UNIVERSITÄT SIEGEN
Naturwissenschaftlich-Technische Fakultät
Lehrstuhl für Computergraphik und Multimediasysteme

Framework zur effektiven Umsetzung von Fluidsimulationen auf GPUs

Masterarbeit im Studiengang Informatik

Rene Winchenbach
Elisabethstraße 7
57072, Siegen
Matrikelnummer 928519

Erstgutachter: Prof. Dr. Andreas Kolb
Zweitgutachter: Dr.-Ing. Martin Lambers

Siegen, November 2018

Abstract

Simulation methods in the past have been used for a variety of purposes, from the simulation of engineering problems over biological systems to the simulation of galaxies. Many of these require significant computational power which makes them ideal candidates for massively parallel computing using GPUs. The ever increasing usage of these simulations, combined with the increasing demand for physically realistic simulations, even in computer graphics, places a large burden on researchers as they not only have to be on the bleeding edge of science but also keep up with the technological advances.

In computer graphics many different standard frameworks exist for applications which range from video game engines like Unreal Engine to procedural rendering and animation programs like Houdini. Using such a framework for research however is impractical as these systems place restrictions on what can be done and limit the freedom of the user. Similarly many frameworks used for simulation, specifically of SPH effects, exist, e.g. DualSPHysics. But these often are either created in a very ad-hoc fashion resulting in bloated and difficult to use code, or optimized to a point where they are not practical in research applications.

The goal of this thesis is to develop a new and versatile framework which allows for the easy development of new SPH simulation methods by providing powerful abstractions to the user that significantly reduce the complexity of the code that has to be written. In order to achieve this, many challenges have to be addressed, e.g. how memory is managed or how mathematical operations are implemented. All of these features should be implemented in a way that still allows the user to, if desired, write any arbitrary code without having to consider the framework the code is being written in.

Additional requirements include the ability of the framework to run on both Windows and Linux machines, as well as providing both CPU and GPU execution. Whilst the performance of GPUs in simulations is significantly higher than that of CPUs, CPUs still provide a valuable tool for research, especially when it comes to debugging and verification of implementations, as CPU based implementations have significantly less parallel effects to deal with.

In order to support physical plausibility the framework should also provide a way to check if the code being written is physically consistent based on the SI-units of the involved operands. This checking should be done at compile time to force the user to write proper code and not ad-hoc a problem away.

Finally the framework should be open source to make these tools available to a wider audience.

Contents

Chapter 1

Introduction

In the past simulation methods were mostly used for numerical simulations, but recently these methods have found significant usage in computer graphics due to the ability of GPU based simulations to simulate complex systems, without compromises to physical plausibility, in a reasonable time. However, the ever increasing demands of performance and capability of simulations requires significant research and developmental effort. Many industry standard frameworks, e.g. Unreal Engine or Houdini, are often restrictive in what a developer can directly express as they are created for a specific purpose. Similarly, other SPH simulation frameworks, e.g. DualSPHysics, apply different restrictions, or abstractions, to the code which can make it very difficult to quickly implement an algorithm or argue about its correctness which puts a significant burden on the researchers.

The aim of this thesis is to introduce a versatile, multi-platform and open source simulation framework focused on Smoothed Particle Hydrodynamics (SPH). To this extent the framework described here should provide the following set of features:

- Safe and reliable mathematic functions which respect physical units to help develop physically plausible methods.
- Quick and easy prototyping of new methods without limiting what can be expressed manually.
- Run virtually the same code on both GPUs and CPUs without restricting the user in how they have to implement functions.
- Create functions, parameters and arrays using a straightforward meta modeling system.
- Modularize individual functions to easily encapsulate functionality and analyze dependencies between functions.
- Windows and Linux support, with a low number of special libraries.

Even though these features aim to provide a generic framework, they are applied in an exclusively SPH context to provide a more practical approach to these problems. These features might introduce some overhead in the implementation of certain functionality, but as this overhead is the same for all methods the relative gains of individual methods should remain the same allowing for a representative evaluation.

Overall, this thesis is split into 4 parts:

In the first part the fundamental building blocks of the framework are introduced. Chapter 2 introduces the underlying fundamentals of SPH and how this method can be derived from continuous equations to obtain some practical operators, which are easily implemented in this framework. Chapter 3 covers a set of basic modern C++ constructs to create a solid foundation of C++ terminology so the explanations in later chapters using these constructs are more approachable. These two chapters only cover the fundamentals to provide some motivation and insight into the problems faced here. Following this chapter 4 summarizes the goals of this framework into more concrete criteria, which are used to develop the framework throughout this thesis.

The second part covers the mathematical aspects of the framework and how these are implemented here. In chapter 5 the systems for mathematical functions and operators for low dimensional math using the built-in CUDA types are introduced with some insight into how these are implemented using generic programming. Chapter 6 extends these functions and operators to support strict enforcement of SI-units within the program at compile time where the details are mostly left vague as the way they are represented is more important for their understanding.

The third part covers the more general purpose aspects of the framework. These include, in chapter 7, a meta-modeling approach which uses JSON strings to represent parameters, arrays, and modules used within the framework to allow for an easy and central management of these aspects. Chapter 8 covers how the framework manages arrays and module calling to provide an insight into how memory is intended to be used here to expand upon the previous meta descriptions. Finally, in this part chapter 9 covers how functions in this framework are launched both in a GPU as well as CPU context and how the corresponding framework aspects should be used in practice.

The last part covers the more practical aspects of the framework. First, in chapter 10 the overall results from this framework to the prior one are compared, with some examples of basic SPH constructs and how they are implemented, as well as an example which shows how the unit enforcing system identifies problems in practice. Next chapter 11 investigates the generated assembly code of the mathematical operators to evaluate, whether or not the implementation has any theoretical drawbacks compared to manually implemented functions. Chapter 12 then covers some miscellaneous aspects of the framework, e.g. rendering, and gives an overview of the general structure of the code base.

In chapter 13 some final conclusions are drawn about the framework and possible drawbacks.

Chapter 2

Foundations of Smoothed Particle Hydrodynamics

Smoothed Particle Hydrodynamics as it has been introduced by Gingold and Monaghan [1977] has been around for a long time, but has recently found significant improvements. These improvements allow for real time simulations of complex systems, or very large, and highly detailed, simulations for computer graphics, which makes SPH a very attractive method. There exist other simulation methods for fluids, e.g. Eulerian grid based methods [Ando et al. [2013]] or more closely related methods like FLIP [Cornelis et al. [2014]], but the choice of simulation method is not topic of this thesis and SPH serves as an example of how this framework can be applied.

This section in general covers the theoretical foundations of SPH to give a reader that is not as familiar with the method some intuition, and appreciation, for the way the method tries to replicate real phenomena. This will only include the basic formulations, e.g. for gradients, and no complex algorithms, like incompressible pressure solvers or implementation specific data structures, which are often used in practice.

2.1 Governing equations

The underlying basis of SPH are the **Navier-Stokes equations**. In this derivation the compressible version was chosen due it's relative simplicity and guaranteeing incompressibility is beyond the scope of this section. These equations in general are stated as [Monaghan [1992]]:

$$\begin{aligned} \frac{D\rho}{Dt} &= -\rho\nabla\cdot\mathbf{v} \\ \frac{D\mathbf{v}}{Dt} &= \frac{1}{\rho}\nabla\sigma + \mathbf{f_e} \\ \frac{D\mathbf{x}}{Dt} &= \mathbf{v} \end{aligned} \tag{2.1}$$

In these equations D/Dt denotes the material derivative, ρ denotes the density, $\mathbf{x}$ denotes positions and $\mathbf{v}$ denotes velocities, which are continuous field expressions. $\mathbf{f_e}$ denotes the body force per unit mass and σ denotes the stress tensor calculated as:

$$\sigma = -\mathbf{PI} + \eta\left[-\frac{\mathbf{2}}{\mathbf{3}}(\nabla\cdot\mathbf{v})\mathbf{I} + (\nabla\mathbf{v} + \nabla\mathbf{v^T})\right] \tag{2.2}$$

$\mathbf{I}$ being the identity tensor and η denoting dynamic viscosity. The pressure P, in a basic SPH formulation, is calculated using an equation of state, where a common choice is the *Tait equation* [Müller et al. [2003]]:

$$P = \frac{\rho_0 c_0^2}{\gamma}\left[\left(\frac{\rho}{\rho_0}\right)^{\gamma} - 1\right] \tag{2.3}$$

The Tait equation uses the rest density ρ_0, which describes the density of the fluid under no compression, a compression constant γ and the speed of sound. In practice the real density of water $\rho_0 \approx 1000\frac{\mathrm{kg}}{\mathrm{m}^3}$ and a factor of $\gamma = 7$ is often used. The speed of sound is chosen significantly lower than the real speed of sound in water, e.g. $c_0 = 10 - 30\frac{\mathrm{m}}{\mathrm{s}}$, as the speed of sound directly influences the stability of the simulation. This method is still used as an example, even though the physical accuracy is limited by the low speed of sound, whereas in practice more complex schemes are often employed [Ihmsen et al. [2014a], Bender and Koschier [2015]].

2.2 Discretization

The continuous Navier-Stokes equations cannot be directly implemented and need to be discretized. The SPH method is derived by first considering the *integral identity* of a function f, here in a 3 dimensional real space $\mathbb{R}^3$, written as:

$$f(\mathbf{x}) = \int_{\mathbb{R}^\mathbf{3}} \mathbf{f}(\mathbf{x}')\delta(\mathbf{x}-\mathbf{x}')\mathbf{dx}' \tag{2.4}$$

This is based on the *Dirac delta function* [Price [2010]]:

$$\int_{\mathbb{R}^3} \delta(\mathbf{x}) = \begin{cases} \infty, & \mathbf{x} = \mathbf{0} \\ 0, & \mathbf{x} \neq \mathbf{0} \end{cases} \tag{2.5}$$

In order to actually use this identity the Dirac delta function is replaced with a so called **smoothing kernel function** over a finite closed space Ω resulting in the following equation, with $\langle f(\mathbf{x})\rangle$ denoting an estimated quantity:

$$\langle f(\mathbf{x})\rangle = \int_{\mathbf{\Omega}} \mathbf{f}(\mathbf{x}')\mathbf{W}(\mathbf{x}-\mathbf{x}',\mathbf{h})\mathbf{dx}' \tag{2.6}$$

The smoothing kernel function, or often times referred to as just a kernel, uses a finite **smoothing length** h which determines the contribution of the kernel based on the distance between the integral center $\mathbf{x}$ and the integrated point $\mathbf{x'}$. This equation can now be discretized by using Lagrange points, more commonly called particles, where two particles i and j are located at positions $\mathbf{x_i}$ and $\mathbf{x_j}$, with W_{ij} being short hand notation for $W(\mathbf{x_i} - \mathbf{x_j}, \mathbf{h})$ which, using a *numerical quadrature*, results in:

$$\langle f(\mathbf{x_i})\rangle \approx \sum_{\mathbf{j}} \mathbf{V_j f(x_j) W_{ij}} \tag{2.7}$$

Which is a *Riemann summation* over all particles j within a spatial region Ω, where particles with $W_{ij} > 0$ denote so called *neighboring particles*. The **apparent volume** for the numerical quadrature is defined as:

$$V_j = \frac{m_j}{\rho(\mathbf{x_j})} \tag{2.8}$$

With m_j denoting the mass represented by a particle and ρ_j being the density of the fluid sampled at position $\mathbf{x_j}$. As all SPH estimates based on context are assumed to be estimates $\langle f(\mathbf{x_i})\rangle$ is often replaced with just f_i. For an estimate of the density $\rho_i = \langle \rho(\mathbf{x_i})\rangle$ this results in:

$$\rho_i = \sum_j \frac{m_j}{\rho(\mathbf{x_j})} \rho(\mathbf{x_j}) \mathbf{W_{ij}} = \sum_{\mathbf{j}} \mathbf{m_j W_{ij}} \tag{2.9}$$

If an even kernel function is used, namely one with the following property:

$$\nabla_{\mathbf{x'}} W(\mathbf{x} - \mathbf{x'}, \mathbf{h}) = -\nabla_{\mathbf{x}} \mathbf{W}(\mathbf{x} - \mathbf{x'}, \mathbf{h}) \tag{2.10}$$

Then the *Gaussian theorem* can be applied to approximate the **gradient** of a scalar function as:

$$\begin{aligned}
\langle \nabla f(\mathbf{x_i})\rangle &= \int_\Omega \nabla_{\mathbf{x'}} f(\mathbf{x'}) \mathbf{W}(\mathbf{x} - \mathbf{x'}, \mathbf{h}) \mathbf{dx'} \\
&= \int_{\delta\Omega} f(\mathbf{x'}) \mathbf{W}(\mathbf{x} - \mathbf{x'}, \mathbf{h}) \mathbf{ndS} - \int_{\boldsymbol{\Omega}} \mathbf{f}(\mathbf{x'}) \nabla_{\mathbf{x'}} \mathbf{W}(\mathbf{x} - \mathbf{x'}, \mathbf{h}) \mathbf{dx'} \\
&= \int_\Omega f(\mathbf{x'}) \nabla_{\mathbf{x}} \mathbf{W}(\mathbf{x} - \mathbf{x'}, \mathbf{h}) \mathbf{dx'}
\end{aligned} \tag{2.11}$$

And the **divergence** of a vector function as:

$$\begin{aligned}
\langle \nabla \cdot \mathbf{f}(\mathbf{x_i})\rangle &= \int_\Omega [\nabla_{\mathbf{x'}} \cdot \mathbf{f}(\mathbf{x'})] \mathbf{W}(\mathbf{x} - \mathbf{x'}, \mathbf{h}) \mathbf{dx'} \\
&= \int_{\delta\Omega} \mathbf{f}(\mathbf{x'}) \mathbf{W}(\mathbf{x} - \mathbf{x'}, \mathbf{h}) \mathbf{ndS} - \int_{\boldsymbol{\Omega}} \mathbf{f}(\mathbf{x'}) \cdot \nabla_{\mathbf{x'}} \mathbf{W}(\mathbf{x} - \mathbf{x'}, \mathbf{h}) \mathbf{dx'} \\
&= \int_\Omega f(\mathbf{x'}) \cdot \nabla_{\mathbf{x}} \mathbf{W}(\mathbf{x} - \mathbf{x'}, \mathbf{h}) \mathbf{dx'}
\end{aligned} \tag{2.12}$$

Which can be discretized using discrete position $\mathbf{x_i}$ and $\mathbf{x_j}$ as before, with ∇_i denoting the differentiation carried out with respect to particle i's position [Monaghan [2005]]:

$$\langle \nabla f(\mathbf{x_i}) \rangle \approx \sum_{\mathbf{j}} \mathbf{V_j f(x_j) \nabla_i W_{ij}} \tag{2.13}$$

$$\langle \nabla \cdot \mathbf{f}(\mathbf{x_i}) \rangle \approx \sum_{\mathbf{j}} \mathbf{V_j f(x_j)} \cdot \mathbf{\nabla_i W_{ij}} \tag{2.14}$$

2.3 Discrete governing equations

Using the prior approximations and discretization the Navier-Stokes equations can be rewritten as:

$$\begin{aligned} \frac{D\rho_i}{Dt} &= -\rho_i \nabla \cdot \mathbf{v_i} \\ \frac{D\mathbf{v_i}}{Dt} &= \frac{1}{\rho_i} \nabla(-P_i) + \mathbf{f_e} \\ \frac{D\mathbf{x_i}}{Dt} &= \mathbf{v_i} \end{aligned} \tag{2.15}$$

With a discretized Tait equation:

$$P_i = \frac{\rho_0 c_0^2}{\gamma} \left[(\frac{\rho_i}{\rho_0})^\gamma - 1 \right] \tag{2.16}$$

2.4 Smoothing kernel functions

In general a suitable kernel should have a set of properties [Monaghan [1992]]:

$$\lim_{h \to 0} W(\mathbf{x} - \mathbf{x}', \mathbf{h}) = \delta(\mathbf{x} - \mathbf{x}'), \tag{2.17}$$

$$\int_\Omega W(\mathbf{x} - \mathbf{x}', \mathbf{h}) \mathbf{dx}' = \mathbf{1}, \tag{2.18}$$

$$\int_\Omega (\mathbf{x} - \mathbf{x}', \mathbf{h}) \mathbf{W}(\mathbf{x} - \mathbf{x}', \mathbf{h}) \mathbf{dx}' = \mathbf{0}. \tag{2.19}$$

Additionally to satisfy the requirements for gradient operators the kernel needs to have a compact support and be an even symmetric function:

$$W(\mathbf{x} - \mathbf{x}', \mathbf{h}) = \mathbf{0}; ||\mathbf{x} - \mathbf{x}'|| > \kappa \mathbf{h} \tag{2.20}$$

$$W(\mathbf{x} - \mathbf{x}', \mathbf{h}) = \mathbf{W}(\mathbf{x}' - \mathbf{x}, \mathbf{h}) \tag{2.21}$$

$$\nabla_{\mathbf{x}'} W(\mathbf{x}-\mathbf{x}',\mathbf{h}) = -\nabla_{\mathbf{x}} \mathbf{W}(\mathbf{x}-\mathbf{x}',\mathbf{h}) \tag{2.22}$$

The kernel should also be positive, monotonically decreasing and sufficiently smooth. A **kernel** in general is often written as:

$$W(\mathbf{x}-\mathbf{x}',\mathbf{h}) = \frac{\alpha_\mathbf{d}}{\mathbf{h}^\mathbf{d}}\hat{\mathbf{W}}(\mathbf{q}) \tag{2.23}$$

With a scaling factor α_d to satisfy $\int_\Omega W(\mathbf{x}-\mathbf{x}',\mathbf{h})\mathbf{dx}' = \mathbf{1}$, d being the spatial dimension and q being a dimensionless number representing a unit length as:

$$q = \frac{||\mathbf{x}-\mathbf{x}'||}{h}. \tag{2.24}$$

Using this notation the **derivative of a kernel** function can be written as:

$$\nabla W(\mathbf{x}-\mathbf{x}',\mathbf{h}) = \frac{\alpha_\mathbf{d}}{\mathbf{h}^{\mathbf{d}+\mathbf{1}}}\nabla\hat{\mathbf{W}}(\mathbf{q}) = \frac{\alpha_\mathbf{d}}{\mathbf{h}^{\mathbf{d}+\mathbf{1}}}\frac{\mathbf{d}\hat{\mathbf{W}}(\mathbf{q})}{\mathbf{dq}}\nabla\mathbf{q} = \frac{\alpha_\mathbf{d}}{\mathbf{h}^{\mathbf{d}+\mathbf{1}}}\frac{\mathbf{d}\hat{\mathbf{W}}(\mathbf{q})}{\mathbf{dq}}\frac{\mathbf{x}-\mathbf{x}'}{||\mathbf{x}-\mathbf{x}'||}. \tag{2.25}$$

Common choices for these functions include gaussian and B-spline functions or Wendlandt kernel functions [Dehnen and Aly [2012]], where the choice has implications on accuracy and computational cost. For computer graphics the **cubic spline kernel** has found wide adoption [Ihmsen et al. [2014b]] and has been used throughout this framework and is often defined as:

$$W(\mathbf{x}-\mathbf{x}',\mathbf{h}) = \frac{\alpha_\mathbf{d}}{\mathbf{h}^\mathbf{d}}\hat{\mathbf{W}}(\mathbf{q}); \hat{\mathbf{W}}(\mathbf{q}) = \begin{cases} 1-\frac{3}{2}q^2+\frac{3}{4}q^3, 0 & \leq q \leq 1 \\ \frac{1}{4}(2-q)^3, & 1 \leq q \leq 2 \\ 0, & \text{else} \end{cases} \tag{2.26}$$

With the normalization constant $\alpha_d = \frac{1}{\pi}$ for 3 spatial dimensions. The **gradient of the cubic spline kernel** can be calculated as:

$$\nabla W(\mathbf{x}-\mathbf{x}',\mathbf{h}) = \frac{\alpha_\mathbf{d}}{\mathbf{h}^{\mathbf{d}+\mathbf{1}}}\frac{\mathbf{d}\hat{\mathbf{W}}(\mathbf{q})}{\mathbf{dq}}\frac{\mathbf{x}-\mathbf{x}'}{||\mathbf{x}-\mathbf{x}'||}; \frac{\mathbf{d}\hat{\mathbf{W}}(\mathbf{q})}{\mathbf{dq}} = \begin{cases} \frac{9}{4}q^2-3x, & 0 \leq q \leq 1 \\ -\frac{3}{4}(2-q)^2, & 1 \leq q \leq 2 \\ 0, & else. \end{cases} \tag{2.27}$$

The smoothing length h of a particle can be calculated as [Monaghan [1992]]:

$$h_i = \eta\left(\frac{m_i}{\rho_i}\right)^{\frac{1}{3}} \tag{2.28}$$

With η being a parameter to tune the smoothing length chosen around 1.3. In practice more complex schemes are often employed but the concept remains the same.

2.5 Improved spatial gradients

Although the previous approximate gradients are sufficient in some cases, a more robust formulation can be derived [Price [2010]] by considering the following alternative problem:

$$\nabla f(\mathbf{x}\cdot\mathbf{1}) = \mathbf{1}\nabla\mathbf{f}(\mathbf{x}) + \mathbf{f}(\mathbf{x})\cdot\nabla\mathbf{1} \Rightarrow \nabla\mathbf{f}(\mathbf{x}) = \nabla\mathbf{f}(\mathbf{x}) - \mathbf{f}(\mathbf{x})\nabla\mathbf{1} \tag{2.29}$$

Where an intuitive consideration would be $\nabla 1 = 0$. However, for SPH $\nabla 1$ is non zero. Thus using the previous definition of the gradient an alternative gradient formulation can be found which vanishes for constant functions:

$$\nabla f(\mathbf{x}) = \sum_{\mathbf{j}} \mathbf{V_j}\left[\mathbf{f}(\mathbf{x_j}) - \mathbf{f}(\mathbf{x_i})\right]\nabla\mathbf{W_{ij}} \tag{2.30}$$

Alternatively the *vector calculus identity* can be employed:

$$\nabla f(\mathbf{x}\rho^{\mathbf{n}}) = \mathbf{n}\rho^{\mathbf{n-1}}\mathbf{f}(\mathbf{x})\nabla\rho + \rho^{\mathbf{n}}\nabla\mathbf{f}(\mathbf{x}) \tag{2.31}$$

Which leads to the following gradient:

$$\nabla f(\mathbf{x}) = \frac{\mathbf{1}}{\rho^{\mathbf{n}}}\left[\nabla(\mathbf{f}(\mathbf{x}\rho^{\mathbf{n}}) - \mathbf{n}\rho^{\mathbf{n-1}}\mathbf{f}(\mathbf{x})\nabla\rho\right] \tag{2.32}$$

Which for $n = 1$ and $n = -1$ respectively leads to:

$$\nabla f(\mathbf{x_i}) = \frac{\mathbf{1}}{\rho(\mathbf{x_i})}\sum_{\mathbf{j}} \mathbf{m_j}(\mathbf{f}(\mathbf{x_i}) - \mathbf{f}(\mathbf{x_j}))\nabla\mathbf{W_{ij}} \tag{2.33}$$

$$\nabla f(\mathbf{x_i}) = \rho(\mathbf{x_i})\sum_{\mathbf{j}} \mathbf{m_j}\left(\frac{\mathbf{f}(\mathbf{x_i})}{\rho(\mathbf{x_i})^{\mathbf{2}}} + \frac{\mathbf{f}(\mathbf{x_j})}{\rho(\mathbf{x_j})^{\mathbf{2}}}\right)\nabla\mathbf{W_{ij}} \tag{2.34}$$

Where the first formulation is not anti-symmetric but exact for constant functions, and the second formulation being pairwise symmetric but not exact for constant functions. Similar derivations can be made for the divergence of a vector field as:

$$\nabla\cdot(\rho^{n}\mathbf{f}(\mathbf{x_i})) = \rho^{\mathbf{n}}\cdot\nabla\mathbf{f}(\mathbf{x_i}) + \mathbf{n}\rho^{\mathbf{n-1}}\mathbf{f}(\mathbf{x_i})\nabla\rho \tag{2.35}$$

Which for the $n = 1$ and $n = -1$ cases respectively yields:

$$\nabla\cdot\mathbf{f}(\mathbf{x_i}) = \frac{\mathbf{1}}{\rho(\mathbf{x_i})}\sum_{\mathbf{j}} \mathbf{m_j}(\mathbf{f}(\mathbf{x_j}) - \mathbf{f}(\mathbf{x_i}))\cdot\nabla\mathbf{W_{ij}} \tag{2.36}$$

$$\nabla\cdot\mathbf{f}(\mathbf{x_i}) = \rho(\mathbf{x_i})\sum_{\mathbf{j}} \mathbf{m_j}\left(\frac{\mathbf{f}(\mathbf{x_i})}{\rho(\mathbf{x_i})^{\mathbf{2}}} + \frac{\mathbf{f}(\mathbf{x_j})}{\rho(\mathbf{x_j})^{\mathbf{2}}}\right)\cdot\nabla\mathbf{W_{ij}} \tag{2.37}$$

These basic building blocks will be used as examples of functions later on to show how SPH equations could possibly be implemented and serve as a strong foundation for the development of SPH methods in general.

Chapter 3

Foundations of Modern C++

Many code bases are created over a long period of time, with many developers working collectively on them, with all of them trying to reach an optimized program that solves a task as quickly as possible. In numerical applications, like simulations, this often leads to very bloated code, or code relying on older constructs which are deemed to be working very well and touching the foundations of the system again is often avoided at all cost. In case of the prior framework that is being replaced here the code became almost impossible to use in any reasonable manner, which in turn hurt research efforts significantly.

The goal here is to present approaches in modern C++ that could help solve problems in numerical code to allow for a framework in which it is very easy to express complex statements by relying upon the underlying compiler. Whilst this approach might not lead to the most optimal code, it will lead to a very concise and easy to argue about code where it should be simple to draw parallels from equations to code and back.

In order to achieve this certain features of C++ are needed which might not be familiar to some readers as some of them are part of the newest C++ standard which is just getting supported by compiler vendors. This chapter will offer a brief overview of some basic constructs and approaches to solving problems which are applied throughout the framework.

3.1 GPGPU Programming

C++ itself is only a programming language for code that is to be executed on a CPU and the C++ standard only provides multi threading constructs [ISO C++ [2017] [thread]] which are not directly intended for numerical code but for a more general task parallelization. In practice many other standards have been created for CPU parallelization, e.g. **OpenMP [2015]** and **OpenACC [2017]**, which are more oriented towards numerical applications. OpenACC on the one hand allows for easy parallelization, even for GPU targets, but limits the expressiveness of what can easily be done using OpenACC constructs. OpenMP on the other hand is a much more generic system which has been used for a long time and allows for easy parallelization of code for CPUs, however the accelerator support is not straight forward and again limits what can be done freely.

The two main candidates for GPU programming that were thus considered for this framework are **CUDA [2018]** and **OpenCL [2018]**. Whilst OpenCL is an open standard created by a conglomerate of vendors and would allow for a hardware agnostic implementation that works on multiple GPU vendors devices, CUDA offers a significantly more powerful programming interface where code itself can be written as mostly ISO C++ [2014] compliant code. This makes CUDA significantly more attractive as the benefit of being able to write standard code outweighs the support of other GPU vendors devices in an academic background where no external customer base exists.

3.2 CUDA

The standard execution model of CUDA, ignoring memory allocation, is the following

```
__global__ void add_fn(float* A, float* B, float* C, int32_t N){
  int32_t i = threadIdx.x + blockDim.x * blockIdx.x;
  if(i > N) return;
  C[i] = A[i] + B[i];
}
//...
add_fn<<<blocks,threadsPerBlock>>>(d_A, d_B, d_C, N);
```

whereas a sequential CPU version could look like this:

```
for(int32_t i = 0; i < N; ++i)
  C[i] = A[i] + B[i];
```

The basic idea of CUDA is to write a **kernel** function (not to be confused with the SPH meaning of the word kernel) that is executed once for each element of "a loop". The call is configured into *blocks* of *thread-groups*, where every thread-group contains a certain number of *threads per block*. Each of these blocks is executed in parallel as sets of 32 threads (*warps*) and they share resources on a processing core (*SMM*) and have the option to use *shared memory* on a block level.

The details of CUDA are not the topic of this thesis but will be covered where necessary. Note, however, that writing perfectly optimized code can include significant usage of either inline assembly or complex intrinsic functions which can offer increased performance at the cost of creating a less readable code basis. For most problems within a framework similar, basic, approaches can be used which might not be the best solution for every method but provide a very solid, and more importantly comparable, basis for all methods. Additionally SPH, and other similar methods like FLIP, are *data-parallel* simulations with very few, if any, dependencies allowing for easy parallelization. For concrete GPU specifics to SPH see, for example, Green [2010] which offers a basic overview.

3.3 Types of variables

The following code segment is a very simple example which calculates some value based on the input of two other values:

```
double fn(double a, double b){
  return a * b * b * a;
}
int main(){
  float f = fn(1.0, 2.0);
}
```

The important part here is that even though the function returns a *double* value, the result is stored within a *float* value causing a possibly unwanted *conversion* which causes a loss of arithmetic precision. In this example the type of f could simply be changed to the correct one, or an explicit cast added to signal that the conversion is intended. However, consider the following example:

```
#ifdef SINGLE_PRECISION
float unitCircumference(){
  return 2.f * 3.141592f;
}
#else
double unitCircumference(){
  return 2.0 * 3.14159265359;
}
#endif
int main(){
  float f = unitCircumference();
}
```

In this case there still is a mismatch of types as the function returns a double which again is stored as a float value. Changing the type of f to match now would create a problem as the return type of the function depends on a macro. Carrying this *dependence* through the code would make the code significantly less readable and more difficult to maintain. ISO C++ [2011] introduced the following construct to address this problem:

```
declspec(expression)
```

Which represents the *type* and not the *value* of the expression. In the previous example the mismatch could then be resolved by changing the call to the following:

```
declspec(unitCircumference()) f = unitCircumference();
```

However, repeating the call on both sides is very verbose and instead **auto** is used:

```
auto f = unitCircumference();
```

This auto construct uses the type of the return value of the function which allows relying on the strong type system of C++ without having to check, or remember, the exact return types of functions and consider difficult cases like above. An additional benefit of auto lies with more complex types, e.g. those created from iterator functions, or those created within deeply nested namespaces which are difficult to write out correctly by hand, whereas the compiler will known on it's own what the correct type is.

3.4 Function return types

An additional place where auto can be used is as a *function return type*. Consider the following, somewhat construed example, where ISO C++ [2011] introduced the following way to write it:

```
auto fn(double a, float b) -> declspec(a+b){
  return a + b;
}
```

Which is called a *trailing return type*. This trailing return type stems from templated functions where often times the return type depends on the types of the arguments and how they interact and replicating that behavior without such a construct in the type system can be very difficult and error prone. ISO C++ [2014] allows for functions to, in general, have an **auto return type** without a trailing return type:

```
auto fn(double a, float b){
  return a + b;
}
```

Which works as long as all return paths return the exact same type. This can make some template code significantly easier to write as it does not require large constructs to recreate what the compiler knows anyways.

3.5 Structured bindings

C++17 [ISO C++ [2017] [dcl.struct.bind]] introduced a further usage of auto for so called **structured bindings**. Whilst CUDA [2018] does not yet support these constructs in device code, they are still worth mentioning as they find wide usage in the GUI code of the framework to simplify some complex statements. A simple example of structured bindings is the following:

```
float2 getPos(){/*...*/}
std::tuple<int32_t, float, std::string> getPerson(){/*...*/}
//...
auto pos = getPos();
auto x = pos.x;
auto y = pos.y;
auto person = getPerson();
auto id = std::get<0>(person);
auto income = std::get<1>(person);
auto name = std::get<2>(person);
```

Where structured bindings simplify the code into the following:

```
float2 getPos(){/*...*/}
std::tuple<int32_t, float, std::string> getPerson(){/*...*/}
// ...
auto[x, y] = getPos();
auto[id, income, name] = getPerson();
```

These structured bindings can bind to tuples as well as any *Plain Old Data*-type (POD-type). These work great for functions with multiple return values, where manually operating on tuples with std::get becomes cumbersome. Additionally these types can also be *bound to references*, similar to normal auto variables.

3.6 Templates

Templates [ISO C++ [2017] [temp]] have been around in C++ for a long time, and they allow for very generic and flexible code, but they can also easily be abused. In this section we will first look at a very basic example of both function and class templates to set a solid foundation of what templates are.

```
template<typename T, typename U>
auto add(T&& lhs, U&& rhs){
  return lhs + rhs;
}
//...
auto result = add(1.0,2.f);
```

In this simple example a few different concepts are coming together. When a **function template** is called without specifying the template parameters the compiler will execute a **template argument deduction** [ISO C++ [2017] [temp.deduct.call]]. During this deduction the compiler will try and figure out what kinds of template parameters would best fit the description, which for example in the following:

```
template<typename T>
auto len(const std::vector<T>& vec){ return vec.size();}
//...
len(std::vector<float>{1,2,3,4,5});
```

Would deduce T to be float. In the previous example however this is not as straight forward due to the usage of **&&**. In normal contexts C++ distinguishes between left and right references [ISO C++ [2017] [dcl.ref]] where a **left reference** represents a left side of an assignment, so a named variable, and a **right (or value) reference** represents a right side of an assignment, so a value. A value reference behaves just like a value, however in contrast to a left reference certain optimizations can be used, e.g. *moving* which means to pull resources out of the value without having to copy them. In the context of template argument deduction denoting a type **directly** with && turns it into a **forwarding reference** [ISO C++ [2017] [temp.deduct.call]] which can turn into either a left or right reference. In this example T would be deduced as *double*&& and U as *float*&& giving:

```
auto add(double&&&& lhs, float&&&& rhs){...}
```

If either one had been a left reference the compiler would have deduced float& for example. This results in literally float&&& which **collapses** to float&. Similarly in the given example the numbers are already value references thus giving float&&&& which collapses to float&& [ISO C++ [2017] [dcl.ref]].

Consider, however, the following example:

```
template<typename T, typename U>
struct fn{
  static auto add(T lhs, U rhs){return lhs + rhs;}
};
//...
auto val = fn<double,float>::add(1.0,1.f);
```

Here a **template struct** is used where no template argument deduction can take place (ignoring template deduction guidelines added in C++17). This means that the template parameter needs to be *manually specified* or wrapped inside of a helper function:

```
template<typename T, typename U>
auto fn_tad(T&& lhs, U&& rhs){
  return fn<T,U>::add(std::forward<T>(lhs),std::forward<U>(rhs));
}
//...
auto val = fn_tad(1.0,1.f);
```

Where the call to fn_tad can use template argument deduction, where the *deduced* types are used to instantiate fn itself. In order to pass along forwarding references exactly as they are std::forward<T>(val) is used.

Templates can also be overloaded, or more precisely **specialized**, where certain versions of a template have more specific implementations. The exact rules of this are complex but the general principle "*the match that uses less conversions wins*" usually holds true.

3.7 Template parameter packs

Consider the following problem: A function should be written which takes another function and the arguments to call that function with and prints some statement to the console before and after the call. In order to solve this problem C++ has so called **template parameter packs** [ISO C++ [2017] [temp.param]], or sometimes called **variadic templates**, which allow for packs of parameters to be bound to a single identifier. The following is a possible solution to the prior problem using these packs:

```
template<typename Func, typename... Ts>
auto callFn(Func fn, Ts&&... args){
  std::cout << "before" << std::endl;
  auto retval = fn(std::forward<Ts>(args)...);
  std::cout << "after" << std::endl;
  return retval;
}
```

Here typename... Ts denotes a template parameter pack. This pack consists of a list, which can be empty, of types. The individual elements of a pack cannot be accessed directly and instead the pack can only be used by *expanding* it using an ellipsis **...**. These parameter packs are often used to iterate over a list of arguments in a *recursive* manner where a base case of the function is one with no template argument, for example:

```
void tprintf(const char* format){
    std::cout << format;
}
template<typename T, typename... Ts>
void tprintf(const char* format, T value, Ts... args){
    for ( ; *format != '\0'; format++ ) {
        if ( *format == '%' ) {
          std::cout << value;
          tprintf(format+1, args...); // recursive call
          return;
        }
        std::cout << *format;
    }
}
```

Which implements a printf function using variadic templates from C++ instead of *variadic functions* from C. These variadic templates can often significantly increase compile time, but as they are widely supported by all compilers and offer significant abstractions which makes the tradeoff usually worth it. However implementing every function as a variadic template is not necessarily ideal, similar to templates in general.

3.8 Lambda functions

Consider the following example:

```
template<typename T, typename Func>
auto apply(std::vector<T>& vec, Func fn){
  for(auto& elem : vec)
    elem = fn(elem);
}
```

Which applies a unary function to each element in a vector. These apply operations often use a very specific function that is only used *locally* and creating a global function and taking a pointer to this function to pass it, or creating a functor, is a significant programmatic overhead. Instead C++ contains **lambda** functions which are specified as follows [ISO C++ [2017] [expr.prim.lambda]]:

```
[ capture ] ( params ) -> ret { body }
```

Where **capture** describes a list of variables which are either explicitly captured by reference & or value = or implicitly using a **default capture mode** [ISO C++ [2017] [expr.prim.lambda.closure]]. *Params* are the same as normal function parameters, *ret* is the return value, which usually is implicitly used as auto, and *body* is the body of the function being created where the parameters and all captured variables are accessible. For example a lambda to negate all elements using the apply function and one to increment every element by the vector length could be:

```
std::vector<float> vec{...};
apply(vec, [](float val){ return -val;});
auto lambda = [&](float val){ return val + vec.size();};
apply(vec, lambda);
```

In the first example no captures are required. In the second example the stack is captured by reference (to avoid copying vec) and the lambda is stored in a variable of type auto. The type of a lambda function is is a unique, unnamed nonunion class type called the **closure type** [ISO C++ [2017] [expr.prim.lambda]].

In addition to these aspects, lambda functions can also use arguments of type auto. These lambdas are then called **generalized** (or polymorphic) **lambdas**, where for example a lambda function can be passed to a function without requiring knowledge of the exact types being used. A common example for this would be:

```
std::sort(vec.begin(), vec.end(), [](const auto& lhs, const auto& rhs){
  return lhs > rhs;
});
```

3.9 Template control

Often times when writing generic tasks a problem that arises is that the specialized version for some kind of template parameters is not just *different in types* but relies on *different members or functions*. This means that the function, if it was instantiated with different types, would create an *ill-formed program* for some types. However, due to using a template, simply overloading the function or providing an alternative template does not work directly, and instead a concept called SFINAE needs to be used.

Substitution Failure Is Not An Error describes a mechanism where if the compiler during *type substitution*, for a candidate template, detects a failure the candidate is discarded and no error is emitted. Only if *all* candidates are discarded will an error be emitted. These failures are often caused by **std::enable_if<c>** which is a construct that if the boolean condition c is false forms an ill-formed statement. For example:

```
template <typename T, typename std::enable_if_t<std::is_integral_v<T>>
void do_stuff(T& t) {/*...*/}
template <typename T, typename std::enable_if_t<std::is_class_v<T>>
void do_stuff(T& t) {/*...*/}
```

Compared to a more simple version using **static_assert**:

```
template <typename T>
void do_stuff(T& t) {static_assert(std::is_integral_v<T>);/*...*/}
template <typename T>
void do_stuff(T& t) {static_assert(std::is_class_v<T>);/*...*/}
```

Where the failure happens inside of the actual function body and not during type substitution. A failure here is an actual error *as the substitution has already finished*. In general the basic rule is that anything that is part of the signature, e.g. return type, parameters and template arguments, is part of the substitution process which allows for SFINAE behavior and the actual definition of the function or struct is not part of the substitution process.

There are many different variants of how SFINAE can be used, especially on functions, however many of them are unreliable with CUDA and the various compilers being used with CUDA. As such the following pattern is used throughout the framework as it is the most widely supported version for **function SFINAE**:

```
template<typename T, std::enable_if_t<C1>* = nullptr>
auto fn(T&& arg){/*...*/}
template<typename T, std::enable_if_t<!C1>* = nullptr,
  std::enable_if_t<C2>* = nullptr>
auto fn(T&& arg){/*...*/}
```

Which is relatively verbose by requiring repeated arguments. The mechanism used here is using **template variables** instead of types which in this case are unnamed parameters of type void* due to enable_if_t being void if the condition is valid. These variables are assigned a default parameter, meaning they do not need to be specified manually, but in order to satisfy some CUDA compiler versions an increasing number of arguments needs to be used.

3.10 constexpr

A much derided short coming of C++ for some people is the following example:

```
int32_t some_fn(){/*...*/}
//...
const int32_t N = some_fn();
double data[N];
```

In this example a fixed size array of size N should be created, however this syntax only allows for compile time fixed size arrays. This fixed size does not refer to a **run-time const** size but a **compile-time constexpr** value. This distinction was added in ISO C++ [2011] [expr.const] and allows parts of the code to be *executed at compile time* and not at runtime. The above example could then be written as:

```
constexpr int32_t some_fn(){/*...*/}
//...
constexpr int32_t N = some_fn();
double data[N];
```

Which is now valid code. This code creates an array with a size that is calculated and set at compile-time and thus fulfills the requirements of a fixed size array. These constexpr expressions and functions can for example be used *within templates*, where a common use-case are **global constexpr variables** where a given trait, e.g. std::is_integral<T>, is turned into a global boolean variable:

```
template<typename T>
constexpr bool is_integral_v = std::is_integral<T>::value;
```

Where is_integral_v<T> can be used within template parameters directly. ISO C++ [2017] [stmt.if] introduced an extension to if branches which allows them to be picked at compile time which, similar to SFINAE, allows writing different code for different types, e.g.

```
template<typename T>
auto fn(){
  if constexpr(std::is_integral_v<T>){/*...*/}
  else { /*...*/ }
}
```

However, this feature is not yet supported in CUDA but is still useful for host side code.

3.11 Member detection idiom

A problem often faced in code is to detect whether or not a certain type contains a member of a certain name. This problem for example is faced in certain math code to detect whether or not a type has an x member. C++ Library TS2 introduced the following mechanism

```
template< class Default, template<class...> class Op, class... Args >
using detected_or = /* ... */;
template< class Default, template<class...> class Op, class... Args >
using detected_or_t = typename detected_or<Default, Op, Args...>::type;
```

Where *detected_or* is an **alias template** to an unnamed type which contains a value_t and type entry. If the Op<Args> denotes a valid type then value_t is an alias for std::true_type and type is an alias for Op<Args>. If Op<Args> does not denote a valid type then value_t is an alias for std::false_type and type is an alias for Default. In order to utilize this the following macro (ignoring the line splice operators) can be constructed:

```
#define HAS_MEMBER(ty)
template <class T>
using ty##_t = decltype(std::declval<T>().ty);
template <class Ptr>
using ty##_type_template = detected_or_t<std::ptrdiff_t, ty##_t, Ptr>;
template <typename T>
constexpr bool has_##ty =
    !std::is_same<ty##_type_template<T>, std::ptrdiff_t>::value;
```

Where first a template alias is constructed that checks for the member ty in T by trying to declare a value of type T and accessing the member ty which is only stored as a type. This type is thus only valid if ty exists within T. Next the detector_or_t is constructed with the default type of std::ptrdiff_t, although any fixed type could be used. If the type of the detector_or_t is the default value ty does not exist within T and if this is not the case ty exists within T which is wrapped inside the template constexpr variable has_ty.

Do note however that these functions are *not yet part of the standard*, however implementing them is fairly straight forward as the Library specification provides possible implementations which are valid current C++ code as they do not require new language features. Up until recently this idiom was not possible in CUDA and a much more difficult idiom had to be used.

Chapter 4

Goals of the framework

The goal of the framework presented here is to implement a simulation based on SPH, which conceptually was shown in chapter 2, using the foundations built in chapter 3. However, before this can be done a few *guiding principles* have to be set so in the end a valid conclusion can be drawn as to whether or not the framework works as planned and if all problems have been adequately addressed. Many of these aspects are similar to those presented in chapter 1 but are now concretized into actual criteria.

In the introduction the main goal that was outlined was **usability**. This goal, whilst difficult to measure, builds the foundation of this framework and is the reason many of these features have been designed in the way they are. Here the question in general is one of **back-end vs front-end** complexity.

A *direct* solution to creating a framework might be to not provide any back-end features and require the user to manage everything *manually*. This is fairly simple to construct but requires significant effort on the users part, but it also allows for the most flexibility. On the complete *opposite* site is a framework with a very complex and large back-end which aims to provide the most *powerful abstractions* possible to the user making it possible to very easily express complex statements. The second solution in general seems trivially superior but in practice due to so called **impedance mismatch**es, meaning a difference in what the framework creator wants and what the user needs, these systems can become very difficult to use. An example of such a mismatch would be using Unreal Engine for simulation research where the backend of the engine provides large amounts of very powerful abstractions but those are a complete mismatch for generic simulations. The goal thus is to create a very powerful back-end which is tailored to the task at hand but is essentially optional to *not restrict* the user.

This **balance** of back to front-end was one of the most fundamental problems of the previously employed framework where not only an impedance mismatch, due to using a scene graph for the simulation, existed, but no real powerful back-end functionality existed which amplified the problem. To avoid such a problem the framework introduced here should require as little as possible from the user to actually follow and *not force a concept*, e.g. of a scene-graph, onto the simulation. This should still be combined with a powerful back-end which can be used for powerful abstractions.

Another problem often faced is that it is very easy in practice to accidentally write bugs in a system due to *misusing units*. In normal frameworks this is fairly difficult to catch as physical units are usually not part of the calculations and doing this in a way that does not impact performance would be a great addition. This goes together with providing not just a **unit system** but also a **math system** for common operations which creates fast and efficient code without requiring the user to be aware of the underlying complexity.

Additionally, often times systems are created once for a specific purpose and then are extended ad-hoc to add functions over time, or even just simple modules, which can bloat the code up as the framework was not constructed with a concept of **modules with dependencies** in mind, which should easily be extended. As such the goal here is to facilitate a system where it is easy to not just add a module of functions to the framework, but also to easily figure out where the components come from, what the requirements are and if the module can safely be removed.

A fourth problem often faced is the **difficulty of debugging** GPU based code. In theory tools like gdb exist, but these often slow the overall execution down to such an extent that the simulation not just takes a very long time to cause an error but might not cause any error at all due to different parallelization effects under debugging. Similarly just adding debug messages to parallel GPU code can easily timeout kernels causing a crash in a program when just a check was intended. As such the framework should provide the option to run parts, or all, of the simulation on the CPU and provide the option to **step back in "time"** to a previous point to then switch to a CPU version to investigate problems more closely.

A final problem is obviously **performance**. A large set of back-end functionality and abstractions is nice to have, but if these features cause the simulation to be unacceptably slow then there is no real benefit in these abstractions. In general C++ nowadays has a concept of **Zero-Cost-Abstractions** where complex expressions still generate ideal code and this should also be the goal for this framework.

Overall the goals for this framework can be summarized as creating a powerful back-end within the framework which allows for abstractions that allow for very **easy to maintain and understand** code, whilst providing strong features, like unit checking and modularization, at **no measurable overhead** over a direct solution whilst allowing for a user, if desired, to write any arbitrary code that is standard compliant as to not force the user to adhere to the system if there is an impedance mismatch.

Whilst most of these goals are difficult to measure, i.e. how can an impedance mismatch be measured, some of them are more simple to quantify. The most powerful metric with these goals in mind are **Lines Of Code** (LOC) which should be significantly reduced compared to the previous framework, especially with regard to user code. Additionally there should be no negative performance impact compared to the previous framework and the abstractions should be investigated as to whether or not they produce **bad assembly code**, as compared to a *down to the metal* implementation.

Chapter 5

Implementing low dimensional math in CUDA

CUDA provides many *built-in types* for common computer graphics operations, namely 1D, 2D, 3D and 4D vectors for a multitude of types. However, CUDA directly does *not include any operations* on these types, neither simple operators like additions nor more complex functions like dot products. These built-in types however are available everywhere and are well understood by the compiler so including proper math functions for these types is a good idea. Additionally providing a solid basic mathematical foundation offers valuable back-end functionality if it is done in a generic and unobtrusive way.

5.1 Alignment

In order to guarantee more efficient memory accesses many types, or programs, include hints on the **alignment** of types. This is no different for the CUDA types where, for example, for float4 the definition of this type is as follows [CUDA [2018]]:

```
struct __device_builtin__ __builtin_align__(16) float4
{
        float x, y, z, w;
};
typedef __device_builtin__ struct float4 float4;
```

Where __builtin_align__ is a macro that defines the memory alignment of the type in bytes. Using a custom vector type would require using something similar so that the CUDA compiler can perform the same optimizations. However, using these CUDA internal macros and specifiers might lead to *undefined behavior*, and whilst CUDA does provide an align macro for user code, the performance is not always the same due to compiler specific issues. Additionally using alignment specifiers in general in C++, for custom types, quickly becomes challenging as every compiler vendor uses a slightly different syntax.

The problem thus becomes using these built-in types as the underlying data types for everything and *adding functionality* for them on top. This could easily be done by manually covering all the possible cases, e.g. float4 + float4, float4 + float, float + float4 for addition, but this becomes difficult to maintain or read and is an ideal candidate for templates. Additionally certain comfort features like accessing the n-th value might be useful but this is not directly possible with the built-in definition except by doing something like

```
float4 vec{1,2,3,4};
float y = ((float*)&vec)[1];
```

Which is not safe and *might not be properly optimized.*

5.2 Type-traits

The first question that needs to be asked for a generic template is:

What defines a vector?

This question might seem obvious but it requires certain fixed **rules and assumptions** about the underlying types and how they are created. The most basic assumptions we can make is that, for example, a 2D vector contains a member called x, a member called y but no member called z. We could create further restrictions, e.g. the types of the members are all the same, but these additional constraints make the detection more difficult and do not increase the hit rate of these **traits**.

The problem with this assumption however is that this creates operators which are valid for *all types that follow these rules.* This might interfere with external libraries, like glm, which define template operators on their built-in types. As those operators and these created here would be equally correct an **ambiguous overload** would exist and as such a preprocessor define *NO_OPERATORS* can be used to block the operators of being visible at global scope where they are required for the CUDA built-in types. Normal math functions are all contained within a separate **math namespace**.

Actually detecting the vectors then becomes straight forward using the **member detection idiom** to create traits for the individual members x, y, z and w. These need to be combined into more generic terms by simply creating logical conjunctions:

```
template<typename T>
constexpr bool has_x = has_mem_x<T>;
template<typename T>
constexpr bool has_xy = has_mem_x<T> && has_mem_y<T>;
//...
```

For the 4 cases (x, xy, xyz, xyzw). Using these traits a helper function can be constructed which calculates the **dimension** of a vector where 0 is returned for integral and floating point types, e.g. double and int32_t, and *0xDEADBEEF* is returned as a constant for non vector types. It would be possible to filter these out by creating no SFINAE case where they are valid, but this leads to more complex calling code. This function is defined as:

```
template <typename T>
static constexpr uint32_t dimension_fn() {
        using Ty = std::decay_t<T>;
        if (std::is_integral<Ty>::value
                || std::is_floating_point<Ty>::value)
                return 0;
        else if (has_x<Ty> && !has_xy<Ty>)
                return 1;
        else if (has_xy<Ty> && !has_xyz<Ty>)
                return 2;
        else if (has_xyz<Ty> && !has_xyzw<Ty>)
                return 3;
        else if (has_xyzw<Ty>)
                return 4;
        else
                return 0xDEADBEEF;
}
template<typename T>
constexpr uint32_t dimension_v = dimension_fn<T>();
```

Note that even though the conditions within the if branches are compile time constant expressions that if constexpr cannot be used due to lacking support in CUDA, but the expressions within the if branches always compile to correct code so this is not an issue here.

5.3 Accessing elements

A problem described before is accessing the n-th element of a vector as they are defined within CUDA. This can effectively be implemented using the following code

```
template <uint32_t idx, typename T,
        std::enable_if_t<idx == 3 && !(dimension_v<T> < 3)>* = nullptr>
hostDeviceInline auto&& get(T &&a) {
return a.z;
}
```

Which starts the index at 1 to be the same as the dimension number. This function returns an l-value reference for l-value references and an rvalue otherwise. The function itself takes a forwarding reference and returns a forwarding auto value which can become either an l or r value reference here. The function is only enabled via *SFINAE* if the requested idx is 3, to provide for different versions for 1, 2, 3 and 4 as the index, and if the dimension of the argument is not less than the dimension requested. This reverted test is needed as checking if the dimension is larger to enable it would conflict with using a large value to define non dimensioned values. Without using auto this function would become significantly more complex.

Often times however, it is beneficial to return 0 instead of causing a compile time error, e.g. to allow for easier function creation where the 0 can be used to *filter* out the non existent dimensions. This can easily be realized by copying the definition of get but adding a SFINAE case for dimension $<$ idx. In code this addition could be written as:

```
template<uint32_t idx, typename T,
        std::enable_if_t<(dimension_v<T> != 0)
        && (dimension_v<T> < idx)
        && (dimension_v<T> != 0xDEADBEEF)>* = nullptr>
hostDeviceInline auto weak_get(T a) {
        return static_cast<decltype(std::decay_t<T>::x)>(0);
}
```

In order to avoid type conversions issues the zero is manually cast to the **decayed element type**. In these functions in general a *simple value* instead of a forwarding reference is used to avoid potential divergent behavior in functions which could modify, possibly accidentally, the input values for some dimensions but not for other dimensions and so returning a value regardless of input works well in practice. For scalar values weak_get always *returns the value itself*.

Using the weak_get function a relatively simple **cast function** can be defined to cast between arbitrary vector types:

```
template <typename T, typename U,
                typename std::enable_if<(dimension<T>::value == 2)>
hostDeviceInline T to(U &&a) {
using Ty = decltype(weak_get<1>(std::declval<T>()));
return T{static_cast<Ty>(weak_get<1>(a)),
                static_cast<Ty>(weak_get<2>(a))};
}
```

Which is repeated once for all dimensions. This allows conversions between any arbitrary type including scalar values. For scalar values the cast results in every element of the casted vector having the value of the scalar, for non scalars the dimensions that were not present in the input vector are set to 0.

5.4 Functions on vectors

In general there are two kinds of functions on vectors to consider. The first kind of function takes a single vector as input and applies a **unary function** to all elements of the vector and the other kind being a function which takes two vectors as input and applies a **binary function** to the corresponding elements of the vectors. Functions that modify the elements of a vector could be implemented in theory, however they can easily be emulated using the first kind. For unary functions all inputs are valid, however for binary functions the following relationship with input and output exists, with one vector having n dimensions and the other m The output would be **n**-dimensional if $n == m \vee m == 1$, **m**-dimensional if $n == 1$, and **undefined** else.

Based on these functions a trait can be written which checks if for two input types T and U their output would be d-dimensional:

```
template<typename T, typename U, uint32_t d>
constexpr bool dimension_compatible = /* ... */;
```

Using this a function to apply a binary function to two inputs can be defined, using a helper template to determine the return type,:

```
template <typename T, typename U, typename C,
        std::enable_if_t<dimension_compatible<T, U, 2>>* = nullptr>
hostDeviceInline auto fn(T&& lhs, U&& rhs, C fn) {
  return return_type<T, U>{fn(weak_get<1>(lhs), weak_get<1>(rhs)),
                           fn(weak_get<2>(lhs), weak_get<2>(rhs))};
}
```

And analogous for unary functions. Using these two functions and lambda functions many of the traditional mathematic functions can very easily be implemented, e.g. a function to round values up to the nearest value:

```
template <typename T> hostDeviceInline auto ceilf(T lhs) {
  return fn(lhs, [](auto a) { return ::ceilf(a); });
}
```

5.5 Operators

Normal operators [+,-,*,/,%] can easily be implemented using the get functions:

```
template <typename T, typename U,
        std::enable_if_t<dimension_compatible<T, U, 2>>* = nullptr>
hostDeviceInline auto operator +=(T&a, U&& b){
  math::get<1>(a) += math::get<1>(b);
  math::get<2>(a) += math::get<2>(b);
  return a;
}
```

For non assignment operators the return_type helper is used again and they are implemented the same as normal binary functions. These operators, due to their similarity, are defined based on a macro in the actual framework.

Comparison operators are not implemented for built-in vector types as no definitive interpretation of the comparison of two vectors can be created. The value aware math introduced later however implements these using lexicographical ordering.

Chapter 6

Implementing SI-Units in C++

Many problems nowadays are at least in parts based on *real physical systems*. In physics usually one of the most important steps to writing or developing a method or calculation is making sure the **units are respected**. In computer graphics however it is fairly common for solutions to problems being created *ad-hoc* by employing "scaling" factors that make the system not physically sound. Even ignoring these problems accidentally writing bugs is easy, but some kinds of bugs also cause a difference in units, e.g. in the following example:

```
auto l = 10.0m;
auto v = 5.0m_s;
auto s = 0.1s;
l += l * s;
// should be l += v * s;
```

Where the wrong variable leads to **incompatible units**. The goal in this chapter is to provide the basic overview of the way these units can be enforced in C++ and CUDA. Providing this functionality again helps to build a solid back-end foundation in the framework and allows for very powerful abstractions which, due to the compile time nature of them, come at a zero run time cost.

6.1 Tagged types

In the **SI unit system** only a small number of basic units exist where every other unit is based on a combination of base units and coefficients. The basic units can simply be represented using a *strongly typed enumeration* with the coefficients being represented by a ratio type:

```
enum struct Base { m, kg, s, A, K, mol, cd };
template <Base _unit, typename T = ratio<1, 1>> struct SI_Unit {
  constexpr static Base unit = _unit;
  using ratio = T;
};
```

Where ratio is a struct that stores its template arguments as values:

```
template <std::intmax_t n, std::intmax_t d> struct ratio {
  static constexpr int num = n;
  static constexpr int den = d;
};
```

which implements a ratio $\mathbb{Q}$ based on a nominator and denominator. Using template alias definitions and constexpr functions basic math functions can be implemented on these types within the type system. For example:

```
template<typename R1, typename R2>
using ratio_add = ratio_reduce<
  ratio<R1::num * R2::den + R2::num * R1::den, R1::den * R2::den>>;
inline constexpr auto gcd(int a, int b) {/*...*/}
```

However, this only represents *a single unit* and not a combination of units. A natural choice within C++ to combine these units together is **std::tuple** [ISO C++ [2017] [tuple]]. These tuples can, in addition to storing values of different types, be seen as a way to *list different types* and pass them around together. Additionally certain basic operations, like tuple_cat already exist for them.

Using these a **value_unit** can be defined as a tagged type, which represents an actual value with a normal type combined together with its unit as:

```
template <typename Value, typename Unit> struct value_unit {
  Value val;
  using unit = Unit;
  using type = Value;
  constexpr hostDevice value_unit() { val = vec<Value>::zero(); };
  template <typename... Ts>
  constexpr explicit hostDevice value_unit(Ts... args) : val{args...} {}
  template <typename T, typename U>
  constexpr hostDevice value_unit<Value, Unit>(
      const value_unit<T, U> &rhs,
      std::enable_if_t<SI::is_same_unit<Unit, U>::value, int *> = 0) {
    val = rhs.val;
  }
  constexpr explicit hostDevice operator Value() const { return val; }
};
```

Instances of this class template need certain **constraints** with regards to their constructors to avoid accidentally creating objects of these classes based on wrong physical units. Sometimes accessing the value directly is still useful so the variable is made accessible to the outside as it is a *public member*. In the actual implementation a separate value_unit is used to handle units with *Unit = void*.

6.2 Operations on units

For the **basic units** a simple template alias can be used to turn the enumeration into SI_Unit entries:

```
using m = SI_Unit<Base::m>;
```

And for **derived units** a *variadic template alias* derived_unit<Ts...> can be used. In order to help with these derived units a set of helper aliases exists, e.g. for reciprocal units:

```
template <typename T> using recip = multiply_ratio<T, ratio<-1, 1>>;
template <typename T> using recip_2 = multiply_ratio<T, ratio<-2, 1>>;
template <typename T> using recip_3 = multiply_ratio<T, ratio<-3, 1>>;
template <typename T> using id = multiply_ratio<T, ratio<1, 1>>;
template <typename T> using square = multiply_ratio<T, ratio<2, 1>>;
template <typename T> using cubic = multiply_ratio<T, ratio<3, 1>>;
```

Which can be used to define a set of common derived units, e.g. $\mathrm{N} = \frac{\mathrm{kg \cdot m}}{\mathrm{s}^2}$ as:

```
using N = derived_unit<kg, m, recip_2<s>>;
```

Such a derived unit, even with just a single unit, is a tuple of units. In order to use these units however a set of functions needs to be created to allow for certain basic functionality. As an example of these utility functions **filter_unit**'s implementation is shown below which is used to *remove certain basic units from a tuple*:

```
template <Base B, typename Tuple> struct filter_unit;
template <Base B>
struct filter_unit<B, std::tuple<>> { using type = std::tuple<>; };
template <Base B, Base C, typename U, typename... Ts>
struct filter_unit<B, std::tuple<SI_Unit<C, U>, Ts...>> {
  using type = decltype(
    std::tuple_cat(std::declval<std::tuple<SI_Unit<C, U>>>(),
      std::declval<typename filter_unit<B, std::tuple<Ts...>>::type>()));
};
template <Base B, typename U, typename... Ts>
struct filter_unit<B, std::tuple<SI_Unit<B, U>, Ts...>> {
  using type = typename filter_unit<B, std::tuple<Ts...>>::type;
};
```

This implementation looks intimidating at first, however when breaking it down into separate blocks it becomes fairly straight forward. filter_unit is a template struct which requires a base unit B and a tuple it works on. The first version defines a basic struct which can be used for specialization and the second version defines the variadic template recursion anchor. The third version concatenates a tuple based on the current SI_Unit with the value returned by a recursive call. The fourth version is only allowed if the first SI_Unit has the same basic unit as the SI_Unit that should be filtered out, in which case the type is set to just be the result of the recursive call without tuple_cat.

There are other basic functions like *has_unit* which is of true_type if the given derived unit contains a certain base unit.

A more interesting utility is **equivalent_types**. This set of templates, which is too long to show here, compares two types and checks if they are equal. Conceptually this is implemented as a recursion on one template checking for the value of the same basic unit in the other template. The types are only equivalent if every basic unit of a is equal to the same basic unit in b ignoring ordering. Additionally is **compatible_unit** returns true if they are equivalent or one, or both, are void. This is useful to allow operations that scale a unit type by a standard float, e.g.

```
auto l = 0.5f * 1._m;
```

6.3 Utility things

The value_unit defined before works fine, however often times it is too verbose to use. Instead a simple *alias template* can be defined for the most commonly used types e.g.:

```
template<typename... Ts>
using float_u = value_unit<float, SI::derived_unit<Ts...>>;
```

Similarly to directly create scalars with a unit attached to them **user defined literals** [ISO C++ [2017] [lex.ext]] can be defined. These UDLs work similar to *f* for floating point numbers, however to be conforming to the standard they need to start with an underscore. They can be implemented using a macro for all basic, and analogous for derived units:

```
#define UNIT_UDL(u) \
constexpr value_unit<float, SI::u> operator "" _##u(long double wgt) \
      {return value_unit<float, SI::u>{static_cast<float>(wgt)};}
UNIT_UDL(m);
```

6.4 Functions

Functions on types with units can be defined as they would be on normal math functions, e.g. the ceilf function from before can be implemented as

```
template<typename T, typename U>
hostDeviceInline auto ceilf(value_unit<T, U> arg) {
  return value_unit<T, U>{math::ceilf(arg.val)};
}
```

Which takes the actual value of the unit and passes it to the standard function. Of note here is that *no forwarding reference* can be used on these derived types. In this case there is no influence on the SI-Unit associated with the variable so this is all that is needed. Similarly get functions can be defined that work the same as the unit-less versions, however the element *has to be casted* to a unit based type. This has the consequence that get on value_unit's does not return a reference but a value.

To still assign values to individual elements a separate **unit_assign** function is defined. Which implements assigning a scalar to a 1D vector value. This obviously leads to many different specializations of this function which makes it very long in code but conceptually relatively straight forward.

Some functions have *no correspondence* with unit systems, e.g. sinus functions, as there is no way to modify a type to be for example *sin(SI::m)*. Similarly the **power function** is restricted to *compile time fixed powers* as the unit checking needs to run at compile time. Thus the power function for units is defined as:

```
template<typename Ratio, typename T, typename U >
hostDeviceInline constexpr auto power(value_unit<T, U> a) {
  constexpr float n = static_cast<float>(Ratio::num);
  constexpr float d = static_cast<float>(Ratio::den);
  constexpr float r = n / d;
  using ret_t = value_unit<T, typename SI::multiply_ratio<U, Ratio>>;
  return ret_t{ math::pow(a.val, r) };
}
```

Where the return type is calculated based on the ratio that is provided as the first template argument. Operators can also be defined on these types, however due to the complexity of unit compatibility and how they interact they become relatively complex. An example of this would be division where dividing a number by a number with an associated unit creates a result with the reciprocal unit of the second argument, however dividing a number associated with a unit by a normal number changes nothing. These rules follow ordinary unit rules most people are familiar with so providing them here in code would serve no greater purpose.

6.5 SPH kernel functions

In the section dealing with the SPH fundamentals **kernel functions** were described which are used here as an example of more complex mathematical functions.

As a basis for kernels a base class, which provides a set of functions to interact with the kernel, is used which a specific kernel inherits. An individual kernel then provides the following *interface*:

- **neighbor_number**, a constant value representing the ideal number of neighbors for this kernel, defined per kernel

- **kernel_size()** returnes κ for a given kernel, defined per kernel
- *gradient(a,b)* returns ∇W_{ab}
- *derivative(a,b)* returns $\frac{d\hat{W}(q)}{dq}$
- *norm_derivative(a,b)* returns $\frac{d\hat{W}(q)}{dq}\frac{1}{||\mathbf{x_a}-\mathbf{x_b}||}$
- **value(a,b)** returns W_{ab}, defined per kernel
- **derivative_impl(q)** returns $\frac{d\hat{W}(q)}{dq}$, defined per kernel

The actual implementations of the kernel functions are almost *agnostic* to whether or not unit based types are used to call them. The only exception to this is an optimization that can easily be done to SPH simulations where *positions* are stored as 4D values, for alignment reasons, where the fourth component stores the *smoothing length* of the particle. To access the fourth component with unit types unit_get has to be used and for non unit types just get. This is wrapped in a separate helper function calculate_support which uses SFINAE to pick the right version. The cubic spline kernel function itself can then be implemented as:

```
template <typename T, typename U> hostDeviceInline
auto spline4_kernel(T a, U b) {
  auto difference = a - b;
  auto r = math::length3(difference);
  auto half = calculate_support(a, b);
  auto H = half * kernel_size();
  auto q = r / H;
  auto C = 16.f / CUDART_PI_F;
  auto kernel_scaling = C / (H * H * H);
  auto kernel_value = 0.f;
  if (q <= 0.5f) {
    auto q1 = 1.f - q;
    auto q2 = 0.5f - q;
    kernel_value = (q1 * q1 * q1) - 4.f * (q2 * q2 * q2);
  } else if ((q <= 1.0f) && (q > 0.5f)) {
    auto q1 = 1.f - q;
    kernel_value = q1 * q1 * q1;
  }
  return kernel_value * kernel_scaling;
}
```

In order to make the code more readable certain helper macros are defined using common kernel calls to allow for more concise code, e.g. *W_ij* for *spline4_kernel(pos[i], pos[j])*.

Chapter 7

Meta-Modeling for simulations

A general problem, and what motivated this development effort, was managing **dependencies**. In most systems, and especially simulations, there are often different ways to solve a problem, e.g. different pressure solvers. These methods interact in certain ways where often times the *output* of one **module** is used as the *input* of another, where certain modules are used to process certain data etc. This often leads to practical problems as defining what something depends on, or even making all information available in *a single place*, is often difficult with a spread out implementation.

The goal here is to create a system in which modules can be defined in a centralized manner and this information can be used to create **meta information** about the program, memory and parameters in a central location where looking up properties and dependencies is easily possible. These central files are written in a *mostly human readable format* instead of actual C++ *source code* which significantly helps understanding.

This central information is stored within **JSON** (Java Script Object Notation) files as this format strikes a reasonable balanced between human readability and expressiveness.

7.1 Using JSON to describe data

The following is an excerpt from the JSON file used to create configuration parameters from the framework and should serve as an example of the structure of JSON itself:

```
{
  "modules": {
    "resorting": {
      "identifier": "sorting",
      "description": "Used to select the underlying sorting algorithm.",
      "type": "std::string",
      "default": "hashed_cell"
    }
  }
}
```

Which describes the resorting *parameter* which in itself is an entry *within modules*. This parameter has a certain *identifier*, a helpful *description*, a *default value* and a C++ *type*. In order to translate this into actual C++ source code a program is required that can execute a **source code transformation**, or as a more general problem, compiles JSON to C++. This obviously is no generic programming language but a **limited domain specific language**.

To read a JSON file within C++ an external library is helpful as recreating the logic of *parsing* a JSON file into some *abstract tree* representation is not the topic of this thesis. In order to load the JSON file **boost** is used due to its ubiquitous nature. In order to actually load a JSON file in boost the file first is read into a standard string and then transformed into a property tree:

```
std::string json = "...";
boost::property_tree::ptree pt;
boost::property_tree::read_json(json, pt);
```

Once this tree has been created a **node** class is used which represents a property tree node combined with its value as a string. This node can then be used to define a **transformation** from the corresponding JSON node to **C++ code artifacts**. The actual implementation of this logic is not important as it could be done in a variety of ways and the general concept is more important. Actually using this node class however is more relevant as this might be something a user of the framework would consider modifying to fit some new requirement.

The most simple transformation possible is a transformation that simply *recursively* parses the JSON tree, which is stored in a **std::function** object with a default lambda closure assigned to it:

```
std::function<void(transformation&, transformation*, node_t&)> transfer_fn =
[](auto& node, auto, node_t& tree) {
  for (auto& t_fn : node.children)
          for (auto& children : tree.second)
                  t_fn->transform(children);
};
```

Once this recursive step has been done on the highest level additional transformations can be created for the individual nodes which in the JSON sample data above would be the actual resorting parameter node. The *output* of these transformations in general is based on some **pattern** where *placeholders* are replaced to insert the specific information for this parameter or array. These patterns, once filled out, are pushed into string streams representing the actual source code to iteratively build the actual files.

In addition to these string based operations certain steps require **lists** to be created, e.g. a list containing all parameters called uniforms_list, which are created by pushing back elements into a *global map* which maps list identifiers to list entries in string form:

```
std::map<std::string, std::vector<std::string>> global_map;
global_map["uniforms_list"].push_back(
  node.second.template get("identifier", std::string(parent->node->first))
);
```

These are transformed into code after all nodes have been processed and are thus placed at the bottom of the file. In the following subsections the basic patterns for parameters, arrays and modules are shown to demonstrate how these can be used in practice and what kinds of inputs are expected.

7.2 Array creation using JSON

Creating **arrays** has certain requirements that have to be met for this system to be useful. An array in general, besides a type, has various interesting **properties** which, based on their JSON attribute name, are the following:

- the name of the **node** is used as the name of an array, or the variable name more commonly
- **type**, contains the underlying C++ type
- **unit**, contains the SI unit of the array, based on the unit node containing an SI unit, e.g. SI::m or SI::derived_unit<SI::area, SI::recip_2<SI::s»
- **kind**, within our SPH framework there are 4 kinds of allocation types that are allowed. These are:

  particleData, one element allocated for every particle in the simulation

  singleData, a single element allocated

  cellData, one element allocated per grid cell

  customData, no default allocation
- **size**, contains the number of elements allocated per element, e.g. a size of 2 using particleData allocates 2 array entries per particle
- **swap**, contains a boolean attribute that describes if the array has a front and rear buffer used for updating values or resorting
- **resort**, indicates whether or not the array, only if it contains particleData, should be resorted in every timestep
- **depends_any**, indicates possible dependencies on configuration parameters. These dependencies are arrays of parameter : value pairs
- **depends_all**, same as depends any but requires all conditions in the array to be true. Exclusive with depends_any

An example JSON entry would be the following:

```
{
  "basicArrays": {
    "position": {
      "description": "...",
      "type": "float4",
      "kind": "particleData",
      "unit": "SI::m",
      "size": 1,
      "swap": true,
      "resort": true
    }
  }
}
```

This entry describes the position array which contains particleData and thus one element per particle in the simulation. The unit of this arrays data is SI::m. The array also contains a back buffer used for updating values and is resorted every timestep. If the entry also contained

```
{
  "depends_any": [
    { "pressure": "IISPH" },
    { "pressure": "IISPH17" }
  ]
}
```

Then the array would only be allocated if the parameter *pressure* had either the value *IISPH* or *IISPH17* within a configuration.

There are some additional details that are not directly obvious in these descriptions. The swap attribute not only creates a back buffer but also makes the data **ephemeral**, which means that the memory associated with the array is never fully reclaimed so the data is kept intact between timesteps. On simulation startup only arrays marked as *swap* and *single* data are allocated, where the following might be a relatively common solution to allocating custom data fields on startup which are kept throughout the simulation:

```
{
  "kind": "singleData",
  "size": "get<parameters::mlm_schemes>() * get<parameters::hash_entries>()"
}
```

Note that size can contain arbitrary C++ **expressions** as long as they evaluate to a number. Another important distinction has to be made on the unit attribute. This attribute allows for a special case where the unit of the array is declared **none**. If an array is declared none then there will be *no associated value_unit pointer* with the array and instead the only way to refer to the data is using the type directly. This is different to the array being declared void which creates a value_unit<T,void> which might not be desirable for certain kinds of arrays, e.g. particle indices or hash maps.

The general replacement form for an array creates an individual type per identifier in a header file:

```
struct $identifier{
        using type = $type;
        using unit_type = $unit;
  static constexpr const array_enum identifier =
                                            array_enum::$identifier;
        static constexpr const auto variableName = "$identifier";
        static $type* ptr;
        static size_t alloc_size;
        static constexpr const auto description = "$description";
$swap_h
        static constexpr const memory_kind kind = memory_kind::$kind;
$default_alloc_h
        static void allocate(size_t size);
        static void free();
        operator type*();
        type& operator[](size_t idx);
        static bool valid();
        template<class T> static inline auto& get_member(T& var) {
    return var.$identifier;
  }
};
```

7.3 Parameter creation using JSON and configurations

Parameter descriptions face a similar problem to arrays, but the benefits of centralizing their definitions is even more valuable. There are not many reasons to find out what arrays exist in the simulation except to write code, whereas finding out what configuration parameters exist and what they mean has benefits even to users of the simulation. As such *automatically* generating a user interface in the simulation and making this information available to users without having to scour source files is beneficial.

The general principle of parameters is equal to that of the arrays, however there are additional attributes for parameters. The attributes of parameters are:

- **identifier**, is used as the name of the C++ struct

- the name of the **node**, is combined with the name of the node one level up to create the JSON name of the parameter
- **unit**, same as arrays
- **type**, same as arrays, however std::vector<Type> is also allowed
- **visible**, a boolean parameter that makes this parameter invisible to the GUI if set to false
- **const**, boolean parameter which makes the value unmodifiable at runtime through the GUI
- **default**, a C++ expression used to initialize the parameter, initialization works as type vdefault
- **range**, a sub JSON node which contains a min, max and step entry to define slider ranges for the GUI, if desired
- **depends_any**, **depends_all**, same as arrays

There are certain special behaviors of parameters that are worth pointing out. First of all the simulation creates a JSON identifier from the parent node name and the name of the node which are used to automatically load a value from a *configuration* as a string value. This parsing is created automatically and works for a variety of types. These include: bool, integral types, floating point types, CUDA vectors, std::vector and std::string.

CUDA vectors are parsed as *comma separated values*, e.g. "default": "1.f,0.f,0.f,0.f" for a 4D float value of $[1, 0, 0, 0]$. For std::vector the following would be an example entry in the parameter file where the individual vector entries are sub nodes in a JSON object:

```
{
  "inlet_volumes": {
    "volume$": {
      "identifier": "inlet_volumes",
      "type": "std::vector<std::string>"
    }
  }
}
```

With an example **configuration file** entry of:

```
{
  "inlet_volumes":{
    "volume1":"...",
    "volume2":"..."
  }
}
```

Where the number is indicated by a **$** symbol in the identifier. Additionally parameters can define **complex types** which are implemented as PODs with complex_entry members. These are parsed for each of the entries, for example in the prior example the actual type is std::vector<inlet_volume> which is defined as:

```
{
  "complex_type": {
    "name": "inlet_volume",
    "description": {
      "file": {
        "identifier": "fileName",
        "type": "std::string",
        "default": ""
      }
    }
  }
}
```

Where an entry in the configuration would be

```
{
  "volume1": {
    "file": "Volumes/Inlet1.vdb"
  }
}
```

These complex types are also defined within the header that contains all parameters and is accessible across the simulation. However for actual CUDA code they should *manually* be transformed into arguments as passing along these complex types automatically to GPU code would be difficult and restrict their usefulness.

The **configuration parsing** in general works **fully automated** and allows for easy creation of new parameters. Because of this automation, and the configuration being a JSON file itself, JSON attributes can be provided on the command line, e.g. calling

```
sim -j=modules.resorting="hashed",simulation_settings.max_numptcls=1024
```

Would set these parameters and overwrite any value within the simulation allowing for easy creation of test cases. To further this the configuration can also contain **options** which contain a set of configuration parameters. These options can be accessed when starting the simulation using

```
sim -o \$num
```

An example of an option would be the following:

```
{
  "options":[
    {
      "modules":{
        "resorting": "linear_cell"
      }
    }
  ]
}
```

The general replacement form for a parameter creates an individual type per identifier in a header file:

```
struct $identifier{
  using type = $type;
  using unit_type = $unit;
  static constexpr const uniforms identifier = uniforms::$identifier;
  $identifier(const type& val){*ptr = val;}
  $identifier() = default;
  operator type() const{return * ptr;}
  static constexpr const auto variableName = "$identifier";
  static $type* ptr;
  static $unit* unit_ptr;

  static constexpr const auto jsonName = "$json";
  static constexpr const bool modifiable = $constant;
  static constexpr const bool visible = $visible;

  template<class T> static inline auto& get_member(T& var) {
    return var.$identifier;
  }$range_statement
};)";
```

7.4 Module creation using JSON

Functions, or more generally here **modules**, are significantly different to parameters and arrays, but they follow the same structure. A module in this framework consists of a *namespace*, with the name of the module, a function to check if the module is *valid*, a *structure* describing the arrays and parameters used by the module and the actual *functions*. All of these components are automatically generated in a generated header file *per module* which the implementations of the functions can simply include. The functions themselves take *a single parameter*, which is the memory defined within the modules namespace, which is filled out by the framework with all necessary pointers and parameters. To get a better understanding of this structure consider the general replacement form:

```
#pragma once
#include <utility/identifier.h>
/*
$description
*/
namespace SPH{
  namespace $name{
    struct Memory{
    // basic information$basic_info
      // parameters$parameter
      // temporary resources (mapped as read/write)$temporary
      // input resources (mapped as read only)$input
      // output resources (mapped as read/write)$output
      // swap resources (mapped as read/write)$swap
      // cell resources (mapped as read only)$cell_info
      // neighborhood resources (mapped as read only)$neighbor_info
      // virtual resources (mapped as read only)$virtual_info
      // volume boundary resources (mapped as read only)$boundaryInfo
      $using
      $properties
    };
    //valid checking function
    inline bool valid(Memory){
      $valid_str
      return condition;
    }
    $functions
  } // namspace $name
}// namespace SPH
```

Some basic values are fairly uninteresting as they are the same as those from parameters and arrays, namely depends_any and depends_all, as well as a description attribute and a name. In addition to the name a module also contains a folder attribute which names the subfolder in which the module will be generated. The functions attribute describes the name of all functions as an array that the module provides. The remaining parameters describe the input and output of the module:

- parameters, JSON array with the name of all parameters required by the module
- input, JSON array with the name of all arrays required by the module, read only

- output, JSON array with the name of all arrays that are intended outputs of the module, read/write
- temporary, JSON array with the name of all arrays that are used only internally, read/write
- swap, JSON array with the name of all arrays that should be bound as a pair of front/rear buffer, read/write
- x-info like attributes describe sets of arrays and parameters that can be included implicitly, e.g. all arrays required for a neighborhood iteration
- units is a boolean that defines whether or not arrays and parameters should be bound as types with associated units
- target can either be cuda or cpu to decide whether a .cpp or .cu file should be generated

An example would be the module description for IISPH:

```
{
  "iisph17": {
    "description": "...",
    "folder": "pressure",
    "name": "IISPH17",
    "target": "cuda",
    "units": true,
    "cell_info": false,
    "virtual_info": true,
    "boundaryInfo": true,
    "neighbor_info": true,
    "functions": [ "pressure_solve" ],
    "depends_any": [ { "pressure": "IISPH17" } ],
    "parameters": [ "eta", "iterations", "density_error"],
    "input": [ "position", "density", "volume" ],
    "temporary": [ "iisphVolume", "iisphSource", "..." ],
    "output": [ "acceleration", "velocity" ],
    "swap": [ "pressure" ]
  }
}
```

In addition to the arrays being bound into variables that have the same name as the array the module memory structure also contains additional meta information. This mostly includes lists of all parameters bound and all arrays bound as input, output, temporary etc. This information is used to fill out an instance of the memory object automatically before the function is called so the function itself doesn't have to worry about these problems. This process will be discussed more in-depth in the next section.

Chapter 8

Module and memory management

So far only descriptions of parameters, arrays and modules have been created and the challenge now is to bring these descriptions, with all of their meta information, together in a **system** that automates many of the traditionally busy work involving steps. These steps are done fully automated as to not require user influence at any point of the actual function calling, but the user still has to add **module calls** in the right spot. Additionally, whilst it is not important to the average user, it is still important to understand how the **memory manager** conceptually works, and how in more general terms memory is allocated.

8.1 Adding modules

Within this framework a central simulation object exists which carries a list of module pointers in the order in which they are called in each *simulation step*. This object is setup within the setup_simulation function where an empty simulation "loop" is created and modules are added using a **then** function. Internally this uses a tuple, as the modules all are of different types to avoid performance penalties from using inheritance, to which modules are appended. The syntax of this .then call in general is:

```
template <typename T> simulation<Ts..., array_clear<T>>
then(array_clear<T> arr);
template <typename T> simulation<Ts..., function_call<T>>
then(void(*func)(T), std::string name, Color col, bool graph) ;
```

Where the first overload is used to *clear an array* as a simulation step, e.g. setting all accelerations to zero would be .then(array_clear<arrays::acceleration>()). Adding an actual module call is done using the second version of .then which takes a *function pointer* to the actual module function being called, and some helpful information, which is used to create a timer, and possibly error messages within the GUI. An example call to the function estimate_density of the Density module would be .then(&SPH::Density::estimate_density, "Density estimate", Color::orange4).

Overall the current version of the simulation framework contains 28 individual then calls. One thing to note about this approach of building the simulation loop is that using a tuple in this way tends to be relatively *slow to compile* and the simulation struct can cause certain syntax highlighting systems to not highlight any code if the header of the simulation struct is included. This however is a fairly minor problem and not serious enough to warrant using a less versatile approach as this header is only included in a single file whose sole purpose is compiling this header.

8.2 Preparing a module call

When a module is called using the simulation construct the framework has to step in and do a few things. The first and most obvious step is checking if the module call is even *valid*. This could be relatively tricky as only the function pointer to the actual function and not the call to the valid function is given. However, there is a C++ feature which allows us to do the following

```
template<typename T>
void adl(void(*func)(T)){
  T val;
  if(!valid(val)) throw std::exception(/*...*/);
  //...
}
```

Due to the function pointer to a module containing the *type* of the memory for this module an instance of this memory class can be created. Using this instance and **Argument Dependent Lookup** [ISO C++ [2017] [basic.lookup.argdep]] it is possible to call functions within the namespace of parameters implicitly. Due to the valid functions being defined in an automatic way, which guarantees them being in the *same namespace* as the memory struct, it is possible to do this reliably for all modules.

If this valid check fails the function simply returns, however if the valid check succeeds the memory object needs to be filled out with the appropriate values for parameters and pointers for arrays. In general it would be possible to leave the memory for all arrays to be allocated at all times, making this trivial to implement, but this would increase memory consumption significantly which in a GPU based simulation is not acceptable if it can be avoided.

In order to assign the array pointers a **memory manager** is used which keeps track of allocated arrays in a **pool** of allocations. Before the function is called *prepareMemory* is used to fill out the memory object and after the call *clearMemory* is used to return no longer needed memory to the pool. At the end of each simulation step an additional method called *reclaimMemory* is called. All three of these functions will be discussed in the following sections.

8.3 The memory manager

Due to the meta generation of arrays only a limited number of different *flavors* of arrays exist. These arrays are different in their intended purposes and sizes, with some of them being allocated outside the *sphere of control* of this manager. Most arrays are allocations which are taken from an internal pool of allocations by the memory manager which are represented using the following struct:

```
struct Allocation {
        size_t allocation_size = 0;
        void *ptr = nullptr;
        bool inUse = false;
        std::string last_allocation = "";
};
```

Where new allocations made by the manager are added to a *vector* of all allocations. The different kinds of arrays, as seen before in the meta-modeling sections, are handled here as follows:

- **customData** are the most straight forward kind of memory as they are based on some user defined allocations, which are outside the control of the memory manager, so no management needs to be done for these.
- **singleData** are treated as memory allocations which are allocated at simulation startup and never returned to the memory manager as their sizes are either too small to re-use (if they are truly single element arrays) or they have lengths which do not make their allocations something that can be reused except for anything but the specific array itself.
- **cellData** are always of the same basic size and do not contain any persistent data. As such it might make sense to allocate these arrays when they are first accessed using the pool.
- **particleData** are more complex. If an array is marked as swap the array is considered to contain persistent data so there is no need to allocate the memory in a pool, for the front array, as it will never be available for other uses. However, if an array is not marked as swap the memory is allocated using the pool as this data is not persistent.

Memory that is not managed can just be assigned to the entries in the object used to call a module and does not present a challenge. Memory that is allocated using a pool can have a few different states based on the intention for the specific module and whether or not the array has memory assigned to it (meaning the pointer in the array structure is not null). If the member of the structure is null the following things have to happen:

- **output**: A free allocation is searched for in the pool, and if no fitting memory is found a new pool entry is allocated. After the function call the memory is not returned to the pool.

- **temporary**: Are treated the same as output but the memory is returned to the pool.
- **swap**: The rear pointer of the array is treated as a temporary.
- **input**: An exception is raised as there is no memory to be used as an input in a reasonable way.

If the member of the structure is not null the following things happen:

- **output**: The pointer is assigned to the memory instance.
- **temporary**: Same as output but the allocation is returned to the pool.
- **swap**: The rear pointer of the array is treated as a temporary.
- **input**: Same as output.

The important thing to note is that if something is used as a temporary by a function, even though it is the output of something before and the input of something later the memory is still returned to the pool. This makes memory management significantly easier and forces the user to not just throw everything into something as a temporary.

8.4 Preparing memory

The prepareMemory function is fairly trivial considering the previous section, however there are still some noteworthy points. The prepareMemory function also assigns all *parameter* values to the structure, as they were at the time of the module call. Additionally if a module is marked as a *resorting* module all rear pointers of all valid swap arrays are allocated. And finally if an array is not valid (based on the arrays valid function) a warning is generated but no exception is raised as sometimes it might be easier to allow for optional inputs for testing as making arrays depend on modules that might not be used.

8.5 Clearing memory

The clearMemory function is similarly trivial considering the conditions described in the memory manager with the same additional behavior that all rear pointers of all valid swap arrays are returned to the pool in a *resorting* function. An additional important note is that whilst the values at the call of the module of parameters are set in the memory object, changes to these *parameters* are not reflected back onto the global parameter value to avoid unintentional side effects. Additionally two arrays are never cleared. These two are the density array, as this is used as a fallback array for *visualizing* something, and the array used for visualization is also not cleared as the value needs to stay around even if the array is just temporary.

8.6 Reclaiming memory

Using the memory manager described before memory is allocated on the fly using the pool for non persistent data. This data, due to the nature of it not being persistent, is no longer needed after a timestep finishes and as such a*ll pool allocations are returned* to the memory manager in the reclaim memory step. However, the same restriction to the density array and the array being visualized as with clearing memory apply.

8.7 Unified memory

All of the previous discussion of the memory manager makes no distinction whether or not a module is marked as CUDA or CPU code. This distinction is not necessary for CUDA as memory can easily be allocated that is valid on *both host and device* as:

```
T* ptr = nullptr;
cudaError_t err = cudaMallocManaged(&ptr, sizeof(T) * N);
```

Which allocates a block of **unified memory**. This memory automatically is transferred between host and GPU contexts when needed which significantly simplifies writing code. This is one of the biggest benefits of using CUDA over for example OpenCL where this transition has to be done manually. An example of where this would be useful would be in *debugging* where in between CUDA kernel calls (and adding a cudaDeviceSynchronize) the results can be checked for errors or for correctness using a CPU loop which is much more easily debugged and created without worrying about parallelism and GPU execution. Even though the transfer is automatic, the transfer is *not free*. This cost associated with the transfer however only occurs when the context is switched mid simulation which usually is not the case except for debugging or verification where the added overhead is acceptable.

8.8 Shared memory

Shared memory for CUDA is often a point where a lot of optimization can be found, and this is no different for SPH code. Whilst certain advanced optimizations are sometimes possible one optimization that is always possible, even for non SPH functions, is using the shared memory as an **additional level of cache**. This method is similar to a *tiling* algorithm where a section of data is loaded into shared memory, where in this framework one element per thread is loaded into shared memory.

For example when using shared memory to cache the position array for threads n to $n+m$ a shared memory allocation of $m *$sizeof(float4) is required. In this shared memory array the entry i contains the global entry $n+i$ with $0 \leq i < m$. Using two caches, and a **dynamic shared memory** array, the second cache would require an **offset** equal to the size of the first cache. Using this logic the following implementation could be written:

```
extern __shared__ float sm_cache[];
__global__ void some_fn(float* A, float* B, float * C, int32_t N){
  int32_t i = threadIdx.x + blockDim.x * blockIdx.x;
  if(i >= N) return;
  float* cacheA = sm_cache;
  float* cacheB = sm_cache + sizeof(float) * blockDim.x;
  cacheA[threadIdx.x] = A[i];
  cacheB[threadIdx.x] = B[i];
        __syncthreads();
  int32_t j = /* some other index */;
  int32_t block_offset = blockDim.x * blockIdx.x;
  float a_j = ( j > block_offset) && ( j < block_offset + blockdim.x)
    ? cacheA[j - block_offset] : A[j];
  float b_j = ( j > block_offset) && ( j < block_offset + blockdim.x)
    ? cacheB[j - block_offset] : B[j];
  C[i] = a_j + b_j;
}
```

Which can be significantly simplified by using the **cache class** which results in, ignoring the function signature and thread calculation:

```
  auto cacheA = cache_array(A);
  auto cacheB = cache_array(B, cacheA.offset);
  int32_t j = /* some other index */;
  C[i] = cacheA[j] + cacheB[j];
```

Where all of the complexity has been abstracted away. The cache internally still uses the dynamic shared memory but has hidden this away where everything except calculating the offset is done automatically. The cache class also realizes a CPU compatible access function where the same cache class is used but with the difference that no caching is actually done. One thing to note about this access is that the internal decision whether or not to use the shared memory leads to **divergence** which cannot be avoided.

This can be further improved upon by using a, very complex to implement, macro called **cache_arrays**, which simplifies the previous example into the following, as long as the memory object is called arrays, where the macro checkedThreadIdx(i) is a simple macro which calculates the thread index and checks if this is less than the threads parameter.

```
struct memory{float *A, *B, *C; int32_t threads;}
__global__ void some_fn(memory arrays){
  checkedThreadIdx(i);
  cache_arrays((cachedA, A),(cachedB, B));
  int32_t j = /* some other index */;
  C[i] = cacheA[j] + cacheB[j];
}
```

A similar macro **alias_arrays** also exists, which simply creates short hand pointers to help with the readability in certain complex functions. The implementation of these macros uses recursion solely using pre processor expressions, which require different implementations for Windows and Linux due to different interpretations of the C standard, with respect to variadic macros. The boost library could be used to reduce this complexity, however boost explicitly blocks out the preprocessor macros when it detects a CUDA compiler.

Chapter 9

Function calling

In a generic program calling a function is fairly straight forward, however in a complex simulation this can become difficult due to various versions of each function with different underlying algorithms, data structures etc. Managing assigning pointers from arrays has already been covered, so this section will cover how different versions of actual functions are chosen and kernels in general can be created and called. This builds an essential part of the back-end functionality of the framework as it allows CPU and GPU execution as well as easy management of variability and certain classes of methods.

9.1 Occupancy

An often faced problem in CUDA code is figuring out what **thread configuration** to use for a function. A traditional solution to this problem would be to either go with a value assumed to be good or use the **occupancy** calculator, which is an excel spreadsheet provided by NVIDIA. However, CUDA provides a built-in function to figure out the ideal (regarding occupancy) threads per block size:

```
// defined in cuda_runtime.h
template <class UnaryFunction, class T>
__host__ cudaError_t cudaOccupancyMaxPotentialBlockSizeVariableSMem
  ( int* minGridSize, int* blockSize, T func,
    UnaryFunction blockSizeToDynamicSMemSize, int  blockSizeLimit = 0 )
```

Which calculates the block size to achieve *maximum occupancy* for functions where dynamic shared memory is used. The first and second argument are the actual outputs of the function as pointers. The third argument is a function pointer to the kernel function, the last parameter gives an upper user defined limit on the block size. The interesting argument is the **unary function**. This parameter is a template argument that expects a function which takes the number of threads per block as an argument and returns the shared memory usage based on this number of threads.

For example in a function that was using two floating point entries per thread in a block the unary function would be:

```
auto fn = [](int32_t blockSize){return blockSize * (sizeof(float) * 2);};
```

Defining these functions manually however is not ideal as changing something about the calculation would now require changing every call. Instead a **caches template** is added to the framework which simplifies the prior example into:

```
auto fn = caches<float,float>{}
```

Where caches represents a functor where the function call operator simply multiplies the argument by the *added size* of the template arguments. This is implemented in the usual fashion of a recursive variadic template and not very interesting here. Calculating the occupancy thus can be encapsulated into an easy to interface function with a significantly shorter name.

9.2 Picking a template to call

In a simulation framework often times multiple algorithms exist to solve some underlying problems and SPH is no different. For example multiple neighborhood iterations exist and switching with if statements in the kernel function itself might not be the right choice as this increases the code complexity and size significantly. Instead of this approach **partial template specializations** can be used which represent an interface that can provide efficient implementations for various algorithms without creating unnecessary code. An example of this would be:

```
template<neighborhood neigh>
__global__ void fn(Memory arrays){
  //...
  interpolate<neigh>([&](int32_t j){/*...*/});
  //...
}
```

This however creates a problem as the kernel function no longer is a simple function but a template, and C++ does not allow passing around *un-instantiated template functions*, only un-instantiated template classes. As such the kernel could be **wrapped** within a functor:

```
template<neighborhood neighbor_list, typename... Vs>
struct functor{
  template<typename... Ts>
  static void launch(int32_t threads, Ts... args){
    // get configuration
    fn<neigh, Vs...><<<blocks,tpbs>>>(args...);
  }
}
```

Which just passes along the call and calculates the proper configuration for this template. The actual implementation of this functor is significantly more complex and this here just represents a conceptual model. Do note however that this template requires two separate template parameter packs. The outer template parameter pack (Ts) represents the types of the actual arguments to the function and the inner template parameter pack (Vs) is passed along to the function which might require explicit template parameters.

Using this functor a function can be created to launch these functors which also chooses the correct template instantiation to call:

```
template < template <neighbor_list, typename...> class T,
              class... Vs, class... Ts>
void launch(int32_t threads, Ts... args) {
  if (parameters::neighborhood_kind{} == /* ... */)
    // ...
  else
    / ...
}
//...
launch<functor>(threads, ...);
```

The template template argument here allows passing along non specialized class templates where the function itself can create the actual instantiation. The template that needs to be specified here is just the *basic template* name without any template parameters applied to it. These launch functions exist for various different kinds of choices, e.g. neighborhood iterations or cell structures, and are wrapped up in a set of launch functions and macros. The above example within the framework for example would be:

```
neighborFunctionType fn(Memory arrays){
  //...
  interpolate<neigh>([&](int32_t j){/*...*/});
  //...
}
//...
neighFunction(functor, fn, "Example function", caches<float, float>{});)
//...
launch<functor>(threads, arrays);
```

Where the neighborFunctionType = "template<neighborhood neigh> __global__ void" and neighFunction wraps up the functor inside of a macro. This macro takes the name of the functor to be created as the first argument, the function name as the second, and the caches discussed before as the third entry. The functor can then simply be called using a **launch function** which takes the number of threads as the first argument and the arguments to pass along to the function itself as a *parameter pack*.

9.3 Picking a context to launch in

So far only GPU side execution was considered, however to make this framework more useful executing functions on CPU side as well. However CUDA does not allow this in a straight forward manner. For example the following kernel (that was shown before as well):

```
__global__ void some_fn(memory arrays){
  checkedThreadIdx(i);
  cache_arrays((cachedA, A),(cachedB, B));
  int32_t j = /* some other index */;
  C[i] = cacheA[j] + cacheB[j];
}
```

Even if checkedThreadIdx could be made to work on CPU side, and the caching forms valid code as well, the function would still be **impossible** to call. This is due to the function being a textbf__global__ function which can only be called as a configured kernel launch which always executes on the GPU. Instead using a __host__ __device__ function would make the function executable in a CPU context but would not allow us to execute the function in a GPU context as a *function pointer* generated for a __device__ function in CPU code cannot be passed to a GPU context. There is however an exception to this. The exception is that **lambda** functions can be declared as __host__ __device__ and passed along to kernel functions (if -expt-extended-lambda is used for compilation). Thus the following can be constructed:

```
__device__ __host__ void some_fn(Memory arrays){/*...*/}
template<typename Func, typename... Ts>
__global__ void kernelDispatcher(Func fn, Ts... args){
  fn(args...);
}
some_launch_fn([]__host__ __device__(Memory arrays){some_fn(arrays);});
```

Where the some_launch_fn function can either call the lambda in a host context or use the kernelDispatcher kernel function to launch the lambda on the device. The actual implementations, as usual, are more complicated due to having to handle more generic problems, where the biggest challenge is generating the functors which are done in the macros from the previous section. The overhead of this on the GPU is negligible and the actual function some_fn can still be a templated function as before to allow for variability.

So far functions can be called on the GPU, however to make them useful in a CPU context an additional step is required as the **index calculation** is not straight forward. A simple solution would be adding an additional parameter to some_fn which simply contains the current index, however this is not ideal as it adds unnecessary complexity and would only work for standard indexing. In order to solve this problem the following global variables can be defined in CPU code:

```
thread_local int3 h_threadIdx;
thread_local int3 h_blockDim;
thread_local int3 h_blockIdx;
```

Where the checkedThreadIdx macro can be defined differently in a GPU and CPU context. The names are not exactly the same as the built-in GPU variables to avoid linkage issues. The new macros then are:

```
#ifdef __CUDA_ARCH__
#define getThreadIdx() \
  blockIdx.x * blockDim.x + threadIdx.x
#define checkedThreadIdx(x) \
  int32_t x = getThreadIdx(); if(x >= threads) return;
#define checkedParticleIdx(x) \
  int32_t x = getThreadIdx(); if(x >= arrays.num_ptcls) return;
#else
#define getThreadIdx() \
  h_threadIdx.x
#define checkedThreadIdx(x) \
  int32_t x = getThreadIdx();(void)(threads);
#define checkedParticleIdx(x) \
  int32_t x = getThreadIdx();
#endif
```

Which works in general as long as proper values are assigned to h_threadIdx. In this framework **Intel's TBB library** was chosen for parallelization as it offers very powerful C++ constructs for parallelism and, as it is required by OpenVDB, is easily available whereas OpenMP traditionally is not well supported in Windows. As these values are declared as thread_local variables the following construct can be used to launch a function in parallel in a CPU context:

```
tbb::parallel_for(tbb::blocked_range<int32_t>(0, elements),
        [&](const tbb::blocked_range<int32_t> &range) {
        for (int32_t i = range.begin(); i < range.end(); ++i) {
                h_threadIdx.x = i;
                m_kernel(args...);
        }
}
```

which implements a basic blocked parallelization. The performance of the CPU version is not the fastest as the code itself is not hand optimized using vector operations and a loop defined this way is not trivial to optimize automatically for the compiler using recent SIMD instruction sets like AVX2. The CPU version still provides a very valuable debugging and evaluation platform making development significantly easier.

The actual implementations of many of these constructs are significantly more complex, especially the actual launch function due to having to handle multiple different types of configurations (templates and contexts) which need to be handled using relatively intricate template specializations and lookups. However, the backend complexity (which has no measurable performance impact) is worth it as it provides a very simple frontend. Launching code on a CPU this way can be achieved by simply replacing launch with launchHost. Or by setting *modules.context=host* in the configuration.

9.4 Implementing variability

So far as an example for variability interpolate<neigh>([&](int32_t j)/*...*/); was used. Whilst this is a valid approach to doing a neighborhood interpolation, the syntax is not ideal and capturing the full stack by reference can create suboptimal code in CUDA. Instead it would be desirable to use **range based for loops**, where in order to achieve this the following template is specialized for each algorithm:

```
template <neighbor_list neighbor> struct neighbor_span;
```

These structures need to fulfill the requirements of ISO C++ [2017] [stmt.ranged], namely providing an iterator returned by a begin() and end() function. This allows for the following code:

```
for (const auto& var : neighbor_span<neighborhood>(i, arrays)){/*...*/}
// or as a short hand macro
iterateNeighbors(var){/*...*/}
```

Implementing this for a constrained neighbor list requires about 50 lines of code and implementing this for a cell based neighbor list requires about 60 lines of code and neither of these two show any measurable negative performance impact compared to manually implementing them as the actual iterators are very simple.

The problem with these iterators however is with cell based iterators. For some methods, especially neighborhood searches, searching over all possible neighbors is required, which in turn requires iterating over a 3D grid. In a for loop, or lambda based version, this simply requires stacking three for loops, whereas for an iterator based version these three way loops have to be wrapped into a linear iterator. For cell based iterators the implementation is around 120 to 140 lines of code and even though the performance impact, if any, is very small, the implementation of new methods is made more difficult. However, as this overhead is only required once and allows for very short and abstract code throughout the simulation the tradeoff is justifiable. Additionally for testing any code can be used and only once it should be made available everywhere is the iterator based version required.

Chapter 10

Implementing SPH

The actual implementation of more complex algorithms, e.g. IISPH is relatively interesting but just showing the code is not all that helpful in understanding how this framework is used. However, a table of the lines of code used for the module definition (parameters, arrays, functions) and implementation length of some common building blocks are given in the following table with the total lines of code required in the previously used framework:

Module name	Array JSON	Parameter JSON	Module JSON	LOC New	LOC Old	Ratio
IISPH Ihmsen et al. [2014a]	50	36	17	120	2154	1:18
Density Solenthaler and Pajarola [2008]	5	0	15	22	1170	1:53
Constrained Neighbors Winchenbach et al. [2016]	21	17	13	278	1864	1: 7
Resorting Green [2010]	40	5	13	97	6231	1:64
Adaptivity Winchenbach et al. [2017]	100	59	17	605	3107	1: 5
Integration Vacondio et al. [2012]	5	6	14	165	2747	1:17
Surface Tension Akinci et al. [2013]	5	17	13	52	2957	1:57
Viscosity Monaghan [2002]	0	17	11	36	2919	1:81
Total framework	733	1057	330	17418	395921	1:16

The high number of lines of code for the JSON descriptions is mostly due to the semi-verbose nature of JSON and, as seen before, in some cases most of theses lines are simple "name":"value" pairs and not complex statements. Similarly the lines of code for the functions could be reduced by removing unnecessary false statements for packs of values. Additionally the implementation lines of code for the new framework include documentation, whereas the old framework was mostly undocumented and unformatted. Similarly the new framework, especially in the neighborhood list and adaptive sections, contains a significant number of improvements for faster execution as well as alternative versions for CPU execution. However, the old framework contains large portions of code which are not used or probably not used. Due to the challenge of figuring out what is, and what is not used and what is referred to where, it is not feasible to dissect these modules. Additionally the previous framework already includes some ideas from the new framework to significantly simplify memory management.

Overall, the lines of codes are just reduced by a factor of 23, however individual modules are reduced by factors up to **80**. These differences are in line with what was expected as much of the new framework is backend functionality to allow for shorter module code. The new framework also includes a significant number of advanced features, e.g. the unit enforcement, which are not present in the old framework. Similarly some of the features of the old framework are not present in the new one as some of them were artifacts remaining from trying out things or half-baked ideas which never went anywhere. The difference still is staggering. In the new framework a basic module could be created in 10 lines of JSON plus a basic SPH estimate of 15 lines of C++. The old framework provided a basic example, to allow copy-pasting parts of it, which alone was 868 lines of C++ code **(ratio of 1:58)**. Additionally the new framework consists of about 200 files, whereas the old framework consisted of 1494 files.

In the following subsections some example implementations of basic SPH estimates, as they were shown in chapter 2, are demonstrated. Additionally an example is shown which demonstrates a practical example of what kind of errors the unit based system can catch. An additional important observation are the lines of code for the individual parts of the framework and not just the relative lines of code compared to the previous framework.

10.1 SPH estimates

The first, and most basic SPH estimate is a density estimate

$$\delta_i = \sum_j V_j W_{ij} \tag{10.1}$$

where δ denotes the number density and $V_j = \frac{m_j}{\rho_j^0}$ [Solenthaler and Pajarola [2008]]. This can be implemented using the following code, which utilizes the macros seen before, as:

```
neighborFunctionType densityEstimate(Memory arrays){
  checkedParticleIdx(i);
  cache_arrays((pos,positions),(vol,volumes));
  auto delta_i = 0.f;
  iterateNeighbors(j)
    delta_i += vol[j] * W_ij;
  arrays.deltas[i] = delta_i;
}
```

A more general form where an array A is interpolated and stored in fA

$$\mathrm{fA}_i = \sum_j \frac{V_j}{\delta_j} A_j W_i j \tag{10.2}$$

Can be implemented as the following code, leaving out the index calculation and function signature for simplicity:

```
auto fA_i = 0.f;
iterateNeighbors(j)
  fA_i += vol[j] / arrays.deltas[j] * arrays.As[j] * W_ij;
arrays.fA[i] = fA_i;
```

As a representative of the gradient functions the following, which is commonly used for velocity gradients, is calculated:

$$\nabla \cdot f(\mathbf{x_i}) = \frac{\mathbf{1}}{\rho(\mathbf{x_i})} \sum_{\mathbf{j}} \mathbf{m_j}(\mathbf{f}(\mathbf{x_i}) - \mathbf{f}(\mathbf{x_j})) \cdot \nabla \mathbf{W_{ij}} \tag{10.3}$$

The results are stored in gradF and values are loaded from F.

```
auto gradF_i = 0.f;
iterateNeighbors(j)
  gradF_i += arrays.masses[j] * math::dot3(arrays.F[i] - arrays.F[j], GW_ij);
arrays.gradF[i] = gradF_i / (arrays.deltas[i] * arrays.rest_density);
```

10.2 Density estimates in the actual framework

In the prior subsection the density estimate was shown as a basic example which just showed the actual kernel. This subsection will show the remaining code required to implement this in the presented framework. The first file to consider is the actual implementation in density.cu:

```
#include <SPH/density/density.cuh>
#include <utility/include_all.h>
// Kernel goes here with argument SPH::Density::Memory arrays
neighFunction(estimateDensity, estimate_density,
                  "Estimate Density", caches<float4, float>{});
void SPH::Density::estimate_density(Memory mem) {
        launch<estimateDensity>(mem.num_ptcls, mem);
}
```

Which is all of the code required to run the density estimate. The array entry for density is:

```
{
  "density": {
    "description": "...",
    "type": "float",
    "unit": "void",
    "kind": "particleData"
  }
}
```

And the module description is:

```
{
  "density": {
    "description": "...",
    "folder": "density",
    "name": "Density",
    "units": true,
    "neighbor_info": true,
    "functions": [ "estimate_density" ],
    "input": [ "position", "volume" ],
    "output": [ "density" ]
  }
}
```

This is all that is required to create a module with functionality except for adding the correct .then entry at the correct location in the simulation loop which was demonstrated in chapter 8

10.3 Where units become important

A well known paper in literature implements a **surface tension** effect and in order to do this the algorithm first calculates the fluid normal which is implemented as:

```
neighFunctionType tensionFirst(SPH::TensionModule::Memory arrays) {
  checkedParticleIdx(i);
  cache_arrays((pos, position), (vol, volume), (dens, density));
  float4_u<SI::recip<SI::m>> kernelSum;
  iterateNeighbors(j) kernelSum += vol[j] / dens[j] * GW_ij;
  arrays.particleNormal[i] = (kernelSum * support_H(pos[i])).val;
}
```

Using this surface normal and a modified kernel (called **cohesion**) the surface tension is then calculated as:

```
neighFunctionType tensionSecond(SPH::TensionModule::Memory arrays) {
  checkedParticleIdx(i);
  cache_arrays((pos, position), (normal, particleNormal),
    (vol, volume), (dens, density));
  float4_u<SI::N> cohesionForce, curvatureForce;
  iterateNeighbors(j) {
    auto kernel = Kernel<cohesion>::value<kernel_kind::spline4>(pos[i], pos[j]);
    if (kernel == 0.f)
      return;
    auto scaling = 2.f * arrays.rest_density /
      (dens[i] * arrays.rest_density + dens[j] * arrays.rest_density + 1e-6f);
    auto cohesion = -arrays.tension_strength *
      vol[i] * arrays.rest_density * vol[j] *  arrays.rest_density * kernel;
    auto curvature = -arrays.tension_strength *
      vol[i] * arrays.rest_density * (normal[i] - normal[j]);
    curvatureForce += scaling * curvature;
    cohesionForce += scaling * cohesion;
  }
  arrays.acceleration[i] += (curvatureForce + cohesionForce) /
    (vol[i] * arrays.rest_density);
}
```

However trying to compile this code, which exactly follows the paper, will not compile. It will fail with an error on the statement **cohesionForce += scaling * cohesion;**.

The cohesionForce and curvatureForce should naturally have the units of **N** where the acceleration then is $\frac{\mathrm{N}}{\mathrm{m}}$ which would be the correct unit. Inside of the loop the scaling factor has no unit, and the cohesion (ignoring the tension strength) has the unit $\frac{\mathrm{kg}^2}{\mathrm{m}^3}$ whereas curvature has the unit kg and both of them are multiplied by the same tension strength parameter and the unitless scaling factor. To fulfill this equation the tension strength has to be of unit $\frac{\mathrm{m}^4}{\mathrm{kg}\cdot\mathrm{s}^2}$ or $\frac{\mathrm{m}}{\mathrm{s}^2}$. This obviously leads to a problem where resolving it would involve changing the underlying mistake in the algorithm. Alternatively one can set the tension strength to be of unit $\frac{\mathrm{m}}{\mathrm{s}^2}$ and change the assignment of the cohesionForce to

```
cohesionForce += (scaling * cohesion).val;
```

These kinds of problems can be "annoying" to deal with but ultimately lead to a more physically sound method with less ad-hoc parameters and equations that cannot be physically correct. However the disadvantage is actually figuring out the exact units of arrays and parameters. Doing this however often helps with understanding and verifying an algorithm regardless so this drawback is not that significant.

10.4 Lines of Code and back-end balance

One of the main goals of the framework was to provide a **solid back-end** of the framework with an **easy to use front-end**. The front-end complexity has been shown in the previous few sections and is very simple compared to the previous framework and even simple in a general sense due to the abstractions. This simplification is only possible due to the backend provided by the framework.

In general the new framework consists of about **17 thousand Lines of Code** (LoC) in manually written C++ which is a significant reduction from the previous almost **400 thousand LoC**. However this does not tell the full story as there is a significant amount of generated code in the new framework which amounts to a total of almost *12 thousand LoC of generated C++ code*, where the number is not very important due to the verbose nature of generated code. This generated code is concentrated (with over 7 thousand LoC) in the generated code for arrays and parameters.

Additionally, the framework now consists of just over **2 thousand LoC of JSON**, which again is relatively verbose, which on it's own might look like a lot but even compared to just the respective generated C++ code is a reduction of a factor of over 4 whilst providing a significantly more abstract *domain specific* way to handle data within a simulation. The framework also uses a single CMake file with about 400 LoC where much of the complexity deals with setting up various reusable macros.

The presented mathematical functions from chapter 5 require about 1 thousand LoC and the unit based system from chapter 6 requires about 1200 LoC. The launcher, memory handling and general CUDA functionality, including caches, requires a total of about 2 thousand LoC.

The overall utility back-end of the framework contains a total of 6900 LoC of written C++ whereas the actual SPH simulation code, meaning the module implementation, requires about 3800 LoC. The IO functionality requires 1200 LoC and the user interface, with all rendering functions, an additional 3500 LoC. The JSON parsers require a total of about 1400 LoC.

Overall the balance of front-end, meaning actual module code, to back-end code, including meta descriptions, is 3800 LoC to 15700 or about 1:4 which seems to be a reasonable ratio considering the complexity and number of SPH methods implemented in this framework.

Considering the old framework all parts required just to implement resorting require significantly more LoC than all of the SPH functionality in the new framework combined and comes close to the size of the complete back-end of the new framework. Additionally the new framework is purpose built to be expandable to keep the complexity of the whole system manageable as the individual modules can be separated and investigated as individual components instead of requiring a deeper and in depth understanding of the underlying framework due to the back-end hiding much of the complexity, which in turn makes the front-end more concise and powerful.

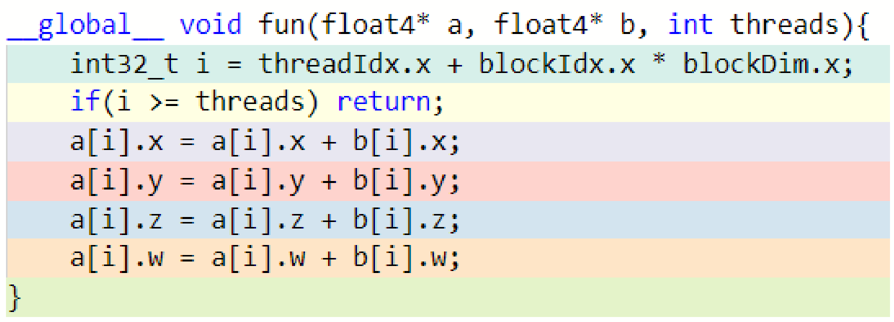
Chapter 11

Evaluating performance of mathematical operators

Considering the mathematical operations introduced in Section 3 (without units) one might question how fast these operands are compared to a direct "down to the metal" implementation and an important goal of this framework was to provide zero cost abstractions where this aspect plays an important role. This section aims to provide some insight into the actual generated assembly for the GPU code to see what the potential overheads are, and whether or not these are acceptable.

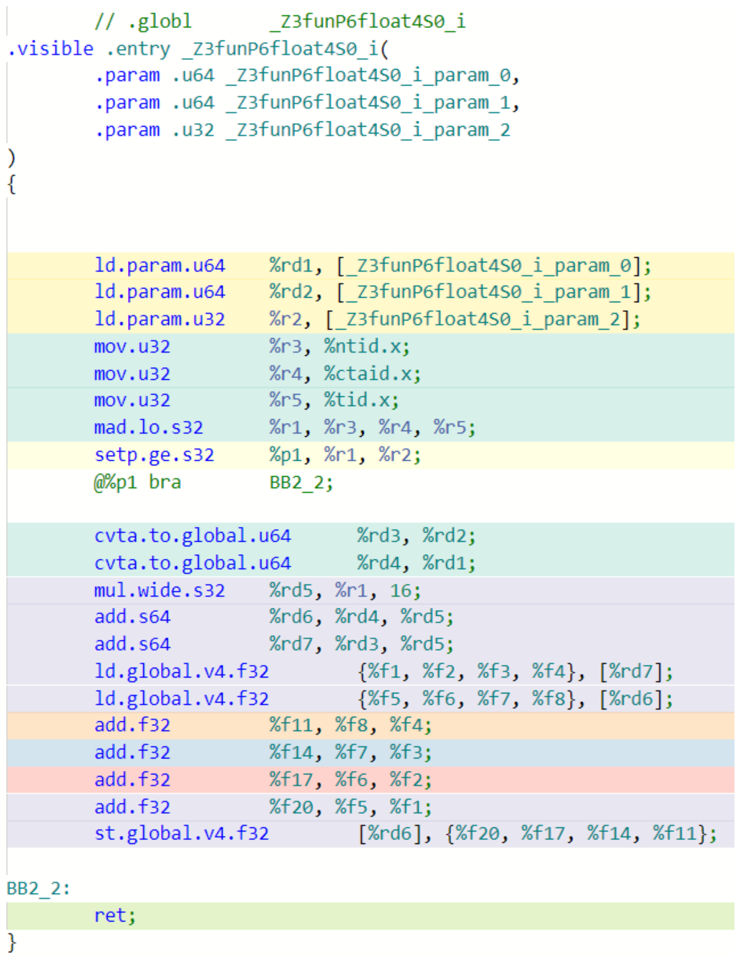
11.1 A basic example

The most basic example that could be of interest would be a simple vector addition which stores the result back into one of the operands. This implements an operation $a \mathrel{+}= b$ which in a manually implemented kernel function would look as follows:



```
__global__ void fun(float4* a, float4* b, int threads){
    int32_t i = threadIdx.x + blockIdx.x * blockDim.x;
    if(i >= threads) return;
    a[i].x = a[i].x + b[i].x;
    a[i].y = a[i].y + b[i].y;
    a[i].z = a[i].z + b[i].z;
    a[i].w = a[i].w + b[i].w;
}
```

All of the ptx assembly was generated using CUDA v9.2 by utilizing a tool called Compiler Explorer which automatically adds syntax highlighting to the kernel and code to refernce the ptx back to the CUDA code. These results were all verified manually in CUDA 10.0 and 9.2 as well. The ptx of the previous kernel then looks as follows:



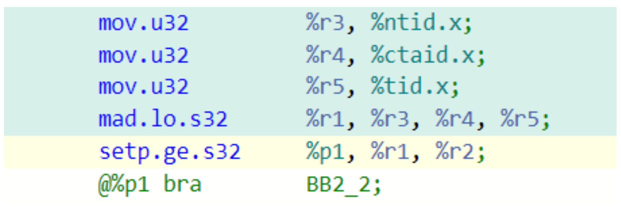
```
        // .globl       _Z3funP6float4S0_i
.visible .entry _Z3funP6float4S0_i(
        .param .u64 _Z3funP6float4S0_i_param_0,
        .param .u64 _Z3funP6float4S0_i_param_1,
        .param .u32 _Z3funP6float4S0_i_param_2
)
{


        ld.param.u64    %rd1, [_Z3funP6float4S0_i_param_0];
        ld.param.u64    %rd2, [_Z3funP6float4S0_i_param_1];
        ld.param.u32    %r2, [_Z3funP6float4S0_i_param_2];
        mov.u32         %r3, %ntid.x;
        mov.u32         %r4, %ctaid.x;
        mov.u32         %r5, %tid.x;
        mad.lo.s32      %r1, %r3, %r4, %r5;
        setp.ge.s32     %p1, %r1, %r2;
        @%p1 bra        BB2_2;

        cvta.to.global.u64      %rd3, %rd2;
        cvta.to.global.u64      %rd4, %rd1;
        mul.wide.s32    %rd5, %r1, 16;
        add.s64         %rd6, %rd4, %rd5;
        add.s64         %rd7, %rd3, %rd5;
        ld.global.v4.f32        {%f1, %f2, %f3, %f4}, [%rd7];
        ld.global.v4.f32        {%f5, %f6, %f7, %f8}, [%rd6];
        add.f32         %f11, %f8, %f4;
        add.f32         %f14, %f7, %f3;
        add.f32         %f17, %f6, %f2;
        add.f32         %f20, %f5, %f1;
        st.global.v4.f32        [%rd6], {%f20, %f17, %f14, %f11};

BB2_2:
        ret;
}
```

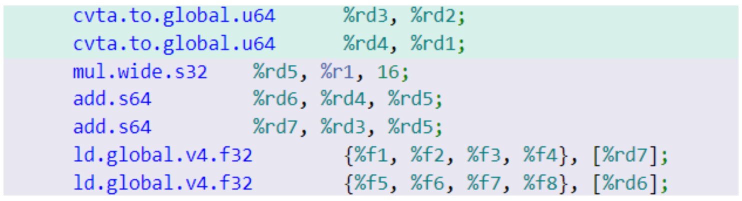
The first section of the ptx contains the function signature and entry code. The first few lines describe the function **.entry** which is what is called by CUDA when a kernel is launched. This **.entry** contains the three parameters of the function stored as unsigned integers which are loaded into the function stack using **ld.param**. The second section executed the thread index calculation:



```
        mov.u32         %r3, %ntid.x;
        mov.u32         %r4, %ctaid.x;
        mov.u32         %r5, %tid.x;
        mad.lo.s32      %r1, %r3, %r4, %r5;
        setp.ge.s32     %p1, %r1, %r2;
        @%p1 bra        BB2_2;
```

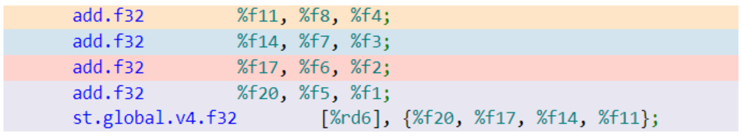
In these lines of ptx the thread indices are calculated from built-in variables **ntid**, **ctaid** and **tid** using a **mad** (multiply add) instruction. This thread index is compared to the number of threads using **setp.ge.s32** which jumps to **p1** which is a pointer for **BB2_2** which causes the execution to jump to the end of the generated function and thus causing it to return immediately.

The actual computation step is split into two distinction section where the first section is:



```
cvta.to.global.u64      %rd3, %rd2;
cvta.to.global.u64      %rd4, %rd1;
mul.wide.s32    %rd5, %r1, 16;
add.s64         %rd6, %rd4, %rd5;
add.s64         %rd7, %rd3, %rd5;
ld.global.v4.f32        {%f1, %f2, %f3, %f4}, [%rd7];
ld.global.v4.f32        {%f5, %f6, %f7, %f8}, [%rd6];
```

Which first converts the parameters into global memory pointers using a conversion instruction called **cvta.to.global.u64**. Next the thread index is multiplied by 16 to account for the size of 16 byte per float4 using **mul.wide.s32** which stores the result as a 64 bit integer. Next the actual addresses to load from are calculated using simple **add.s64** instructions which are used as input for **ld.global.v4.f32** which loads 4 32 bit floating point values from a specified global address. The actual computation then is simply executing four floating point additions with **add.f32** and storing the result back using **st.global.v4.f32**:

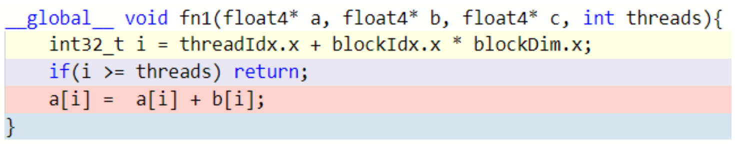


```
add.f32         %f11, %f8, %f4;
add.f32         %f14, %f7, %f3;
add.f32         %f17, %f6, %f2;
add.f32         %f20, %f5, %f1;
st.global.v4.f32        [%rd6], {%f20, %f17, %f14, %f11};
```

An interesting note here is that the order of additions is reversed from the source code which should have no impact on the performance. Due to the simplicity of this case this generated ptx can be assumed to be as close to ideal as possible.

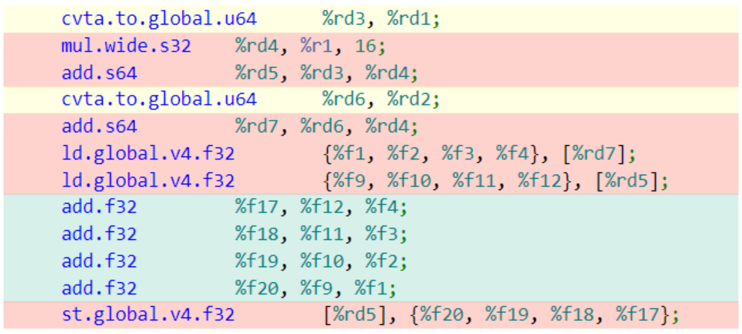
11.2 Using our math functions

Using the mathematical functions from chapter 5 we can instead create the following kernel:



```
__global__ void fn1(float4* a, float4* b, float4* c, int threads){
    int32_t i = threadIdx.x + blockIdx.x * blockDim.x;
    if(i >= threads) return;
    a[i] =  a[i] + b[i];
}
```

Where the question is how closely this resembles the ptx of the manually generated example. Note that the ptx is equal regardless of using $a += b$ or $a = a + b$. The **.entry** section as well as the thread index calculation are exactly equal and are thus not shown here. The ptx however differs for the actual computaation part which now is assembled as:



```
cvta.to.global.u64      %rd3, %rd1;
mul.wide.s32    %rd4, %r1, 16;
add.s64         %rd5, %rd3, %rd4;
cvta.to.global.u64      %rd6, %rd2;
add.s64         %rd7, %rd6, %rd4;
ld.global.v4.f32        {%f1, %f2, %f3, %f4}, [%rd7];
ld.global.v4.f32        {%f9, %f10, %f11, %f12}, [%rd5];
add.f32         %f17, %f12, %f4;
add.f32         %f18, %f11, %f3;
add.f32         %f19, %f10, %f2;
add.f32         %f20, %f9, %f1;
st.global.v4.f32        [%rd5], {%f20, %f19, %f18, %f17};
```

Whilst the actual computation is the same, loading the values from global memory is not exactly the same. The actual loads are still identical calls to **ld.global.v4.f32**, however instead of first converting both parameters to pointers and then calculating the offset pointer for both parameters, the function instead calculates one after the other. In practice this has no measurable impact as the conversion and addition operations are, on most architectures, single cycle operations, but the difference still is noteworthy.

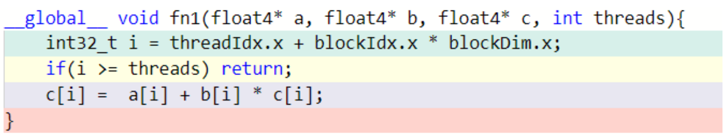
One could argue that this change is due to the overhead caused by our methods, however if one was to write a simple direct operator:

```
float4 operator+(float4 lhs, float4 rhs){
  return float4{
    lhs.x + rhs.x,
    lhs.y + rhs.y,
    lhs.z + rhs.z,
    lhs.w + rhs.w
  };
}
```

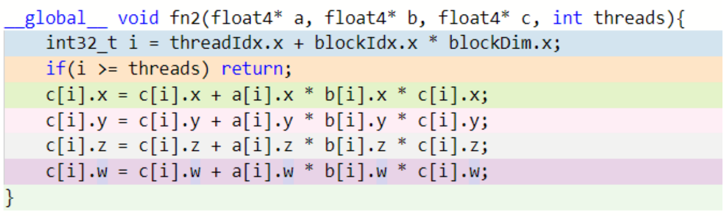
The same changed order would be generated and as such is probably an effect of using a function to execute the addition which has to evaluate all function arguments in sequence before doing any operations.

11.3 FMAD vector operations

A common goal of code is to use so called **fused-multiply-add** instructions which combine an addition with a multiplication, $d = a + b\ {*c}$ into a single instruction and as these instructions are executed with the same speed as the individual operations doubles the throughput, in an ideal case. As such it would be important for a mathematical library to still allow for fmad operations even though complex functions are used underneath. To test this the following two evaluation kernels were used:

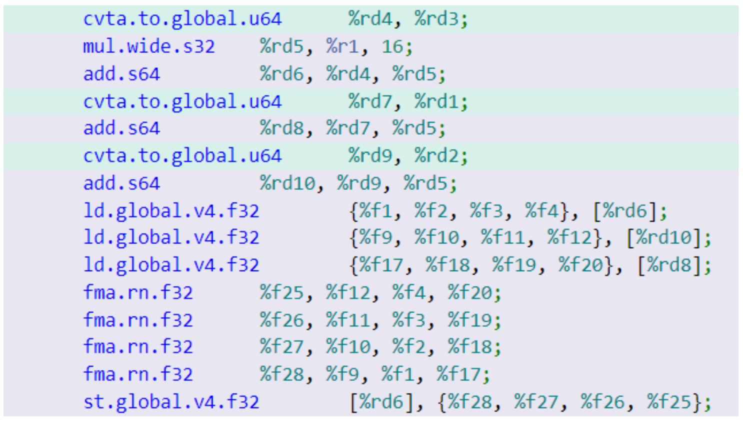


```
__global__ void fn1(float4* a, float4* b, float4* c, int threads){
    int32_t i = threadIdx.x + blockIdx.x * blockDim.x;
    if(i >= threads) return;
    c[i] =  a[i] + b[i] * c[i];
}
```



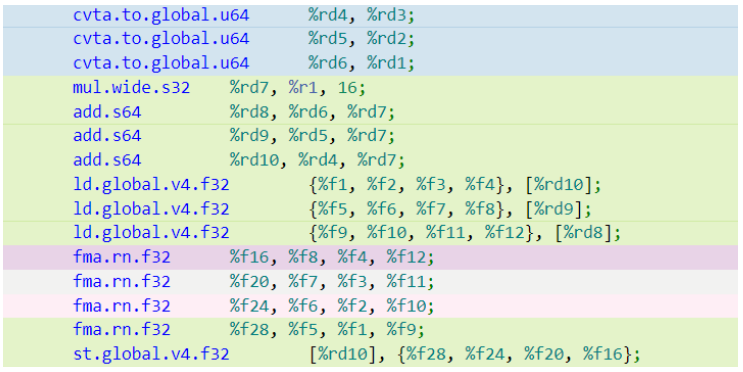
```
__global__ void fn2(float4* a, float4* b, float4* c, int threads){
    int32_t i = threadIdx.x + blockIdx.x * blockDim.x;
    if(i >= threads) return;
    c[i].x = c[i].x + a[i].x * b[i].x * c[i].x;
    c[i].y = c[i].y + a[i].y * b[i].y * c[i].y;
    c[i].z = c[i].z + a[i].z * b[i].z * c[i].z;
    c[i].w = c[i].w + a[i].w * b[i].w * c[i].w;
}
```

These functions generate equivalent entry and thread calculation ptx and as such those ptx sections are left out. However, the actual computations for these kernels are assembled as:



```
        cvta.to.global.u64      %rd4, %rd3;
        mul.wide.s32    %rd5, %r1, 16;
        add.s64         %rd6, %rd4, %rd5;
        cvta.to.global.u64      %rd7, %rd1;
        add.s64         %rd8, %rd7, %rd5;
        cvta.to.global.u64      %rd9, %rd2;
        add.s64         %rd10, %rd9, %rd5;
        ld.global.v4.f32        {%f1, %f2, %f3, %f4}, [%rd6];
        ld.global.v4.f32        {%f9, %f10, %f11, %f12}, [%rd10];
        ld.global.v4.f32        {%f17, %f18, %f19, %f20}, [%rd8];
        fma.rn.f32      %f25, %f12, %f4, %f20;
        fma.rn.f32      %f26, %f11, %f3, %f19;
        fma.rn.f32      %f27, %f10, %f2, %f18;
        fma.rn.f32      %f28, %f9, %f1, %f17;
        st.global.v4.f32        [%rd6], {%f28, %f27, %f26, %f25};
```

Whereas the manual implementation results in the following ptx:

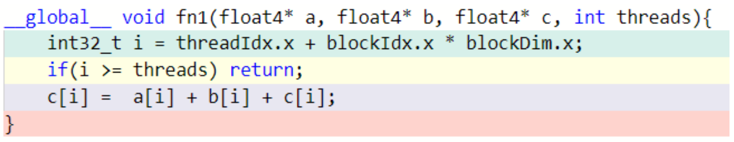


```
        cvta.to.global.u64      %rd4, %rd3;
        cvta.to.global.u64      %rd5, %rd2;
        cvta.to.global.u64      %rd6, %rd1;
        mul.wide.s32    %rd7, %r1, 16;
        add.s64         %rd8, %rd6, %rd7;
        add.s64         %rd9, %rd5, %rd7;
        add.s64         %rd10, %rd4, %rd7;
        ld.global.v4.f32        {%f1, %f2, %f3, %f4}, [%rd10];
        ld.global.v4.f32        {%f5, %f6, %f7, %f8}, [%rd9];
        ld.global.v4.f32        {%f9, %f10, %f11, %f12}, [%rd8];
        fma.rn.f32      %f16, %f8, %f4, %f12;
        fma.rn.f32      %f20, %f7, %f3, %f11;
        fma.rn.f32      %f24, %f6, %f2, %f10;
        fma.rn.f32      %f28, %f5, %f1, %f9;
        st.global.v4.f32        [%rd10], {%f28, %f24, %f20, %f16};
```

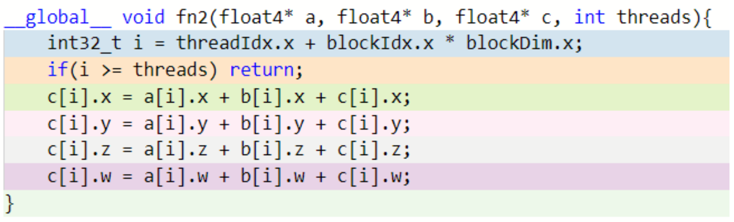
Both versions use **fma.rn.f32** instructions and properly load the data using the calls to **ld.global.v4.f32**. The same changed order of address calculations as before can be observed. This difference again can be seen as negligible and the desired fmad instructions were still generated.

11.4 Double additions

Using a seemingly more simple example of $c = a + b + c$ the following evaluation kernels are used:

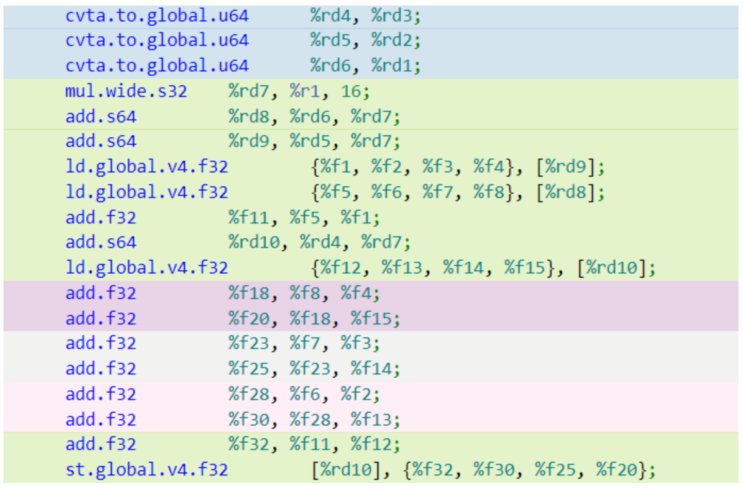


```
__global__ void fn1(float4* a, float4* b, float4* c, int threads){
    int32_t i = threadIdx.x + blockIdx.x * blockDim.x;
    if(i >= threads) return;
    c[i] =  a[i] + b[i] + c[i];
}
```



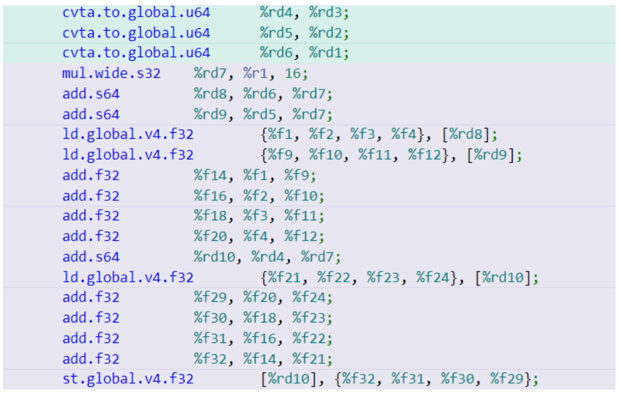
```
__global__ void fn2(float4* a, float4* b, float4* c, int threads){
    int32_t i = threadIdx.x + blockIdx.x * blockDim.x;
    if(i >= threads) return;
    c[i].x = a[i].x + b[i].x + c[i].x;
    c[i].y = a[i].y + b[i].y + c[i].y;
    c[i].z = a[i].z + b[i].z + c[i].z;
    c[i].w = a[i].w + b[i].w + c[i].w;
}
```

Which, again, produce equivalent entry and thread index calculation, but this time they produce significantly different ptx for the computational part of the kernel. The manual version generates the following ptx:



```
    cvta.to.global.u64      %rd4, %rd3;
    cvta.to.global.u64      %rd5, %rd2;
    cvta.to.global.u64      %rd6, %rd1;
    mul.wide.s32    %rd7, %r1, 16;
    add.s64         %rd8, %rd6, %rd7;
    add.s64         %rd9, %rd5, %rd7;
    ld.global.v4.f32        {%f1, %f2, %f3, %f4}, [%rd9];
    ld.global.v4.f32        {%f5, %f6, %f7, %f8}, [%rd8];
    add.f32         %f11, %f5, %f1;
    add.s64         %rd10, %rd4, %rd7;
    ld.global.v4.f32        {%f12, %f13, %f14, %f15}, [%rd10];
    add.f32         %f18, %f8, %f4;
    add.f32         %f20, %f18, %f15;
    add.f32         %f23, %f7, %f3;
    add.f32         %f25, %f23, %f14;
    add.f32         %f28, %f6, %f2;
    add.f32         %f30, %f28, %f13;
    add.f32         %f32, %f11, %f12;
    st.global.v4.f32        [%rd10], {%f32, %f30, %f25, %f20};
```

The combined cvta can still be observed but only for two out of three loads packed together. These load the entries from vectors a and b, but not c. After the load a single addition is executed and then the third address is calculated and loaded. After this addition the remaining 7 additions are executed. Comparing this to the alternative ptx:



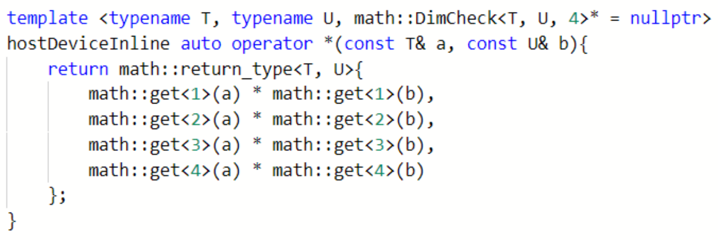
```
cvta.to.global.u64      %rd4, %rd3;
cvta.to.global.u64      %rd5, %rd2;
cvta.to.global.u64      %rd6, %rd1;
mul.wide.s32    %rd7, %r1, 16;
add.s64         %rd8, %rd6, %rd7;
add.s64         %rd9, %rd5, %rd7;
ld.global.v4.f32        {%f1, %f2, %f3, %f4}, [%rd8];
ld.global.v4.f32        {%f9, %f10, %f11, %f12}, [%rd9];
add.f32         %f14, %f1, %f9;
add.f32         %f16, %f2, %f10;
add.f32         %f18, %f3, %f11;
add.f32         %f20, %f4, %f12;
add.s64         %rd10, %rd4, %rd7;
ld.global.v4.f32        {%f21, %f22, %f23, %f24}, [%rd10];
add.f32         %f29, %f20, %f24;
add.f32         %f30, %f18, %f23;
add.f32         %f31, %f16, %f22;
add.f32         %f32, %f14, %f21;
st.global.v4.f32        [%rd10], {%f32, %f31, %f30, %f29};
```

The same split can be seen where only two vectors (a and b) are loaded together. Interestingly the address calculations are now more closely resembling the manual versions instead of being one after the other. However, in the manual version after adding the first elements (a.x and b.x) together the third vector had to be immediately loaded to allow for the third addition (of c.x). In this assembly however the addition of a and b together is done as a block to calculate the result of their addition and only once this is completed the entry of the third vector is loaded. This difference is due to the evaluation being restated as a chain of operations of vectors and not as a chain of element operations.

In the presented approach the third load is executed after the first four additions, whereas in the manual approach only a single addition exists in between the load instructions. Measuring this influence in practice is within the error margin of the measurement as the generally advertised concept of latency hiding would allow for other warps to execute a chain of four additions whilst another warp is waiting for data to load instead of only a single instruction. However, as the difference of loading times compared to four additions or one addition is still in the same order of magnitude no benefit could be generated from this in this example. As such no definitive conclusion can be drawn as to which implementation is superior to the other and as such both can be considered as valid implementations.

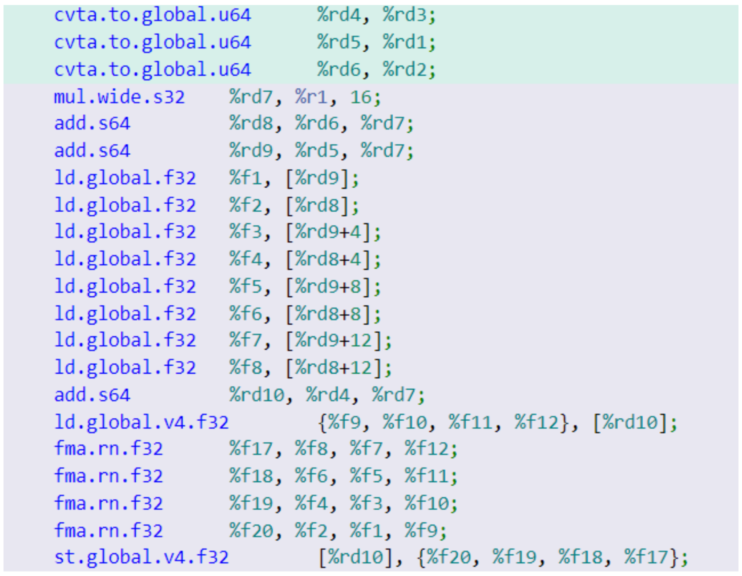
11.5 Universal references

An often stated optimization in C++ is to use constant reference where possible as these require no copying of values. As such the multiplication operator of two 4D vectors might have been implemented as follows:



```
template <typename T, typename U, math::DimCheck<T, U, 4>* = nullptr>
hostDeviceInline auto operator *(const T& a, const U& b){
    return math::return_type<T, U>{
        math::get<1>(a) * math::get<1>(b),
        math::get<2>(a) * math::get<2>(b),
        math::get<3>(a) * math::get<3>(b),
        math::get<4>(a) * math::get<4>(b)
    };
}
```

With the same kernel as used for the FMAD evaluation which generates the following ptx for the actual computational step of the function:



```
cvta.to.global.u64      %rd4, %rd3;
cvta.to.global.u64      %rd5, %rd1;
cvta.to.global.u64      %rd6, %rd2;
mul.wide.s32    %rd7, %r1, 16;
add.s64         %rd8, %rd6, %rd7;
add.s64         %rd9, %rd5, %rd7;
ld.global.f32   %f1, [%rd9];
ld.global.f32   %f2, [%rd8];
ld.global.f32   %f3, [%rd9+4];
ld.global.f32   %f4, [%rd8+4];
ld.global.f32   %f5, [%rd9+8];
ld.global.f32   %f6, [%rd8+8];
ld.global.f32   %f7, [%rd9+12];
ld.global.f32   %f8, [%rd8+12];
add.s64         %rd10, %rd4, %rd7;
ld.global.v4.f32        {%f9, %f10, %f11, %f12}, [%rd10];
fma.rn.f32      %f17, %f8, %f7, %f12;
fma.rn.f32      %f18, %f6, %f5, %f11;
fma.rn.f32      %f19, %f4, %f3, %f10;
fma.rn.f32      %f20, %f2, %f1, %f9;
st.global.v4.f32        [%rd10], {%f20, %f19, %f18, %f17};
```

This code still generates the desired fmad instructions with the same structure of **cvta** instructions, but there is a significant problem. Instead of using **ld.global.v4.f32** to load a and b the function loads these elements using single 4 byte reads from global memory using **ld.global.f32** with 4 byte offsets and interleaved for both values. This requires eight accesses to global memory instead of 2 and whilst the values should be cached in this simple example in a more complex example the later load instructions might be un-cached again, at least in L1 cache, causing an obvious performance degradation.

In the simulation framework as a whole using the implementation with const T& for all mathematical functions instead of forwarding references T&& for non unit math or simple values for unit math (due to limitations of forwarding references) the overall simulation is 10% faster for the non const T& version, which is significant for such a minor detail.

The question however is: Why does this happen? This behavior can be reproduced in situations when a temporary, e.g. the result of an operation like a * b is immediately used as an argument to another function expecting a const& argument. This happens in nvcc 9.x and 10.0 but does not happen in other compilers, e.g. GCC 8.2 or clang 7.0. Except for having access to the compiler to exactly investigate what is causing this there is no real way to address this behavior and should just be kept in mind for GPU code for now.

Chapter 12

Other parts of the framework

Besides the main computational part of the framework in the individual function modules there is a large selection of back-end features provided by the framework which serve a more utilitarian purpose, e.g. IO functions, which are still important to build a functional and useful program but are not part of the core framework itself. These features will briefly be described here to give an overview of some of the less general purpose features.

12.1 OpenVDB & Alembic

A common question faced by programs is how data is input into the simulation. The goal would be to provide an easy to use interface to quickly create new simulations and to this extent using relatively **standard data formats** is important as these formats are supported by a variety of standard computer graphics tools, e.g. Houdini. In this framework fluid volumes are initially represented as *OpenVDB* volumes which are loaded and inserted in the simulation at their world space coordinates and then sampled via a hexagonal grid of particles. These volumes in the configurations are described as:

```
{
  "particle_volumes": {
    "volume1": {
      "file": "Volumes/vol.vdb"
    }
  }
}
```

These volumes, as they are created at their original position, can simply be exported via Houdini. Similarly fluid inlets are also described as OpenVDB volumes:

```
{
  "inlet_volumes": {
    "volume1": {
      "file": "Volumes/Inlet1.vdb",
      "dur": 900,
      "delay": 0.0,
      "vel": "0 -30 0 0"
    }
  }
}
```

With attributes denoting how long these emit for (in simulated seconds) and the velocity of the emitted liquid. Complex boundary objects in the simulation are also created based on OpenVDB volumes that are loaded and transformed into 3D CUDA textures which are used as distance and normal fields. These are described as:

```
{
  "boundary_volumes": {
    "volume1": {
      "file": "Volumes/Dragon.vdb"
    }
  }
}
```

Which again can simply be created within Houdini. The overall simulation domain often times is simply represented as floor and walls at some plane in the simulation. These are created using an AABB around an obj file which is treated as the simulation bounds

```
{
  "simulation_settings": {
    "boundaryObject": "Objects/domain.obj",
    "domainWalls": "x+-y+-z+-"
  }
}
```

Where the domainWalls parameter controls in what axis direction the simulation should be bounded. The obj file can, similar to the OpenVDB files, be exported from Houdini.

Exporting data from the simulation could be done in many ways. However, the most simple way in practice is to use some **standard format** used to represent particles. In the case of this framework this is done via Alembic. Alembic is an OpenSource and relatively standard format which for the framework is configured via

```
{
  "modules":{
    "alembic":true
  },
  "alembic":{
    "fps":24,
    "folder":"export/alembic/"
  }
}
```

These exported particles can then be loaded back into Houdini. This usage of standard formats allows us to change some aspect in Houdini and save those changes into OpenVDB files, run the simulation on the updated volume data, write the particle data back out, and then render the data after reimporting it into Houdini. The only step that has to be done manually is adjusting the .json configuration file that describes all of the simulation parameters. Compared to the previous framework this is a significant improvement as creating a new simulation setting, or even adjusting something, could take hours or even days of time with no way of using proprietary rendering software.

12.2 Snapshots

When running the simulation often times certain instabilities can cause errors or undesirable behavior in the simulation. To investigate these, or even to just test the influence of tweaking a parameter like surface tension, the framework has a **snapshot** feature. While the simulation is running a snapshot can be taken of the full simulation state and later reloaded. Implementing these snapshots (ignoring some of the complexity) is relatively simple using all of the meta information provided by the framework. These snapshots are taken by hitting the x key and loaded using z. In the future it would be nice to have the option of storing these snapshots on disk and reloading them when needed but so far this has not been required.

Snapshots reset all parameters, which includes those used for visualization, time information etc. and reset all arrays regardless of their content. The only thing that does not get reset is the camera position in the GUI as this position tracks independent of the camera position in the parameters.

12.3 Structure of the Simulation code

The root folder of the simulation contains a number of sub folders:

- **cmake**: Contains a number of CMAKE modules used to find various dependencies that are not found by the standard CMAKE distribution, e.g. OpenVDB.
- **consoleParticles**: Contains the source code of a simple command line program to run simulations. Mostly used for benchmarking and testing features as the overhead of displaying a GUI is relatively small but measurable.
- **gui**: Contains the source code of the GUI, including the main file of a Qt based interface to the simulation. This GUI is the preferred way of using the simulation.
- **IO**: Contains the parsing functionality for configurations and library functions to deal with Alembic and OpenVDB files.

- **LUTCode**: Contains a program used to generate certain lookup tables for some more complex simulation features.
- **metaCode**: Contains, in sub folders, the programs used to transform the JSON files to actual C++ code.
- **openGLRenderer**: Contains all code required to render the simulation within the GUI. This will briefly be covered in a later section.
- **simulation**: Contains the main simulation loop file where new modules have to be added using then.
- **SPH**: Contains all modules of the simulation created using the JSON descriptions.
- **utility**: Contains many library features that are used throughout the simulation including math and function launching.

Out of these the utility folder is the most interesting as it contains a number of headers which are useful to know about when implementing a module (they are included by default for a module):

- **algorithm.h** is used to include a number of algorithms like reduction, scan and sort. These are implemented based on NVIDIAs *Thrust* library and can be executed either in CPU or GPU context. An example call would be:

  ```
  struct is_valid {
          hostDeviceInline bool operator()(const int x) {
      return x != INT_MAX;
  }
  };
  int32_t compacted_elems = (int32_t) algorithm::copy_if
    (particleIndex, particleIndexCompact, num_ptcls, is_valid());
  //...
  algorithm::stable_sort_by_key
    (num_ptcls, resortIndex, particleparticleIndex);
  ```

- **atomic.h** is used to provide an *atomic class* that can be used on CPU and GPU code. The GPU implementation relies on the built-in atomic functionality of CUDA and the CPU version is implemented in an OS dependent way for Linux and Windows using their specific intrinsic functions. Using the standard C++ atomics in CPU code would be preferable, but this is not possible due to the C++ atomics not being able to bind to arbitrary addresses. An example of their usage would be:

  ```
  cuda_atomic<int32_t> res_atomic(res + j);
  int32_t snapped_mlm, old = res_atomic.val();
  do {
  ```

```
snapped_mlm = old;
        if (snapped_mlm <= r_i)
                break;
        old = res_atomic.CAS(snapped_mlm, r_i);
} while (old != snapped_mlm);
```

- **cuda.h** is used to provide some basic utilities like error checking and calls to synchronize the device.
- **helpers.h** includes logging and timing functionality as well as a lookup table for color values. The logging class provides 5 levels of logging which are displayed in the GUI and filtered there or printed on the console. The timer provides host only timers using std::chrono, device only timers using CUDA events and hybrid timers which time the default compute context. An example would be:

```
logger(log_level::debug) << "Debug message" << std::endl;
LOG_DEBUG << "Debug, LOG macros add line and file info" << std::endl;
LOG_ERROR << "Error" << std::endl;
LOG_VERBOSE << "Verbose, filtered by default" << std::endl;
auto timer = TimerManager::createTimer
  ("Test Timer", Color::rosemadder, false);
timer->start();
//... code
timer->stop();
TIME_CODE("Another Timer", Color::grey10, X) // X is a C++ expression
```

- **identifier.h** is used to include the generated array and parameter information.
- **include_all.h** is used to include every utility header for convenience.
- **iterator.h** provides the iterators for neighborhood and data structure iterations.
- **launcher.h** provides the launcher functionality, see chapter 9.
- **macro.h** is used to include some of the more complex macros e.g. cached_arrays.
- **math.h** is used to provide the math functionality discussed in chapter 5.
- **MemoryManager.h** is used to provide the memory manager from chapter 8.
- **SPH.h** is used to include some boundary functions as well as indexing functions used for resorting. Kernels however are provided via the math header.
- **unit_math.h** provides the unit aware math from chapter 6.

The overall simulation only contains a single CMake file which is located in the root directory. The root directory also contains the JSON files used for code generation called arrays.json, functions.json and parameters.json.

12.4 OpenGL

Rendering often is an important consideration for fluid simulations which can be split into two separate categories:

Production renders, or anything used to reproduce the fluid faithfully in a fully lit environment with complex rendering effects like caustics and reflections. These are not handled within the framework as the goal of the framework is to simulate, not to render. For these kinds of renderings the fluid data can be exported from the simulation, using alembic files, which can be rendered in Houdini or from Houdini processed into different formats for other renderers. This removes the additional complexity that integrating a production quality renderer into this framework would cause. Including a renderer like that however would still be possible if desired, e.g. for developing not just simulation methods but also rendering methods.

Visualization renders, or anything used to display the fluid quickly in a way that represents information and doesn't focus on visual quality. These kinds of renderings are much more useful for a simulation framework to observe how the simulation is running and for investigating possible causes of instabilities. Additionally these visualizations can be done within the simulation and serve as a quick and easy way to create videos or screenshots used to show some result or simulation to someone else whilst portraying some underlying effect in the simulation, e.g. velocity or density.

To keep the simulation modules free of rendering code the rendering functionality needs to be cleanly separated. In some frameworks, especially the prior one that is being replaced with this one, the concept of rendering and scene graphs has been woven into the simulation causing significant problems. Some renderers or frameworks, e.g. Unreal Engine, require using their own way of thinking about rendering which can be great for production systems but not for research where flexibility often is the key.

Visualization itself is implemented using a corresponding module within the simulation which usually is the last sub-step of a simulation step. This function simply maps some array to a value range of 0 to 1, e.g. using vector magnitude or other measures. These metrics can either use a reduction to get the minimum and maximum value for auto scaling or can use manual scaling between a minimum and maximum value.

The actual rendering is done in a very user friendly manner. The OpenGLRenderer itself contains a main render loop which integrates standard openGL code into a Qt openGL widget. Renderers execute in the order they were added to the rendering loop and are written as sub classes of a BaseRenderer. These rendering classes can use any standard openGL construct, or if necessary could also execute ray tracing via CUDA. The only difference between the rendering code and standard openGL code is that the renderer needs to call initializeOpenGLFunctions() as well as compile OpenGL shaders via QOpenGLShaderProgram. These restrictions however are fairly minor as compiling shaders would need to be done using some wrapping functionality anyways. Usually the renderer, and it's associated VAOs etc, are initialized in the renderers constructor and during a rendering step the render function is called which for the visualization renderer looks like this:

```
void ParticleRenderer::render() {
  glBindVertexArray(vao);
  m_program->bind();
  glDrawElementsInstanced(GL_TRIANGLES, 6,
    GL_UNSIGNED_INT, (void *)0, get<parameters::num_ptcls>());
  m_program->release();
  glBindVertexArray(0);
}
```

One challenge though is binding parameters and arrays to OpenGL from the simulation. CUDA has the benefit of having interoperability with OpenGL which allows us to create VBO objects that represent an array and uniforms that represent the parameters. An iteration over all parameters and arrays can be executed which tries to bind these to uniforms, or attributes, with their respective names, which if this succeeds indicates wether or not a specific shader uses a specific parameter or array. Using this logic the following section of a shader can be written:

```
in vec4 posAttr;
in vec2 uvAttr;
in vec4 position;
in float renderIntensity;
in float volume;

uniform mat4 perspective_matrix;
uniform mat4 view_matrix;
```

Where *position*, *volume* and *renderIntensity* are automatically bound using the previously described iteration. The view and perspective matrices use another mechanism which allows the creation of custom parameter bindings for user defined values, which here are the camera matrices defined by some camera instance, as:

```
m_uniformMappings["view_matrix"] = new gl_uniform_custom<QMatrix4x4>(
    &Camera::instance().matrices.view, "view_matrix");
m_uniformMappings["perspective_matrix"] = new gl_uniform_custom<QMatrix4x4>(
    &Camera::instance().matrices.perspective, "perspective_matrix");
```

The rendering loop takes care of binding the current parameters for everything which means that this complexity can be completely ignored when implementing a renderer. The visualization renderer additionally uses a color map, which is bound to programs by manually registering with the color map handler:

```
colorMap::instance().bind(m_program, 0, "colorRamp");
```

An example configuration of this renderer could be the following, where the map name is used to find a png colormap in the simulations resource folder.

```
{
  "color_map": {
    "auto": false,
    "buffer": "velocity",
    "min": 50,
    "max": 0,
    "map": "Blues"
  }
}
```

12.5 GUI Interactivity

The same methodology used to create the automatic configuration parsing can be used to create a list of all parameters within the simulation in the GUI. In the GUI this is represented as an individual Qt docker widget which contains a tree of widgets based on their JSON identifiers. This tree contains all parameters, even if the module they are influencing is not loaded as there is no depends attribute for parameters. The actual implementation of this, especially to enable sliders and text editing, is very complex and requires a large number of templates with if constexpr conditions. Additionally this requires a very complex method that allows us to iterate over all *members of a struct* to generate these displays for arbitrary complex types. In practice however, none of the implementation detail matter as the system works very reliably on Windows and Linux. The only drawback is that if a variable is not defined as not const, it cannot be edited and forgetting to set this attribute can often be done on accident, however this is no short coming of the framework.

A simple example of adding a custom object to this widget would be (in gui/qt/propertyviewer.cpp):

```
struct test {
        float4 a;
        int b;
};
test reflect{ {1, 2}, 2 };
//...
addToTree(&reflect, "reflection test", m_parameterTree);
```

Chapter 13

Conclusions

The framework outlined in this thesis readily achieved all the goals set out in chapter 4, and even exceeded some of them. The code complexity has been reduced to a point where simple methods can be displayed on a single printed page making it very easy not just to implement but also to argue about whether or not the function does what it is supposed to do. The separation of front-end and back-end to the degree shown here allows for significant abstractions at negligible overhead which not just makes the front-end code more concise but also easier to explain. In the previous framework creating and executing a simulation was a task that took days but now, with support of industry standard formats, can be done in minutes where changes to fluid volumes can be applied in seconds instead of hours. Creating new arrays and parameters is done in a single place, creating not just the actual data but also creating GUI elements and function parsing, simplifying this task, yet again, from hours to seconds. Modules in this framework encapsulate their functionality fully, where the actual use code does not require any system to be followed but could be written in any way the user wants to write it. The introduction of the mathematical operators for built-in types not just simplifies code in a generic and simple way but also produces ideal assembly code which can be extended easily to support unit enforcement.

The only significant limitation to this framework is a lacking support of Multi-GPU systems, which could be added into the back-end in the future.

Thus, in conclusion, the new framework presented here provides a relatively simple, abstract, way to implement SPH simulations on GPUs using modern programming techniques and domain specific languages in an open source fashion that, due to the low number of external libraries, could easily be used by independent researchers, where new algorithms could easily be shared and added to the framework by simply providing the small amounts of JSON and C++ required for the front-end of a module.

Bibliography

Nadir Akinci, Gizem Akinci, and Matthias Teschner. 2013. Versatile Surface Tension and Adhesion for SPH Fluids. *ACM Trans. Graph.* 32, 6 (2013), 182:1—-182:8. https://doi.org/10.1145/2508363.2508395 55

Ryoichi Ando, Nils Thürey, and Chris Wojtan. 2013. Highly Adaptive Liquid Simulations on Tetrahedral Meshes. *ACM Trans. Graph.* 32, 4, Article 103 (July 2013), 10 pages. https://doi.org/10.1145/2461912.2461982 3

Jan Bender and Dan Koschier. 2015. Divergence-Free Smoothed Particle Hydrodynamics. *Proc. 2015 ACM SIGGRAPH/Eurographics Symp. Comput. Animat.* 1 (2015). https://doi.org/10.1145/2786784.2786796 4

Jens Cornelis, Markus Ihmsen, Andreas Peer, and Matthias Teschner. 2014. IISPH-FLIP for incompressible fluids. *Comput. Graph. Forum* 33, 2 (may 2014), 255–262. https://doi.org/10.1111/cgf.12324 3

CUDA. 2018. *CUDA Toolkit Documentation v10.0.130.* NVIDIA Corporation. https://docs.nvidia.com/cuda/ 10, 12, 22

Walter Dehnen and Hossam Aly. 2012. Improving convergence in smoothed particle hydrodynamics simulations without pairing instability. *Mon. Not. R. Astron. Soc.* 425, 2 (2012), 1068–1082. arXiv:1204.2471 7

R. A. Gingold and J. J. Monaghan. 1977. Smoothed particle hydrodynamics-theory and application to non-spherical stars. *Monthly Notices of the Roy. Astronomical Soc.* 181 (1977), 375–389. 3

Simon Green. 2010. Particle Simulation using CUDA. *Cuda 4.0 Sdk* May (2010). 10, 55

Markus Ihmsen, Jens Cornelis, Barbara Solenthaler, Christopher Horvath, and Matthias Teschner. 2014a. Implicit incompressible SPH. *IEEE Trans. Vis. Comput. Graph.* 20, 3 (2014), 426–435. https://doi.org/10.1109/TVCG.2013.105 4, 55

Markus Ihmsen, Jens Orthmann, Barbara Solenthaler, Andreas Kolb, and Matthias Teschner. 2014b. SPH Fluids in Computer Graphics. *Eurographics STARS* 2 (2014), 21–42. 7

ISO C++. 2011. *ISO/IEC 14882:2011 Information technology — Programming languages — C++* (third ed.). ISO/IEC JTC1/SC22/WG21. `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372` 11, 12, 18

ISO C++. 2014. *ISO/IEC 14882:2014 Information technology — Programming languages — C++* (fourth ed.). ISO/IEC JTC1/SC22/WG21. `http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=64029` 10, 12

ISO C++. 2017. *ISO/IEC 14882:2017 Information technology — Programming languages — C++* (fifth ed.). ISO/IEC JTC1/SC22/WG21. 1605 pages. `https://www.iso.org/standard/68564.html` 9, 12, 13, 14, 15, 16, 18, 28, 30, 44, 54

J. J. Monaghan. 1992. Smoothed particle hydrodynamics. *Annual review of astronomy and astrophysics* 30, 1 (1992), 543–574. `https://doi.org/10.1146/annurev.astro.30.1.543` arXiv:arXiv:1007.1245v2 3, 6, 7

J. J. Monaghan. 2002. SPH compressible turbulence. 335 (Sept. 2002), 843–852. arXiv:astro-ph/0204118 55

J J Monaghan. 2005. Smoothed particle hydrodynamics. *Reports Prog. Phys.* 68, 8 (2005), 1–34. arXiv:astro-ph/0507472v1 6

Matthias Müller, David Charypar, and Markus Gross. 2003. Particle-Based Fluid Simulation for Interactive Applications. *Proc. ACM SIGGRAPH/Eurographics Symp. Comput. Animat.* 5 (2003), 154–159. 4

OpenACC. 2017. *The OpenACC Application Programming Interface.* OpenACC organization. `https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.6.final.pdf` 9

OpenCL. 2018. *The OpenCL Specification.* Khronos OpenCL Working Group. `https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_API.pdf` 10

OpenMP. 2015. *OpenMP Application Program Interface Version 4.5.* OpenMP Architecture Review Board. `https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf` 9

Daniel J. Price. 2010. Smoothed Particle Hydrodynamics and Magnetohydrodynamics. (2010). `https://doi.org/10.1016/j.jcp.2010.12.011` arXiv:1012.1885 4, 8

B Solenthaler and R Pajarola. 2008. Density Contrast SPH Interfaces. *Proc. ACM SIGGRAPH/Eurographics Symp. Comput. Animat.* (2008), 211–218. 55, 56

R. Vacondio, B. D. Rogers, and P. K. Stansby. 2012. Accurate particle splitting for smoothed particle hydrodynamics in shallow water with shock capturing. *International Journal for Numerical Methods in Fluids* 69, 8 (2012), 1377–1410. `https://doi.org/10.1002/fld.2646` 55

Rene Winchenbach, Hendrik Hochstetter, and Andreas Kolb. 2016. Constrained Neighbor Lists for SPH-based Fluid Simulations. In *Proc. Eurographics/ACM SIGGRAPH Symp. Comput. Animat.* Eurographics Association. 55

Rene Winchenbach, Hendrik Hochstetter, and Andreas Kolb. 2017. Infinite Continuous Adaptivity for Incompressible SPH. *ACM Trans. Graph.* 36, 4, Article 102 (July 2017), 10 pages. `https://doi.org/10.1145/3072959.3073713` 55

Eidesstattliche Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Siegen, ____________________________________