid and/or adding adhesion to surfaces and models.

Should it prove to challenging, the Non-Newtonian fluids. Our implementation may also look into how CUDA can access the graphics memory directly in OpenGL, as well as exploring new CUDA features for our newer CUDA cards such as our new GTX 3090 cards and/or looking into multi-GPU acceleration.

Abstract

With access to more powerful computers, it is becoming more and more feasible to be able to simulate snow and snow-based avalanches more accurately and quickly. Snow is, however, a complex material to simulate because of the complex structure and changing flow characteristics. Depending on, among other factors the temperature, there are different categories of snow with different density levels and bindings that are hard to model.

In this thesis we use the numerical method called **Smoothed Particle Hydrodynamics** (SPH) to solve the Naiver-Stokes equations. We simulate Newtonian and non-Newtonian fluids using a similar SPH method as in Ø. E. Krog [1] work on snow avalanches. We apply the Newtonian model which can simulate fluids such as water, and we use the non-Newtonian model to simulate snow with the Bingham rheological model.

In particular, the parallel capabilities of modern Nvidia **Graphics Processing Units** (GPU's) are exploited to speed up our simulation. The SPH algorithm is has a high computational requirement, but is parallel in nature. This makes the GPU an ideal processor for solving this problem. Aided by GPU- and algorithm optimizations we achieve real-time performance at particle counts up to four and two million on our most powerful testing system using the Newtonian model and non-Newtonian model, respectively.

Our implementation is also integrated into the NTNU HPC-Lab snow simulator. This is a program suite that is developed and maintained by HPC-Lab students. The SPH algorithm is available as a stand alone component in the simulator. We have also added some unique features to our implementation such as simulation scaling and user defined terrain resolution. We also added a user interface to the simulation using ImgGUI framework that Roger Holten implemented in the HPC-Lab snow simulator. Using the GUI one can change physics parameters and change various options for the simulation during run-time. We also create OpenGL buffers for particle positions and intensity values and map and un-map OpenGL buffers to a device pointers for use in device code, both which should reduce memory copies.

Many of our findings are visual in nature. We therefore also include as part of our results, a series of images captured from the NTNU HPC-Lab snow simulator. This includes pictures of snow avalanches and water simulations. Various

functionalities of the user interface, and scenarios that can be initiated with predefined settings, are also added. Our benchmark results in the form of graphs and tables are also included. Several interesting screen shots from the HPC-Lab snow simulator during an avalanche that was flowing in a Newtonian manner when the yield shear stress was exceeded, are also shown. Finally, we also add some suggestions for future work.

Sammendrag

Med tilgang til raskere datamaskiner blir det mer og mer realistisk å simulere snø og snøras mer nøyaktig og raskere. Men snø er et komplisert matriale å simulere på grunn av den komplekse strukturen til snø og dens dynamiske flytegenskaper. Egenskapene til snø bestemmes av flere faktorer som for eksempel temperatur. Det er flere kategorier snø kan defineres i avhengig av for eksempel massetetthet og bindingen mellom snøpartikler. Dette gjør det utfordrende å modellere snø.

I denne masteroppgaven burker vi en nummerisk metode som heter **Smoothed Particle Hydrodynamics** (SPH) for å løse Naiver-Stokes likningene. Vi simmulerer en Newtonsk og en ikke-Newtonsk veske der vi bruker en liknende metode som Ø. E. Krog [1] brukte i sin masteroppgave som omhandlet simulering av snøskred. Vi bruker den Newtonske modellen til å simulere vesker som for eksempel vann og vi bruker den ikke-Newtonske modellen til å simulate snø med Bingham reologi.

Vi bruker de parallelle egenskapene til et moderne Nvidia skjermkort (GPU) for å gjøre simuleringen vår raskere. SPH algoritmen krever mange utregninger men er parallell i sin natur. Dette gjør at et skjermkort er godt egnet prosessor for å løse dette problemet. Ved hjelp av GPU og algoritme optimaliseringer så klarer vi å kjøre vår simulering i sanntid med opptil fire millioner partikler med den Newtonske modellen og to millioner partikler med ikke-Newtonske modellen på vårt beste skjermkort.

Implementasjonen vår er også integrert med NTNU HPC-Lab snøimulatoren. Dette er en programsuite som ble utviklet og er vedlikeholdt av HPC-Lab studentere. SPH-algoritmen er tilgjengelig som en frittstående komponent i simulatoren. Vi har også lagt til noen unike funksjoner til vår implementasjon, slik som skalering av simulasjonene (via parametre) og bruker-definert terrengoppløsning.

Vi la også til et brukergrensesnitt til simuleringen ved bruk av IngGUI som Roger Holten implementerte i HPC-Lab snøsimulatorenm ved å bruke denne GUIen kan en endre de fysiske parameterene og endre diverse valg av andre parametre under kjøring

Vi lagde også en OpenGL buffer for partikkel-posisjoner og intensitetverdier og mappet og av-mappet OpenGL buffere til en enhetspeker ("device pointer") for bruk i GPU koden (device code), som begge burde redusere minnekopieringer.

Mange av våre funn er visuelle av natur. Vi har derfor også tatt med mange bilder av snøskred og vannsimuleringer fra HPC-Lab snøsimulatoren som deler

av resultatene våre. Diverse funksjoner fra det grafiske grensesnittet vi la til, og flere av tilfellene som kan initieres fra forhåndsdefinerte innstillinger, er også tatt med. Vi viser også flere skjermbilder fra HPC-Lab snøsimulatorer under snøskred og flyt av vann. Til slutt tar vi med noen forslag til fremtidige arbeider.

Acknowledgements

I want to thank my supervisor Dr. Anne C. Elster for her valuable input and guidance on this work. I also want to thank the HPC-Lab and the Department of Computer Science at NTNU for access to state of the art hardware and working environment. Without these resources and support I would not be able to complete this project. I also want to express my gratitude to Anne for arranging a few social and professional gatherings for HPC-Lab members in a safe environment during this COVID-19 period. I also want to thank Roger R. Holten for his contributions to the HPC-Lab snow simulator and collaboration.

Lastly, I want to thank my family and friends for their support with a special mentions of HPC-Lab colleague Andreas Hammer and my father Rune M. Sandvik.

Contents

Figures

Tables

Code Listings

Chapter 1

Introduction

Snow is an material formed by ice crystals in the clouds in the atmosphere. Snow has some interesting physical properties. It can act as both a fluid and a solid depending on the configuration, binding of the crystals and these can change with stress. Snow can also support a wide range of density values, from light snow to solid ice. In order to simulate the behaviour of snow in different conditions, one needs to take into account these properties. There have been several projects related to snow simulation in recent years at the NTNU Heterogeneous and Parallel Computing Laboratory HPC-Lab). It started in 2006 when I. Saltvik [2] laid the ground work for the HPC-Lab snow simulator. Several projects have been implemented in the HPC-Lab snow simulator since then, including Robin Eidissen´s master thesis work with Prof. Elster simulating 2 million particles on a GPU back in 2008 [3]. A video of Eidissens simulation can also be found on Youtube video produced by NVIDIA [4]. A important note about the snow simulator is that the main focus has always been to speed up simulation using parallel computing techniques. Since GPU-processing power has outpaced the performance of CPU's, the focus has since Eidissen´s work shifted towards GPU programming. In this work, we will extend the capabilities of the HPC-Lab snow simulator to simulate snow avalanches and fluid flow using a method called **smoothed particle hydrodynamics (SPH)** implemented on the GPU.

1.1 Motivation

In recent years, fluid simulation has gained popularity in mathematics, engineering and computer graphics applications. Over time parts of these fields have merged into a common field called computational fluid dynamics (CFD). There are several approaches for doing computational fluid dynamics, but a few common denominators that usually apply to CFD: A mathematical sound method for the simulation, a practical solution to the mathematical method (engineering aspect), and a graphical visualization of the solution (computer graphics).

In this thesis work we will discuss a way of using CFD to simulate a solid such as snow and fluids in real-time. We will be using a CFD method called smoothed particle hydrodynamics. To aid our goal of a real-time simulation, we will be using a modern graphics processing unit (GPU) to accelerate our simulation.

1.2 Contribution

The original goal of the project was to simulate snow as a fully compressible material, however this proved to be too challenging given the scope and time frame of the project. We instead opted to build upon Ø. E. Krog [1, 5] model for SPH snow simulation. His simulations was not integrated with the HPC-Lab snow simulator, and ran our implementation on newer GPUs such as the RTX 2080Ti and RTX3090, which have emerged since his work back in 2010.
The main contributions of this thesis are thus:

- Extending the functionality of the HPC-Lab snow simulator with two high-performance GPU-based SPH solvers.
    - A weakly compressible Newtonian solver based on Muller [6], on which Ø. Krog [5] also did an implementation on GPUs back in 2010.
    - A weakly compressible non-Newtonian solver based on M. Hossein´s [7] work. Ø. Krog also did an implementation of this solver [5].

Before this contribution the snow simulator had a limited capability to simulate snow and fluid behaviour. The intention of this project was also to enable users of the HPC-Lab snow simulator without deep knowledge about fluid mechanics to be able to simulate various phenomena such as a avalanches and fluid flow.

In addition, another HPC-lab master student, Roger Holten, was working on updating the Graphical User Interface (GUI) for the HPC-Lab simulator by setting up a new library based on IMGUI [8] along with this work. We therefore collaborated to also integrate this work into the snow simulator summarized as the following contribution:

- Adding a user interface to the HPC-Lab snow simulator where the physics parameters, including boundary settings, can easily be tuned to do various simulations.

We also made the following main contribution:

- Integrating into the HPC-Lab snow-simulator a state-of-the-art CUDA grid-based nearest neighbour search algorithm developed by R. Hoetzlein [9] to enable our SPH solvers to scale with the newer GPU hardware developed over the last several years.
- Performing profiling and benchmarking with some of the new tools devel-

oped such as Nvidia Nsight Compute and our own benchmarking utility.
- Optimizing out implementation including simplifying the code (e.g. moving constants out of sum formulas and reducing memory load operations), using constant memory and changing L1 cache size configuration for the newest consumer GPU's currently available.

1.3 Outline

The rest of this thesis is outlined as follows: **Chapter 2:** The first Section 2.1 covers some of the differences between a graphics processing unit (GPU) and a central processing unit (CPU). We also explore some of the building blocks in the GPU core design and the Nvidia CUDA architecture. In the next Section 2.2 we explain and examine earlier projects related to the HPC-Lab snow simulator and the NTNU HPC-Lab. Proceeding this we explain some of the properties of snow in Section 2.3. The last Section 2.4 explains some of the forces present in a fluid system and rheological models for non-Newtonain fluids.

Chapter 3: In Chapter 3 we examine how we can use SPH to solve CFD problems. This chapter includes the author´s fall report on smoothed particle hydrodynamics for the readers convenience. In the first Section 3.1 we cover the foundation of computational fluid dynamics and the Naiver-Stokes equation for in-compressible fluid. We also examine the Navier-Stokes equation for a non-Newtonian fluid. In the next Section 3.2 we explore how we can use smoothed particle hydrodynamics (SPH) to solve the Naiver-Stokes equation numerically. In the final Section **??** we have included a some research done in the fall report where we compare two different fast neighbour search algorithms on the GPU and CPU.

Chapter 4: This chapter is about the implementation of the SPH algorithm in the HPC-Lab snow simulator. In the first Section 4.1 we briefly explain how CUDA kernels are invoked. In Section 4.2 we describe the Newtonian and non-Newtonian SPH algorithms we use. We also give a breif overview of the neighbour search algorithm we use and data structures. In the third Section 4.3 we describe the simulation scaling system we use to run simulations in of different physical dimensions on the same rendering area. In Section 4.4 we explain how our simulation integrates with the terrain in the HPC-Lab snow simulator. In the final Section 4.5 we describe how we integrated our solvers into the snow simulator, the simulation parameters and user interface.

Chapter 5: Chapter 5 is about testing and optimization of the SPH algorithms on modern hardware. The the first Section 5.1 describes our test systems and specifications. The following Section 5.2 we feature the optimization tools that we use for determining bottlenecks. In the next Section 5.3 we apply various optimizations to improve performance. In Section 4.4 we benchmark the application

with different particle counts and record the performance numbers. The last Section 4.5 show a collection of images of the finished implementation.

Chapter 6: In chapter 6 we present the conclusions from our work and present some ideas for future work.

Chapter 2

Background

In this chapter we will cover some of the background material on GPU computing, the HPC-Lab Snow simulator, snow, forces in a fluid system and fluid rheology. Section 2.1 describes the architecture of a modern general purpose graphics processing unit (GP-GPU). Section 2.2 introduces the HPC-Lab snow simulator and the HPC-Lab. In Section 2.3 we examine the physical properties of snow. In Section 2.4 we explain the forces we need to consider in a fluid system and fluid rheology.

2.1 SIMD architecture and GPU programming

In the field of high performance computing and science GPU programming has gained popularity in recent years. While traditional central processing units (CPU's) have seen a modest performance gain, GPU performance has increased more or less exponentially Brodtkorb et al. [10]. An other contributor to the increasing GPU popularity is the ease of use with the introduction of GPU programming frameworks such as Compute Unified Device Architecture (CUDA) and OpenCL in 2007 and 2009 respectively. Prior to this GPU's where almost exclusively used for graphics processing. To use GPU's for compute at this time one had to exploit the graphics pipeline to do compute, requiring expert knowledge[10].

2.1.1 CPU and GPU comparison

In order to understand why GPU's and CPU's have different applications and performance characteristics we have to investigate the underlying architecture of these systems. In 1966 M. J. Flynn [11] coined four terms covering how different architectures execute instructions on data, often referred to as Flynn's taxonomy [10]. These therms only cover the high level parallelism of a computer architecture. The first distinction is the number of instructions the processor can execute at a time and the other distinction is the number of data elements the instructions(s) are executed upon. Flynn's taxonomy covers these architectures:

- **Single instrucion single data stream (SISD):** This characterizes the typical serial processor that executes a single instruction on a single data element. This is the processor architecture used in normal desktop computers and laptops. Parallelism on this architecture can be achieved by having multiple "cores" (processors working in parallel) that operate independently on the same die (MIMD in a sense).
- **Single instruction multiple data streams (SIMD):** This is the parallel architecture used in modern GPU's. These processors can execute a single instruction on multiple data elements. Parallelism is possible during execution since we can execute a instruction on multiple data elements. We can also get MIMD parallelism by having multiple independent "cores" working independently with different instructions on the same chip.
- **Multiple instruction single data stream (MISD):** This is more of a theoretical architecture where multiple instructions are executed on a single data element and is often considered as an impractical architecture[10].
- **Multiple instructions multiple data streams (MIMD):** This is a parallel architecture that can execute multiple instructions on multiple data streams. Since Flynn's taxonomy is very crude, one could argue that a supercomputer with multiple nodes or a multi-core processor is a MIMD system.

When we look at modern CPU's and GPU's, the distinction is not as clear as above. For example some modern CPU's can execute certain instructions in a SIMD fashion, for example Advanced Vector Extentions (AVX). However there are some clear differences between modern CPU's and GPU's making them excel at different applications. CPU's are designed to execute a multitude of different instructions at a hardware level in a fast manner, giving a performance advantage in complex logic where the data elements are few and instruction variety is big Lee et al. [12]. The CPU also utilizes techniques such as branch prediction, pipe-lining, out-of-order execution and has higher clock frequencies. This results in a complex core design making the core physically bigger and and increasing power consumption[12]. This makes the CPU ideal for more sparse workloads where single threaded performance and latency is of importance.

GPU's on the other hand features a much simpler core design and excels in applications where data parallelism is present Kirk and Hwu [13]. As mentioned earlier the GPU was originally intended to process graphics workloads. In these workloads latency was not a primary focus, instead parallel floating point calculations were important. An example of this is shader performance where the colors and positions of objects on the screen was calculated. Since the core structure is simpler without the advanced features of the traditional CPU such as out-of-order execution and large cache (per core), the core count can be higher on the same area compared to a CPU. While a GPU might have a huge performance advantage on paper, its important to note that a GPU is dependent on a workload where massive parallelism is possible. In many cases a CPU will be faster completing a task if the workload is not fit for a GPU [12].

2.1.2 Compute Unified Device Architecture (CUDA)

CUDA is a parallel computing architecture and programming interface (API) that was released by Nvidia in 2007 together with the graphics chip G80 (Geforce 8000 series)[13]. Prior to this GPU programming was restricted to grapics API's such as OpenGL and Direct3D[13]. GPU programs had to be written as shader programs using these API's. The CUDA programming interface made it possible for developers to write custom GPU programs called CUDA kernels. These programs are written in a C/C++ syntax with additional GPU syntax. The kernels in the program are then compiled to PTX (Nvidia GPU instruction set) using the Nvidia CUDA Compiler (NVCC)[13]. Since 2007 all subsequent Nvidia desktop GPU's have supported the CUDA framework. This has lowered the entry point, making GPU programming widely available on consumer grade hardware.

CUDA programming model

Making CUDA programs differs from traditional programs. In a normal C program the programmer would write a **main()** method that takes an argument(s) from the user and returns a result and then be done. In CUDA however the graphics card is running the program (kernel) and therefore the data needs to be transferred to the GPU's random access memory (RAM) before the kernel is invoked. The GPU then runs the program with the data in its memory and stores the result. When the GPU is finished the CPU portion of the program then needs to copy the result from the GPU RAM into its local RAM and then return the result to the user.

To avoid confusion between CPU and GPU execution and RAM two scopes are used to distinguish them.

- **HOST:** This term is used to specify CPU program/serial code and main system memory.
- **DEVICE:** This term is used to specify the GPU scope including parallel kernel code and GPU memory.

2.1.3 Nvidia GPU design

In order to use a GPU efficiently its important with a understanding of the underlying architecture. There are often changes to the GPU architecture between generations of Nvidia products. In this work we will be considering the newest Nvidia architecture called Ampere (Nvidia [14]). At the top level of the GPU we have the GigaThread Engine, Memory controllers, L2 cache and the PCI-Express interface as seen in Figure 2.1. The GigaThread Engine's role is to distribute thread blocks to the streaming multiprocessors (SM) in the GPU. The memory controllers gives the GPU access to the device memory (also called global memory). The L2 cache is shared for all SM units and is used to cache reads and writes to the device memory. The PCI-express interface is used to communicate with the host system (CPU) and other devices connected to the bus (for example another GPU).

Figure 2.1: A top level image of the Nvidia ampere GPU. With permission from Nvidia.

On Figure 5.2 we see that there are seven groups called Graphics Processing Clusters (GPC). These are separated on chip and contains some shared graphics related units and a number of streaming multiprocessors each. In our application we are interested in the SM units (where we do our processing). The SM's in the Amphere architecture is shown in Figure 2.2, each of the SM's contain four processing blocks and a shared L1 cache that is shared between them. Each of the processing blocks have their own register file for the warps that are assigned to the SM, load/store units, special function units, int pipelines and 32 bit precision pipelines (CUDA cores). Further details on CUDA can be found in Appendix A, Chapter 5 in Krog's theis[1] as well as the CUDA C++ Programming Guide [15].

2.2 The NTNU HPC-lab snow simulator

The NTNU HPC-Lab snow simulator as shown in Figure 2.3 is a collection of contributions made by former graduate students at the NTNU HPC-Lab. The snow simulator is a high performance software package aimed at utilizing modern parallel hardware to make realistic snow simulations and visual effects. The project started with Ingar Saltvik's masters thesis on CPU parallel snow simulation in 2006 [2]. Since then the snow simulator software package has capable of simulating various phenomena such as snow accumulation Eidissen [3] and wind simulation Schmid [16]. In this context of the HPC-Lab snow simulator, it should

be mentioned that the NTNU HPC-lab is a research group led and founded by Dr. Anne C. Elster in 2008 [17] consisting of her Masters and PhD students. The focus of this group is primarily on high performance computing.

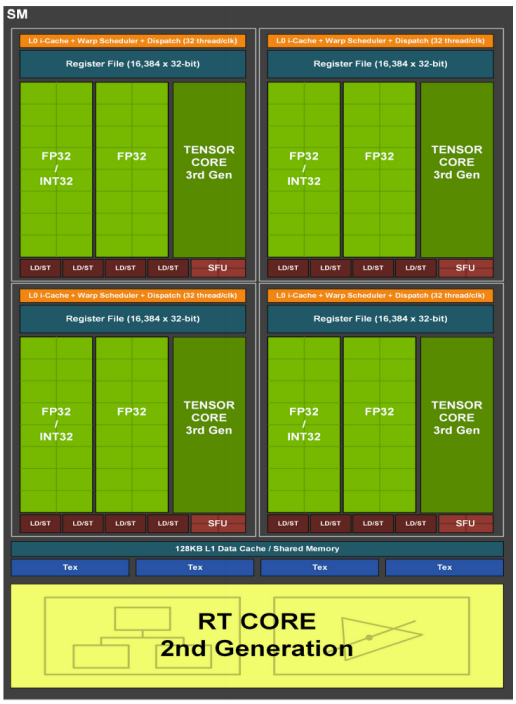


Figure 2.2: A image of a single SM unit of the Ampere architecture. With permission from Nvidia.

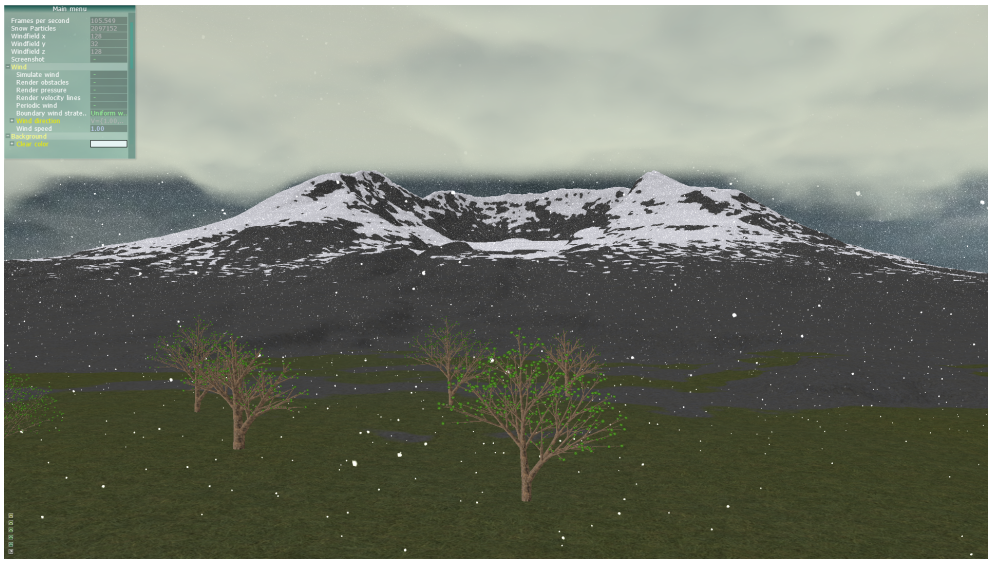


Figure 2.3: A screenshot of the current NTNU HPC-Lab snow simulator.

2.2.1 Saltvik, Ingar: Parallel Methods for Real-Time Visualization of Snow (2006)

This was one of the earlier projects making the foundation for subsequent projects relating to high performance snow simulation at the HPC-lab [2]. Saltvik implemented a parallel algorithm to simulating snow fall and accumulation on the ground. He also implemented a grid based wind solver to simulate the effects of wind on the snow while falling. The algorithm used the CPU for the physics simulation and OpenGL for rendering.

2.2.2 Eidissen, Robin: Utilizing GPUs for Real-Time Visualization of Snow (2009)

Eidissen's masters thesis [3] built upon Saltvik's implementation. In this work the wind and snow simulation part of the snow simulator was implemented on the GPU. It should be mentioned that in both Saltvik and Eidissen's work a incompressible CFD wind solver was used to make a grid based wind field (simulation domain divided into a grid of wind velocities affecting the snow particles with a customizable "resolution/grid block sizes"). Eidissen also implemented a better snow accumulation algorithm to distribute snow on the ground more evenly depending on wind conditions. Another contribution of this project was the implementation of a height map loader that enables modeling of realistic terrain such as mountains.

2.2.3 Vestre, Frederik M. J.: Enhancing and Porting the HPC-Lab Snow Simulator to OpenCL on Mobile Platforms (2012)

Vestre's masters thesis Vestre [18] main focus was to translate the CUDA code in snow simulator to OpenCL and make the simulator run on mobile devices. In the later versions of the snow simulator CUDA is still used as the main GPU framework. The project contributed with some newer visualizations that work in the newer versions such as as pressure field and wind field visualization.

2.2.4 Nordahl, Andreas (2013): Enhancing the HPC-Lab Snow Simulator with More Realistic Terrains and Other Interactive Features

Nordal's masters thesis [19] worked on the visual appearance of the snow simulator implementing visual features such as improved terrain and snow textures, distance fog, shadow mapping and more. The result of these visual improvements can be seen above in the screenshot of the current snow simulator.

2.2.5 Schmid Thomas M.: Real-Time Snow Simulation - Integrating Weather Data and Cloud Rendering (2016)

Schmid's masters thesis [16] is the latest contribution to the snow simulator. Improvements were made to to the snowflake and wind field simulations to make them more realistic. Support for virtual weather stations outside the simulation domain was added and interpolation between those to determine wind and precipitation inside the simulation domain.

2.3 Snow

Snow is raindrops that have been formed as small ice crystals in the atmosphere. When the temperature in the atmosphere falls below the freezing temperature, the snow crystals will be formed. Snowflakes can be created when the crystals are partially melted and binds together, usually at a lower altitude. Snow is very complex to model because it can undergo many changes over time changing its physical properties Arenson et al. [20]. External factors that can determine the structure and density are the temperature, wind and pressure.

2.3.1 Physical properties

After snow has accumulated on the ground it can undergo many changes, depending on the temperature, wind and pressure conditions in the area. Wind forces breaks up the grain structure of the snow making the snowflakes more round and smooth[20]. This allows the snow to sinter forming strong bonds. Temperature changes influences the strengthening of these bonds making the snow harder. This happens on the surface layer of the snow since its exposed to wind and rapid temperature changes[20].

In order to model snow the density aspect is important. Experiments show that the density of snow changes under its own weight over time[20]. Density is given as kg/m^3. When the temperature is around melting point the density changes rapidly. The density gradient with respect to time is lower for lower temperatures, but note that the density also increases in lower temperatures. The density of snow/ice can range from 10 to 917 kg/m^3. Some typical densities of snow are Paterson [21]:

- New snow $50-70kg/m^3$. This is snow that just have accumulated on the ground.
- Damp new snow 100—$200kg/m^3$. Relatively new snow, that has weak bonds and is "fluffy".
- Settled snow 200—$300kg/m^3$. This is typical old snow, that have been metamorphosed and compacted over time.
- Wind packed snow $350-400kg/m^3$. Snow that has been compressed by wind forces, compact.

- Firn $400-830 kg/m^3$. This is multi-seasonal snow that has not melted during summer, very compact.
- Ice $830-917 kg/m^3$.

The strength of snow is closely related to the density, but also to the underlying micro-structure of the snow particles [20]. The mechanical properties of snow is similar to ice, its response to stress and rate of stress is similar to a visco-elastic material [20]. This means that snow has a viscous and elastic property. A viscous material will deform when a strain is applied. Elastic materials will return to their original state after a strain has been applied. Visco-elastic materials exposed to strain will in some cases return to their original form depending on the stress and the stress duration, but can also deform. Brittle failures happens often in snow, this happens when the strain rate (deformation/time), exceeds a certain level. Under this level the snow/ice will behave close to linear-elastically, meaning that it will return close to its original configuration [20].

2.4 Forces in a fluid system

In fluid simulations we want to investigate how a fluid volume evolves over time. The movement of a fluid occurs when the substance is affected by forces affecting acceleration of the fluid to change. To limit the complexity and computational requirement of a simulation we simplify the forces we want to model down to a few major forces. We categorize these into internal and external forces. The internal forces we model are pressure and viscosity. External forces that we model are friction, boundary collision and gravity.

2.4.1 Pressure

Pressure is defined as the perpendicular force per unit area, or the stress at a point within a confined fluid Britannica [22]. In a fluid system we usually determine the pressure as the current level of compression in the fluid with an equation of state relating to the rest density of the fluid (which can vary depending on the fluid and temperature). It should be noted that pressure is a scalar quantity.

$$P = \frac{F}{A} \tag{2.1}$$

The usual cause of pressure in a fluid system is gravity causing fluid to gather at low points resulting in the fluid resisting compression (exerting pressure force) or constrained space in the simulation. Temperature changes can also result in the fluid phase changing taking up more or less space (rest density changing).

2.4.2 Viscosity

Viscosity is a measure of how a fluid resists flow Krishnan et al. [23]. A fluid with higher viscosity will have more resistance to move than a fluid with lower viscosity. There are two categories of viscous fluids: Newtonian fluids and non-Newtonian fluids. Newtonian fluids have a constant viscosity while non-Newtonian fluids can have dynamic viscosity values depending on various factors such as rate of deformation and applied stress. The viscosity in a fluid simulation can be described as the friction between fluid particles causing a reduction in velocity of the fluid.

To show the effect of viscosity a illustration like 2.4 is often used[23]. The figure shows a fluid between one stationary and one moving surface A in contact with the fluid. F is the force applied to move the surface A. The displacement (shear) S is how much the top plate moves when F is applied. We assume non-slip condition on the surfaces where the velocity of the fluid closest to the surface move at the same velocity as the surface. H is the distance between the plates (height of the fluid in the illustration). From this we can express the viscosity as follows:

$$\frac{\frac{F}{A}}{\frac{S}{H}} = \mu \tag{2.2}$$

We can define F/A as τ which is the shear stress (force per unit area). $\frac{S}{H}$ is the velocity derivative in the moving direction (assuming y-direction). We can also define strain as ϵ and strain rate as $\dot{\epsilon}$:

$$\frac{\tau}{\dot{\epsilon}} = \mu \tag{2.3}$$

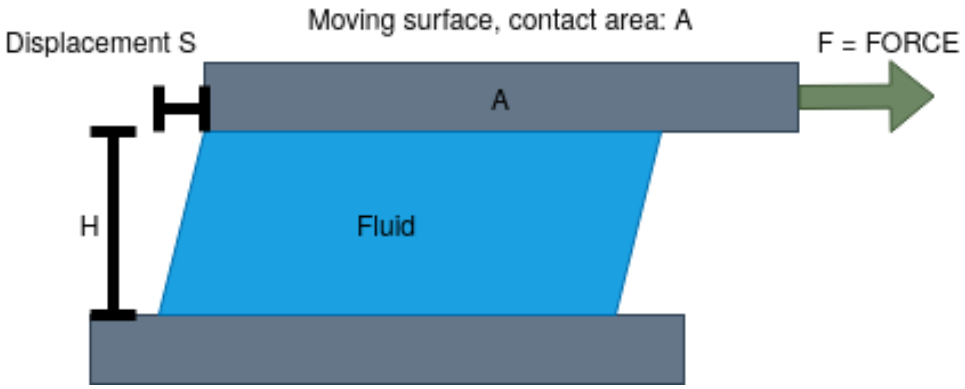


Figure 2.4: Illustration of viscous forces on fluid between two plates, inspired by[23].

2.4.3 Viscosity Newtonian fluids

A Newtonian fluid is a fluid that obey Newton's law of viscosity. It states that: The rate of deformation of a volume element of the fluid is proportional to the applied stress[23]. This means that the relationship between shear stress and deformation is constant. Newton's law of viscosity defines viscosity as Equation 2.4:

$$\mu = \frac{\tau}{\dot{\epsilon}} \tag{2.4}$$

Where τ is the shear stress, $\dot{\epsilon}$ is the strain rate. Newton's viscosity law implies that the viscosity of a fluid is the same for any shear rate. The fluids that follow Newton's viscosity law are referred to as Newtonian fluids [23]. Example of an Newtonian fluid is water.

2.4.4 Viscosity non-Newtonian fluids

The fluids that does not obey Newton's viscosity law are called non-Newtonian fluids. Factors such as pressure, temperature and duration of applied stress can change the flow properties of non-Newtonian fluids[23]. Figure 2.5 shows the rheological models presented below with the relationship between stress and shearing rate. We can use some models Some common non-Newtonian fluid models are[23]:

The power-law model

This model is used for fluids that are shear thinning or thickening where the viscosity is decreasing or increasing with shear rate. It can also be combined with the Herschel–Bulkley model, to have a yield stress before flowing. K is the consistency index, m is a measure of the consistency of the substance[23]. When $n < 1$ the effective viscosity μ will decrease as the shear rate increases and when $n > 1$ the viscosity will increase with shear rate. The power-law model is defined in Equation 2.5

$$\begin{aligned} \tau &= m \cdot \dot{\epsilon}^n \\ \mu &= m(\dot{\epsilon})^{n-1} \end{aligned} \tag{2.5}$$

Visco-Plastic: Bingham model

The Bingham model is a model used for fluids that have a yield stress τ_{yield}, the fluid will only flow if a stress τ_i is applied that exceeds the yield stress threshold[23]. When the yield stress is exceeded the fluid will behave in a Newtonian manner. In Equation 2.6 we can see that once the stress level is above the threshold τ_{yield} the fluid is flowing as a Newtonian fluid. When the applied stress is not greater than the yield stress the rate of deformation is zero (solid).

$$\begin{aligned} \tau_i = \tau_{yield} + \mu \cdot \dot{\epsilon} \qquad & \text{if } |\tau_i| > |\tau_{yield}| \\ \dot{\epsilon} = 0 \qquad & \text{if } |\tau_i| \leq |\tau_{yield}| \end{aligned} \tag{2.6}$$

Visco-plastic: Herschel–Bulkley model

The visco-plastic model is similar to the Bingham model with one exception. After the stress threshold is exceeded the fluid behaves as a shear thinning fluid, when $n < 1$[23]. m is known as the consistency index[23]. The Herschel–Bulkley model is defined as in Equation 2.7

$$\begin{aligned} \tau_i = \tau_{yield} + m \cdot \dot{\epsilon}^n \qquad & \text{if } |\tau_i| > |\tau_{yield}| \\ \dot{\epsilon} = 0 \qquad & \text{if } |\tau_i| \leq |\tau_{yield}| \end{aligned} \tag{2.7}$$

2.4.5 External forces

The external forces we will employ in the simulation is gravity and simulation boundaries. This will be discussed in the implementation chapter.

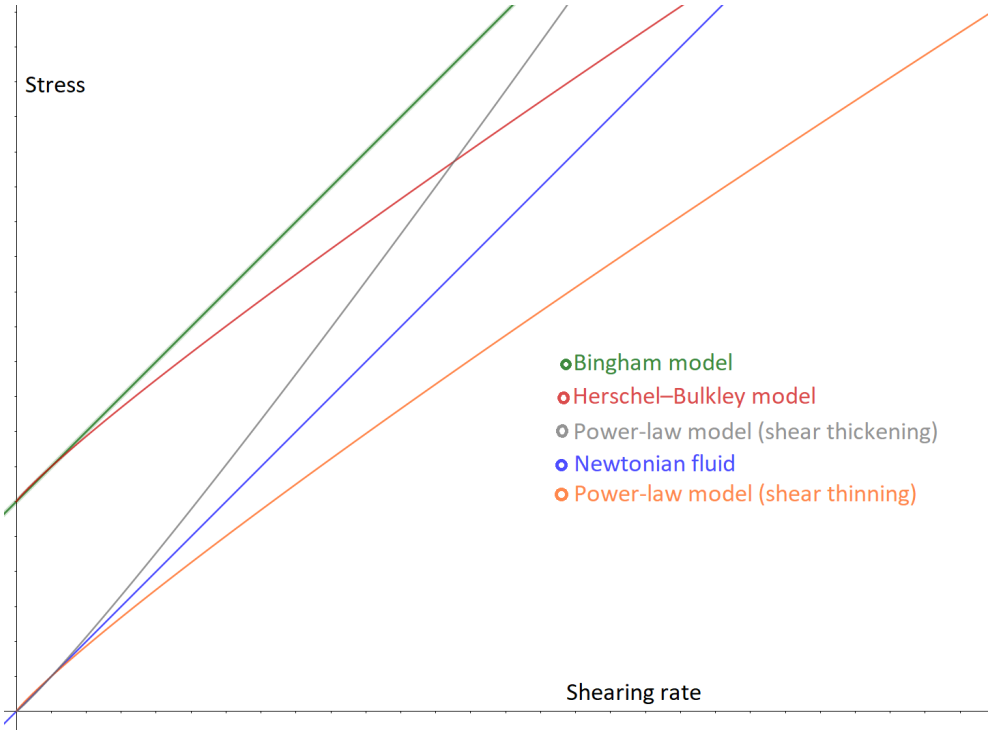


Figure 2.5: Rheological models viscosity illustrated at different stress and shearing rates. Inspired by[23].

Chapter 3

Computational Fluid Dynamics (CFD) and Smoothed Particle Hydrodynamics (SPH)

This chapter describes the basics of Computational Fluid Dynamics (CFD) in Section 3.1. It includes the background chapter based on the author´s fall report on Smoothed Particle Hydrodynamics (SPH) in Section 3.2. Since we will be using SPH in our implementation this material is included for the readers convenience. This chapter will give an brief introduction to computational fluid mechanics and how we can apply a smoothed particle hydrodynamics to find a numerical solution to the Navier-Stokes equations. It also includes a comparison on grid based neighbour search and verlet list neighbour search in Section **??**.

3.1 Computational fluid dynamics

Computational fluid dynamics (CFD) is a branch of fluid dynamics where the goal is to numerically simulate fluid flow using computers. The field integrates typical engineering fluid dynamics with mathematics and computer science. The motion of the fluid can be described using mathematical equations, most commonly in partial differential form.

3.1.1 The governing equations of CFD

Computational fluid dynamics is based on the on the following physical laws Tu [24].

- Mass conservation of the fluid
- Newton's second law. The rate of change of momentum equals the sum of forces acting on the fluid
- The first law of thermodynamics. The rate of change of energy equals the sum rate of heat addition to and the rate of work done on the fluid. *Note,*

due to the scope of this work we are not including this section.

3.1.2 The continuity equation

The mass conservation law stats that matter can neither be created of destroyed. In more general terms this means that given a control volume, mass cannot be created or destroyed inside the volume. If we imagine a fluid passing trough this volume with a constant density ρ. The in flow rate will be given in m^3/s. So the total mass change in the control volume will only be affected by the difference between in and out flow rate given during an given time span. Equation 3.1 is the continuity equation[24]. If we look at the derivative of ρ with respect to time we can see that for a in-compressible fluid the density change will always be zero. The density of the fluid therefore is constant Equation 3.2.

$$\frac{d\rho}{dt} + \nabla \cdot (\rho \boldsymbol{V}) = 0 \tag{3.1}$$

$$\frac{d\rho}{dt} = 0 \tag{3.2}$$

This implies that the divergence of the velocity field of the in-compressible fluid is zero Equation 3.3.

$$\nabla \cdot \rho \mathbf{V} = 0 \tag{3.3}$$

3.1.3 The momentum equation

The momentum equation is based on Newton's second law of motion which states that the sum of forces acting on a body is F equals mass times acceleration. In fluid mechanics we can write the equation in terms of the directional forces[24], example for x direction is shown in Equation 3.4:

$$\sum F_x = ma_x \tag{3.4}$$

The acceleration component can we written as Equation 3.5:

$$a_x = \frac{Du}{Dt} \tag{3.5}$$

With u being the velocity vector in the x direction, it is important to note that D means the substantial derivative. The substantial derivative is used when we consider a general variable property per unit of mass ϕ with respect to time [24]. The substantial derivative can be written as Equation 3.6:

$$\frac{D\phi}{Dt} = \frac{\partial \phi}{\partial t} + \boldsymbol{u}\frac{\partial \phi}{\partial x} + \boldsymbol{v}\frac{\partial \phi}{\partial y} + \boldsymbol{w}\frac{\partial \phi}{\partial z} \tag{3.6}$$

To define the mass in Equation 3.4 we can use the mass density formulation $m = \rho(\Delta x \Delta y \Delta z)$. We end up with the expression for momentum (in x direction) in Equation 3.7.

$$\rho \frac{Du}{Dt} \Delta x \Delta y \Delta z \tag{3.7}$$

Our fluid element experiences two different forces that together make the force F_x. The two forces are body forces and surface forces[24]. The body forces that influences the acceleration are gravity, centrifugal forces, Coriolis and electromagnetic forces. The body forces are added as external forces usually. The surface forces are inflicted by normal stress σ_{xx} and tangential stresses τ_{yx} and τ_{zx} that are acting on the surface of the fluid[24]. The normal stress σ_{xx} is a force that is applied on to the fluid surface an example of this is pressure p. The tangential stresses τ_{yx} and τ_{zx} are stress that acts alongside the fluid surface, example of this is shear stresses caused by the fluid's viscosity[24]. We end up with the following Equation 3.8 for the x-momentum:

$$\rho \frac{Du}{Dt} = \frac{\partial \sigma_{xx}}{\partial x} + \frac{\partial \tau_{yx}}{\partial y} + \frac{\partial \tau_{zx}}{\partial z} + \sum F_x^{bodyforces} \tag{3.8}$$

From this we can derive the Navier-Stokes Equation for a Newtonian in-compressible fluid in Equation 3.9. Referring to [24] for more information.

$$\rho \frac{\partial \boldsymbol{v}}{\partial t} = -\nabla p + \mu \nabla^2 \boldsymbol{u} + \rho g \tag{3.9}$$

We can simplify by multiplying the equation with $1/\rho$:

$$\frac{\partial \boldsymbol{v}}{\partial t} = -\frac{1}{\rho}\nabla p + \frac{\mu}{\rho}\nabla^2 \boldsymbol{u} + g \tag{3.10}$$

On the left side is the derivative of $\mathbf{v}$ (velocity vector) with respect to time t giving us the acceleration vector. The expressions on the right side are as follows: ∇p is the pressure gradient, ρ is the density, μ is the viscosity constant, $\nabla^2\mathbf{u}$ is the Laplacian of flow velocity and g is external acceleration.

3.1.4 Navier stokes equation non-Newtonian fluid

For a non-Newtonian fluid, the viscosity can vary depending on shear stress and shear rate. We need to define the Navier-Stokes equation with the shear stress tensor. Equation 3.11 show the Navier-Stokes equation with the shear stress tensor:

$$\frac{\partial \boldsymbol{v}}{\partial t} = -\frac{1}{\rho}\nabla p + \frac{1}{\rho}\nabla \cdot \boldsymbol{\tau} + g \tag{3.11}$$

Hosseini et al. [7] defines the shear stress tensor as Equation 3.12:

$$\tau = \mu(|\dot{\epsilon}|)\dot{\epsilon} \tag{3.12}$$

Where $\dot{\epsilon}$ is strain rate tensor and $\mu(|\dot{\epsilon}|)$ is the viscosity function. To find the strain rate tensor we first need to take the gradient of the fluid velocity to find the strain tensor. In three dimensions the fluid velocity has three components. That gives us Equation 3.13:

$$\epsilon = \nabla \mathbf{v} = \begin{bmatrix} \frac{\partial v_x}{\partial x} & \frac{\partial v_x}{\partial y} & \frac{\partial v_x}{\partial z} \\ \frac{\partial v_y}{\partial x} & \frac{\partial v_y}{\partial y} & \frac{\partial v_y}{\partial z} \\ \frac{\partial v_z}{\partial x} & \frac{\partial v_z}{\partial y} & \frac{\partial v_z}{\partial z} \end{bmatrix} \tag{3.13}$$

According to [7] strain rate tensor $\dot{\epsilon}$ can then be found by Equation 3.14.

$$\dot{\epsilon} = \frac{1}{2}(\nabla \mathbf{v} + \nabla \mathbf{v}^T) \tag{3.14}$$

We can find the magnitude of the strain rate tensor $|\dot{\epsilon}|$ by:

$$|\dot{\epsilon}| = \sqrt{trace(\dot{\epsilon})^2} \tag{3.15}$$

The trace operation is to sum the elements in the diagonal from the top left of the tensor to the bottom right of the tensor.

3.2 Smoothed Particle hydrodynamics

Smoothed particle hydrodynamics is a simulation method used to simulate solid mechanics and fluid flow. The method was developed by Gingold and Monaghan and presented in the paper "Smoothed particle hydrodynamics: theory and application to non-spherical stars"[25] in 1977. SPH is a meshfree Lagrangian method meaning that the bodies in the simulation are not connected, but instead are individual with their own positions and velocities. The Lagrangian approach makes it easy to obtain approximate solutions fluid dynamics problems. In the case of fluid dynamics the fluid is represented as groups of particles acting together as a fluid. The particle approach also makes it easy to represent different materials in the same simulation Monaghan [26]. An advantage of SPH is that the computation is only done for where the matter is in the simulation compared to grid based Eulerian methods.

SPH is based on interpolation theory, this means that values we find such as pressure forces, density and viscous forces at any location approximated by nearby particles in the simulation space. The kernel (or smoothing function) function is the basis of the SPH method determining weight of the contribution from surrounding particles. The interpolation function is described as Equation 3.16 in Monaghan [26] in a integral form.

$$A_h(\boldsymbol{r}) = \int_V A(\boldsymbol{r'})W(\boldsymbol{r} - r', h)dr' \tag{3.16}$$

For practical applications a numerical approximation of this integral is made Equation 3.17.

$$A_i(\boldsymbol{r}) = \sum_{j=1}^{N} \frac{m_j}{\rho_j} A_j W(\boldsymbol{r_i} - \boldsymbol{r_j}, h) \tag{3.17}$$

In order to separate the current interpolation point and surrounding known neighbour points, values for the neighbour points are referred to with $_j$ and values for the interpolation point is referred to with $_i$. m means the mass and ρ is density. $W(r_i - r_j, h)$ is the kernel function used, r is a coordinate in space and h is the

smoothing radius (defining a spherical or circle radius in 3D and 2D respectively). A_i and A_j is scalar quantities that we substitute in order to find a approximate value. An example of this is if we want to find the density at the position r_i. We set $A_i = \rho_i$ and $A_j = \rho_j$ resulting in the summation in Equations (3.18) and (3.19), giving us an approximation for calculating the density at position r_i.

$$\rho_i(\boldsymbol{r}) = \sum_{j=1}^{N} \frac{m_j}{\rho_j} \rho_j W(\boldsymbol{r_i} - \boldsymbol{r_j}, h) \tag{3.18}$$

$$\rho_i(\boldsymbol{r}) = \sum_{j=1}^{N} m_j W(\boldsymbol{r_i} - \boldsymbol{r_j}, h) \tag{3.19}$$

In this context its important to mention the relation between volume, mass and density in Equation 3.20:

$$V = \frac{m}{\rho} \tag{3.20}$$

Mass is given in kg, volume m^3 and density $\frac{kg}{m^3}$.

3.2.1 Smoothing values

In addition to approximating field values such as density SPH can also be used to find the gradient and Laplacian of the scalar quantity A_i [27]. When approximating the Naiver-stokes equations we need to be able to find the gradient and Laplacian of the scalar quantity. An example of this is approximating the pressure gradient. In order to find the value of a scalar field we use the following Equation 3.21.

$$A_i(\boldsymbol{r}) = \sum_{j=1}^{N} \frac{m_j}{\rho_j} A_j W(\boldsymbol{r_i} - \boldsymbol{r_j}, h) \tag{3.21}$$

When we want to approximate the gradient at a point $\boldsymbol{r}$ we use the following Equation 3.22 [27]:

$$\nabla A_i(\boldsymbol{r}) = \sum_{j=1}^{N} \frac{m_j}{\rho_j} A_j \nabla W(\boldsymbol{r_i} - \boldsymbol{r_j}, h) \tag{3.22}$$

We see that the gradient only affects our kernel since our variable in this function is $\boldsymbol{r}$ (note that the gradient of r can have multiple components). This attribute makes the SPH equations easy derive from the Naiver-Stokes equations, we simply substitute A for the value we want to approximate. The Laplacian approximation is as shown in Equation 3.23 [27]:

$$\nabla^2 A_i(\boldsymbol{r}) = \sum_{j=1}^{N} \frac{m_j}{\rho_j} A_j \nabla^2 W(\boldsymbol{r_i} - \boldsymbol{r_j}, h) \tag{3.23}$$

These equations are the basis for SPH simulations, but it poses an challenge. We need to make sure that the gradient and Laplacian of the smoothing kernel is suitable. We will discuss smoothing kernels and requirements in the next section.

3.2.2 Smoothing function

The smoothing function or kernel function is a function that weights the contributions from surrounding sample points based on distance. The input for a kernel function is $r_i - r_j$ the distance between our interpolation point and neighbour point. The second input is the smoothing radius h, the smoothing radius determines the maximum distance between the interpolation point and neighbour point. A smoothing kernel needs to satisfy the following condition in Equation3.24 [27]. Ideally it should also satisfy the conditions in Equations 3.25 suggested by Kelager [28].

$$\int_V W(r_{ij}, h)dr = 1 \text{ (Normalized)} \tag{3.24}$$

$$\begin{aligned} W(r_{ij}, h) &\geq 0 \text{ (positive)} \\ W(rij, h) &= W(-r_{ij}, h) \text{ (even)} \end{aligned} \tag{3.25}$$

There are several kernel functions developed for SPH. Different kernels are used when evaluating gradients and lapclacian. The kernel needs to be normalized, meaning that the volume V of the function is exactly 1, Monaghan [27] for further explanation. Its also important that the kernel has compact support meaning if $|r_{ij}| \geq h \implies W(r_{ij}, h) = 0$. Its important to choose kernels that are suitable for the force we are calculating. In some situations we need to use the gradient and Laplacian of the kernel function, we will discuss the kernel functions used by Müller [6].

W_{poly6} Kernel

This kernel is recommended by Müller when approximating density where we don't need the gradient or Laplacian[6]. We can see that the gradient and Laplacian turns negative in certain parts of the range $-2 \leq r_{ij} \leq 2$. The gradient and Laplacian of this function is not useful for our experiments. The W_{poly6} function can we written as Equation 3.26. The plot in Figure 3.1 function with the gradient and Laplacian.

$$W_{poly6}(r, h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - r^2)^3, & \text{if } 0 \leq r \leq h \\ 0, & \text{otherwise} \end{cases} \tag{3.26}$$

W_{spiky} Kernel

This kernel is used by Müller when calculating pressure forces between particles[6]. We use the gradient of the kernel function for this. When inspecting the gradient we see that its always positive in the range $-h \leq r_{ij} \leq h$ and the gradient is increasing exponentially as r_{ij} approaches 0. This correlates with our goal for

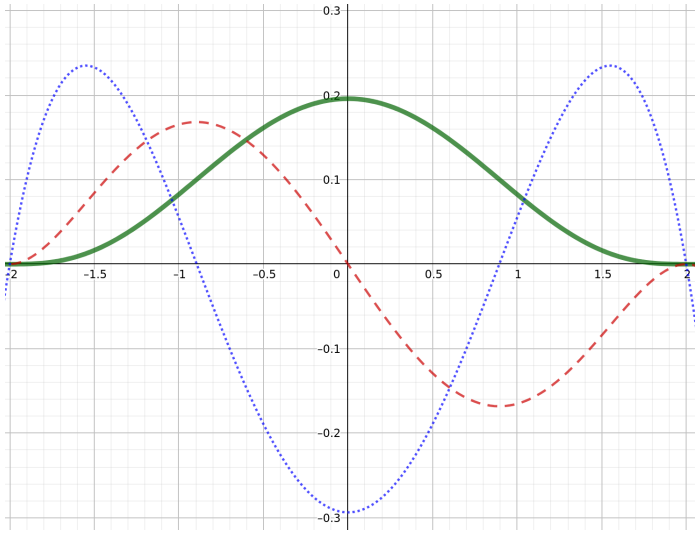


Figure 3.1: W_{poly6} kernel. The green solid line shows the kernel, the red striped line shows the gradient and the blue dotted line shows the Laplacian. The smoothing radius h is the constant 2.

pressure forces, that the pressure increases when particles get closer to one and another. This avoids a common problem in SPH simulations where particles cluster together when they get too close if the derivative of the kernel function does not grow as r_{ij} approaches 0. The kernel is shown in Equation 3.27. The plot of the Laplacian, gradient and kernel is shown in Figure 3.2.

$$W_{spiky}(r,h) = \frac{15}{\pi h^6} \begin{cases} (h-|r|)^3, & \text{if } 0 \leq r \leq h \\ 0, & \text{otherwise} \end{cases} \tag{3.27}$$

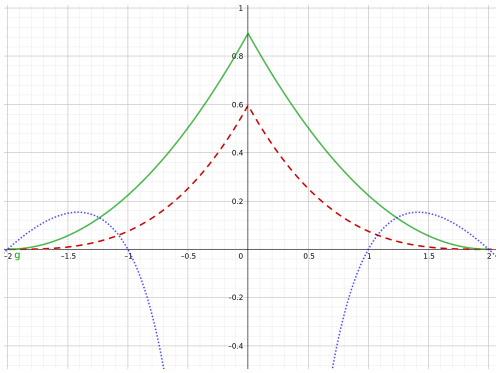


Figure 3.2: W_{spiky} kernel. The red striped line shows the kernel, the green solid line shows the gradient and the blue dotted line shows the Laplacian. The smoothing radius h is the constant 2.

$W_{viscosity}$ Kernel

This kernel is Müller when calculating viscous forces[6]. We use the Laplacian of this function, the kernel is designed to be positive for all values of r_{ij} inside the smoothing radius. The kernel function is shown in 3.28. The plot of the function with gradient and Laplacian is plotted in figure 3.3.

$$W_{viscosity}(r,h) = \frac{15}{2\pi h^3} \begin{cases} -\frac{r^3}{2h^3} + \frac{r^2}{h^2} + \frac{h}{2r}, & \text{if } 0 \leq r \leq h \\ 0, & \text{otherwise} \end{cases} \tag{3.28}$$

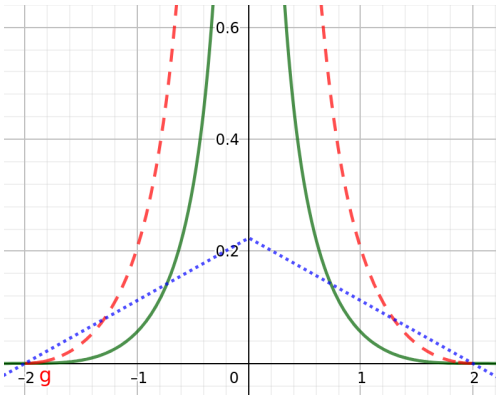


Figure 3.3: $W_{viscosity}$ kernel. The green solid line shows the kernel, the red striped line shows the gradient and the blue dotted line shows the Laplacian. The smoothing radius h is the constant 2.

3.2.3 Density

Monaghan suggests that densisty can be found as state in Equation 3.29 [27].

$$\rho_i = \rho_i(\boldsymbol{r}) = \sum_{j=1}^{N} m_j W(\boldsymbol{r_i} - \boldsymbol{r_j}, h) \tag{3.29}$$

($r_i - r_j$ is the distance between particle i and j).
Note that we also iterate over particle i in this summation, otherwise we could potentially get a zero-division later on. Most literature on SPH recommends a kernel similar to a Gaussian distribution when approximating the density value[1, 7]. An illustration on the influence for particle i is shown in Figure 3.4.

SPH viscosity weakly compressible fluid

The Navier-Stokes equation is given as follows for an in-compressible Newtonian fluid 3.30:

$$\frac{\partial \boldsymbol{v}}{\partial t} = -\frac{1}{\rho}\nabla p + \frac{\mu}{\rho}\nabla^2 \boldsymbol{u} + g \tag{3.30}$$

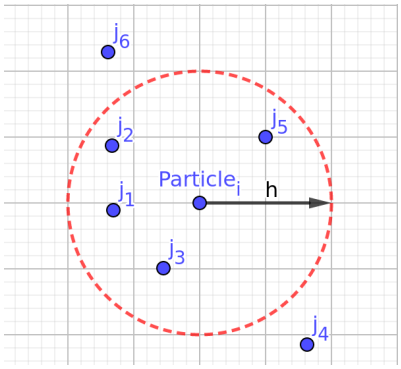


Figure 3.4: Mass density approximation for $particle_i$. The red striped circle is the smoothing radius. The particles inside the radius will contribute to the mass density of $particle_i$ based on the distance between them and $particle_i$.

We want to find the divergence of the velocity field gradient $\nabla^2 p$, μ is the viscosity constant of the fluid. We can expand the expression as In Equation 3.31:

$$\frac{\mu}{\rho}(\nabla \cdot \nabla \boldsymbol{u}) \tag{3.31}$$

In SPH notation viscosity force can be calculated in the following way using the specialized $W_{viscosity}$ kernel [6] (note that v_i, v_j is the current velocity of particle i and j respectively):

$$\frac{\mu}{\rho}\nabla^2 \boldsymbol{u}(\boldsymbol{r}) = \frac{\mu}{\rho}\sum_{j=1,i\neq j}^{N} m_j \frac{\boldsymbol{v_j} - \boldsymbol{v_i}}{2\rho_j}\nabla^2 W_{viscosity}(\boldsymbol{r_i} - \boldsymbol{r_j}, h) \tag{3.32}$$

($r_i - r_j$ is the distance between particle i and j).

3.2.4 Pressure

The Navier-Stokes equation for an in-compressible Newtonian fluid is given as follows 3.33.

$$\rho\frac{\partial \boldsymbol{v}}{\partial t} = -\nabla p + \mu\nabla^2 \boldsymbol{u} + \rho g \tag{3.33}$$

We want to find the pressure gradient $-\nabla p$, in SPH notation the pressure gradient can be described as Equation 3.34 [6].

$$-\nabla p_i(\boldsymbol{r}) = -\sum_{j=1,i\neq j}^{N} m_j \frac{p_j}{\rho_j}\nabla W(\boldsymbol{r_i} - \boldsymbol{r_j}, h) \tag{3.34}$$

($r_i - r_j$ is the distance between particle i and j).

We don't need to sum the forces for particle i for itself $i \neq j$. A problem with this Equation 3.34 is that the forces are not always symmetric when any two particles interact. In order to fix this issue the average of p_i and p_j can be calculated. Two different approaches are proposed in the literature[6, 27] Equations (3.35) and (3.36).

$$\text{Müller 2003:} -\nabla p_i(\boldsymbol{r}) = -\sum_{j=1, i\neq j}^{N} m_j \frac{p_j + p_i}{2\rho_j} \nabla W(\boldsymbol{r_i} - \boldsymbol{r_j}, h) \tag{3.35}$$

$$\text{Monaghan 1992:} -\nabla p_i(\boldsymbol{r}) = -\sum_{j=1, i\neq j}^{N} m_j (\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2}) \nabla W(\boldsymbol{r_i} - \boldsymbol{r_j}, h) \tag{3.36}$$

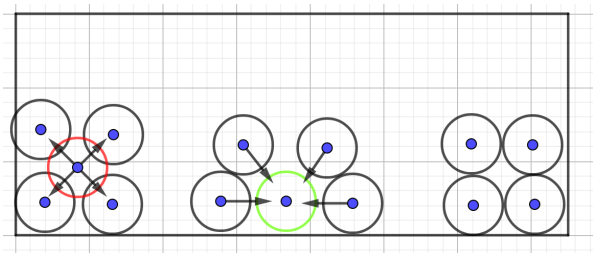


Figure 3.5: A crude illustration of pressure forces, on the left side particles are compressed (red) causing a repulsive force between them. In the center there is negative pressure meaning that the particles attract each other (green). To the right the pressure forces are balanced. The vectors give a indication of acceleration direction caused by pressure force.

Equation of state

In order to find the unknown pressure force p, we can use an equation of state to restore the intended mass density. We can simulate a weakly compressible fluid using this. The equation of state will scale the pressure to preserve a certain mass density. In order to simulate of snow/ice a more complex model is needed, since snow is highly compressible with a wide density range [20]. This is not included in this work, instead we focus on a weakly/in-compressible pressure solver. There are several formulations for an equation of state, (Müller, 2003)[6] suggests deriving a equation of state from the ideal gas state Equation 3.37.

$$p = k\rho \tag{3.37}$$

k is a gas constant that depends on the temperature, an modified version of this expressed as Equation 3.38.

$$p = k(\rho - \rho_0) \tag{3.38}$$

Where k is the gas constant, used for scaling the force, ρ is the current mass density and ρ_0 is the rest density. The equation will try to restore the the rest density

ρ_0. A problem with this equation is that we can get a negative pressure resulting in strong attraction forces where the density is low (see figure 1.5). We can avoid this by adding the condition that $(\rho - \rho_0) = max(\rho - \rho_0, 0)$.

3.2.5 Enforcing boundaries in SPH

In particle simulations it is important to enforce boundaries for the particles. Otherwise the particles can escape our simulation domain or not collide with other objects in the simulation. This is one of the big challenges in SPH simulations[27]. The reason why this is a challenge lies in the of the smoothing radius which SPH calculations are based on. When a particle is near a boundary some parts of the smoothing radius might be covering the boundary or exceeding the boundary. This can cause rapid density/pressure fluctuations for particles at the boundary causing instability. An illustration of this is shown in Figure 3.6. To combat the first

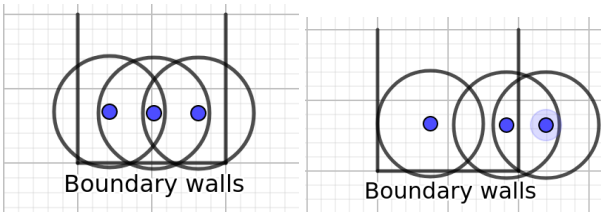


Figure 3.6: The left image illustrates the the cause of fluctuations at the boundary (very simplified illustration with 3 particles). The center particle here has a higher density than the boundary particles since their smoothing radius covers parts of the area outside the boundary wall. The right image shows that a particle inside the boundary can be influenced by particles outside the boundary.

problem with varying density, pressure at the boundary, we can place "dummy particles" at the boundary that contributes to the density and repels particles from going trough the boundary, these particles are stationary (does not move, only acts as a boundary). In order to avoid influence from particles outside a border we need to increase the resolution of the simulation. We can do this by reducing the smoothing radius and adding more particles (adjustment to particle mass is possible as well, but does not increase resolution).

3.2.6 Nearest neighbour search

We have written about neighbour search in SPH previous work, reffering to Appendix A.

Chapter 4

Implementing SPH in the HPC-lab Snow Simulator

In this chapter we describe the core of the master thesis work, which is how our SPH solvers were implemented and the features we added to the program such as the user interface and terrain modeling. The SPH solvers are integrated as a standalone component in the HPC-Lab snow simulator.

4.1 Parallel execution CUDA

The implementation of SPH uses CUDA kernels that are launched on the GPU (device) with one thread for each data element running in parallel. To launch the correct number of CUDA threads we use the following basic formula given a number of threads per block as shown in Code Listing 4.1:

Code listing 4.1: Finding kernel launch thread and block count in 1 dimension

```
uint particleCount = 4000;   // Example particle count
uint threadsPerBlock = 1024; // Common blocksize
uint blockCount = floor(particleCount / threadsPerBlock)+1; // = 4
// Total threads: threadsPerBlock*blockCount = 4096
```

This is an example of a function call that will be executed on the GPU (device). When we call a device function we tell the GPU how many threads to execute by supplying a block count and threads per block. Note that blocks and threads per block can be supplied in 3 dimensions (x,y,z). In these examples we use 1 dimension, referring to CUDA developer guide for more information [29] on this. Code for executing an kernel is shown in Code Listing 4.2.

Code listing 4.2: Launching a CUDA kernel from host

```
// Total threads running in parallel in this call: blockCount*threadsPerBlock
CudaKernel<<<blockCount,threadsPerBlock>>>(particleCount);
```

We can see from this that usually we get more threads than data elements. To avoid this we add a if-statement at the beginning of our CUDA kernels to stop

execution if the thread index exceeds the number of data elements as shown in Code listing 4.3:

Code listing 4.3: Finding kernel launch thread and block count in 1 dimension

```
__global__ void CudaKernel(uint particleCount, float* exampleData) {
uint index = blockIdx.x*blockDim.x+threadIdx.x;  // unique index for each thread
if (i <= particleCount) {
    return; //Index exceeds particle count
}
// Access example data
float value = exampleData[index];
... // Continue execution
}
```

4.2 SPH implementation

The SPH implementation is done in three parts. A Newtonian weakly compressible solver is the basic one, this version is based on Müller et al. [6]. This SPH algorithm is best suited to simulate water and other Newtonian fluids, but it can yield interesting performance numbers and visuals.

The second algorithm implementation is based Hosseini [7] and Krog 2010[5] implementations of a non-Newtonian SPH solver with different rhetorical models for viscous force. This model gave some realistic visuals and behaviour of snow avalanches.

4.2.1 Particle creation

When the program starts it will create a specified number of particles multiple arrays on host system. The particles contains a set of properties each that represent its state in the system. The initialization step allows for multiple initial configurations of particles. The particle properties for the Newtonian and non-Newtonian model is presented in Tables 4.1 and 4.2. The float3 and float4 are groups of 3 and 4 floats respectively representing vectors with x, y, z, w components. The matrix3 datatype is a set of three float3 representing a matrix.

The particles are allocated in different arrays where every particle has a unique index in the arrays. For example there is one array for storing positions for all particles and another array for velocities. In order to access all the properties for a given particle i we use the same index for every array as illustration in Figure 4.1.

Table 4.1: Particle properties used in the Newtonian model.

Property	Type	Byte size	Note
Position	float3	12	
Velocity	float3	12	
Last velocity	float3	12	For leap-frog time integration
Acceleration (force)	float3	12	Storage for pressure and viscous force
Density	float	4	
Pressure	float	4	
Intensity value	float	4	For coloring of forces
Particle cell	uint	4	Used to store current grid cell
Particle cell offset	uint	4	Used to store position offset in grid cell
Sum bytes		**68**	Memory requirement for each particle

Table 4.2: Particle properties used in the non-Newtonian model.

Property	Type	Byte size	Note
Position	float3	12	
Velocity	float3	12	
Last velocity	float3	12	For leap-frog time integration
Acceleration (force)	float3	12	Storage for pressure and viscous force
Stress tensor	matrix3	36	Storage for stress tensor
XSPH	float3	12	Storage for XSPH corrected velocity
Density	float	4	
Pressure	float	4	
Intensity value	float	4	For coloring of forces
Particle cell	uint	4	Used to store current grid cell
Particle cell offset	uint	4	Used to store position offset in grid cell
Sum bytes		**116**	Memory requirement for each particle

4.2.2 GPU Allocation of particles

When the particles have been created with initial values on the host system they need to be transferred to the GPU memory. In order to do this we need to allocate the same amount of memory on the GPU and then transfer the particle data to this memory. When memory is allocated on the GPU (using **cudaMalloc())** we get a pointer to the beginning of memory area we are given on the GPU. Then we use the **cudaMemcopy()** command to copy the data to the GPU. This is done repeatably for all the particle property arrays. In order to have access to the arrays inside a GPU kernel we need to pass pointers to the arrays at run time. When dealing with multiple arrays, we use a custom struct bind the arrays together so that they can be passed as one element to the GPU. We store the pointers to the GPU memory in the struct GPUBuff (Figure 4.2). There are also some functions in the struct to retrieve the pointer for a given particle property array.

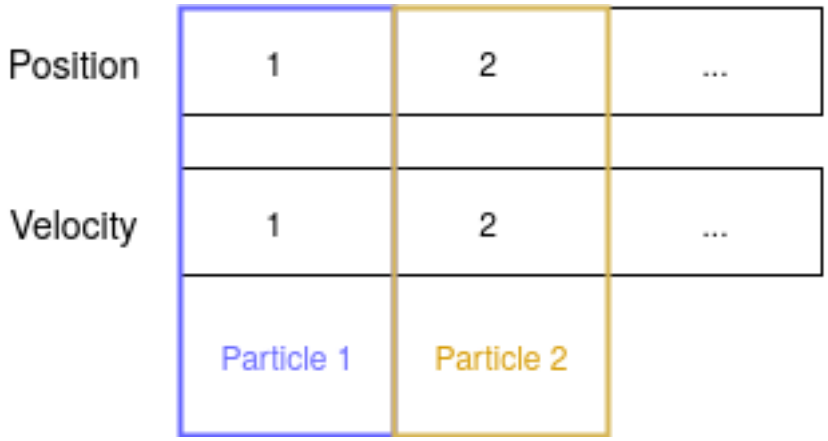


Figure 4.1: Accessing particle properties with index across multiple arrays.

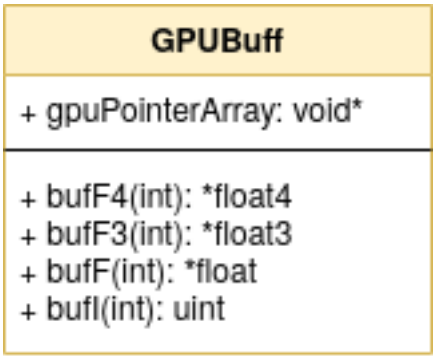


Figure 4.2: GPUBuff struct containing GPU memory pointers to particle property buffers.

4.2.3 Fast neighbour grid search

A SPH implementation is dependent on fast neighbour search in order to find particles that can are within the smoothing radius. In this implementation we employ a grid that spans the simulation space. The grid has a minimum and maximum point in 3D space describing the size of the structure. The grid is then divided into equal sized cells (or cubes) which covers volumes (in 3D) or area (in 2D).

Fast neighbor search in CUDA

In this work we are using a counting sort based neighbour search algorithm. The algorithm is a part of a SPH fluid simulation program called fluids v.3[9] developed by Hoetzlein. Fluids v.3 has later been implemented as a sample program in the Nvidia GVDB software package[30].

The algorithm divides the simulation domain into grid cells in three dimensions.

Each grid cell acts as a bin for particles. When the position of a particle is determined to be in a bin it is indirectly assigned to the bin in memory. When a particle later on is looking for neighbours it can iterate over the particles in the nearby cells. The size of each grid cell is determined by the smoothing radius h used in the simulation. To make the relationship between the smoothing radius and cell size easier we use the diameter of the particles influence sphere instead of smoothing radius, the diameter is $d = 2 \cdot h$. On Figure 4.3 we can see which cells and how many cells we need to search to find particles that could influence a given particle in a given cell in 2D. We can see that $Cellsize = d/1$ and $Cellsize = d/2$ both has 9 cells to check and that $Cellsize = d/3$ and $Cellsize = d/4$ has 25 cells to check.

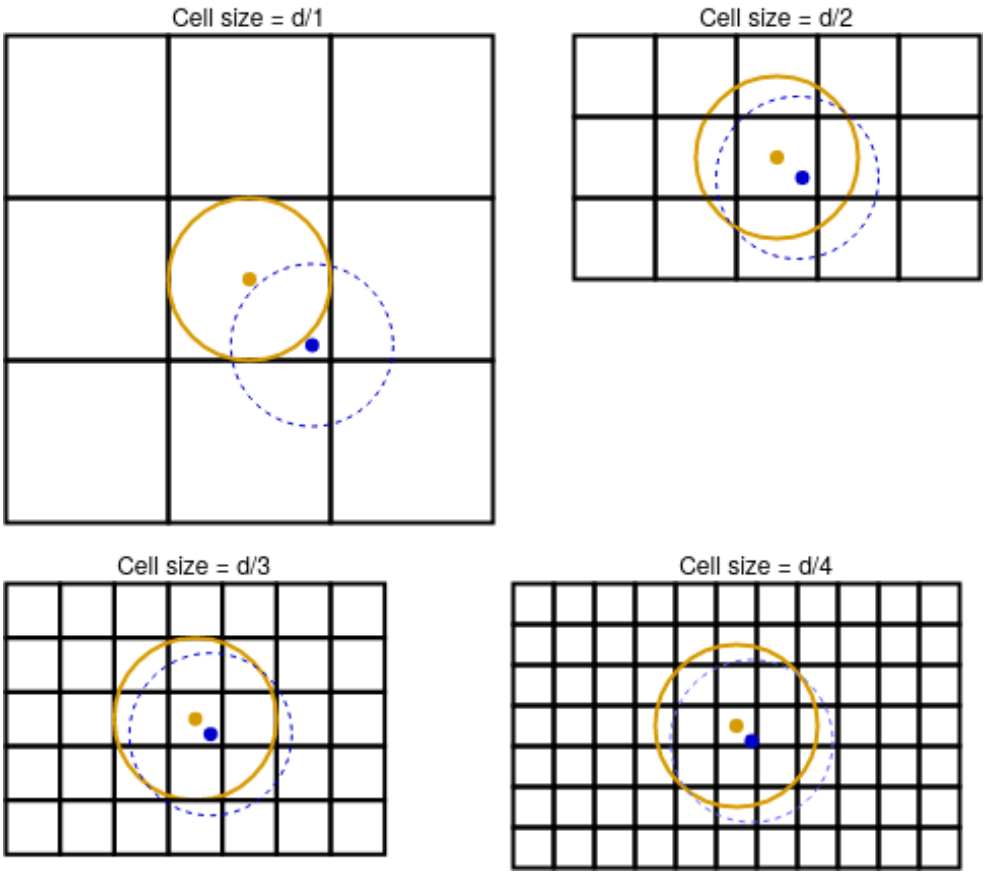


Figure 4.3: Figure of several 2D grid cell sizes with a constant particle influence $diameter$.

The CUDA counting sort algorithm

The algorithms goal is to place particles in the correct cell and at the same time store particles belonging to the same cell beside each other in memory. However the algorithm requires a separate copy of the particle properties (see Figure **??**) causing the memory usage by the particles to double. The algorithm is divided into three separate steps:

- Insert particles
- Prefix sum
- Counting sort

The first part finds which cell in the grid the particles belongs to. For each particle the index of the cell it belongs to is stored. The particle finds its cell using its position in the simulation domain. Each grid cell has a counter to keep track of how many particles is present inside. This counter is incremented every time a particle is assigned to the grid cell. Each particle also know what index it belongs to in its grid cell. If for example two particles have the same grid cell, then the first one stores the index 0 and the last one stores the index 1.

After the particles have a assigned position in the grid the prefix sum of the grid cell counts is calculated and stored for each grid cell. This is essentially done so that each particle later on know know where in memory the grid cell starts. An excellent illustration of this can be found in [30].

The last step of the algorithm is the counting sort. At this point every particle knows which cell it belongs to. And every grid cell knows how many particles should be stored before it. The particles copies its data to the memory location with the offset from the grid (prefix sum) plus its index in the grid cell. This sorts the particles by grid cell in memory.

4.2.4 Newtionan fluid implementation

Our Newtonian fluid implementation is based on Müller et al. [6]. The algorithm describes a weakly-compressible Newtonian fluid. We do not include surface tension in our model. We describe the calculations used to find the acceleration for each particle in the next equations.

$$\rho_i(\boldsymbol{r}) = \sum_{j=1}^{N} m_j W_{poly6}(\boldsymbol{r_{ij}}, h) \tag{4.1}$$

$$p_i = \min(\mathrm{c} \cdot (\rho_i - \rho_{rest}),\, 0\,) \tag{4.2}$$

After this step we make sure to synchronize the program by allowing all threads to finish the calculations above before continuing.

$$-f_i^{pressure} = -\frac{1}{\rho_i} \sum_{j=1, i \neq j}^{N} m_j \frac{p_j + p_i}{2\rho_j} \nabla W_{spiky}(\boldsymbol{r_{ij}}, h) \tag{4.3}$$

$$f_i^{viscosity} = \frac{\mu}{\rho_i} \sum_{j=1, i \neq j}^{N} m_j \frac{\boldsymbol{v_j} - \boldsymbol{v_i}}{2\rho_j} \nabla^2 W_{viscosity}(\boldsymbol{r_{ij}}, h) \tag{4.4}$$

After this step we synchronize all threads.

$$f_i^{external} = g + f_{boundaryforces} \tag{4.5}$$

$$Acceleration_i = (-f_i^{pressure} + f_i^{viscosity} + f_i^{external}) \tag{4.6}$$

The first step is to find the density of each particle as in Equation 4.1. From the density we calculate the pressure for particle i using the equation of state in Equation 4.2 based on the gas constant c and rest density ρ_{rest}. Then we find force contribution from pressure and viscosity in Equation 4.3 and 4.4. We then add gravity g and $f_{boundaryforces}$ in Equation 4.5. Boundary forces include repulsive force from the borders and friction. Then the final acceleration is calculated by multiplying the force contributions with the time step in Equation 4.6.

4.2.5 Non-Newtonian fluid implementation

The non-Newtonian implementation is based on the algorithm presented in Ø. Krog [1] and M. Hosseini et al. [7]. The rheological models in described in Section 2.4.4 can be used to simulate non-Newtonian fluids. The algorithm is more demanding because of the extra computation required to calculate the stress tensor and applying the rheological model.

$$\rho_i(\boldsymbol{r}) = \sum_{j=1}^{N} m_j W_{poly6}(\boldsymbol{r_{ij}}, h) \tag{4.7}$$

$$p_i = \min(\mathrm{c} \cdot (\rho_i - \rho_{rest}),\, 0\,) \tag{4.8}$$

After this step we make sure to synchronize the program by allowing all threads to finish the calculations above before continuing.

$$\nabla \boldsymbol{v_i} = \sum_{j=1, i \neq j}^{N} \frac{m_j}{\rho_j} (\boldsymbol{v_j} - \boldsymbol{v_i}) \otimes \nabla W_{spiky} \tag{4.9}$$

$$\boldsymbol{\tau_i} = rheological\ model \tag{4.10}$$

$$\boldsymbol{v_i^{xsph}} = \boldsymbol{v_i} + \varphi \sum_{j=1, i \neq j}^{N} 2m_j \frac{(\boldsymbol{v_j} - \boldsymbol{v_i})}{\rho_i + \rho_j} W_{poly6} \tag{4.11}$$

After this step we synchronize all threads.

$$-f_i^{pressure} = -\frac{1}{\rho_i}\sum_{j=1,i\neq j}^{N} m_j \frac{p_j + p_i}{2\rho_j}\nabla W_{spiky}(\boldsymbol{r_{ij}}, h) \tag{4.12}$$

$$f_i^{stress} = \frac{1}{\rho_i}\nabla \cdot \boldsymbol{\tau_i} = \frac{1}{\rho_i}\sum_{j=1,i\neq j}^{N} m_j \frac{\boldsymbol{\tau_i} + \boldsymbol{\tau_j}}{\rho_j}\nabla W_{spiky} \tag{4.13}$$

After this step we synchronize all threads.

$$f_i^{external} = g + f_{boundaryforces} \tag{4.14}$$

$$Acceleration_i = (-f_i^{pressure} + f_i^{stress} + f_i^{external}) \tag{4.15}$$

We calculate density and pressure in the same manner as the Newtonian implementation (Equations 4.7, 4.8). In Equation 4.9 find the gradient of the velocity vector. This gives us the the $3x3$ strain tensor, then we calculate the strain rate tensor and finally the strain rate (see Subsection 3.1.4). We then use the strain rate and rheological model to find the stress tensor $\boldsymbol{\tau_i}$. Then we use the XSPH method in Equation 4.11 to average the velocity for particles within the smoothing radius. The velocity calculated here is used in the time integration step later on. In Equation 4.12 the particle pressure is calculated. We use the pressure same pressure gradient calculation in [6]. The divergence of the stress tensor is found in Equation 4.10, this is the f_i^{stress} force. After this the external forces are calculated in Equation 4.14 and finally the acceleration is found in Equation 4.15.

4.2.6 Calculating stress tensor

When we are using the Bingham model and the Herschel-Bulkey model it is necessary to determine if the stress is less than the yield threshold τ_{yield}. M. Hosseini et al. [7] suggests that the stress magnitude for a given particle i can be found by Equation 4.16 where α is a scaling constant and μ is the viscosity and $|\boldsymbol{D}|$ is the magnitude of the strain-rate tensor. α is set to a high number to simulate a "solid" with high viscosity. In our implementation we replace $\alpha\mu$ with a "solid" viscosity μ_s with a high value that can be configured (example $\mu_s = 500$). We use the formulas presented in [7].

$$|\tau_i| = 2\alpha\mu|\boldsymbol{D_i}| \tag{4.16}$$

Bingham model

In the Bingham model we use the following Equation 4.17 to find the stress tensor τ_i. In the if the first condition is true the particle is below the yield stress value and

then get a high viscosity. If the particle exceeds the threshold value the Newtonian viscosity constant μ is used.

$$\tau_i = \begin{cases} 2\mu_s \boldsymbol{D_i}, & \text{if: } 2\mu_s|\boldsymbol{D_i}| \leq \tau_{yield} \\ (\frac{\tau_{yield}}{|\boldsymbol{D}|} + 2\mu)\boldsymbol{D}, & \text{otherwise} \end{cases} \tag{4.17}$$

Herschel-Bulkey model

The Herschel-Bulkey model has the same solid behaviour as the Bingham model, but when the yield threshold is passed the fluid then acts as a shear thinning or thickening fluid in the same manner as the power-law model. The formula for the stress tensor for the Herschel-Bulkey is shown in Equation 4.18. When the variable $N < 1$ the fluid acts as a shear thinning fluid, when $N > 1$ the fluid behaves in a shear thickening manner.

$$\tau_i = \begin{cases} 2\mu_s \boldsymbol{D_i}, & \text{if: } 2\mu_s|\boldsymbol{D_i}| \leq \tau_{yield} \\ (\frac{\tau_{yield}}{|\boldsymbol{D}|} + 2\mu|\boldsymbol{D}|^{N-1})\boldsymbol{D}, & \text{otherwise} \end{cases} \tag{4.18}$$

Power-law model

The power-law model enables us to simulate a dilatant (shear tickening) and shear thinning fluid. We use the formula in Equation 4.19 to find the stress tensor for this model. When the variable $N < 1$ the fluid acts as a shear thinning fluid, when $N > 1$ the fluid behaves in a shear thickening manner.

$$\tau_i = 2\mu|\boldsymbol{D_i}|^{N-N}\boldsymbol{D_i} \tag{4.19}$$

Newtonian fluid

For a Newtonian fluid we use the formula in Equation 4.20 to find the stress tensor.

$$\tau_i = 2\mu\boldsymbol{D} \tag{4.20}$$

4.2.7 Time integration

In the previous Subsections (4.2.4 and 4.2.5) we described how the acceleration of a particle is determined. In order to advance the simulation we need to calculate the new positions and velocities of the particles based on the acceleration.

Leapfrog integration

In this work we are using the leapfrog time integration scheme. With this method we update position at every full step and velocity at every half as shown in Figure 4.4. The leapfrog method can be written in the following way[31] in Equation

4.21:

$$\begin{aligned} a_i &= acceleration_i \\ v_{i+1/2} &= v_{i-1/2} + a_i \Delta t \\ x_{i+1} &= x_i + v_{i+1/2} \Delta t \end{aligned} \tag{4.21}$$

For our SPH implementation we also need to know the velocity at every full step. We approximate the velocity $\boldsymbol{v_{i+1}}$ as shown in Equation 4.22

$$v_{i+1} \approx \frac{\boldsymbol{v_{i-1/2}} + \boldsymbol{v_{i+1/2}}}{2} \tag{4.22}$$

Note that in our non-Newtonian model we use the XSPH corrected velocty when calculating $v_i + 1/2$ shown in Equation 4.23.

$$v_{i+1/2} = v_{xsph} + a_i \Delta t \tag{4.23}$$

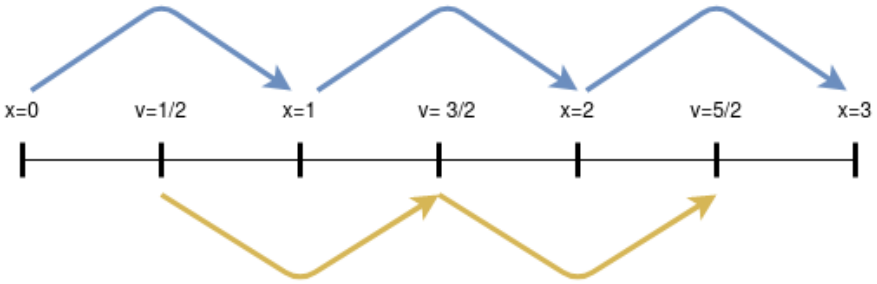


Figure 4.4: Leapfrog time integration. Velocity calculated at every half step and position is updated at every full step

Time step

We have implemented a static time step Δ that is that is configurable by the user. The simulation stability with a given time step can vary greatly depending on the initial simulation parameters and particle configuration. During testing we saw that the parameters that caused most instability at a fixed time step were: high viscosity coefficient μ, high particle velocity and higher gas constant c in the equation of state. A formula to set a time step based on max velocity and viscosity is described in [7].

4.2.8 Enforcing boundaries

To make sure particles stay inside the simulation domain we need to enforce boundaries. We do this by repelling particles that are about to pass the boundaries. Our boundary force implementation is based on T. Amada's [32] Equation

4.24:

$$\begin{aligned} f^{boundary} &= K_s \cdot d \cdot \boldsymbol{n} + K_d \cdot (\boldsymbol{v} \cdot \mathbf{n}) \cdot \mathbf{n} && \textbf{if:} \text{ Particle past boundary} \\ f^{boundary} &= 0 && \textbf{if:} \text{ Particle inside boundary} \end{aligned} \tag{4.24}$$

This implementation creates a mirror like particle reflection. K_s is the spring constant that scales the normal force exerted on the particle. The spring constant is multiplied with d which is how far the particle is past the boundary and $\boldsymbol{n}$ is the surface normal. K_d is the dampening constant and $\boldsymbol{v}$ is the velocity vector of the particle. The parameters K_s and K_d needs to be adjusted to achieve the desired boundary behaviour. It should be noted that d scales the first term in the equation with the distance the particle is past the boundary.

Boundary friction

We also found it useful to implement boundary friction as described in [1]. This reduces the velocity of particles by adding a opposing force based on the velocity of the particle and velocity direction compared to the surface normal of the boundary. There is also a threshold d that determine if the particle is close enough to the surface to be affected by this force. Friction force is shown in Equation 4.25

$$\begin{aligned} f^{friction} &= -K_{friction}(\boldsymbol{v} - \boldsymbol{v}(\boldsymbol{v} \cdot \boldsymbol{n})) && \textbf{if: } d <= threshold_{friction} \\ f^{friction} &= 0 && \textbf{if: } d > threshold_{friction} \end{aligned} \tag{4.25}$$

4.3 Simulation scaling

The snow simulator uses a coordinate system for rendering the environment. The minimum and maximum coordinates are the same at all times. In order to run simulations with custom sizes we want to use our own coordinate system and transform the coordinates of the particles to the rendering coordinate system. Our simulation scaling system has two parts. The first part selects a volume (cube) as a fraction of the rendering coordinate system. An example of this in 2D is shown in Figure 4.5 where we select a fraction from 30% to 70%. This is useful when we want to run our simulation in a more interesting region like a mountain.
When the simulation domain has been selected we can specify the physical size of the simulation domain in meters. This gives us a opportunity to run simulations at different resolutions in the rendering domain. The simulation domain in Figure 4.5 could be set to different sizes such as 100 meters x 100 meters or 1000 meters x 1000 meters. The physical size can be used to reduce or increase the resolution of the simulation and possibly simulate a region as the same physical size as it is in real life.

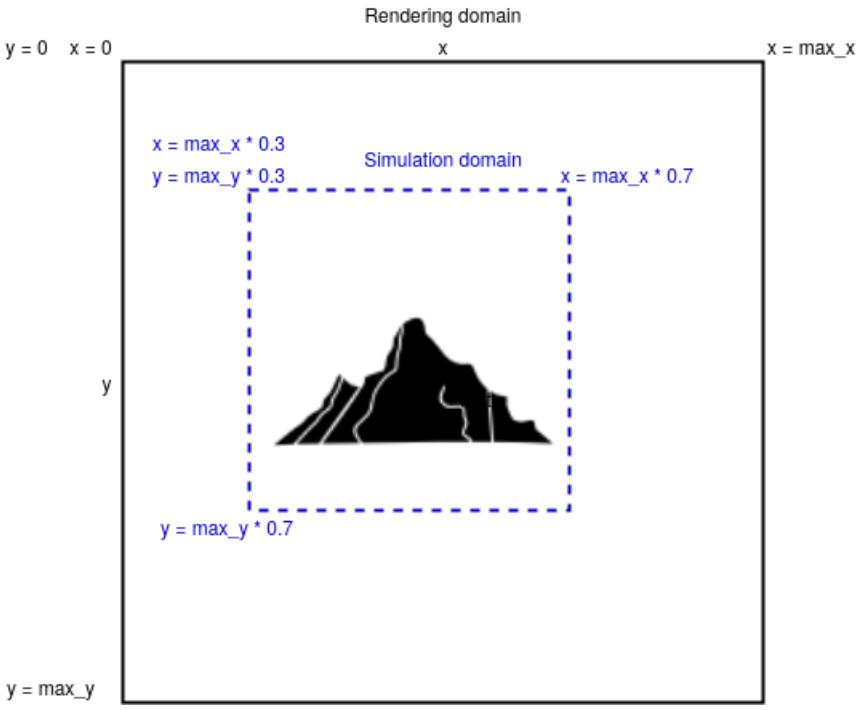


Figure 4.5: Simulation domain as a fraction of rendering domain in 2D.

4.3.1 Smoothing radius and simulation scaling

Since we run the simulation with meters as unit of measure there have been some challenges with the kernel functions. Often we want the smoothing radius h to be less than 1 meter, but this causes the kernel function to give unreasonable high numbers. We can see that h is below the fraction line to the power of 9 in the W_{poly6} kernel for example in Equation 4.3.1.

$$W_{poly6}(r,h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - r^2)^3, & \text{if } 0 \leq r \leq h \\ 0, & \text{otherwise} \end{cases}$$

To avoid h being lower than 1 in the simulation we set a custom smoothing radius that is used when calculating the kernel function value as shown in the Code listing 4.4. The custom smoothing radius is used as the h argument to the kernel function and the distance between particles r is multiplied with $scaleDistance$ to scale the distance to the custom smoothing radius. The drawback of using this method is that it requires an extra multiplication at each smoothing function calculation.

Code listing 4.4: Custom smoothing radius and distance scaling

```
// Actual smoothing radius in meters
float smoothingRadius = 0.2;

// Smoothing radius used in kernel function
float customSmoothingRadius = 2;

// Scale distance to new smoothing radius
float scaleDistance = (1 / smoothingRadius) * customSmoothingRadius;
```

4.4 Modeling terrain

The snow simulator creates its terrain and mountains from a height map. A height map describes the height of the terrain at different points and is usually stored as a gray scale image (0 = black, max = white) where 0 is the lowest point. We use the same height map as the simulator to model the terrain at different positions on the XZ-plane (ground plane). The terrain heights used to create boundaries (collision with terrain) in the y-dimension and to construct surface normal's. The surface normal's are used to make collisions with the terrain smoother (see boundary forces section).

Since we know the height of the terrain at any point in the XZ-plane we just need to find the surface normal's. We start by dividing the XZ-plane of the simulation domain into a grid of tiles. The size of each tile in X and Z direction can be configured. We proceed by finding a average height for each tile (average height from corners of the tile) this will be the height we use for collision testing. We then create the surface normal's by finding the numerical derivative of the height in X and Z direction. Assuming that we are at the position x_i and z_i and that we have a function that returns a height at a X,Z coordinate $H(x,z)$. We then create the following vector in Equation 4.26 in each tile, where we determine the height derivative when moving in x and z direction separately:

$$\boldsymbol{vec} = [\frac{H(x_i + \Delta, z_i) - H(x_i, z_i)}{2 * \Delta}, -1, \frac{H(x_i, z_i + \Delta) - H(x_i, z_i)}{2 * \Delta}] \tag{4.26}$$

The Δ is the step size, in other words how far we move in x and z direction before checking the height. We need to make sure that the Δ reaches far enough that it reaches the next height in the height map. After vec has been calculated we normalize the vector and make sure the normal found points upwards. The normal vectors and heights are then stored in an array and transferred to the GPU memory.

In Figures 4.6 and 4.7 we can see how the simulated particles behaves at different tile sizes. We can see that with larger tiles the particles creates a visible pattern around the edges of the tiles compared to the smooth distribution in the small tile

Figure 4.6: Particles on the ground (slight slope) with a small tile size.

Figure 4.7: Particles on the ground (slight slope) with a larger tile size.

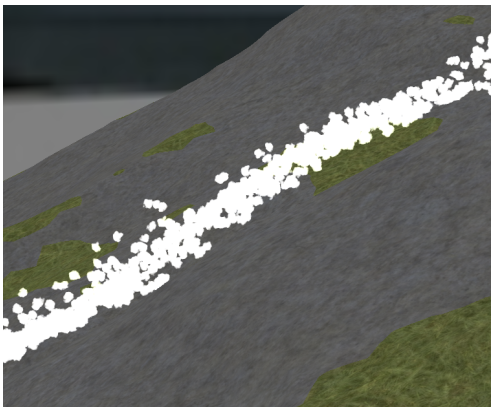


Figure 4.8: Particles sliding down a hill with a small tile size.

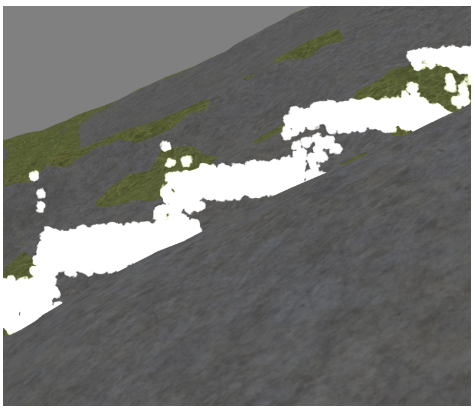


Figure 4.9: Particles sliding down a hill with a larger tile size.

size example. In Figures 4.8 and 4.9 we observe that a larger tile size creates a stairwell pattern when particles slide down the hill. This is even more noticeable in steep hills. The drawback of using smaller tiles is the increased memory usage and possibly more un-coalesced memory access. This will be discussed in the next chapter.

4.5 Implementing SPH in the snow simulator

A main focus of this project is to integrate the SPH solver into the snow simulator and make it accessible to the users of the HPC-Lab snow simulator. When the project started the source code for the snow simulator was outdated and hard to work with. I would like to acknowledge Roger R. Holten for his work on the snow simulator in his fall project, making it easier for us to integrate our solver into the program and setting up a user interface framework for us to use.

4.5.1 Program structure

The snow simulator has a component structure, where each addition to the program has separate files and classes. This is done so that different students can write code that is encapsulated from the other parts of the program at the same time. There is one exception to this, that is the terrain module that loads and imports a height map and makes it accessible to other components. Every component should expose some general functions to the program that can be invoked from the main program such as a initialization, physics, restart, delete and rendering. The snow simulator uses two compilers as well, it uses **gcc** at the top level and **nvcc** (Nvidia compiler) for CUDA code. The snow simulator uses OpenGL for rendering. We have built our program with one top level component that is called from the snow simulator. This component has a child component that is the CUDA part of the program that handles the physics, parameters and device memory. A illustration of the structure is shown in Figure 4.10.

Source code access

To request access to the full source code for the snow simulator Dr. Anne C. Elster can be contacted. The scource code is available on a private GitHub repository.

4.5.2 Simulation parameters

The simulator has quite a lot of parameters that can be set by the end user. The parameters we have included in the simulation is shown in Table 4.3

Table 4.3: Parameters that we use in the SPH implementation. Some less relevant can be seen in the source code and GUI.

Parameter	Unit	Variable(s) in program	Note
Simulation volume	m	xDim, yDim, zDim	Simulation size in x, y, z directions
Particle count	None	particleCount	Number of particles
Smoothing radius	m	smoothingRadius	
Particle mass	kg	particleMass	Particle weight
Rest density	$\frac{kg}{m^3}$	restDensity	Intended fluid density
Gas constant	None	gasConstant	Scales pressure to restore rest density
Viscosity	$\frac{N \cdot s}{m^2}$	viscosity	Viscosity constant
Solid Viscosity	$\frac{N \cdot s}{m^2}$	solidViscosity	Viscosity value to simulate a "solid"
Yield stress	Pa	yieldStress	Stress threshold for "solid", scalar.
Power-law constant	None	powerLawConstant	Controls shear thinning/thickening
Timestep	s	timeStep	Timestep for each iteration.
Gravity acceleration	$\frac{m}{s^2}$	gravityAcceleration	
Maximum velocity	$\frac{m}{s}$	maxVelocity	Maximum particle speed
Maximum acceleration	$\frac{m}{s^2}$	maxAcceleration	Maximum particle acceleration
XSPH constant	None	XSPHConst	For averaging velocity around particle
Boundary friction	None	boundaryFriction	Friction constant on ground boundary
Boundary friction height	m	boundaryFrictionHeight	How far above terrain friction affects particles

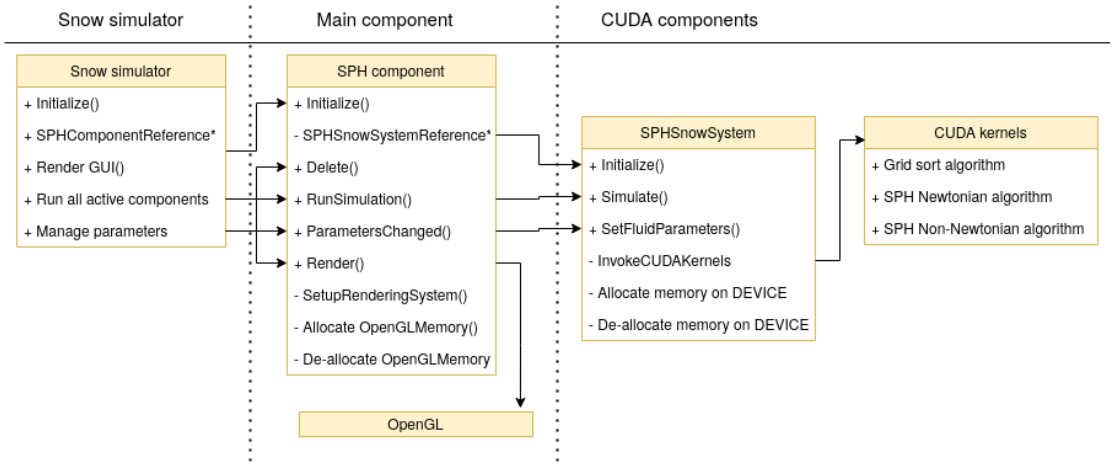


Figure 4.10: Diagram that shows how the SPH component is integrated in the HPC-lab snow simulator. Source code is included in the appendices.

4.6 User interface

The user interface in the HPC-lab snow simulatior is called ImGui O. Cornut [8]. This is a GUI framework for C++ applications. The framework handles rendering and has a lot of controls such as sliders, check boxes and buttons. Using ImGui we created a user interface for the SPH component of the snow simulator as shown in Figures 4.11 and 4.12. We have also added different scenarios in the presets selection that the user can choose from. These presets include parameters and initial particle positions. The user can then edit settings while the simulation is running or restart the simulation with different initial settings. The user can also benchmark the application. The benchmark system is the same as we used for testing in Chapter 5.

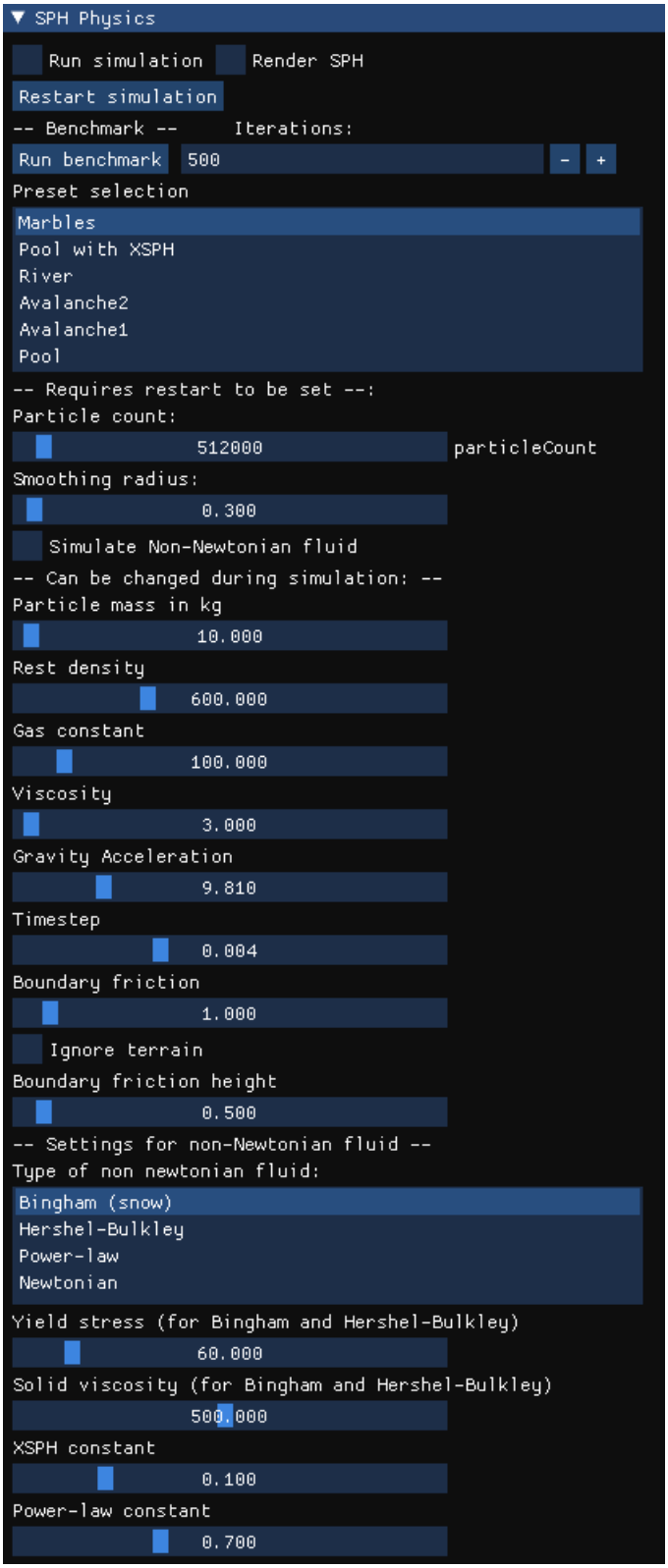


Figure 4.11: Image of the SPH user interface. Main view of physics settings.

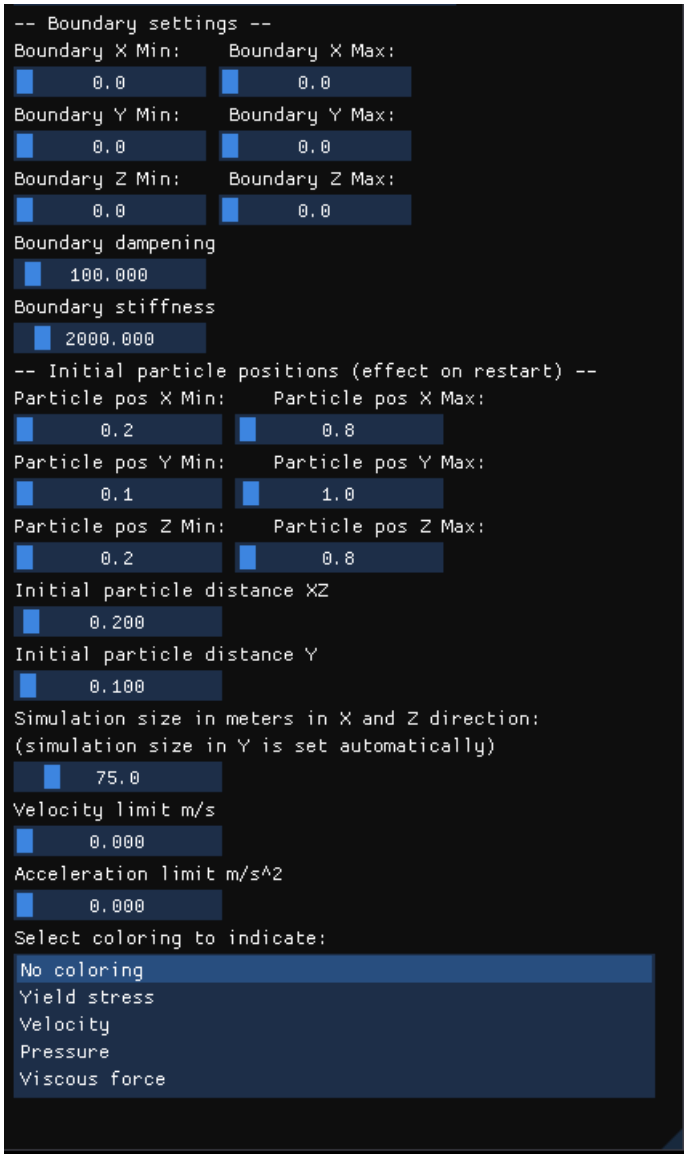


Figure 4.12: Image of the SPH user interface – Boundary settings.

Chapter 5

Profiling and optimizing Our SPH Implementation

In this chapter we will examine our results and describe optimizations we have added to make the simulation run faster. In Section 5.1 we will list the specifications of the hardware and software used. In the following Section 5.2 we will show the tools and methods we use for optimization and then explain the optimization strategy. In the next Section 5.3 we will identity bottlenecks in the program and try to apply changes based on this to improve performance.

5.1 Test systems

For our testing setup we are using two identical systems with the same processor and memory configuration as shown in Table 5.1. Using the same system configuration allows us to compare the GPU performance on an even ground.

Table 5.1: Specifications of the benchmark computers.

CPU	**Base clock**	**Cores**	**Memory capacity**	**Memory speed**	**Storage**
AMD Ryzen 7 5800X	3.6GHz	8	32GB	3200MHz	500GB m.2

The GPU's we use is shown in Table 5.2, at the time of writing the Nvidia RTX 3090 is the most powerful consumer GPU available from Nvidia. The Nvidia RTX 2080 TI was the most powerful consumer GPU from Nvidia in the previous generation (not accounting for the RTX titan).

In Table 5.2 we see that the RTX 3090 GPU has more than double the amount of CUDA cores compared to the RTX 2080 TI. Nvidia uses the term CUDA cores to describe how many 32bit floating point (FP32) units that are capable of executing FP32 instructions[14]. In the previous architecture called Turing (RTX 2080 TI) each streaming multiprocessor (SM) had 64 FP32 compute execution units and

Table 5.2: GPU's used for benchmarking.

GPU	CUDA core count	Base clock	Memory capacity	Memory Bandwidth	FP32 performance
Nvidia RTX 2080 TI	4352	1350MHz	12GB	616GB/s	13.45 TFLOPS
Nvidia RTX 3090	10496	1395MHz	24GB	936GB/s	35.58 TFLOPS

OS	Nvidia driver version	CUDA Version	Memory capacity	Memory Bandwidth	FP32 performance
Nvidia RTX 2080 TI	4352	1350MHz	12GB	616GB/s	13.45 TFLOPS
Nvidia RTX 3090	10496	1395MHz	24GB	936GB/s	35.58 TFLOPS

64 INT32 (32 bit integer) units that could execute simultaneously. In the Ampere architecture the INT32 execution units where changed making them capable of either performing FP32 or INT32 operations. The difference in SM design in shown in Figures 5.1 and 5.2. The changes to the SM gives the RTX 3090 much higher theoretical FP32 performance compared to 2080 TI.

5.2 Tools used and optimization strategy

When doing fluid simulations we want the simulation to run as fast as possible. We want to enable the users of the snow simulator to be able to do large scale simulations with high particle counts at a reasonable frame rate. With our optimization efforts we don't want to sacrifice usability of the software since the simulation package will be a component in the NTNU HPC-lab snow simulator. For example we want the users to be able to change parameters through the user interface without having to change the source code/recompile the program.

5.2.1 Analyzing kernel performance using timers

In the first phase of optimization we want to identify where the program spends most of the time. For the Newtonian implementation we have three different kernels that are called from the host code (not accounting for Counting sort algorithm). In the non-Newtonian implementation we have four kernels that are called from the host code. We utilize CUDA Events to estimate time spent executing each kernel. An example of a CUDA event based timer is shown below in Code listing 5.1. The CUDA event timer has a accuracy of one half microsecond.

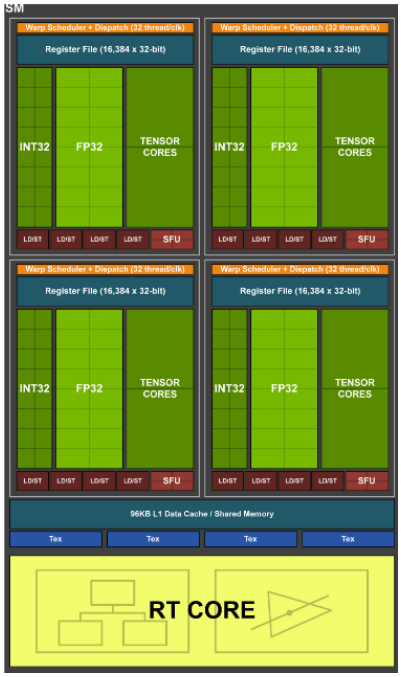


Figure 5.1: Turing architecture SM design. Used with permission from Nvidia.

Figure 5.2: Ampere architecture SM design. Used with permission from Nvidia.

Code listing 5.1: Timing a kernel using CUDA events

```
cudaEvent_t startTime, stopTime; // Initialize structs that CUDA event uses
float timeUsed = 0.0f;

// Create CUDA event for both variables
cudaEventCreate(&startTime);
cudaEventCreate(&stopTime);

cudaEventRecord(startTime); // Start timer
ExampleKernel<<<blockCount,threadsPerBlock>>>(); // Run kernel
cudaEventRecord(stopTime); // Stop timer

// Stop CPU here until "cudaEventRecord(stopTime)" is finished
cudaEventSynchronize(stopTime);
// Store time in milliseconds in variable timeUsed
cudaEventElapsedTime(&timeUsed, startTime, stopTime);
```

Using CUDA timers as shown in Code listing 5.1 we have created a benchmark that allows us to run multiple iterations of the simulation in succession. We disable rendering while the benchmark is running. The benchmark collects the kernel execution time for all kernels and the total run time for each iteration. When the benchmark is finished it calculates the average run time of each kernel. In this way we can identify where the program is spending the most time and compare the total run time between multiple systems. The result from the benchmarking tool is shown in Figure 5.3

```
Running benchmark with 100 iterations
----- Benchmark results ------Average run time in ms: 149.249761     Total run time: 14924.976075ms
Kernel: InsertParticlesCUDA               AverageTimeIteration: 0.333206  Total time used: 33.320640
Kernel: PrefixSumsCUDA                    AverageTimeIteration: 0.355589  Total time used: 35.558880
Kernel: CountingSortFullCUDA              AverageTimeIteration: 1.087551  Total time used: 108.755104
Kernel: IterateParticlesDensity           AverageTimeIteration: 73.984391  Total time used: 7398.439125
Kernel: IterateParticleForcesNewtonian    AverageTimeIteration: 72.935748  Total time used: 7293.574806
Kernel: CalculateVelocityTensor           AverageTimeIteration: 0.000000  Total time used: 0.000000
Kernel: IterateParticleForcesNonNewtonian AverageTimeIteration: 0.000000  Total time used: 0.000000
Kernel: IntegrateTime                     AverageTimeIteration: 0.553275  Total time used: 55.327520
```

Figure 5.3: Output from the benchmarking tool with 100 iterations on the Newtonian model. All time values are given in milliseconds.

5.2.2 Nvidia Nsight Compute

In Subsection 5.2.1 we described the way we time our CUDA kernels. When we have identified kernels we want to optimize we use a tool called Nvidia Nsight Compute[33]. This tool allows us to profile specific CUDA kernels and examine the run of the kernels in detail. During profiling the tool may run the same kernel several times to collect all the metrics it needs. After a kernel have been profiled with the tool we get a visual report with different sections and hints for potential improvements we can make to the kernel code in order to get more performance. The report generated from Nsight Compute is divided into several sections that

we are interested in:

GPU speed of light

This section of the report (Figure 5.4) shows how our kernel performance is relative to the theoretical maximum performance of the GPU. It includes several metrics that illustrate this including SM performance, memory performance and cache. Ideally we want our SM performance (computation) to be as high as possible. This section provides a very high level overview of the kernel performance, but it can give a indication of what could be improved. An example of this is a high memory utilization and a low SM performance, that could indicate that the compute pipeline is stalling while waiting for memory transfers. It should be noted that a knowledge of the kernel code is important before making judgement based on this data. For example a kernel could be doing a limited amount of compute and many memory transfers and/or copies that would not require a high compute performance.

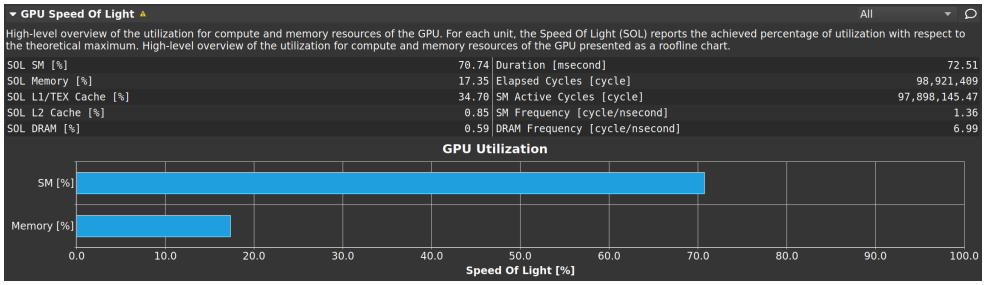


Figure 5.4: Speed of light section of the Nvidia Nsight Compute report.

Memory Workload Analysis

This section of the report shows the memory utilization of the kernel including the L1, L2 caches and device memory. We can see in detail the how often the L1 and L2 cache is hit (hit rate) and how often the global memory is accessed. We also see how much data that was transferred between the memory layers. Global memory reads are orders of magnitude slower than cache reads so ideally we want the global memory reads to be used as little as possible and avoid unnecessary writes to the global memory. A screenshot of this section is provided in Figure 5.5.

Scheduler Statistics and Warp State Statistics

In the Pascal and Ampere architecture Figures 5.1 and 5.2 each streaming multiprocessor (SM) consists of four processing blocks. Each of these blocks has its own warp scheduler (warp is a group of 32 threads), this scheduler can keep a

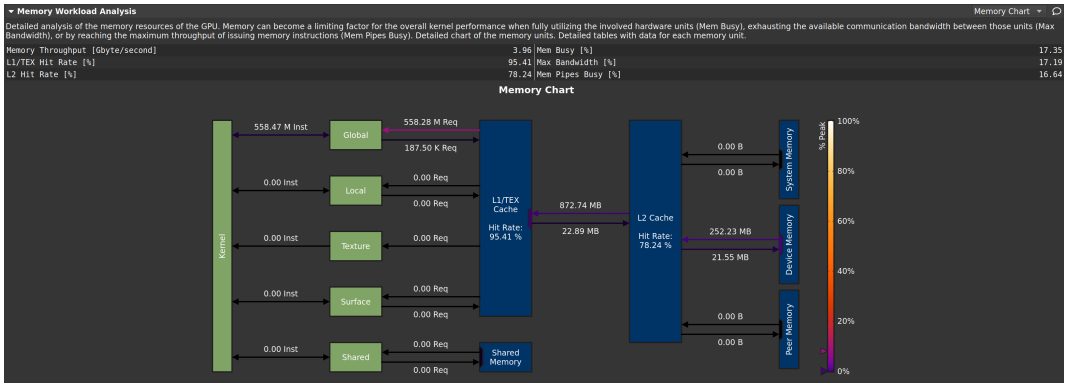


Figure 5.5: Memory Workload Analysis section of the Nvidia Nsight Compute report.

few warps on hold and issue an instruction for the one of these warps at every clock cycle. However if none of the warps that the warp scheduler has on hold is ready no instruction is executed in that processing block at that time (wasted compute cycle for that processing block). A warp is ready for a instruction when the data necessary for that instruction has been fetched from memory/cache. The scheduler statistics Figure 5.6 show how often the scheduler is able to issue an instruction (total instructions issued / total clock cycles) and how many warps on average that are ready to to execute an instruction (eligible warp). If the scheduler has a low ratio of (issued instructions / clock cycles) it could indicate memory bottleneck for example.

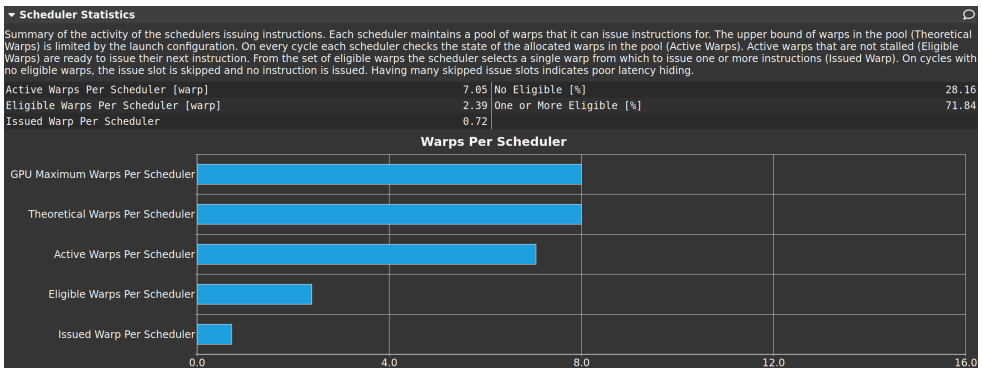


Figure 5.6: Scheduler Statistics section of the Nvidia Nsight Compute report.

The Warp State Statistics section show what state the warps are in most of the times. The state of a warp indicates if the warp is ready to execute an instruction. Warps have two states: A unused state (the slot for the warp in the warp scheduler is not filled), and a active state which has three different sub states: Stalled state (not ready to execute), eligible state (ready for instruction execution) and a selected state (the warp scheduler issue an instruction for this warp in the next

cycle). The most useful feature of this section is to find out why warps are in the stall states (not ready to run a instruction). There are several sub-states of a stall state. The simplified section of the Warp State Statistics is shown in Figure 5.7.

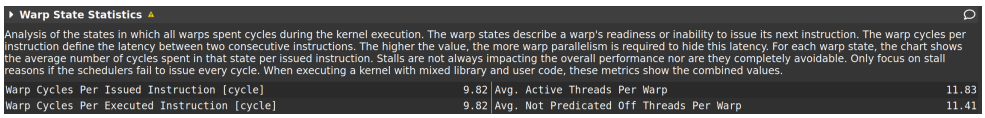


Figure 5.7: Warp State Statistics section of the Nvidia Nsight Compute report.

5.3 Identify bottlenecks and optimization

In this section we will look for potential optimizations for our program. The first step is to identify where the potential for improvement to the total run time lies.

5.3.1 Benchmarking the base application

We run our benchmark on both of our test systems with the GPU's in Table 5.2 to gather the base performance of the un-optimized version of the program. We use a configuration with 1 million particles over 500 iterations to find the average runtime of each kernel. We test both the Newtonian and non-Newtonian implementations.

Table 5.3: Average time spent on kernels in the Newtonian implementation. And 3090 average time relative to 2080 TI.

Kernel name	3090 avg. kernel time	2080TI avg. kernel time
InsertParticlesCUDA	0.100ms	0.130ms
PrefixSumsCUDA	0.161ms	0.272ms
CountingSortFullCUDA	0.263ms	0.394ms
IterateParticlesDensity	15.425ms	21.474ms
IterateParticleForcesNewtonian	15.859ms	21.585ms
IntegrateTime	0.120ms	0.191ms

In the Newtonian implementation in Table 5.3 we can see that the program spends most time on the kernels "IterateParticlesDensity" and "IterateParticleForcesNewtonian". As expected we can see that the 3090 performs better. The 3090 finished the benchmark in 15.80 seconds and the 2080 TI was finished after 27.484 seconds. We continue on in Table 5.4 inspecting the kernel performance of the non-Newtonian implementation: The non-Newtonian implementation was benchmarked in Table 5.4. The 3090 finished its 500 iterations in 27.48 seconds while

Table 5.4: Average time spent on kernels in the non-Newtonian implementation. And 3090 average time relative to 2080 TI.

Kernel name	3090 avg. kernel time	2080TI avg. kernel time
InsertParticlesCUDA	0.0997ms	0.129ms
PrefixSumsCUDA	0.163ms	0.272ms
CountingSortFullCUDA	0.263ms	0.393ms
IterateParticlesDensity	15.099ms	20.699ms
CalculateStressTensor	21.917ms	29.774ms
IterateParticleForcesNonNewtonian	17.290ms	22.653ms
IntegrateTime	0.133ms	0.224ms

the 2080 TI finished in 37.84 seconds. We observe that the kernels "IterateParticlesDensity", "CalculateStressTensor", "IterateParticleForcesNonNewtonian" and "IterateParticleForcesNewtonian", and this is where we should focus on making improvements.

5.3.2 Simplifying sum formulas

We have now identified the kernels that are most time consuming. These kernels share the same processing structure. The kernels start by finding its grid cell and then iterate over particles in its cell and neighbouring cells. While iterating over the particles in a given cell they add the the contribution from these particles to its sum (for the calculated force/density/XSPH/velocity tensor). An example of this is the "IterateParticleForcesNewtonian" shown in Code listing 5.2:

Code listing 5.2: Pseudocode for the Newtonian forces calculation. Note this is simplified "code" is run in parallel for each particle

```
void IterateParticleForcesNewtonian {
// Each thread gets its own index
// (each thread is responsible for different particle)
uint i = particleIndex;

// Get gridcell for particle
uint currentGridCell = getCurrentGridCell( i );

//Get particle properties for current particle
vec3 pos_i = GetPos(ix);
vec3 vel_i = GetVelocity(i);
float pressure_i = GetPressure(i);

// Pressure and viscous forces sums
vec3 pressureSum = [0,0,0];
vec3 viscousSum = [0,0,0];

// For each neighbouring cell (usually 27 including own cell)
Foreach(cell = GetNeighbouringCell(i)) {
    Foreach(j = ParticleInCell(cell)) {
        vec3 pos_j = getPos(j);
        vec_i_j = pos_i - pos_j;
        float dist =  distance(pos_i, pos_j);
        // If particle i and j is close enough to influence
        // Note squared smoothingRadius and dot product is actually used here
        if(dist <= smoothingRadius && i != j) {
            mass_j = GetMass(j);
            pressure_j = GetPressure(j);
            density_j = GetDensity(j);
            vel_j = GetVel(j);
            // Add pressure contribution
            pressureSum += mass_j*(pressure_i + pressure_j) / (2 * density_j) *
            W_Spiky_grad(dist, vec_i_j, smoothingRadius);
            // Add viscous contribution
            viscousSum += mass_j*(vel_j - vel_i) / (density_j) *
            W_Viscosity_lap(dist, smoothingRadius);
        }
    }
}
particleForces(i) = (-pressureSum + viscousSum * viscosityConst) / getDensity(i);
}
```

We see that for there is a possibility to move the mass term outside the loop assuming the mass is constant for all particles. This was not done before since we were originally planning to have particle based borders in the simulation, that would most likely require the mass to be different for border particles. However in the current implementation the mass is the same for all particles. In practice to do this optimization we just move $mass_j$ in in Code listing 5.2 and multiply it to the sum after we have collected the contributions. For all kernel functions we move constants that are separate from variables in the summation and multiply them with the sum instead.

In Code listing 5.2 we use two smoothing functions called "W_Viscosity_lap" and

"W_Spiky_grad" inside the "for each particle in cell". We use three smoothing functions in total in our simulation. The pseudocode for these can be seen in Code listing 5.3. These functions are called often and contain many multiplications, however since the smoothing radius is constant during a iteration we could calculate the constant part of the smoothing function in advance. When doing this we can extract this part from the loop in the same way as the mass and instead multiply the constant part with the sum after the summation is done. We can also compute the the smoothing radius to the power of two h^2 in advance of the "W_poly6" calculation. We also remove the if condition since we check the distance between the particles before calling the methods.

Code listing 5.3: Pseudocode for the unoptimized smoothing functions used.

```
/*
Function parameters:
float r = The distance between particle i and j
float h = The smoothing radius
vec3 r_i_j = pos_j - pos_i (position vectors)
*/
float W_Poly6(float r, float h) {
    if(r <= h) {
// Constant part independent on r: (315 / (64 * PI * pow(h, 9)))
        float res = (315 / (64 * PI * pow(h, 9))) * pow(pow(h, 2) - pow(r, 2), 3);
        return res;
    } else {
        return 0;
    }
}

vec3 W_Spiky_grad(float r, vec3 r_i_j, float h) {
    if(r <= h) {
// Constant part independent on r: ((-45.0f) / (PI * pow(h, 6)))
        float res = ((-45.0f) / (PI * pow(h, 6))) * pow(h - r, 2);
        return res * r_i_j / length(r_i_j);
    } else {
        return [0,0,0]
    }
}

float W_Viscosity_lap(float r, float h) {
    if(r <= h) {
// Constant part independent on r: (45 / (PI * pow(h, 6)))
        return (45 / (PI * pow(h, 6))) * (h - r);
    } else {
        return 0;
    }
}
```

Results

After applying the optimizations we benchmarked the program again and the results are in Tables 5.5 and 5.6. In the Newtonian implementation the 3090 was 2.11 times faster than previously, and 2080 TI was 2.05 times faster. In the non-Newtonian implementation the 3090 performance was improved by 1.81 times

and the 2080 TI had an performance gain of 1.79.

Table 5.5: Average time spent on kernels in the Newtonian implementation after optimizing the smoothing functions and kernels. Also shows performance gain compared to reference benchmark.

Kernel name	**3090 avg. kernel time**	**2080TI avg. kernel time**	**3090 gain**	**2080TI gain**
IterateParticlesDensity	5.557ms	7.960ms	2.77	2.69
IterateParticleForces-Newtonian	8.942ms	12.460ms	1.77	1.73

Table 5.6: Average time spent on kernels in the non-Newtonian implementation after optimizing the smoothing functions and kernels. Also shows performance gain compared to reference benchmark.

Kernel name	**3090 avg. kernel time**	**2080TI avg. kernel time**	**3090 gain**	**2080TI gain**
IterateParticlesDensity	5.876ms	8.275ms	2.57	2.50
CalculateStressTensor	11.826ms	16.423ms	1.85	1.81
IterateParticleForces-NonNewtonian	12.017ms	15.742ms	1.44	1.43

We can see that the biggest performance gain was in the "IterateParticlesDensity" kernel, this is as expected since this is the simplest kernel. The kernel "IterateParticleForcesNonNewtonain" had the smallest performance gain. This kernel calculates the stress and pressure forces. The stress force calculation is has a significantly higher data number operations compared to the other summations. However using the Nsight Compute profiler we see that the GPU spends most of its time at the step where it fetches its neightbour position to determine the distance between particles and the vector between particles (for pressure force calculation). The Nsight Compute profiler records samples during execution to determine the progress of warps in the kernel. When a sample is collected the warp state is also recorded and color coded as shown in Figure 5.10. Slight brown color indicates the warp state "Long scoreboard" which means that the warp is waiting for a global memory read. We see that this is where our kernels spends the majority of the time (those listed in Tables 5.5 and 5.6) on global memory reads. After we implemented the change we ran Nvidia Nshight Compute to profile one kernel before after the change Figures 5.8 and 5.9 to compare the results. The optimized kernel statistics is shown in blue and the old kernel is shown in green.

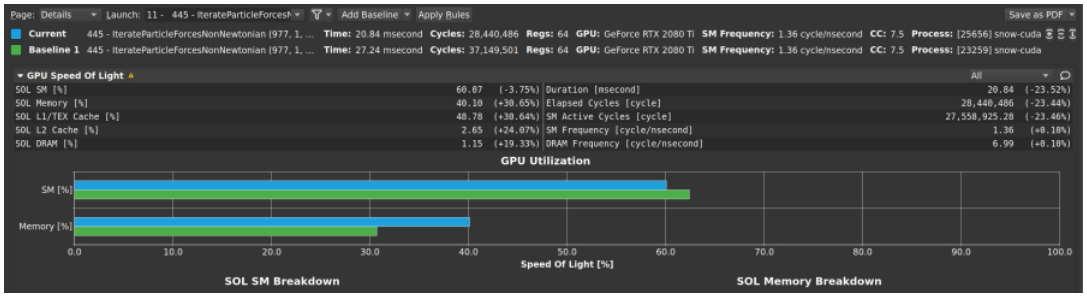


Figure 5.8: Speed of light section the IterateParticleForcesNonNewtonian kernel showing GPU utilization. Here one sees one gets a more even memory/compute ultilization ratio, indicating a smaller compute bottleneck.

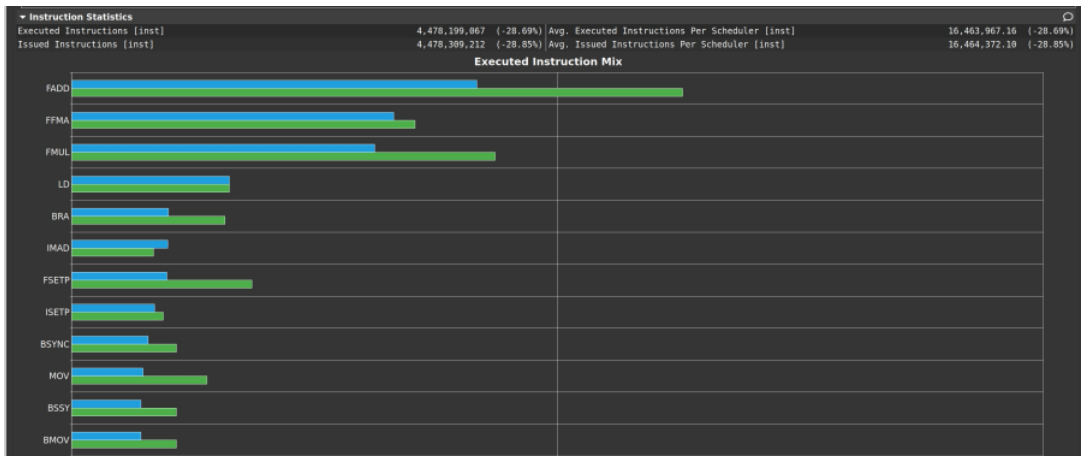


Figure 5.9: Instruction statistics for the IterateParticleForcesNonNewtonian kernel. Here one can see that the changes drastically reduced the overall instruction count.

5.3.3 Utilizing constant memory

Constant memory is a section of the global memory that has a cache in each SM (among four compute blocks). The constant memory is read only when accessed in device functions and can be set in host code. The memory features a broadcasting ability when threads in a warp access the same data location and has the same read time as a register (1 clock cycle) when the memory is hit. However when threads in a warp access a different location in shared memory the access is serialized. An example of this is if 32 threads in a warp access a different location effective read speed would be 1/32 compared to a read on the same location. This makes the shared memory ideal for memory addressed that are accessed simultaneously by threads. There is limited information about how the cache is implemented in the SM. But some information is available in the CUDA programming guide [15]. The constant memory has a size of 64KB and each SM has a 8KB cache reserved for constant memory. The constant data we have is our fluid parameters that are

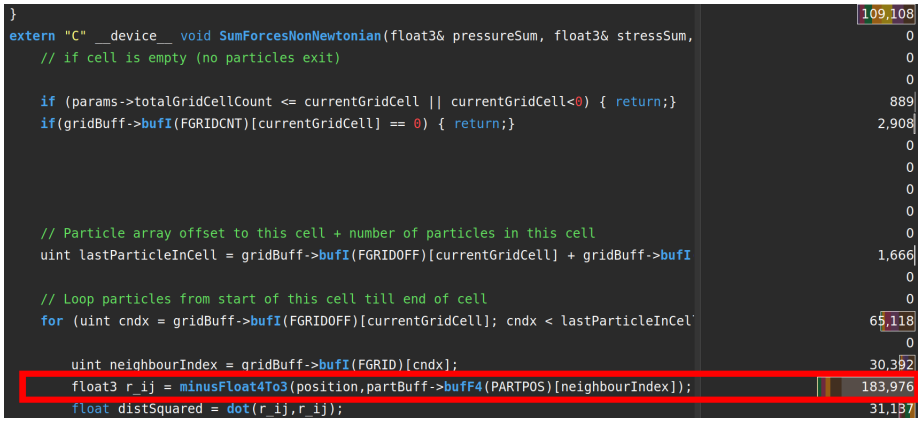


Figure 5.10: Output from Nvidia Insight showing source code on the left and number of samples at the right. The samples also records the warp state.

contained in a struct. We copied the struct to the constant varable defined in the source file as shown in Code listing 5.4:

Code listing 5.4: Copying the fluid parameters struct to a constant variable.

```
// Constant memory varible that we write to
__device__ __constant__ SPHParameters params;

// Kernel access example (viscosityConstant in this case): params.viscosityConstant

// Function invoked from host code before kernel run
void WriteParametersToConstant(SPHParameters* sphParameters) {
    CUDA_SAFE_CALL(cudaMemcpyToSymbol(params,sphParameters,
    sizeof(SPHParameters),0,cudaMemcpyHostToDevice));

    // The SPHParameters struct has a size of 1100 bytes (1.07KB)
}
```

The use of constant memory did not affect the performance noticeably. Over several benchmark runs we don't see any performance gain or degradation.

5.3.4 Changing position datatype

When we were making the initial implementation we wanted to use an extra component in the position vector to store a color to represent the intensity of values for that particle. A particle position was represented as a vector with 4 components $[x, y, z, intensity]$. The "intensity" value was supposed to be used by the fragment shader to color the individual particle based on the value. However we that our kernels spend a lot of time fetching the position of nearby particles. When a warp accesses memory its done in blocks of sizes 32, 64, or 128 bytes called a memory transaction Harris. M [34]. A position vector consists of four float numbers (32bit float) that adds up to 16 bytes for each particle. If 32 threads (warp) need to access different positions, the warp would need to fetch $16 * 32 = 512$bytes

from memory, resulting in $512/128 =$ transactions. If we instead used a 3 component vector for positions (without intensity) the warps would need to fetch $12 * 32 = 384$bytes, giving $384/128 = 3$transactions. It is however likely that the threads in the warp are accessing similar locations when iterating over particles in cells since the threads/particles are sorted by grid cell. But nevertheless it does not hurt to reduce the memory footprint. The results from the benchmark is shown in Tables 5.7 and 5.8.

Table 5.7: Average time spent on kernels in the Newtonian implementation with constant memory use and change to position data. Also shows performance gain compared to previous optimization.

Kernel name	3090 avg. kernel time	2080TI avg. kernel time	3090 gain	2080TI gain
IterateParticlesDensity	5.412ms	7.475ms	1.03	1.06
IterateParticleForces-Newtonian	8.592ms	11.763ms	1.04	1.06

Table 5.8: Average time spent on kernels in the non-Newtonian implementation with constant memory use and change to position data. Also shows performance gain compared to previous optimization.

Kernel name	3090 avg. kernel time	2080TI avg. kernel time	3090 gain	2080TI gain
IterateParticlesDensity	5.730ms	8.275ms	1.02	1.05
CalculateStressTensor	11.826ms	15.675ms	1.02	1.04
IterateParticleForces-NonNewtonian	12.017ms	15.062ms	1.04	1.05

5.3.5 Changing L1 data cache size

In the Turing and Ampere architecture there is support for changing the L1 data cache size[13]. The L1 cache is shared between the processing blocks inside the SM as shown in Figures 5.1 and 5.2. The cache is divided into two parts: L1 data cache and shared memory. The L1 cache is used by the processing blocks inside the SM as a cache for read/writes and texture memory while the shared memory section can be used inside thread-blocks for threads to exchange data. We are not using the latter part in our implementation so we want to dedicate the biggest portion to L1 data cache. The ratio between shared memory and L1 data cache can be set for each kernel. The procedure for setting the ratio is shown in Code listing 5.5:

Code listing 5.5: Changing the L1 data cache size with cudaFuncSetAttribute for a kernel.

```
/* Parameters:
    - Kernel name
    - Attribute name (allways the same)
    - Carve-out preference:
    cudaSharedmemCarveoutMaxL1
    cudaSharedmemCarveoutDefault
    cudaSharedmemCarveoutMaxShared
*/
cudaFuncSetAttribute(IterateParticlesDensity,
                cudaFuncAttributePreferredSharedMemoryCarveout,
                cudaSharedmemCarveoutMaxL1);
```

We did not see any performance impact from changing the L1 data cache size. It is possible that CUDA automatically forces the "cudaSharedmemCarveoutMaxL1" policy, when the kernel is not using shared memory. However we cannot confirm this.

5.3.6 Maximizing occupancy

In a Nvidia GPU each SM has a finite amount of registers, since compute capability 5.0 each SM has 65536 registers shared between the processing blocks in the SM [15]. The register usage of a given kernel is determined at compile time, the compiler attempts to limit the register usage as much as possible while avoiding performance loss. If the register count is too low (depending on amount of reads/writes/temporary storage within kernel) in a kernel then each thread can be forced to store data in global memory causing a potentially massive performance penalty. The register usage of a kernel is given by how many registers each thread need. The register and shared memory usage for a kernel determines how many thread blocks that can be assigned to each SM. Each SM has a maximum thread limit and block limit in addition to this, for us the thread limit is most important. The RTX 2080TI can have 1024 threads assigned to its SM at a given time while the RTX 3090 can handle 1536 threads. Given that the warp size is 32 (32 threads), each processing block in the RTX 2080TI can handle $(1024/32)/4 = 8$ warps while the RTX 3090 can handle $(1536/32)/4 = 12$ warps [15].

The measure of occupancy is the ratio of active warps to the maximum supported warp count. The warp scheduler of each processing block can issue a instruction at every clock for any of its warps. The warp scheduler plays a key role in hiding latency. If a warp is waiting for a memory read for example the scheduler could pick other warps that are assigned to the processing block to execute instructions while waiting for stalled warps. However if the register usage of a kernel is too high each SM could be assigned fewer than the maximum amount of supported threads, resulting in the warp schedulers having fewer warps to choose from and therefore potentially not being able to issue instructions as often. It should also be noted that higher occupancy does not always increase performance. It depends on

where the bottleneck is. If the warp scheduler with fewer warps is able to execute instructions as often as if it had more warps the performance would be similar. However a lower occupancy means that we are not utilizing the latency hiding mechanism of the GPU that well.

Since we are using both the RTX 3090 and 2080TI with different architectures we are instead telling the compiler how many threads per block we want to use and let it determine how many registers to use for the kernels.

Code listing 5.6: Using launch_bounds to tell compiler intended block size. label

```
// Here we are using __launch_bounds__(512) to tell the block size we use (512)
__global__ __launch_bounds__(512) KernelExample() {
    code...
}
// Number of blocks can also be specified: __launch_bounds__(512,<blocksPerSM>)

/* At compile time you can also specify a maximum register count with the flag:
(Affects all kernels)
--maxrregcount <number>
*/
```

5.4 CUDA OpenGL Interoperability

When we render the particles in the simulation we need to place the position array and intensity values in OpenGL buffers in order for our shaders to process the data for every frame. Normally this data would be copied from the device memory to host memory and then to the OpenGL buffer (that resides in device memory). This is a unnecessary operation since our data already is present in device memory. When we allocate memory for our particles we allocate our particle position data and intensity values as a OpenGL buffers instead of "cudaMalloc(..)". Code listing 5.7 shows how we create our OpenGL buffers and Code listing 5.8 shows how we mapped the buffers to a device pointers.

Code listing 5.7: Creating OpenGL buffers for particle positions and intensity values

```
// Allocate opengl vbo
// Create particle buffer (particle positions):
glGenBuffers(1, &snowVBO);
glBindBuffer(GL_ARRAY_BUFFER, snowVBO);
glBufferData(GL_ARRAY_BUFFER, conf.particleCount * sizeof(float) * 3, NULL,
    ↪ GL_DYNAMIC_DRAW);

// Create intensity buffer for coloring particles
glGenBuffers(1, &colorVBO);
glBindBuffer(GL_ARRAY_BUFFER, colorVBO);
glBufferData(GL_ARRAY_BUFFER, conf.particleCount * sizeof(float), NULL,
    ↪ GL_DYNAMIC_DRAW);

// Create and bind vertex array
glGenVertexArrays(1, &snowVAO);
glBindVertexArray(snowVAO);

// Bind particle buffer to VAO
glBindBuffer(GL_ARRAY_BUFFER, snowVBO);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(float) * 3, 0);
glEnableVertexAttribArray(0);

// Bind intensity buffer to VAO
glBindBuffer(GL_ARRAY_BUFFER, colorVBO);
glVertexAttribPointer(1, 1, GL_FLOAT, GL_FALSE, sizeof(float), 0);
glEnableVertexAttribArray(1);

glBindTexture(GL_TEXTURE_2D, 0);
glBindBuffer(GL_ARRAY_BUFFER, 0);

// Unbind VAO
glBindVertexArray(0);
```

Code listing 5.8: Mapping and un-mapping a OpenGL buffer to a device pointer for use in device code.

```
// Map OpenGL buffer to device pointer
void SPHSnowSystem::RegisterCUDAOpenGLResource(cudaGraphicsResource_t& resource,
    ↪ GLuint vboId,void*& opengl,size_t& size){
    /*
    First command: cudaGraphicsGLRegisterBuffer to bind the VBO ID from OpenGl
        ↪ buffer to a cudaGraphicsResource
    */
    CUDA_SAFE_CALL(cudaGraphicsGLRegisterBuffer(&resource, vboId,
        ↪ cudaGraphicsRegisterFlagsNone));
    /*
    Second command: Maps the resource for use:
    */
    CUDA_SAFE_CALL(cudaGraphicsMapResources(1,(cudaGraphicsResource_t*)&resource));
    /*
    Third command: Get pointer and size of the OpenGL buffer that resides in device
        ↪  memory
    */
    CUDA_SAFE_CALL(cudaGraphicsResourceGetMappedPointer(&opengl, &size, resource));
};

void SPHSnowSystem::UnRegisterCUDAOpenGLResource(cudaGraphicsResource_t& resource){
    // When we are finished using the resource we unmap it. (typically when
        ↪ destructor is called on our class)
    CUDA_SAFE_CALL(cudaGraphicsUnmapResources(1, &this->cudaOpenGLResource1, 0));
};
```

Chapter 6

Benchmarks and Visual Results

In this chapter we present several of the benchmarks performed on our implementations as well as some of the visual results from the HPC-Lab snow simulator integration. The following section includes some test scenarios to examine the performance of the optimized version of the program and show the results. Images of snow and water simulation using the non-Newtonian and Newtonian implementation, respectively, follow in Section 6.6.

6.1 Benchmarks method and results

We decided to use two different benchmarks for our implementations. We want to see mainly how our solution scales with a increasing number of particles. However as we have discussed in the previous chapter, Section 5.3 our main bottleneck is the neighbour search so at different particle counts we want to ensure that the particles have approximately the same number of neighbours. We also want to use the same smoothing radius to ensure that our grid configuration is not changed between the runs. With this in mind we have prepared two different benchmarks, one that we call "the pool" which is a simple cube volume with particles and one called "avalanche" that simulates a avalanche descending from a hillside.

In the pool benchmark particles have a high mass value to ensure that they almost achieve their rest state without density contributions from nearby particles. This will ensure a stable neighbour count as particle count increases. Particles are also placed with approximately a smoothing radius delta between them in all dimensions at the beginning. Figure 6.2 show an example of the pool benchmark.

The avalanche benchmark is a benchmark where we simulate a avalanche in a smaller region. The simulation domain is 150x150x75 (x,z,y) meters, but we only utilize some of the space (approximately a region of 30x50x35 meters). In this benchmark we will be reducing the particle mass when we increase the particle count while keeping the smoothing radius fixed. This will not be the total mass will not be exactly the same but we will make sure the result at different particle

counts remains as similar as possible. In this benchmark the number of neighbours (for each particle) will increase as we add more particles and we expect a bigger slowdown at higher particle counts. Figure 6.1 show the initial position of the snow before the benchmark starts.

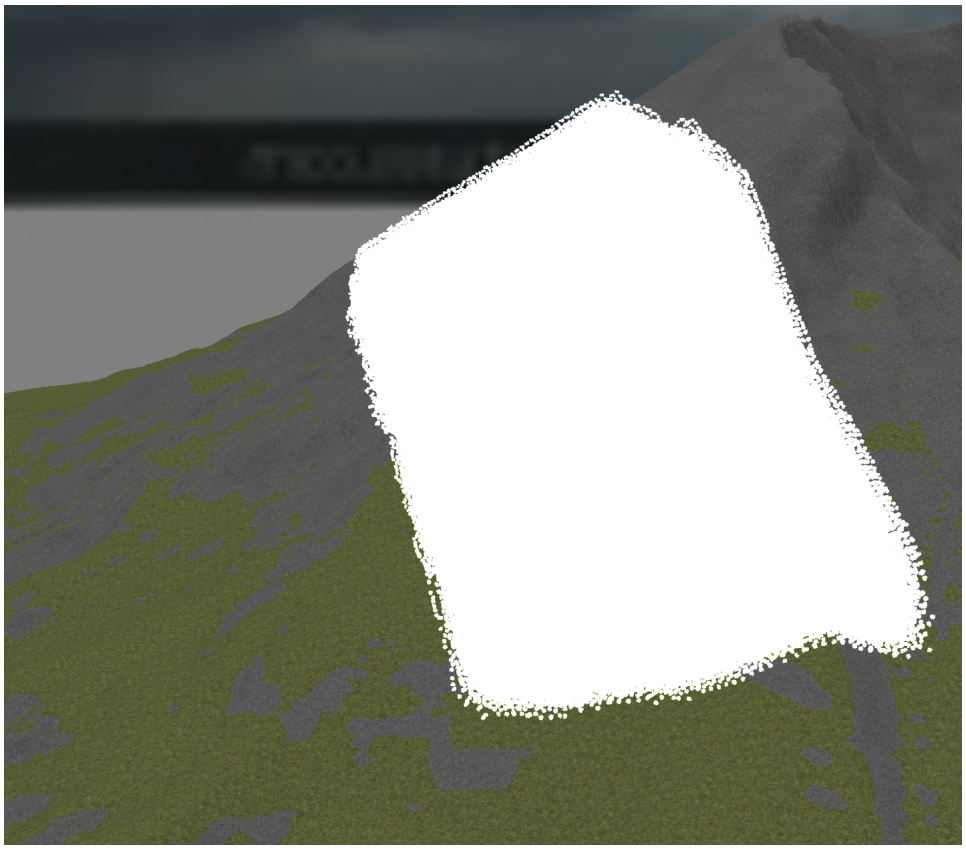


Figure 6.1: The initial snow distribution in the avalanche benchmark with 1 million particles.

We will run the two benchmarks with 1000 iterations 5 times in succession then we take the average. We will be using the Newtonian model and the non-Newtonian model with Binghan fluid rheology since we find this most interesting for a snow avalanche. Figure 6.3 shows the avalanche benchmark after 1000 iterations.

6.2 Benchmark test systems

We ran our benchmarks on the two test systems using the Nvidia RTX 3090 and Nvidia RTX 2080 TI. The benchmark performance was measured in "Iterations per second", meaning how many iterations were completed per second. We divide the total time (in seconds) used, by the iteration count (1000) to find this number. We disabled the rendering while benchmarks were run.

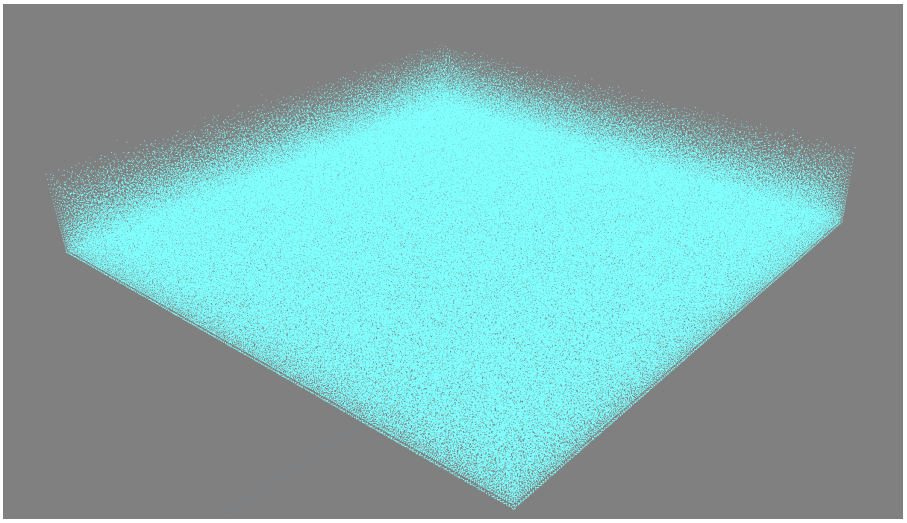


Figure 6.2: Blue pool benchmark on a 150x150x75 (x,z,y) meter pool. Smoothing radius 0.6 meter with 1 million particles.

6.3 The pool benchmark

We ran the pool benchmark for the following particle counts: "256000 (256k), 512000 (512k), 1024000 (1024k), 2048000 (2048k) and 4096000 (4096k)". In Table 6.1 we show the memory usage of the benchmark at various particle counts.

Table 6.1: Pool benchmark GPU memory usage (approximately) at different particle counts with smoothing radius 0.6 meter. Non-Newtonian=NON, Newtonain=NEW

Particle count	**Particle data NEW**	**Particle data NON**	**Grid data**	**Terrain data**	**Sum data NEW**	**Sum data NON**
256k	22MB	33MB	1MB	2MB	25MB	36MB
512k	46MB	69MB	2MB	2MB	50MB	73MB
1024k	92MB	138MB	4MB	2MB	98MB	144MB
2048k	186MB	279MB	8MB	2MB	196MB	289MB
4096k	374MB	561MB	16MB	2MB	392MB	579MB

The results of the benchmark is presented in Figure 6.4. At lower particle counts we get over 500 iterations per second with the RTX 3090 GPU. As the particle count increases the performance decreases rapidly on both test systems. The performance numbers are also shown in Table 6.2. We see that the RTX 2080 TI performs on average 72.1% of the RTX 3090 when the Newtonian model is used. The performance difference is approximately the same with the non-Newtonian model. In a real-time context the RTX 3090 and RTX 2080TI could provide acceptable frame rates (around 30) with a particle count up to 2048k with the Newto-

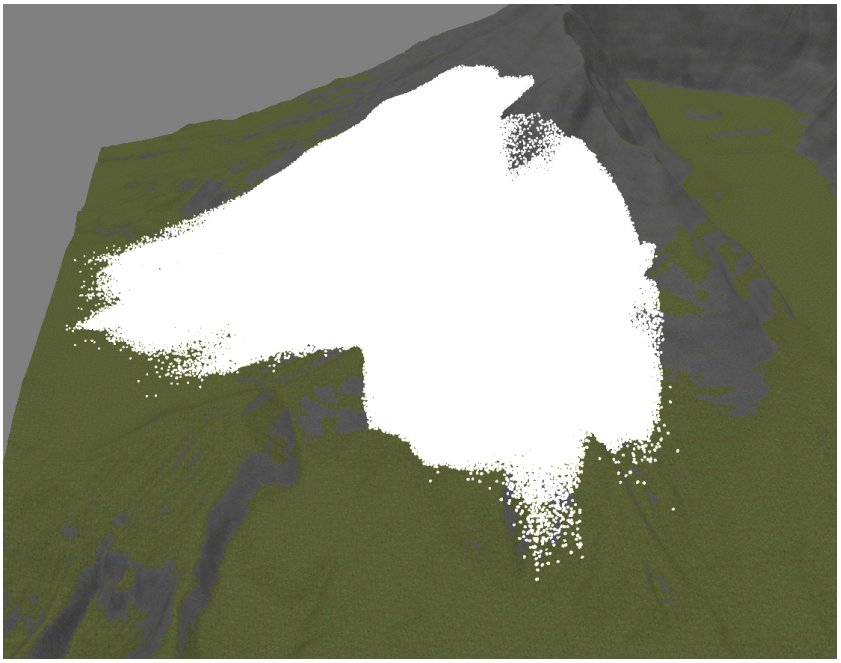


Figure 6.3: Avalanche benchmark with 1 million particles after 1000 iterations with the non-Newtonian model using the Bingham model.

nian model. A particle count around 1024k with the non-Newtonian model could also work.

6.4 The avalanche benchmark

We ran the avalanche benchmark for the following particle counts: "256000 (256k), 512000 (512k), 1024000 (1024k) and 2048000 (2048k)". During testing we found out that at higher than 2048k particle counts the benchmark was unbearably slow so we decided to leave that out. In Table 6.3 we show the memory usage of the benchmark at various particle counts.

The result from the avalanche benchmark is presented in Figure 6.5. We see that the performance degrades quicker when particle count is increased compared to the pool benchmark. For comparing the RTX 3090 Newtonian performance at 256k particles to 512k particles the performance at 512k particles is 0.30 of the performance at 256k. Looking for acceptable real-time performance for this benchmark the RTX 3090 can deliver acceptable interaions per second in between 512k and 1024k particles, and the RTX 2080 TI can deliver acceptable performance around 512k for the Newtonian model. The exact performance numbers can be seen in Table 6.4. The RTX 2080 TI had on average 70% of the RTX 3090 performance in the Newtonian model and 72% of RTX 3090 performance in the non-Newtonian model.

Table 6.2: Pool benchmark performance numbers. Same as Figure 6.4. Given in iterations per second.

Particle count	3090 NEW	2080TI NEW	3090 NON	2080TI NON
256k	646.37	455.51	351.17	258.86
512k	238.77	169.95	132.45	96.56
1024k	93.36	65.55	51.42	37.08
2048k	33.40	26.64	19.36	13.92
4096k	12.64	8.71	7.61	5.44

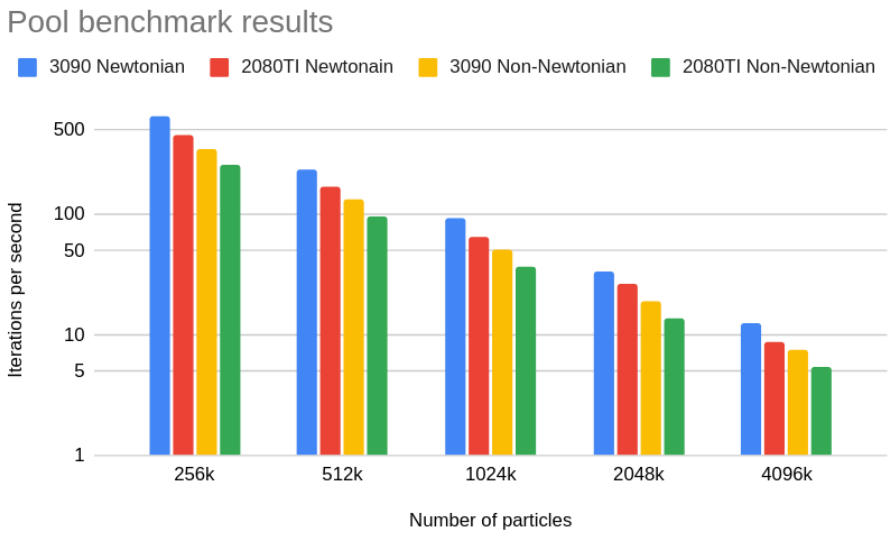


Figure 6.4: Pool benchmark results at various particle counts, measured in iterations per second. The same benchmark is run with the Newtonian and Non-Newtonian model on our testing systems.

Table 6.3: Avalanche benchmark GPU memory usage (approximately) at different particle counts with smoothing radius 0.2 meter. Non-Newtonian=NON, Newtonian=NEW

Particle count	Particle data NEW	Particle data NON	Grid data	Terrain data	Sum data NEW	Sum data NON
256k	22MB	33MB	26MB	19MB	67MB	78MB
512k	46MB	69MB	27MB	19MB	92MB	115MB
1024k	92MB	138MB	29MB	19MB	140MB	186MB
2048k	186MB	279MB	33MB	19MB	238MB	331MB

Table 6.4: Avalanche benchmark performance numbers. Same as Figure 6.5. Given in iterations per second. Non-Newtonian=NON, Newtonain=NEW.

Particle count	3090 NEW	2080TI NEW	3090 NON	2080TI NON
256k	213.52	157.38	108.55	80.50
512k	64.53	45.09	31.22	22.57
1024k	17.34	12.14	8.46	6.10
2048k	4.73	3.29	2.33	1.69

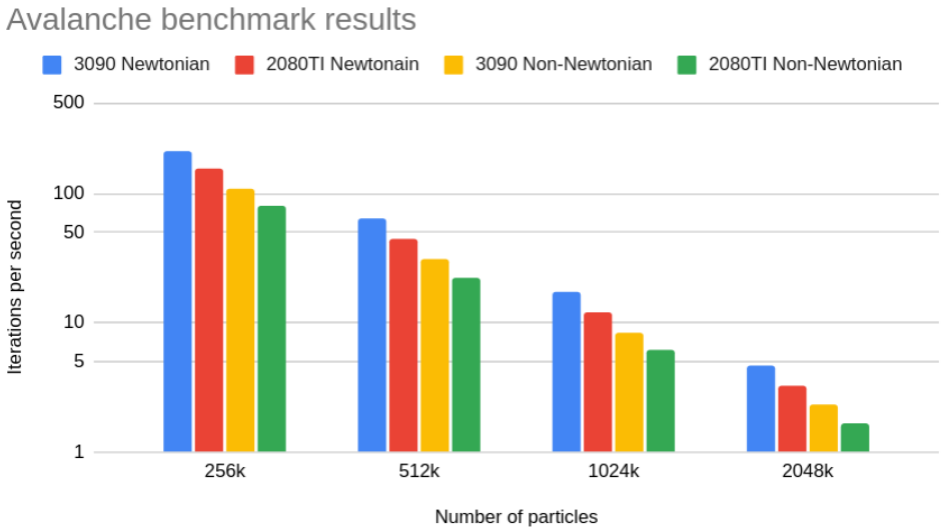


Figure 6.5: Avalanche benchmark results at various particle counts, measured in iterations per second. The same benchmark is run with the Newtonian and Non-Newtonian model on our testing systems.

6.5 Ideal performance vs. actual performance

In our results from the benchmarks we also want to look at how performance scales when particle count is increased. When we double the problem size (example from 256k to 512k particles) the ideal scenario would be that our performance was halved. We set a reference performance at our lowest particle count (256k) and for each doubling of problem size we divide our reference performance by two, then we compare this number to the actual performance at that problem size. For the pool benchmark the ideal vs. actual performance is shown in Table 6.5. The same data for the avalanche benchmark is shown in Table 6.6. From the tables we can see that the performance falls off more quick in the avalanche benchmark. This is expected since the particles are distributed in the same constrained area with a very low mass (low repulsive force between particles). This keeps the neighbour particle count high resulting in many distance searches for

nearby particles and contributions. Ideal performance vs. actual performance can be shown in a graph Figures 6.6 and 6.7.

Table 6.5: Actual performance relative to ideal performance at different particle counts for the pool benchmark. Closer to one is better. Non-Newtonian=NON, Newtonain=NEW.

Particle count	**3090 NEW**	**2080TI NEW**	**3090 NON**	**2080TI NON**
256k	1	1	1	1
512k	0.739	0.746	0.754	0.746
1024k	0.578	0.576	0.586	0.573
2048k	0.413	0.468	0.441	0.430
4096k	0.313	0.306	0.347	0.337

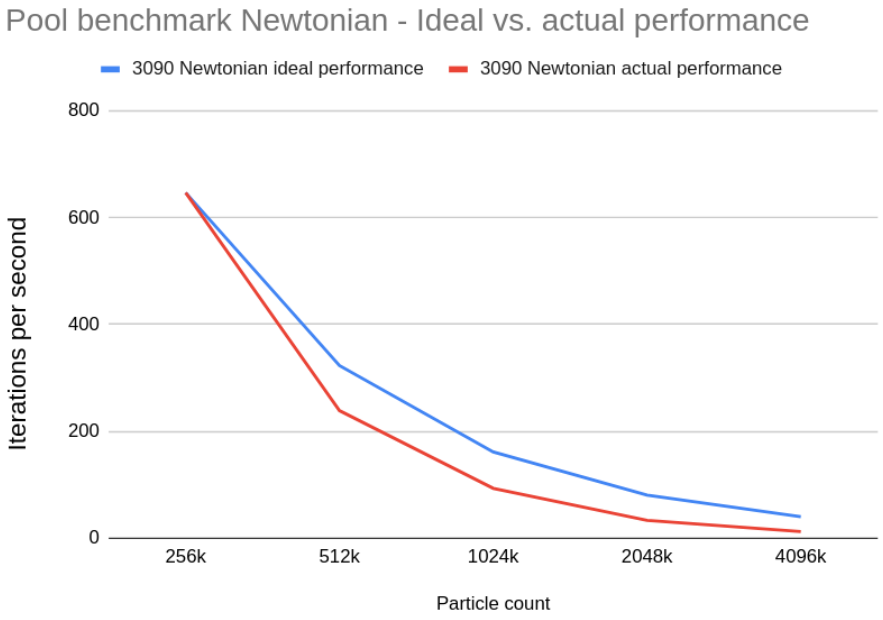


Figure 6.6: Ideal performance vs. actual performance in the pool benchmark using Newtonian model with RTX 3090.

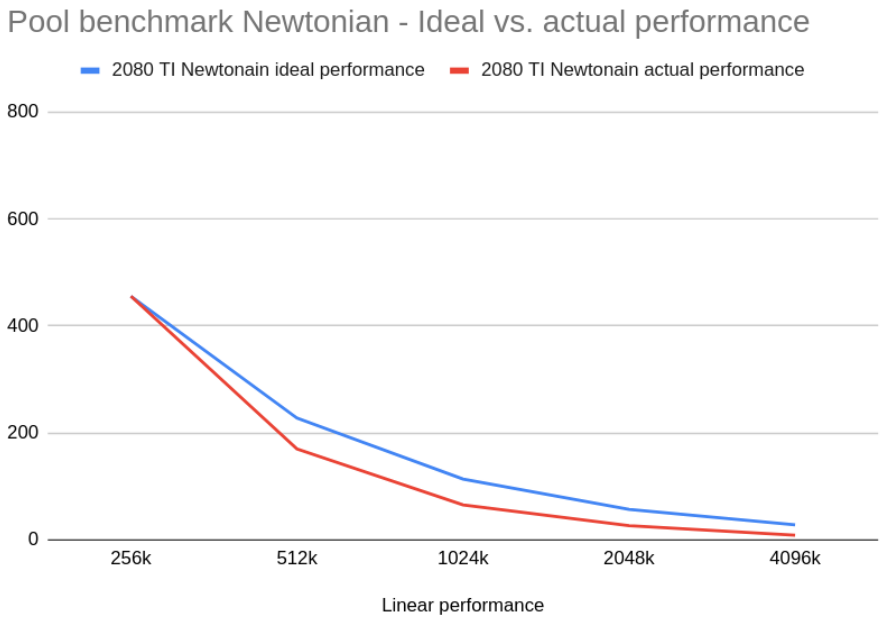


Figure 6.7: Ideal performance vs. actual performance in the pool benchmark using Newtonian model with RTX 2080 TI.

Table 6.6: Actual performance relative to ideal performance at different particle counts for the avalanche benchmark. Closer to one is better. Non-Newtonian=NON, Newtonain=NEW.

Particle count	3090 NEW	2080TI NEW	3090 NON	2080TI NON
256k	1	1	1	1
512k	0.604	0.573	0.575	0.561
1024k	0.325	0.309	0.312	0.303
2048k	0.177	0.168	0.172	0.168

6.6 Additional visual results

Our implementation can also generate graphics that illustrate using color-coded particles which of the particles are above the yield stress threshold using the Bingham model. As shown in Figure 6.8, yellow particles are flowing with reduced viscous stress while the white particles have a high viscosity.
The Figure 6.9 shows the flow of a Newtonian fluid traveling from the valley on the left to the end of the simulation domain.
The Figure 6.10 shows a issue that occurs when particles are push towards a skewed wall such as a hillside. The boundary force is then applied and the particles are pushed upwards.

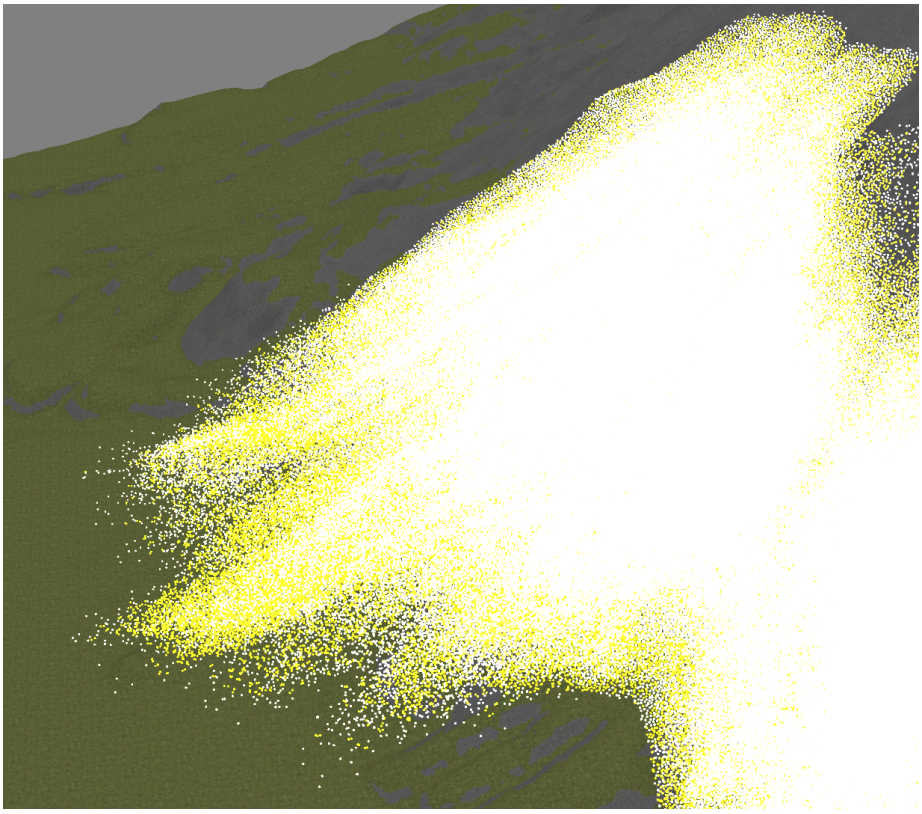


Figure 6.8: Color-coded particles that are above the yield stress threshold using the Bingham model. Yellow particles are flowing with reduced viscous stress while the white particles have a high viscosity.

The Figure 6.11 show the initial snow distribution in the Avalanche preset.

The Figure 6.12 show the avalanche after a period of time.
The Figure 6.13 shows the final snow distribution when the avalanche have reached the ground.
The Figure 6.14 shows the places with high pressure (yellow) during the avalanche.

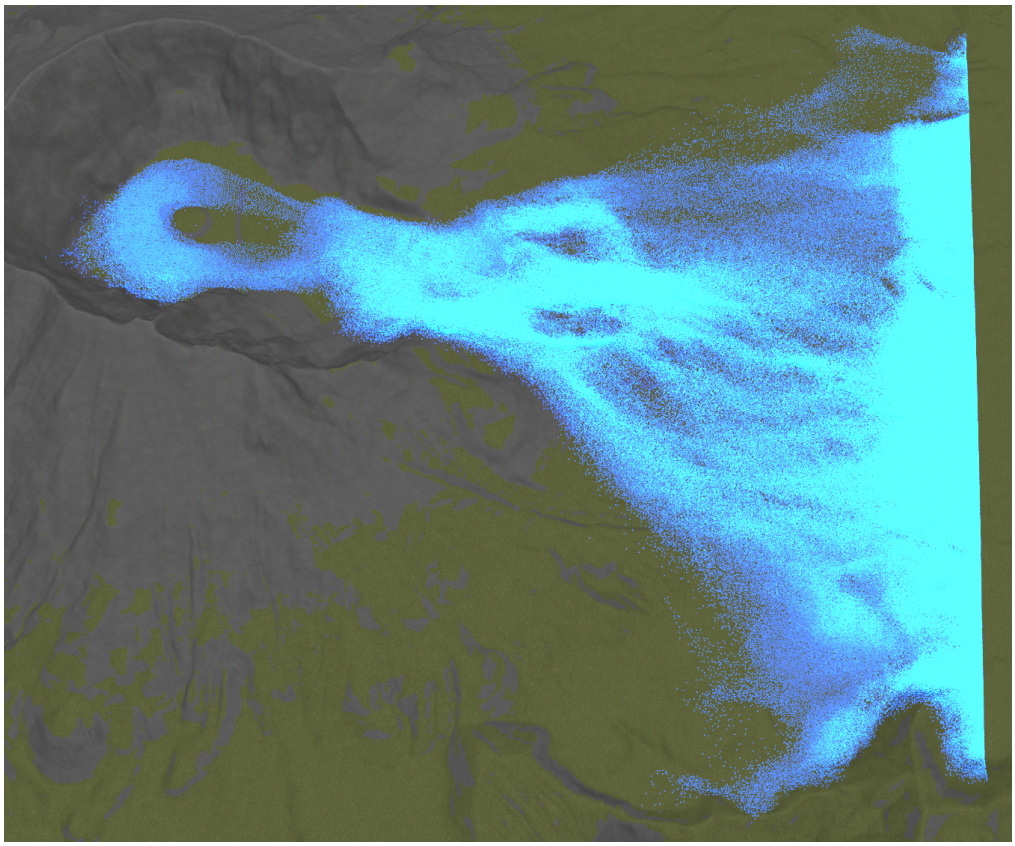


Figure 6.9: River of blue particles that started in the valley on the left.

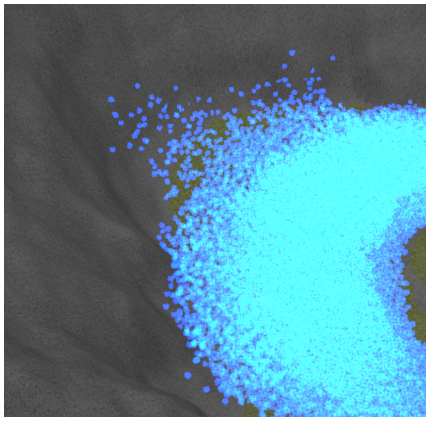


Figure 6.10: Particles climbing up a wall when the terrain normal is not perfectly perpendicular to the ground.

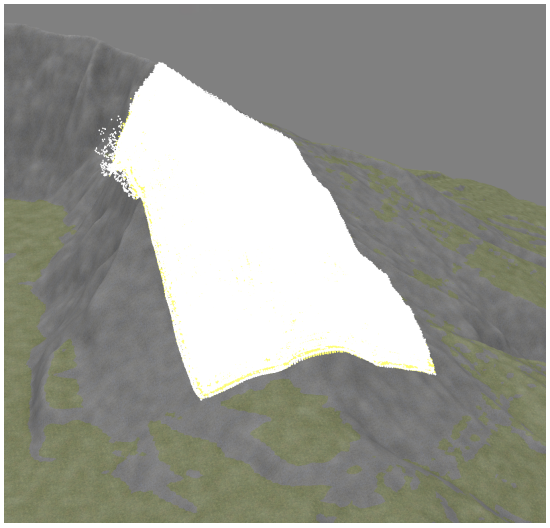


Figure 6.11: Initial snow distribution in the avalanche preset.

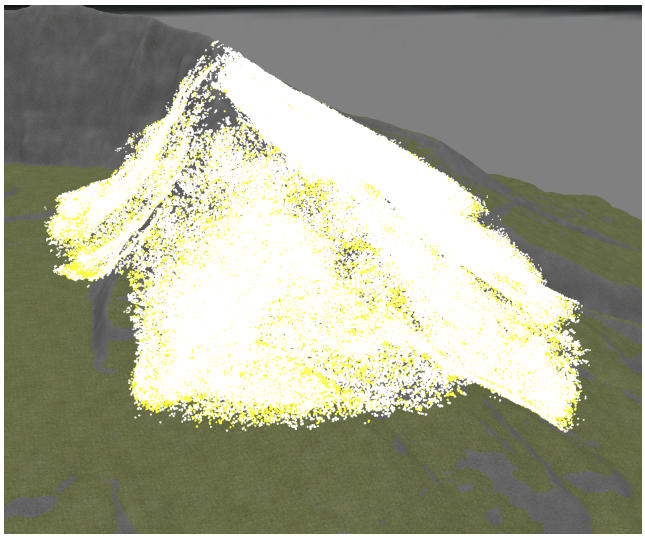


Figure 6.12: The avalanche in progress the colors indicate the particles that are flowing in a Newtonian manner.

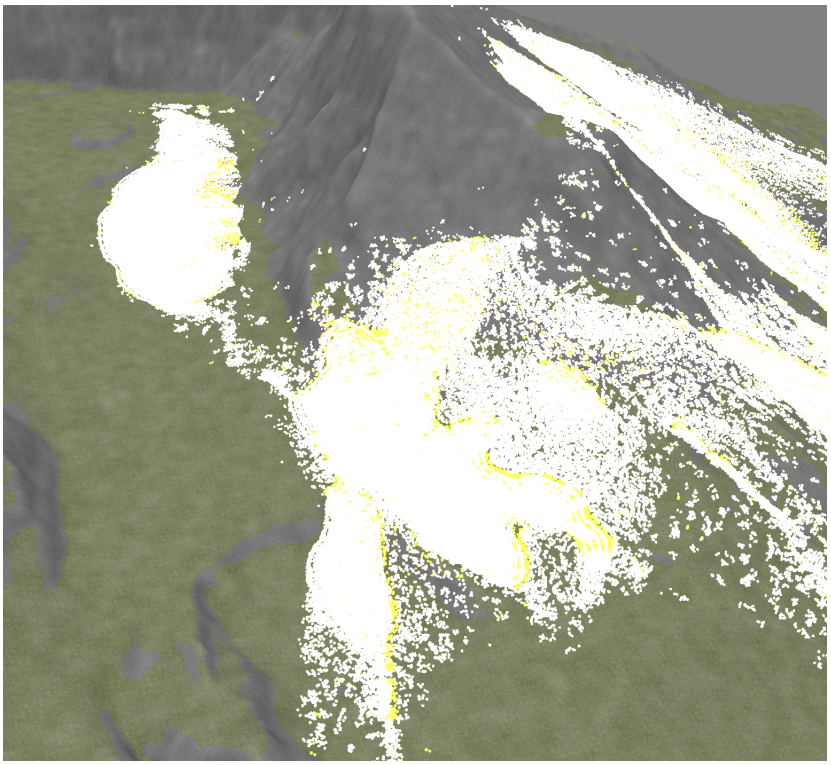


Figure 6.13: Shows when the avalanche has almost reached the ground.

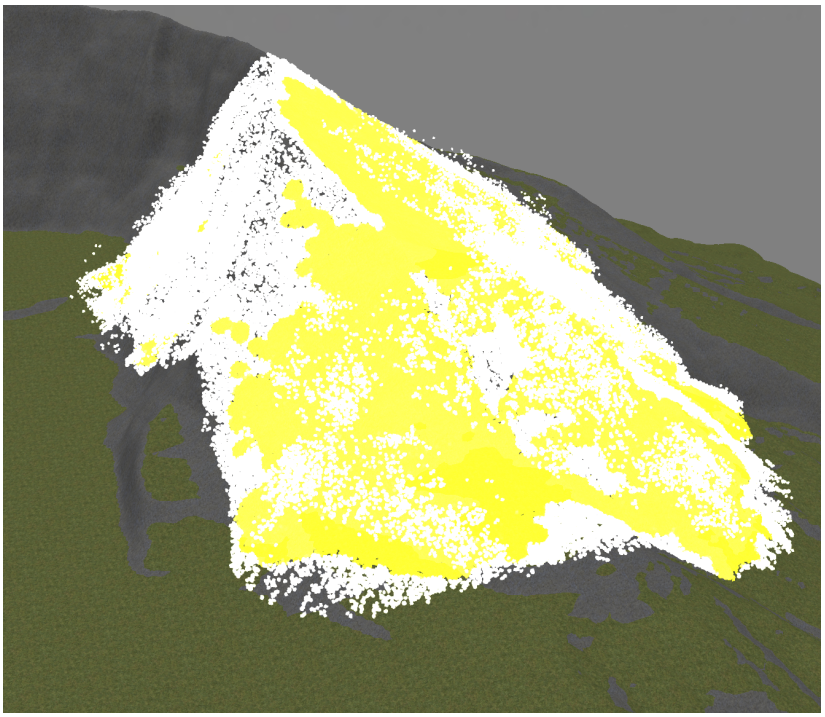


Figure 6.14: Figure shows points of high pressure during the avalanche.

Chapter 7

Conclusions and Future Work

With more extreme weather and weather-related events on the horizon it will become more and more important to be able to simulation snow and snow-based avalanches accurately and quickly.
Our original intention was to use a fully compressible simulations based on Smoothed Particle Hydrodynamics (SPH). However, we had to make a choice during our project to drop implementing the fully compressible version which proved to be too challenging given the 20-week time-constraints imposed by NTNU for masters theses, we instead opted to use the methods that were similar to what Ø. E. Krog used in his 2010 masters thesis [1]. We thus developed a novel particle simulation implementation using both a weakly compressible non-Newtonian methods and a Newtonian fluid method. We then integrated these implementation into the HPC-Lab snow simulator.

We also integrated a high-performance neighbour search framework[9] into the HPC-Lab snow simulator.

Other goals accomplished include making the simulator user friendly with a GUI and support for changing parameters and creating different presets for the simulation using the new ImgGUI-based interface provided by Roger Holten. We also created OpenGL buffers for particle positions and intensity values and mapped and un-mapped OpenGL buffers to a device pointers for use in device code, both which should reduce memory copies. We were also able to implement a simulation scaling system and a terrain modeling with adjustable resolution based on height maps that the HPC-Lab snow simulator uses. The result is a user friendly SPH component in the HPC-Lab snow simulator.

We also utilized new tools such as Nvidia Nsight and our own performance benchmarks to find bottlenecks and optimize our code. Tests results of our implementations on the newer consumer GPU's from Nvidia such as the RTX 2080Ti and RTX 3090 including associated benchmark results were also included achieving 33 iterations per second with 2 million particles on the RTX 3090 and 26 iterations

per second on the RTX 2080TI in the pool benchmark with the Newtonian solver.

Finally, we also included several visual results illustrating the GUI and snow dynamics we can now simulate avalanches and fluid flow in interesting ways.

7.1 Future work

There are several directions that can be taken to build on our work. Here are some of our main ideas:

7.1.1 Optimization

There is definitely potential to optimize the algorithm further. Here are some ideas:

- We saw that our main bottleneck was when conducting distance calculations to neighbours in the physics and density kernels.
- In the non-Newtonian implementation there is also the additional calculations and memory usage with the calculation of the stress tensor and summation of stress forces.
- One could possibly find a way to use for example shared memory or texture memory to make the memory reads quicker (for position data), however one might also reach a point where the compute is the limiting factor. We think this is something that should be investigated further.
- On newer hardware such as the RTX 3090 (in compute capability 8.6 and 8.0) there exists a feature called "L2 Persisting Accesses" (Section 3.2.3 in the CUDA programming guide)[15] where frequently used data can be permanently stored in L2.
- On the professional Nvidia cards such as A100 L2 cache is 40MB (compute capability 8.0) one could possibly store particle positions here to make the distance calculation faster.

7.1.2 Rendering and visuals

In our work we used some existing snow shaders already present the HPC-Lab snow simualtor. We have modified these slightly to fit with our data and scaling system. However we think that the visuals could be improved by rendering them more like a fluid volume instead of rendering them as individual particles.

7.1.3 Features

We were intentionally planning to make a recording system for the SPH simulation that could record particle positions in a file and then "play back" the simulation. This could be a interesting feature to use for recording demanding simulations and then play them back in real time when its finished. Other features that could be

interesting is particle based borders and particle based objects (from 3D models) in the simulation.

Bibliography

[1] Ø. E. Krog, "Gpu-based real-time snow avalanche simulations," M.S. thesis, Dept. of Computer, Information Science, Norwegian University of Science, and Technology (NTNU), Trondheim, Norway, 2010. [Online]. Available: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/252305.

[2] I. Saltvik, "Parallel methods for real-time visualization of snow," M.S. thesis, Dept. of Computer, Information Science, Norwegian University of Science, and Technology (NTNU), Trondheim, Norway, 2006. [Online]. Available: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/251490.

[3] R. Eidissen, "Utilizing gpus for real-time visualization of snow," M.S. thesis, Dept. of Computer, Information Science, Norwegian University of Science, and Technology (NTNU), Trondheim, Norway, 2009. [Online]. Available: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/251316.

[4] NVIDIA, *Anne elster of norwegian university of science and technology (ntnu) talks about cuda*, Includes interview with Robin Eidissen and the HPC-Lab Snow Simulator. [Online]. Available: https://www.youtube.com/watch?v=XWIFSvXqvEo.

[5] Ø. E. Krog and A. C. Elster, "Fast gpu-based fluid simulations using sph," 2010.

[6] M. Müller, D. Charypar, and M. Gross, "Particle-based fluid simulation for interactive applications," 2003.

[7] S. Hosseini, M. Manzari, and S. Hannani, "A fully explicit three-step sph algorithm for simulation of non-newtonian fluid flow," 2007.

[8] O. Cornut, *Imgui*. [Online]. Available: https://github.com/ocornut/imgui (visited on 05/22/2021).

[9] *Fluids v.3 website*. [Online]. Available: http://www.fluids3.com/ (visited on 05/30/2021).

[10] A. R. Brodtkorb, T. R. Hagen, and M. L. Sætra, "Graphics processing unit (gpu) programming strategies and trends in gpu computing," *Journal of Parallel and Distributed Computing*, vol. 73, no. 1, pp. 4–13, 2013, Metaheuristics on GPUs, ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2012.04.003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731512000998.

[11] M. J. Flynn, "Very high-speed computing systems," *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966. DOI: 10.1109/PROC.1966.5273.

[12] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10, Saint-Malo, France: Association for Computing Machinery, 2010, pp. 451–460, ISBN: 9781450300537. DOI: 10.1145/1815961.1816021. [Online]. Available: https://doi.org/10.1145/1815961.1816021.

[13] D. B. Kirk and W.-m. W. Hwu, *Programming massively parallel processors - A hands-on approach*. Burlington, Massachusetts: Morgan Kaufmann Publishers, 1993, ch. 1.

[14] Nvidia, "Nvidia ampere ga102 gpu architecture - the ultimate play," 2020. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf.

[15] Nvidia, *CUDA C++ Programming Guide - Design guide*. Santa Clara, California, USA: Nvidia, digital book, 2021, ch. 1. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[16] T. M. Schmid, "Real-time snow simulation - integrating weather data and cloud rendering," M.S. thesis, Dept. of Computer, Information Science, Norwegian University of Science, and Technology (NTNU), Trondheim, Norway, 2016. [Online]. Available: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2418700.

[17] D. C. Elster, *Hpc-lab*. [Online]. Available: https://www.ntnu.edu/idi/hpc-lab/ (visited on 03/24/2021).

[18] F. M. J. Vestre, "Enhancing and porting the hpc-lab snow simulator to opencl on mobile platforms," M.S. thesis, Dept. of Computer, Information Science, Norwegian University of Science, and Technology (NTNU), Trondheim, Norway, 2012. [Online]. Available: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/253004.

[19] A. Nordahl, "Enhancing the hpc-lab snow simulator with more realistic terrains and other interactive features," M.S. thesis, Dept. of Computer, Information Science, Norwegian University of Science, and Technology (NTNU), Trondheim, Norway, 2013. [Online]. Available: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/253357.

[20] L. U. Arenson, W. Colgan, and H. P. Marshall, *Physical, Thermal, and Mechanical Properties of Snow, Ice, and Permafrost*. Cambridge, Massachusetts, USA: Academic Press, 2015, ch. 2.

[21] W. S. B. Paterson, *The Physics of Glaciers*. Amsterdam, Netherlands: Elsevier, 1994, ch. 1.

[22] T. E. o. E. Britannica, *Pressure*. [Online]. Available: `https://www.britannica.com/science/pressure` (visited on 05/05/2021).

[23] K. J., D. A., and K. P., *Non-Newtonian Fluids: An Introduction: Rheology of Complex Fluids*. New York, USA: Springer, 2010, ch. 1.

[24] J. Tu, G. H. Yeoh, and C. Liu, *Computational Fluid Dynamics: Governing Equations for CFD—Fundamentals*. Oxford, United Kingdom: Butterworth-Heinemann, 2008, ch. 2.

[25] R. A. Gingold and J. J. Monaghan, "Smoothed particle hydrodynamics: Theory and application to non-spherical stars," 1977.

[26] J. J. Monaghan, "Smoothed particle hydrodynamics: Reports on progress in physics," 2005.

[27] J. J. Monaghan, "Smoothed particle hydrodynamics," 1992.

[28] M. Kelager, "Lagrangian fluid dynamics using smoothed particle hydrodynamics," M.S. thesis, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark, 2006.

[29] Nvidia, "Cuda c++ programming guide," 2020. [Online]. Available: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[30] R. K. Hoetzlein and N. Grapics Devtech, *Fast fixed-radius nearest neighbors: Interactive million-particle fluids*. [Online]. Available: `https://on-demand.gputechconf.com/gtc/2014/presentations/S4117-fast-fixed-radius-nearest-neighbor-gpu.pdf` (visited on 05/30/2021).

[31] P. Young, *The leapfrog method and other "symplectic" algorithms for integrating newton's laws of motion*, Santa Cruz ,United States. [Online]. Available: `http://physics.ucsc.edu/~peter/242/leapfrog.pdf` (visited on 05/20/2021).

[32] T. Amada, "Real-time particle-based fluid simulation with rigid body interaction," in *Game Programming Gems 6*, M. Dickheiser, Ed., Charles River Media, 2006, pp. 189–205.

[33] Nvidia, "Nvidia nsight compute," [Online]. Available: `https://developer.nvidia.com/nsight-compute`.

[34] M. Harris, "How to access global memory efficiently in cuda c/c++ kernels," [Online]. Available: `https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/`.

Appendix A

GPU optimization and neighbour search for SPH

An investigation of smoothed particle hydrodynamics, neighbour search and GPU optimization

Ivar Andreas Helgestad Sandvik
NTNU
iasandvi@stud.ntnu.no

***Abstract*— When implementing complex spacial numerical algorithms on a graphics processing unit (GPU), there are many challenges mainly branching and locality of data. The implementation needs to utilize the GPU resources in a efficient manner to avoid stalling. In particle based simulation algorithms such as SPH there is also a need for fast neighbour search to simulate interactions between bodies.**

I. INTRODUCTION

With the evolution of high performance general purpose graphics processors in recent years new applications for this hardware has emerged[6]. The GPU is a massively parallel processor that can run thousands of threads simultaneously compared to the limited amount of parallelism in a CPU. The difference is clear in the execution method. A typical CPU can execute for example only a set number of instructions per clock depending on the number of cores available. A GPU on the other hand can execute thousands of instructions per clock cycle. The drawback for GPU's is that groups of ALU's (arithmetic logic unit) and FPU (floating point unit) that can only execute same instruction at a given time, the GPU is a SIMD processor (single instruction multiple data). This poses some challenges when implementing neighbour search and SPH as we will discuss later.

Why do we need fast neighbour search in particle based simulations? In a simulation domain you want to simulate the interaction between bodies. Examples of this are pressure and collision interactions. In order to do this you need to know the distance between particles in the simulation. An naive approach to this is to calculate the distance between all particles and then determine which bodies affect each other based on the distance. An simple approach is to find the euclidean distance between all bodies. The problem with this approach is that the time complexity is $O(n^2)$ for n particles. For large scale simulations this will be too inefficient. There exist algorithms to limit the number amount of distance computations such as the verlet list and linked-cell algorithms[3]. These algorithms are aimed at limiting the search scope around a given particle. In practice this means to avoiding to check the distance between particles that are far apart without knowing the exact distance. These algorithms use supporting data structures that needs to be updated regularly depending on the implementation.

Smoothed particle hydrodynamics is an particle based simulation method that is frequently used for computational fluid dynamics. The SPH method is a discrete method for simulating fluid flow according to the Newtonian equations[5]. SPH makes heavy use of weighted summation of surrounding particles to determine density and forces acting on a given particle. In order to find weighted sums a kernel function is used. This is often referred to as a smoothing kernel $W(r, h)$ where r is the distance between two particles and h is the cut-off distance[5]. Particles with a distance greater than the cut-off distance are not considered in the weighted sums. Because of the nature of the SPH method cut-off distance we can exploit algorithms which lowers the search scope for surrounding particles.

II. GRAPHICS PROCESSING UNIT (GPU) PROGRAMMING STRATEGIES AND TRENDS IN GPU COMPUTING

In recent times the GPU's have taken the performance crown over traditional CPU's. Graphics processing units today are often referred to as general purpose graphics processing units (GP-GPU). However in the early 2000's GPU's where more restricted. Back then the GPU had a limited set of fixed operations that were purely meant for graphics processing[6]. As a result of this the hardware was difficult to use for other applications such as compute. Some third-party languages where developed to make the hardware work for compute workloads, but soon abandoned after the introduction of CUDA and OpenCL. These languages in combination with GPU hardware innovations made it easier to compile compute programs for GPU's. However graphics processing units were not always faster than traditional CPU's. Around 2002-2004 the GPU caught up with CPU performance in floating point operations and available memory bandwidth and has been outgrowing the CPU performance ever since[6]. The challenge today is to utilize the GPU in a efficient manner.

A. GPU computation

There are two major GPU manufactures today in the PC-market. Advanced micro devices (AMD) and Nvidia, in this paper we will focus on Nvidia hardware. A GPU consists of a core type called streaming multiprocessor (SM), Nvidia refers to this as a group of CUDA cores. A GPU

has a number of these CUDA cores. Each of them can execute one or two instructions (3000-series) at a given time on multiple data. For example the Nvidia GA10x (3000-series) GPU can execute 32FP32 instructions or 16FP32 and 16INT32 operations in one clock cycle for example[8]. The execution model on the GPU is based on a compute grid, blocks and warps. A grid is an overarching structure that encapsulates blocks in 3 dimensions. The blocks are 3 dimensional containers that encapsulates the warps. The warps inside the same block have access to shared memory between each other. The warps are groups of threads that are executed together on a CUDA core.

B. Nvidia GPU scheduling

In order to keep the GPU occupancy on a hardware level as high as possible Nvidia has implemented advanced scheduling on different levels[7]. The scheduler hierarchy consists of the application, stream, thread block, and the warp-scheduler. In this context the application scheduler is not relevant.

1) The stream scheduler: The stream scheduler enables the execution on multiple CUDA kernels concurrently in the same application. This feature was introduced in the Maxwell architecture (2014). Streams can have two priority levels (high and low). In example if two low priority kernels are running, and a high priority stream is added the execution of the low priority kernels may stall. In order to use streams efficiently memory transfers between host and device, and kernel launch should be invoked asynchronously.

2) Thread block scheduler: The thread block scheduler is responsible for mapping blocks to SM (streaming multiprocessor). The thread block scheduler checks the occupancy level of the SM's before assigning a block. The scheduler uses the following information about the kernel in order to determine the SM that will be assigned to that block.[7]:

- The number of threads/warps per thread block
- Shared memory per thread block
- Number of registers per warp

The main goal of the scheduler is to keep the utilization of each SM as high as possible, meaning that an instruction can be executed for every clock cycle on the CUDA cores inside that SM.

3) The warp scheduler: Each SM in in the Nvidia GPU contains a set number of warp schedulers. The warp schedulers task is to to assign warps of threads to the individual CUDA cores inside the SM. Warps can be in two different states, ready and not ready to execute depending on if the data for that warp is ready. The warp scheduler can stall warps that are not ready and continue to execute ready warps. If the SM has enough ready warps at a given time it can effectively hide memory access stalls while waiting for other warps to get ready. A notable improvement with the Volta architecture is that each CUDA core has two pipelines that can execute two warps at the same time given one is FP32 and the other one executing INT32 operations. In the Ampere architecture the INT32 pipeline is also capable of FP32[8].

C. Programming strategies

As mentioned in the previous section the warp scheduler is dependent on having many available thread groups queued for execution. A GPU can execute a given number of threads on every clock cycle. The ratio between current thread execution number and the maximum supported threads is referred to as occupancy. A 100 percent occupancy means that the warp schedulers in all the SM are able to queue instructions every clock on the CUDA cores.

1) Branching: Branching can affect performance significantly since all the threads in the warp executes the same instruction at a given time. An example of this is if a thread goes on a different branch than the others. The CUDA core then has to execute both of the branches, resulting in a huge slow down in that section (example 32 times slower for a 32-thread warp)[6]. One way to avoid branching is to build the warps in such a way that you execute the same branch on each warp. You can also use the CPU to perform the branching before and then select the correct kernel according to the branch, but this is most applicable when there are many parameters in the if condition.

2) GPU memory optimization: Nvidia GPU's have several different memory types. To achieve the best performance a programmer should be aware of the memory structure and utilize the functionality that the different memory types offer[9].

- Global memory (R/W), the global memory is the RAM of the GPU. This memory is large and but slow compared to the caches.
- Shared memory (R/W), this is a memory pool shared between all executing thread blocks. This memory is very fast approximately 100 times faster than global memory. Each SM gets a portion of this memory.
- Constant memory (R), this is a read-only memory stored in global memory, but parts of this memory can be stored in a separate cache in the SM.
- Texture memory (R) is primarily meant for textures, but in some advanced scenarios it can be utilized as constant memory. This memory is stored in the global memory, but it has a cache available that can potentially make access faster than global memory.
- Registers (R/W) is fastest cache in the GPU, its managed by the compiler and stores kernel variables for the different threads.

III. Fast neightbour search: Verlet list and linked-cell algorithms

In most SPH simulations we operate with a fixed smoothing radius h. The smoothing radius determines which particles in the simulation domain that can affect the current particle i. The fixed radius opens opportunities to utilize verlet list and linked-cell algorithms to find neighbours inside the smoothing radius.

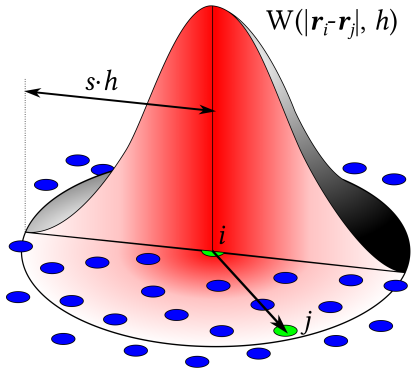


Fig. 1. Illustration: SPH Gaussian kernel, Wikipedia 2020

A. Verlet list algorithm

The verlet list algorithm is a method for reducing the amount of neighbour searches. In this method each particle i stores two or one list of nearby neighbouring particles j. There are two cut-off distances used. The first one R_h which is equal to the smoothing radius of the SPH kernel. The second distance R_p is larger than R_h by a certain number of units. The idea of the algorithm is to use the radius R_p as a buffer for particles that might move into the cut-off distance R_h after time integration in the simulation. Each

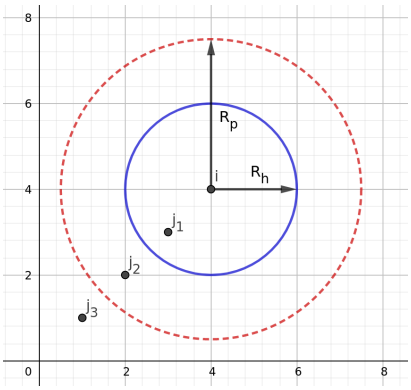


Fig. 2. Verlet list search radiuses illustrated. The red dotted circle represents the buffer radius R_p and the solid blue circle R_h represent the smoothing radius. The center of the circles is the particle i. There are also some surrounding particles j_x in different areas.

particle in the simulation will store lists of the other particles that are encapsulated by the circles. This structure can allow the simulation to continue for several time steps until any particle (such as j_3) outside R_p could potentially reach the inner radius R_h[10]. When this happens the verlet list needs to be updated. In the update or the construction of the verlet lists for all particles a $O(n^2)$ distance search needs to be executed. In order to avoid updating the list too often the R_p radius can be increased, but this increases the number of particles in the list.

B. Linked cell algorithm

In the linked cell algorithm the simulation domain is partitioned into cells with a uniform length, height, depth[10]. The size of the cell in n-dimensions is set to the smoothing radius of the SPH simulation. The idea is that around a given particle you only have to check direct neighbouring cells and the current cell for possible influence on particle J_i. From

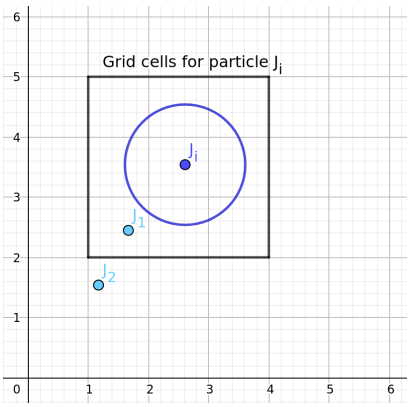


Fig. 3. The grid cell structure illustrated in 2D. The point J_i represents the current particle. The blue circle represents the smoothing radius. J_x illustrates other particles in the system.

the figure we see that a given particle will have a maximum of 8 neighbouring cells. This limits the search scope. In 3D the number of neighbouring cells is 26. For each time step in the simulation particles need to be moved to the correct cell based on their position in the simulation domain[3].

C. Comparison on CPU

In the publication[3] the performance of the two algorithms are evaluated. There are several benchmarks used to evaluate the algorithms. The first comparison is with different number of particles in the system. The performance is within margin of error when the particle count is low. When the particle count exceeds 2000 in the simulation the verlet list performs worse than the cell list algorithm. We can reasonably assume that this is due to the huge memory consumption and utilization when storing the lists of neighbouring particles for each particle.

Another test conducted in[3] is to increase the smoothing radius in the simulation. For the grid cell algorithm this means that the cell size is bigger and fewer cells in the simulation domain. In the verlet list algorithm this means that the buffer radius and smoothing radius is increased. Increased radius resulted in much worse performance for the grid cell algorithm compared to the verlet list implementation. This is expected since when the grid expands it covers more of the simulation domain and in

worst case it results in a $O(n^2)$ brute force search plus the overhead of the data structure. The verlet list algorithm does not suffer in the same degree when increasing the radius. The conclusion from[3] is that the grid cell algorithm performance degrades in a slower manner as the particle count increases and is therefore a better choice for SPH simulation on the CPU.

D. Comparison on GPU

In publication[1] the performance of the two algorithms is evaluated running a SPH simulation on the GPU. In the linked cell implementation an optimization has been done in the way particles in a given cell are stored. The particles have a reference to the next particle, and each cell stores the reference to the first particle in that cell and the last particle in that cell. However the particles must be mapped in the correct order according to their cell for this to work. This requires a sorting of the particles for each time integration. This is a similar approach as in[2]. The verlet list implementation is the same as in the CPU implementation. However the article[1] mentions that there is a possibility to dynamically adjust the search radius (Fig.2) based on the maximum possible displacement of particles to get a even time interval between verlet list updates.

The article[1] suggest that the verlet list implementation is faster on the GPU than linked cell when the particle count is lower than 10^5. The memory consumption is about 18 times higher for the verlet list compared to linked cell. Both solutions suffer in when accessing the global memory. The verlet list implementations suffers a lot because of the non-coalesced memory access, since the threads in the warp might have different neighbors. The same goes for the linked cell, in 3D simulations each grid cell might have more than 32 threads in that cell, this can result in warps that partly belong to different cells, resulting in non-coalesced memory access when checking neighbouring cells. The conclusion is that the best approach for neighbouring search is the linked cell algorithm for SPH simulations on GPU[1].

REFERENCES

[1] Daniel Winkler, Massoud Rezavand, and Wolfgang Rauch. *Neighbour lists for smoothed particle hydrodynamics on GPUs*. Unit of Environmental Engineering, University of Innsbruck, Austria, 2017.

[2] Øystein E. Krog and Anne C. Elster. *Fast GPU-based Fluid Simulations Using SPH*. Dept. of Computer and Information Science, Norwegian University of Science and Technology (NTNU), Trondheim, 2010.

[3] Wan-Qing Li, Tang Ying, Wan Jian and Dong-Jin Yu. *Comparison research on the neighbor list algorithms: Verlet table and linked-cell*. Grid and Service Computing Lab, Hangzhou Dianzi University, Hangzhou, 2010.

[4] Sara S. Baghsork, Isaac Gelad, Matthieu Delahay, Wen-mei W. Hwu. *Efficient Performance Evaluation of Memory Hierarchy for Highly Multithreaded Graphics Processors*. Urbana-Champaign, University of Illinois, Illinois, 2012.

[5] R. A. Gingold and J. J. Monaghan *Smoothed particle hydrodynamics: theory and application to non-spherical stars*. Institute of Astronomy, Cambridge, United kingdom, 1977.

[6] André R. Brodtkorb, Trond R. Hagen and Martin L. Sætra *Graphics processing unit (GPU) programming strategies and trends in GPU computing*. Department of Applied Mathematics, Blindern, Oslo, 2012.

[7] Ignacio Sanudo Olmedo, Nicola Capodieci, Jorge Luis Martinez, Andrea Marongiu and Marko Bertogna *Dissecting the CUDA scheduling hierarchy: a Performance and Predictability Perspective*. University of Modena and Reggio Emilia, Modena, Italy, 2020.

[8] Nvidia *NVIDIA AMPERE GA102 GPU ARCHITECTURE - THE ULTIMATE PLAY*. 2020, https://www.nvidia.com/content/dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf

[9] Nvidia *CUDA C++ Programming Guide*. 2020, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[10] Giacomo Viccione, Eugenio Pugliese Carratelli and Vittorio Bovolin *A fast neighbour-search algorithm for free surface flow simulations using SPH*. Dipartimento di Ingegneria Civile, Università degli Studi di Salerno, Italy, 2007.

Ivar Andreas Helgestad Sandvik

SPH-based Avalanche Simulations on GPU

NTNU
Norwegian University of
Science and Technology