Effiziente SPH-basierte Flüssigkeitssimulation mit Visualisierung auf einem GPU-Cluster

Tim Werner

Bayreuth Reports on Parallel and Distributed Systems

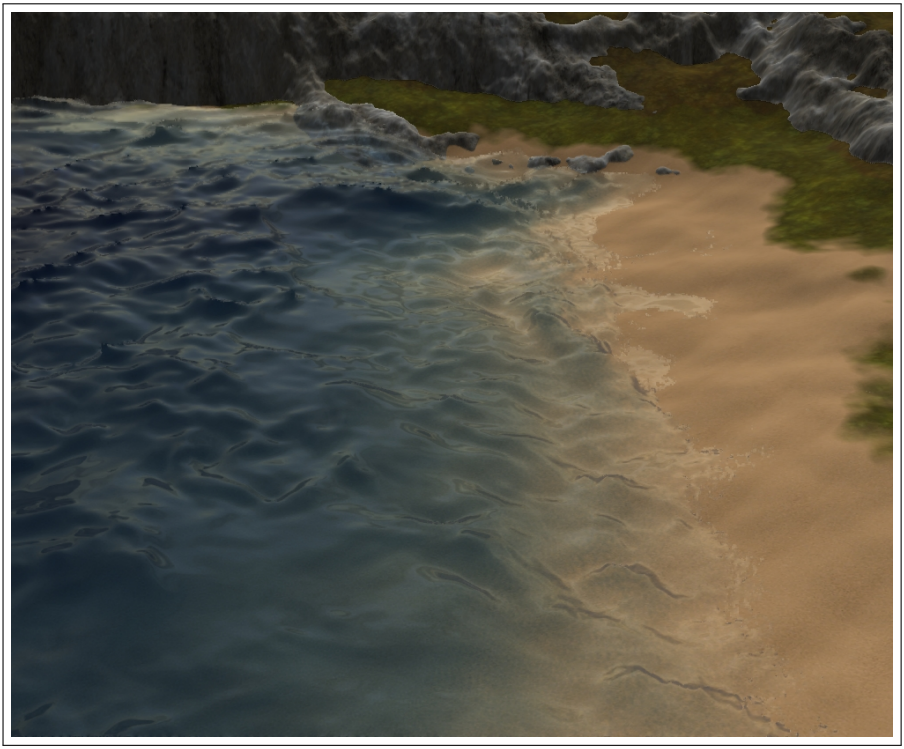
No. 7, September 2015

University of Bayreuth
Department of Mathematics, Physics and Computer Science
Applied Computer Science 2 – Parallel and Distributed Systems
95440 Bayreuth
Germany

Phone: +49 921 55 7701
Fax: +49 921 55 7702
E-Mail: brpds@ai2.uni-bayreuth.de

Effiziente SPH-basierte Flüssigkeitssimulation mit Visualisierung auf einem GPU-Cluster

Master-Thesis



Tim Werner

Eingereicht am 22. Januar 2015
Überarbeitete Version (redaktionelle Änderungen) vom 2. September 2015

Angewandte Informatik 2 — Parallele und Verteilte Systeme

Universität Bayreuth
95440 Bayreuth

Zusammenfassung

Dreidimensionale Flüssigkeitssimulationen sind in vielen Bereichen der Forschung und Entwicklung unverzichtbar. Eine sehr beliebte Methode für die dreidimensionale Flüssigkeitssimulation ist die sogenannte Smoothed-Particle-Hydrodynamics-Methode, welche kurz auch als SPH-Methode bezeichnet wird. Solche SPH-basierte Flüssigkeitssimulationen sind sehr rechenaufwändig. Dafür sind sie aber sehr gut parallelisierbar und besitzen einen mittelmäßigen Grad an Datenparallelität. Des Weiteren sind moderne GPUs parallele Rechner, welche die Instruktionen datenparallel per Single-Instruction-Multiple-Data (SIMD) abarbeiten. Deshalb sind sie für SPH-Simulationen gut geeignet. Auch sind die Speicherzugriffe einer SPH-Simulation extrem lokal. Deshalb eignen SPH-Simulationen sich auch sehr gut für einen GPU-Cluster. Es ist das Ziel der Masterarbeit eine SPH-Simulation auf einen GPU-Cluster zu implementieren, optimieren und untersuchen. Die Optimierungen und Untersuchungen sollen dabei im Hinblick auf die auf die Skalierbarkeit und die bestmögliche Ausnutzung der GPUs stattfinden. Des Weiteren ist für die Auswertung einer SPH-Simulation oft eine Visualisierung von Vorteil. Eine solche Visualisierung ist jedoch sehr rechenaufwändig. Deswegen beschäftigt sich die Arbeit zusätzlich damit, wie sie eine Visualisierung der SPH-Simulation in einem GPU-Cluster implementieren kann.

Hierfür erklärt die Arbeit zunächst OpenGL- und CUDA-Grundlagen. Dafür wird als erstes der Hardware-Aufbau einer GPU erklärt. Auch wird die Ausführung von Programmen auf der GPU, den sogenannten Kernels, erläutert. Hierfür werden insbesondere die SIMT- beziehungsweise SIMD-Eigenschaften der GPU und die damit verbundene Warp-Ausführungseffizienz hervorgehoben. Ebenso wird erläutert, wie die GPU Latenzen überbrückt. Auch stellt die Arbeit die stark mit der Latenzüberbrückung verbundene Occupancy vor. Zudem wird auf die einzelnen Speicherbereiche einer GPU in CUDA eingegangen. Zusätzlich wird kurz die Programmierung von CUDA mit CUDA-C gezeigt. Schließlich demonstriert die Arbeit die grundlegende OpenGL-Funktionalität an einer einfachen OpenGL-Pipeline.

Als Nächstes werden die physikalischen Grundlagen einer SPH-Simulation diskutiert. So diskretisiert ein SPH-Verfahren die Flüssigkeit zunächst durch einzelne Partikel. Dabei übernehmen die einzelnen Partikel die Strömungsgrößen der Flüssigkeit, die Dichte, und die Geschwindigkeit, an ihrer Position. Die Partikel dienen zudem als Stützstellen zur kurzreichweitigen Interpolation der Vektorfelder, Skalarfelder und Gradientenfelder von denjenigen Differentialgleichungen, welche die Flüssigkeit beschreiben. Für die Beschreibung der Flüssigkeit werden in dieser Arbeit die Kontinuitätsgleichung der Dichte und die Impulsgleichung verwendet. Durch die Interpolation wird die Ortsdiskretisierung ausgeführt, womit die SPH-Simulation die zeitlichen Änderungen der Strömungsgrößen der Partikel bestimmt. Da die Interpolation kurzreichweitig ist, handelt es sich bei ihr algorithmisch gesehen um ein Fixed-Radius-Near-Neighbors-Problem. Bei einem solchen Problem geht es immer darum, für jedes Partikel alle benachbarten Partikel innerhalb einer Cutoff-Distance zu finden. Anhand der gefundenen Nachbarn werden dann problemspezifische Berechnungen durchgeführt. Dabei ist zu beachten, dass die Lösung der Fixed-Radius-Near-Neighbors-Probleme bei einer SPH-Simulation immer einen Großteil der Rechenzeit ausmachen. Anschließend kann die Zeitdiskretisierung beziehungsweise Zeitintegration der zeitlichen Änderungen erfolgen, wofür die Arbeit das Prädiktor-Korrektor-Verfahren zweiter Ordnung verwendet. Mit Hilfe dieser Zeitintegration werden die Partikel dann bewegt.

Danach beschreibt die Arbeit, wie die physikalischen Grundlagen zunächst in eine SPH-

Simulation für eine einzige GPU umgesetzt werden. Für die Lösung der Fixed-Radius-Near-Neighbors-Probleme der SPH-Simulation wird eine Space-Partitioning-Datenstruktur benötigt. Die Arbeit verwendet eine spezielle Gitterdatenstruktur, in welche die Partikel einsortiert werden. Für die Konstruktion des Gitters wird ein atomarer Counting-Sort-Ansatz verwendet. Für die Berechnungen selbst verwendet die Arbeit den Linked-Cell-Ansatz. Bei ihm wird über die Partikel parallelisiert. Für jedes Partikel werden die benachbarten Gitterzellen nach Nachbarpartikeln innerhalb der Cutoff-Distance durchsucht, um mit ihnen die SPH-Interpolation ausführen zu können. Da die Fixed-Radius-Near-Neighbors-Berechnungen einen Großteil der Laufzeit des Programms ausmachen, wird versucht diese in Kombination mit dem verwendeten Linked-Cell-Ansatz weiter zu optimieren. Dabei wird besonderen Wert darauf gelegt, die Auswirkungen der limitierenden Latenzen zu reduzieren und die Datenparallelität zu erhöhen. Die Untersuchungen finden beispielhaft anhand von zwei Fixed-Radius-Near-Neighbors-Problemen aus der SPH-Simulation statt. Das eine Problem berechnet die zeitlichen Änderungen der SPH-Simulation, während das andere Problem die Dichten der SPH-Simulation mit einem Shepard-Filter normiert. Durch die Optimierungen lassen sich zwar die Laufzeiten beider Fixed-Radius-Near-Neighbors-Probleme deutlich reduzieren. Dennoch verursachen die Latenzen und die nur mittelmäßige Datenparallelität weiterhin große Performanceverluste.

Als Nächstes wird die Simulation um eine Visualisierung erweitert. Bei der Visualisierung verwendet die Arbeit eine Ellipsoid-Splatting-Technik, um die einzelnen Partikel als Ellipsoide auf den Bildschirm zu projizieren. Für dieses Splatting muss zunächst für jedes Partikel die Anisotropie der Partikelverteilung an dessen Stelle per Fixed-Radius-Near-Neighbors-Berechnung bestimmt werden. Aus dieser Anisotropie lassen sich die Hauptachsen der Ellipsoide berechnen. Anschließend werden die Ellipsoide mit OpenGL gezeichnet, wodurch die GPU ein Tiefenbild und ein Dickebild erstellt. Nun wird das Tiefenbild per Screen-Space-Curvature-Flow geglättet, damit der Betrachter die Wölbungen der Ellipsoide im finalen Renderergebnis nicht mehr erkennen kann. Zuletzt wird das geglättete Tiefenbild und Dickebild zum finalen Renderergebnis zusammengesetzt. Dabei dient das Dickebild dafür die Durchsichtigkeit der Flüssigkeit zu bestimmen und das Tiefenbild dafür die Reflexion der Flüssigkeitsoberfläche zu bestimmen.

Zu Letzt wird die Simulation mitsamt Visualisierung so erweitert, dass sie effizient auf einem GPU-Cluster ausgeführt werden kann. Anschließend wird sie in Hinblick auf die Skalierbarkeit und die Ausnutzung der GPUs untersucht. Für die Modifikationen wird das Volumen der Simulation und damit die in dem Volumen enthaltenen Partikel eindimensional in Scheiben auf die einzelnen GPUs aufgeteilt. Für sowohl die Bewegung der Partikel durch die Zeitintegration als auch für die Berechnungen der Fixed-Radius-Near-Neighbors-Probleme ist jedoch eine Kommunikation zwischen den GPUs des Clusters notwendig. Deshalb wird die Simulation so modifiziert, dass sie gleichzeitig einen Netzwerktransfer und ein Berechnen auf den GPUs erlaubt. Zusätzlich verwendet die Arbeit eine Lastbalancierung, die die Scheibendicke anhand der Kernel-Laufzeiten variiert, so dass die GPUs des Clusters gleichmäßig ausgelastet werden. Ebenso werden die Berechnungen der Visualisierung auf die Knoten des Clusters per Compositing-Technik aufgeteilt. Dafür berechnet jede GPU zunächst mit ihren eigenen Partikeln ein Tiefenbild und ein Dickebild, die dann über das Netzwerk per Min-Operation beziehungsweise per Additionsoperation zu einem einzigen Tiefenbild und Dickebild reduziert werden. Die Reduktion der Tiefenbilder und Dickebilder findet asynchron in einer Pipeline statt, damit die GPUs währenddessen weiter die SPH-Simulation berechnen können. Anschließend findet der Screen-Space-Curvature-Flow und das finale Shading auf einem separaten Knoten des Clusters statt, so dass die GPUs der anderen Knoten nicht ungleichmäßig ausgelastet werden. Die Untersuchungen zeigen, dass die Simulation sehr gut mit der Partikelzahl bei konstanter

GPU-Zahl skaliert. Auch skaliert sie sehr gut mit der GPU-Zahl bei konstanter Partikelzahl. Wird die Partikelzahl der Simulation bei konstanter GPU-Zahl zu groß, so treten kleine Performanceeinbrüche wegen verringerter Cache-Trefferraten auf. Werden zu wenige Partikel bei zu vielen GPUs simuliert, so geht ebenfalls Performance verloren, da in diesem Fall die Netzwerkbandbreite limitiert. Zusätzlich wird die Laufzeit dadurch ein bisschen erhöht, dass die GPUs des Clusters trotz Lastbalancierung ein wenig auf ihre Berechnungen untereinander warten müssen. Wird die Problemgröße groß genug gewählt so lässt sich im Cluster beinahe der optimale Speedup erzielen.

Efficient SPH based Liquid Simulation on a GPU Cluster

Abstract

Three dimensional liquid simulations are very important in many fields of research and design. A very popular method for a three dimensional liquid simulation is the so called smoothed particles hydrodynamics method, which is also abbreviated as the SPH method. Such a SPH based liquid simulation is very computationally expensive. But it is also very parallelizable and has a mediocre data parallelism. In addition modern GPUs are parallel computers, which process their instruction data parallel according to the single instruction multiple data model (SIMD model). That is why they are very eligible for such a SPH simulation. Furthermore a SPH simulation also performs very local memory accesses. Thus a SPH simulation is eligible for a GPU cluster, too. That is why this master thesis aims to implement a SPH based liquid simulation in a GPU cluster. Also the thesis aims to optimize and examine the simulation regarding the scalability and the utilization of the GPUs. Additionally a visualization of a liquid simulation is advantageous for the evaluation of its results. But a liquid visualization is also computationally expensive. Therefore this thesis will implement a visualization of the SPH simulation, which is performed by the GPU cluster.

For this purpose the thesis first explains OpenGL and CUDA basics. First the hardware structure of a GPU will be explained. Furthermore the execution of GPU programs, so called kernels, is explained. Therefore the SIMD properties or precisely the SIMT properties of a GPU and the related warp execution efficiency are illustrated. Also this thesis explains the latency hiding behavior of the GPUs and the related concept of the occupancy. Next the different memory types of CUDA are presented. Finally the thesis demonstrates the basic OpenGL functionality based on a simple OpenGL pipeline.

Subsequently the thesis explains the physical basics of a SPH based liquid simulation. The SPH method discretizes the liquid by particles. Each particle carries properties of the liquid within the volume of the particle, namely the mass, the density and the velocity. Those particles and their properties are used for the short ranged and spatial interpolation of the scalar, vector and gradient fields of the partial differential equations, which define the liquid. This thesis uses the continuity equation of the density and the momentum equation for the definition of the liquid. The interpolation performs a spatial discretization of the partial differential equations. This discretization results in the temporal changes for the variables of each particle. Since the interpolation is short ranged, it is algorithmically considered as a fixed radius near neighbors problem. Such a problem always consists of finding all neighboring particles for each particle within a given cutoff distance. The so found neighbors are used for performing problem specific computations. The solutions of the fixed radius near neighbors problems always need most of the run time of a SPH simulation. After the spatial discretization the liquid simulation performs the temporal discretization by using the temporal changes. Therefore this thesis uses a second order predictor corrector method. Thus the particles are moved.

After that the thesis illustrates, how the physical basics are implemented within a SPH simulation, which is initially only computed by a single GPU. An efficient solution of a fixed radius near neighbors problem requires a spatial partitioning data structure. For that this thesis uses a special grid data structure, into which the particles are sorted. Furthermore it uses an atomic version of the counting sort algorithm for the construction of the grid. For the fixed radius near neighbors computation itself this thesis uses the linked cell approach. This approach computes

each particle in parallel. For each particle the approach traverses the adjacent grid cells around the particle in order to find all neighbors within the cutoff distance. Then the found neighbor particles are used for the SPH interpolation. Since the fixed radius near neighbors computations are the most expensive part of a SPH Simulation, this thesis tries to optimize those computations in combination with the chosen linked cell approach. Therefore it tries especially to increase the quite low data parallelism and to improve the latencies, which both have a big impact on the performance. Those improvements are carried out by using two fixed radius near neighbors problems of the SPH simulation as a example. The first problem calculates the temporal changes of the particle attributes. The other problem is the so called Shepard filter, which normalizes the densities of the particles. By those optimizations the run time of both problems are greatly reduced. But still latencies and a mediocre data parallelism cost much performance.

Next this thesis extends the SPH simulation by a visualization. Therefore is uses an ellipsoid splatting technique, which projects every particle as an ellipsoid onto the screen. Before the visualization can perform the splatting, it has to calculate the main axis of the ellipsoids by determining the anisotropy of the particle distribution around each particle. The anisotropy has again to be calculated by solving a fixed radius near neighbors problem. Then those ellipsoids are rendered with OpenGL in order to create a thickness image and a depth image. Since each ellipsoid causes a curvature in the depth image, which will be visible in the final image later on, the depth image has to be blurred. For blurring the thesis uses the screen space curvature flow method. Finally the thickness image and the depth image are both composed to create the final result of the visualization. Therefore the visualization uses the thickness image to determine the transparency of the liquid and the depth image to determine the reflectivity of the liquid.

Finally the thesis extends the SPH simulation together with visualization, so that both of them can be performed within a GPU cluster. After that both of them are examined regarding the GPU utilization and the scalability. For the extension the volume of the simulation and thereby the particles within the volume have to be partitioned among the GPUs of the Cluster. The thesis uses a one dimensional partitioning scheme, where every GPU obtains a single slice of the volume and all particles within this volume. Furthermore a slow network communication between the different GPUs of the cluster is needed in order to move the particles and to perform the fixed radius near neighbors computations. In order to increase the GPU utilization the thesis implements the communication in a way that it will be performed simultaneously with the SPH computations. Additionally the thesis implements a load balancing algorithm in order to increase the utilization of the GPUs. The load balancing algorithm changes the thickness of the slices according to the run times of the SPH kernels. The computations of the visualization are also distributed among the GPUs of the cluster by using a compositing technique. For this compositing technique each GPU renders a depth image and a thickness image using only its own particles. After that the thickness images and the depth images of all GPUs are reduced to a single thickness image and a single depth image over the network. The reduction is performed asynchronously in a pipeline, so that the GPUs do not have to wait in order to continue with the SPH calculations. Furthermore the screen space curvature flow and the final Shading are executed on a special node with a single GPU, in order that the renaming GPUs are more evenly utilized. Finally the examinations revealed, that the simulation scales very well with the amount of used GPUs and simulated particles. If the amount of particles, which are simulated by a constant amount of GPUs, becomes too big, the performance degrades a bit because of lower cache hit rates. Furthermore there is some performance loss if too few particles are simulated by too many GPUs. This is caused by a limitation of the network bandwidth. The simulation also loses some performance because the GPUs still have to wait for the computations of each

other a bit in spite of the load balancing. If the problem sizes are chosen large enough the cluster almost achieves an optimal speed up.

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

1 Einleitung

1.1 Motivation

Bei vielen Anwendungen ist es nötig eine Flüssigkeit dreidimensional zu simulieren. Eine oft verwendete Simulationsmethode für Flüssigkeiten ist die sogenannte Smoothed-Particle-Hydrodynamics-Methode, welche auch kurz als SPH bezeichnet wird. Mögliche Anwendungsfälle für dreidimensionale Flüssigkeitssimulationen mit SPH sind:

- Medizinische Simulationen, wie der Blutfluss in den Adern
- Katastrophensimulationen, beispielsweise das Modellieren einer Tsunamiwelle oder die Überflutung eines Schiffes
- Ingenieurtechnische Simulationen, wie Strömungssimulationen, hydraulische Simulationen, das Design von Wellenbrechern oder vom Küstenschutz
- In der physikalischen und mathematischen Forschung
- Computerspiele, wie „Alice Madness Returns“ oder „Cryostasis“
- CGI-Filme zum Beispiel mit der Flüssigkeitssimulation des 3D-Programms Blender

Drei Beispiele für die Visualisierung einer solchen SPH-Flüssigkeitssimulation sind in Abbildung 1.1 zu sehen.

Das SPH-Verfahren wurde ursprünglich in 1977 von Monaghan und Gingold im Artikel [GM] vorgestellt, um damit nicht kugelsymmetrische, astrophysikalische und hydrodynamische Phänomene zu simulieren, zum Beispiel wie sich ein Protostern aus einer dichten interstellaren Wolke formt. Jedoch lässt es sich auch leicht allgemein für eine drei dimensionale Flüssigkeitssimulation verwenden. Bei einer SPH-Flüssigkeitssimulation wird die Flüssigkeit durch eine Vielzahl von Partikeln diskretisiert (siehe Abbildung 1.1), welche sich mit der Flüssigkeit mitbewegen. Deswegen handelt es sich um eine lagrange'sche Diskretisierung. So basiert SPH im Gegensatz zu vielen anderen Flüssigkeitssimulationsmethoden auf keinem Gitter oder allgemeiner keinem Mesh, weshalb es zu den Meshfree-Methods gehört. Diese Partikel dienen in ihrer näheren Umgebung als Stützstellen für die Interpolation von Vektor-, Skalar- und Gradientenfeldern mit sogenannten Smoothing-Kernels. Bei den Feldern handelt es sich um die Felder derjenigen partiellen Differentialgleichungen, die die Flüssigkeit beschreiben. Mit Hilfe den interpolierten Feldstärken an den Positionen der Partikel lassen sich Kräfte berechnen, die die Partikel auf sich untereinander gegenseitig auswirken. Die Kräfte resultieren nun wieder in Beschleunigungen der einzelnen Partikel. Mit Hilfe der Beschleunigungen kann das SPH-Verfahren dann wiederum die Zeitintegration ausführen. Selbst wenn die Grundideen, nämlich die Diskretisierung durch Partikel und die Interpolation von Feldern, bei allen SPH-Flüssigkeitssimulationen identisch sind, so gibt es bei der SPH-Flüssigkeitssimulation kein einziges starres sondern eine Vielzahl von unterschiedlichen Verfahren. Auch lassen sich die unterschiedlichen SPH-Verfahren zum Teil beliebig kombinieren. Des Weiteren sind die SPH-Verfahren allesamt algorithmisch gesehen sehr ähnlich. Jedoch gibt es auch fortgeschrittenere SPH-Verfahren, welche eine größere physikalische Komplexität besitzen. Wegen den eben

genannten Gründen sowie der Tatsache, dass es sich bei dieser Arbeit um eine Informatik-Masterarbeit handelt, wird im Folgenden allerdings nur exemplarisch ein einfaches Verfahren für die SPH-Flüssigkeitssimulation verwendet.

SPH-Simulationen sind parallelisierbar auf Partikelbasis und dabei mittelmäßig datenparallel. Des Weiteren ist jede etwas größere SPH-Simulation extrem rechenaufwändig. Die Ursache hierfür ist, dass wegen der Dreidimensionalität einer solchen Simulation die Flüssigkeit für eine gewünschte Auflösung durch sehr viele Partikel diskretisiert und berechnet werden muss. So bestehen einfachere SPH-Simulationen aus mehreren hunderttausend Partikeln. Aber auch Simulationen von mehreren hundert Millionen Partikeln sind keine Seltenheit. Da moderne Graphikprozessoren parallele SIMD-Rechner sind, eignen sie sich gut für SPH-Simulationen. Dennoch sind bei den Implementierungen von SPH-Simulationen GPU-spezifische Optimierungen nötig, damit die Simulation die Rechenleistung der GPU gut ausnutzt. Aus diesem Grund beschäftigt sich diese Masterarbeit damit, wie sie eine SPH-Simulation für GPUs implementieren und optimieren kann. Zusätzlich eignen SPH-Verfahren sich, wegen der kurzen Reichweite der Interpolation und wegen den damit verbundenen lokalen Speicherzugriffen, sehr gut für NUMA-Rechner oder für Rechner mit verteilten Speicher. Aus diesem Grund ist es ein weiteres Ziel der Masterarbeit eine SPH-Implementierung zu verwirklichen, welche auf einem GPU-Cluster lauffähig ist.

Ein Weiteres Problem ist, dass ein Anwender für die Auswertung einer Flüssigkeitssimulation oft eine Visualisierung der Flüssigkeit benötigt. Dabei ist es prinzipiell vorteilhaft, wenn die Visualisierung zeitgleich mit der SPH-Simulation durchgeführt wird. Denn eine SPH-Simulation erzeugt große Datenmengen. Deshalb ist die langfristige Speicherung dieser Daten problematisch. Eine solche Visualisierung ist ebenfalls extrem rechenaufwändig. Zusätzlich liegen die Daten der Flüssigkeit nur verteilt im Cluster vor. Deswegen ist es bei der Visualisierung auch erforderlich, dass alle GPUs des Clusters die Visualisierung berechnen. Aus dem Grund wird zusätzlich in dieser Masterarbeit eine Visualisierung für die SPH-Flüssigkeitssimulation auf einem GPU-Cluster implementiert.

1.2 Zielstellung und Gliederung

Nachdem ein grober Überblick über SPH und die Motivation hinter einer GPU-Cluster-Simulation mit Visualisierung gegeben worden ist, sollen nun die Ziele und die Gliederung dieser Masterarbeit erläutert werden. So ist es ein primäres Ziel eine CUDA-Implementierung einer einfacheren SPH-Simulation für eine einzige GPU zu verwirklichen. Anschließend soll die Masterarbeit diese Implementierung optimieren und untersuchen. Um das zu verwirklichen müssen zuerst diverse CUDA- und SPH-Grundlagen erklärt werden. Dabei ist es jedoch kein Ziel der Arbeit tiefer in die physikalischen und mathematischen Grundlagen der SPH-Methode vorzudringen, eine Variante einer SPH-Simulation mit komplexeren physikalischen Berechnungen zu implementieren, oder die physikalischen Berechnungen an sich zu optimieren. Zudem soll eine Visualisierung für die SPH-Simulation mit OpenGL implementiert werden. Dafür müssen ebenfalls zunächst einige OpenGL-Grundlagen erklärt werden. Anschließend ist es das letzte Ziel die SPH-Simulation und die Visualisierung so zu erweitern, dass sie performant auf einem GPU-Cluster lauffähig sind. Danach soll die auf dem Cluster lauffähige Version auch untersucht werden.

Demnach lässt sich die Masterarbeit in folgende Kapitel gliedern:

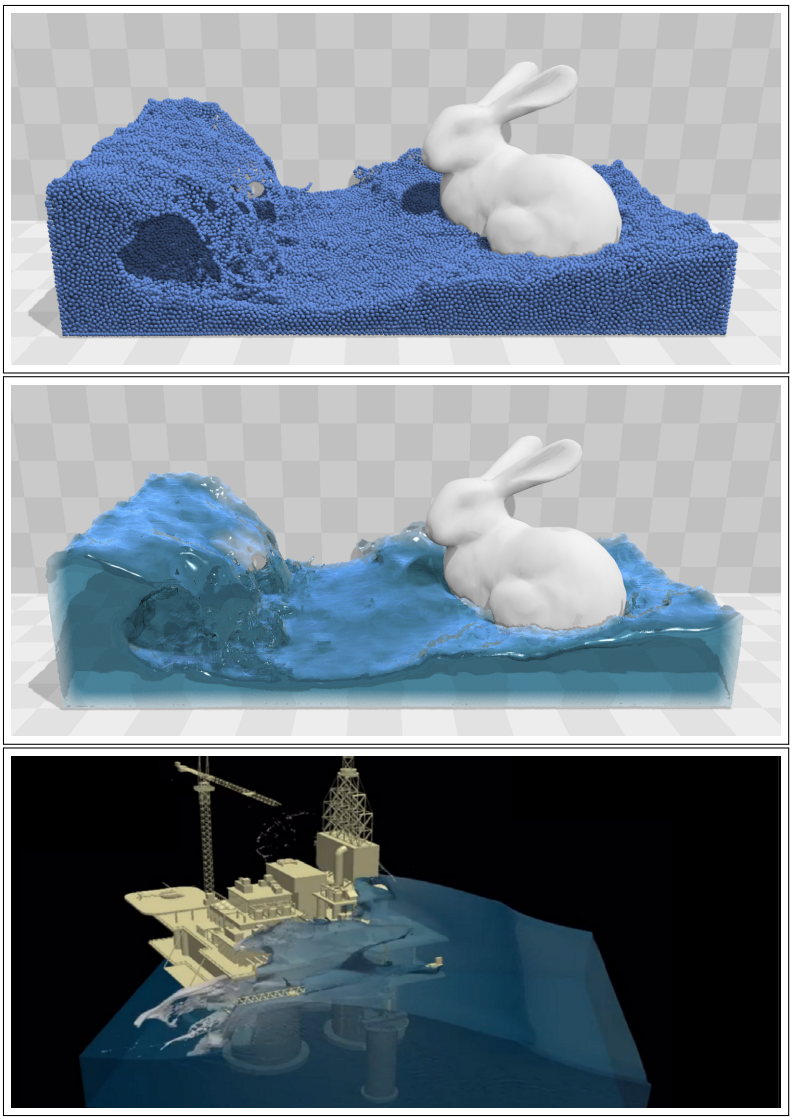


Abbildung 1.1: Beispiele für unterschiedliche SPH-Simulationen: Auf den oberen beiden Bildern ist die SPH-Flüssigkeitssimulation aus [MM] mit 128 000 Partikeln zu sehen. Auf den obigen Bild sind die einzelnen Partikel als Kugeln gerendert, während das mittlere Bild die Flüssigkeit mit einer wirklichkeitsnäheren Visualisierung darstellt. Das unterste Bild zeigt das Auftreffen einer Riesenwelle auf eine Ölplattform, die durch die SPH-Flüssigkeitssimulation aus [DSPHa] mit 226 Millionen Partikeln simuliert wurde.

- **Kapitel 2:** Als erstes erläutert die Arbeit die benötigten CUDA- und OpenGL-Grundlagen.
- **Kapitel 3:** Hier werden die benötigten SPH-Grundlagen vorgestellt.
- **Kapitel 4:** In diesem Kapitel stellt die Arbeit zunächst eine SPH-Implementierung, welche nur auf einer einzigen GPU lauffähig ist, vor. Zusätzlich finden Optimierungen und Untersuchungen dieser Implementierung statt.
- **Kapitel 5:** Nun wird die implementierte Visualisierung erläutert, die ebenfalls zunächst nur eine GPU verwendet.
- **Kapitel 6:** Das Kapitel handelt davon, wie für diese Masterarbeit die SPH-Simulation mit ihrer Visualisierung auf einen GPU-Cluster portiert wurde. Zusätzlich finden wieder Untersuchungen statt.
- **Kapitel 7:** Schließlich wird das Fazit aus der Arbeit gezogen.

2 GPU-Grundlagen

2.1 CUDA-Grundlagen

2.1.1 Allgemeines

Damit später die Diskussion der SPH-Implementierung für einen Leser ohne GPU-Vorkenntnisse nachvollziehbar ist, sollen in diesem Kapitel kurz die benötigten GPU-Grundlagen erläutert werden. Dafür wird zuerst CUDA, welches für die Implementierung der physikalischen SPH-Simulation verwendet wurde, vorgestellt. Danach erklärt die Arbeit diejenigen OpenGL-Grundlagen, die die Visualisierung benötigt.

So ist CUDA eine Plattform für GPGPU, also dem Durchführen von allgemeinen Berechnungen auf GPUs. Dabei ist CUDA nur auf auf NVIDIA-GPUs lauffähig. Es erlaubt einer herkömmlichen CPU, auch Host genannt, Programme, sogenannte Kernels, auf einer GPU, auch Device genannt, zu starten. Dafür spezifiziert CUDA folgendes:

- Die Hardware der GPUs und das Verhalten dieser Hardware
- Eine API für die Ansteuerung der GPUs und für den Datenaustausch zwischen dem Host und den GPUs
- Eine Modifikation des C++-Standards und des C-Standards für die Programmierung von Graphikprozessoren

Die Hardware eines CUDA-Devices wird durch die sogenannte Compute-Capability beschrieben. Dabei unterscheiden sich GPUs von unterschiedlichen Compute-Capabilities primär dadurch, welche Funktionalitäten die GPUs besitzen, wie ein Multiprozessor aufgebaut ist, und wie die Speicherzugriffe zwischengespeichert werden. Da die in der Masterarbeit verwendeten GPUs die Compute Capability 3.5 besitzen, beschreibt folgendes Kapitel nur eben diese Compute-Capability. Die Quellen für die folgende Beschreibung stammen primär aus dem CUDA-C-Programming Guide [NVa]. Dort sind auch Beschreibungen für die übrigen Compute-Capabilities zu finden.

2.1.2 Hardware-Aufbau

Zunächst soll der Hardware-Aufbau erläutert werden. Von der Compute-Capability 3.5 gibt es nur eine einzigen GPU-DIE, nämlich den GK-110. Dessen Blockdiagramm ist in Abbildung 2.1 zu sehen. So besteht eine Graphikkarte mit dem GK-110 primär aus folgender Hardware:

- **Bis zu 12 GiByte GDDR5-DRAM als Device-Memory:** Dieser Speicher dient als Hauptspeicher der GPU für Daten und Befehle. Er ist über einen 384 Bit breiten Speichercontroller angebunden. Als GDDR5 besitzt der DRAM eine Zugriffsgranularität von 32 Byte. Bei einem DRAM-Zugriff führt der DRAM also immer 32-Byte große Transaktionen aus.
- **1536 kiByte L2-Cache:** Sämtliche Speicherzugriffe auf den DRAM werden über diesen Cache abgewickelt. Er besitzt 128-Byte Cache-Lines. Eine Cache-Line ist noch einmal in vier 32-Byte große Transaktionsblöcke unterteilt. Der Grund hierfür ist, dass der

Abbildung 2.1: Blockdiagramm einer GPU der Compute-Capability 3.5: Quelle des Bildes ist [NVb].

GDDR5-DRAM durch 32-Byte große Transaktionen angesteuert werden muss. Werden gloable Speicherzugriffe direkt über die Load-Store-Units (siehe Punkt 2.1.6.3) angesteuert, so werden keine ganzen Cache-Lines sondern nur die tatsächlich benötigten Transaktionsblöcke aus dem DRAM geladen. Dadurch kann wiederum der Overfetch reduziert werden.

- **15 Multiprozessoren:** Die Multiprozessoren sind die einzelnen Prozessorkerne der GPU, ähnlich wie die Prozessorkerne in einer CPU.
- **5 Raster-Engines:** Sie sind die Hardwarebeschleunigung für den Rasterisierungsprozess und können nicht mit CUDA sondern nur mit OpenGL verwendet werden.
- **2 Copy-Engines:** Diese sind für Kopieroperationen zwischen dem Host und dem Device, zwischen dem Device und einem anderen Device, und innerhalb des Devices selbst verantwortlich.
- **GigaThread-Engine:** Diese Steuerhardware verteilt die Arbeit beziehungsweise die Thread-Blocks auf die einzelnen Multiprozessoren.

Es ist hierbei wichtig anzumerken, dass wegen des Product-Binnings einige Teile des GK-110, wie zum Beispiel einige Multiprozessoren, einige Raster-Engines, oder eine Copy-Engine auf

einer bestimmten Graphikkarte deaktiviert sein können. Während sich deshalb obige Werte und Taktraten bei mehreren Graphikkarten von ein und der selben Compute-Capability unterscheiden können, so ist der Aufbau der Multiprozessoren an sich immer identisch. Ein solcher Multiprozessor ist in Abbildung 2.2 gezeigt und lässt sich noch einmal feiner in folgende Hardware unterteilen:

- **Caches und Speicher:**
  - **65536 32-Bit Register:** Diese beinhalten die automatischen Registervariablen während der Kernel-Ausführung. Dabei fungieren sie weitgehend analog wie die Register bei herkömmlichen Prozessoren, abgesehen davon, dass sie dynamisch auf die Threads eines Multiprozessors aufgeteilt werden.
  - **16 bis 48 kiByte Shared-Memory:** Hierbei handelt es sich um einen schnellen Scratchpad-Speicher, welcher sich die selbe Hardware mit dem L1-Cache teilt. Dabei kann das Hostprogramm das Verhältnis von beiden auf 48 kiByte zu 16 kiByte, 32 kiByte zu 32 kiByte und 16 kiByte zu 48 kiByte einstellen. Der Shared-Memory wird meist für Blocking-Algorithmen oder für das Austauschen von Zwischenergebnissen innerhalb eines Thread-Blocks verwendet.
  - **16 bis 48 kiByte L1-Cache:** Der L1-Cache wird nur für den lokalen Speicher - also für automatische Variablen, welche auf den Stack liegen - verwendet. Er speichert keine globalen Speicherzugriffe zwischen.
  - **48 kiByte Texture-Cache:** Diesen Read-Only-Cache verwendet die GPU für die Texturzugriffe. Allerdings kann die GPU ihn auch für globale Speicherzugriffe verwenden, sofern dieser Speicherbereich sich während der Kernelausführung nicht verändert. NVIDIA bezeichnet den Texture-Cache deshalb in diesem Zusammenhang auch oft als Read-Only-Data-Cache.
  - **8 kiByte Konstanten-Cache:** Er ist ebenfalls read-only und wird für den Konstantenspeicher in CUDA verwendet.
- **Hardware für Speicherzugriffe:**
  - **32 Load-Store-Units:** Sie arbeiten Shared-Memory-Zugriffe, lokale Speicherzugriffe und globale Speicherzugriffe, die nicht den Texture-Cache verwenden, ab.
  - **16 Texture-Units**: Die Texture-Units werden bei Texturzugriffen verwendet und führen dabei sowohl die Speicherzugriffe als auch die Interpolation aus. Zudem arbeiten sie globale Speicherzugriffe, die den Texture-Cache verwenden, ab.
- **Hardware für Rechenoperationen:**
  - **192 CUDA-Cores:** Dabei handelt es sich um eine Kombination aus Single-Precision-FPU und ALU. Sie sind untereinander noch einmal spezialisiert: Während alle CUDA-Cores Gleitkomma-Operationen ausführen können, können bestimmte Integeroperationen nur von manchen CUDA-Cores ausgeführt werden.
  - **32 Special-Function-Units:** Diese berechnen transzendente Funktionen und die Kehrwertsfunktion mit einfacher Gleitkommagenauigkeit.
  - **64 DP-Units:** Sie sind für Gleitkommaberechnungen mit doppelter Genauigkeit verantwortlich.

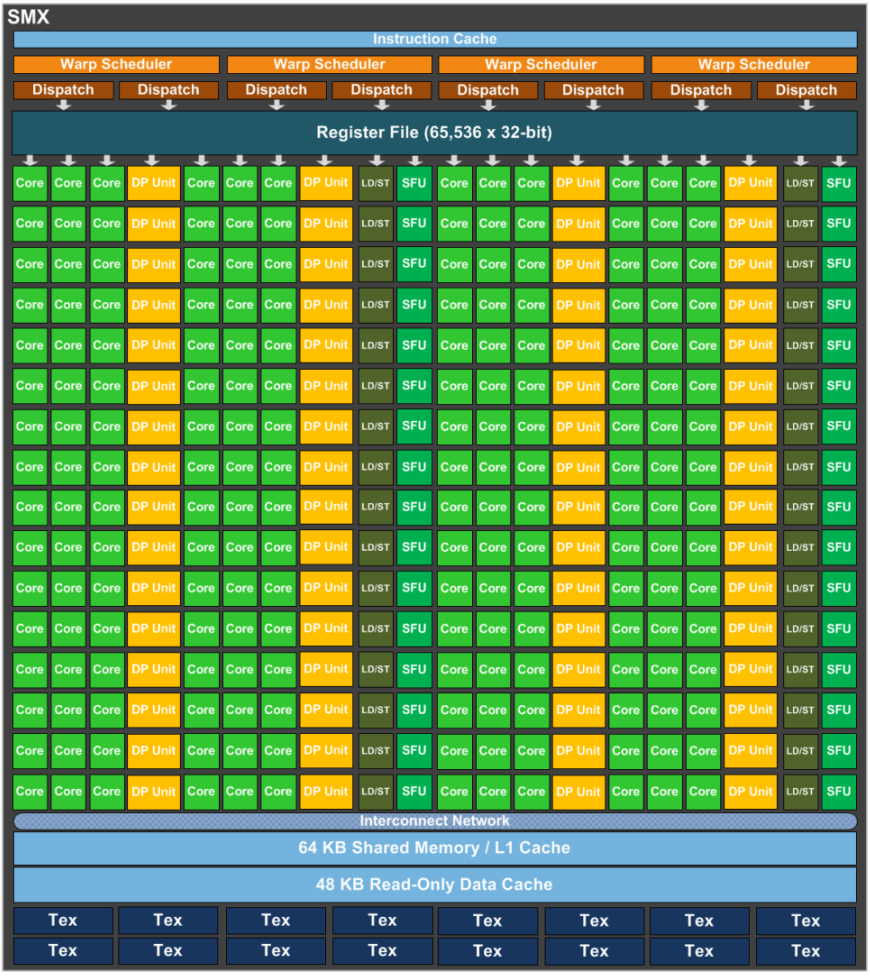


Abbildung 2.2: Blockdiagramm eines Multiprozessors der Compute-Capability 3.5: Quelle des Bildes ist [NVb].

- **4 Warp-Scheduler:** Die Warp-Scheduler sind das Steuerwerk des Multiprozessors und geben die Befehle an die einzelnen Warps heraus.

2.1.3 Kernelausführung

Soeben gab der vorherige Punkt einen groben Überblick über den Hardwareaufbau einer GPU. Anschließend erläutert dieser Punkt wie die GPU ein Kernel abarbeitet.

Um Berechnungen auf einer GPU zu starten muss der Host über einen Aufruf der CUDA-API ein Kernel mit einem bis zu dreidimensionalen Gitter aus Thread-Blocks angeben. Ein jeder dieser Thread-Blocks besteht wiederum aus einem gleich großen und bis zu dreidimensionalen Gitter aus Threads. Dabei muss der Host die Größe des Gitters aus Threads innerhalb eines Thread-Blocks ebenfalls beim Kernelaufruf angeben. Die GPU arbeitet nun die Thread-Blocks ab. Für die Abarbeitung teilt die Gigathread-Engine der GPU den einzelnen Multiprozessoren die Thread-Blocks nacheinander zu. Dabei wird ein Thread-Block immer nur von einem Multiprozessor bearbeitet, während ein Multiprozessor je nach dem Ressourcenverbrauch eines Thread-Blocks eine bestimmte Anzahl von Thread-Blocks gleichzeitig bearbeiten kann. Ein solcher „GPU“-Thread innerhalb eines Thread-Blocks ist einem CPU-Thread sehr ähnlich. So besitzt der GPU-Thread seinen eigenen Befehlszähler, seinen eigenen Stack und seine eigenen Register. Zudem kann er seine Position innerhalb des Thread-Blocks und die Position seines Thread-Blocks innerhalb des Gitters aus Thread-Blocks abfragen und anhand dieser Position seinen eigenen Kontrollfluss beschreiten. Des Weiteren können sich die Threads innerhalb eines Thread-Blocks per Barriere synchronisieren oder mit Hilfe von Shared-Memory Daten untereinander austauschen.

Die Threads innerhalb eines Thread-Blocks sind noch einmal zu je 32 Stück in sogenannten Warps gruppiert. In diesem Kontext werden die Threads eines Warps auch oft als Lanes des Warps bezeichnet. Wird ein Thread-Block einem Multiprozessor zugewiesen, so werden die Warps auf die vier Warp-Scheduler aufgeteilt. Bei der Programmausführung selektiert jeder der vier Warp-Scheduler jeden Takt einen Warp, der bereit ist, und gibt dessen nächsten Befehl oder falls möglich dessen nächsten beiden Befehle in Auftrag. Dabei führen alle Threads eines Warps die Befehle gemeinsam aus. Kommt es wegen Sprungbefehlen vor, dass sich der Kontrollfluss der Threads innerhalb eines Warps aufspaltet, so wollen nicht alle Threads des Warps als Nächstes den selben Befehl ausführen. In diesem Fall arbeitet der Warp denjenigen Befehl ab, der am frühesten im Programm auftritt. Der Warp divergiert hierbei, wodurch der Vorgang allgemein als Warpdivergenz bezeichnet wird. Die Warpthreads, deren nächster Befehl ein anderer ist, bleiben bei der Ausführung dieses Befehls untätig. Die Gruppierung von Threads zu Warps und die Abarbeitung der Befehle auf Warpbasis wird von NVIDIA als das Single-Instruction-Multiple-Threads-Programmiermodell bezeichnet. Das Programmiermodell wird oft als SIMT abgekürzt und stellt einen Spezialfall von SIMD dar.

Ein Warp-Scheduler kann jedoch nur den Befehl eines Warps in Auftrag geben, wenn der Warp bereit ist. Ein Warp gilt als bereit, wenn alle Threads im Warp bereit sind den nächsten Befehl auszuführen. Dies ist nicht der Fall, wenn ein oder mehrere Threads innerhalb des Warps noch auf einen Operanden für den nächsten Befehl warten. Das Warten kann durch die Latenzen der Speicherzugriffe und der Rechenkerne entstehen. Die Wartezeit wird dadurch überbrückt, dass ein Warp-Scheduler mehrere Warps gleichzeitig verwaltet, von denen im Optimalfall immer ein Warp bereit ist.

Da jeder der vier Warp-Scheduler zwei Instruktionen pro Takt herausgibt, können die acht Warp-Scheduler pro Multiprozessor und pro Takt maximal acht Instruktionen an Warps herausgeben. Da ein Multiprozessor jedoch nur 6x32 CUDA-Cores besitzt, können von diesen acht Instruktionen maximal sechs Instruktionen für die CUDA-Cores dabei sein. Es ist hier-

bei interessant anzumerken, dass die CUDA-Cores Skalarprozessoren sind. Somit führen die Warpthreads immer skalare Befehle aus, weshalb NVIDIA seine CUDA-GPUs auch als Skalararchitekturen bezeichnet. Die Vektorisierung des Quelltextes für die einzelnen GPU-Threads ist somit nicht nötig. Dies steht im Gegensatz zu AMD-GPUs der HD 5000er und HD 6000er Serie, die AMD als VLIW-Architektur bezeichnet. Bei der VLIW-Architektur sind die Threads ebenfalls zu Warps, die der AMD-Nomenklatur als Wavefronts bezeichnet, gruppiert. Allerdings führen die einzelnen Warpthreads wiederum Vektorbefehle aus, weshalb diese Architektur stark auf Vektorisierung des Quelltextes angewiesen ist.

2.1.4 Warp-Ausführungseffizienz

Nun soll eben das Konzept der Warp-Ausführungseffizienz vorgestellt werden. Wie im vorherigen Punkt erwähnt, so kann es wegen Sprungbefehlen vorkommen, dass sich der Kontrollfluss der Threads eines Warps aufspaltet und dadurch Warpdivergenz auftritt. In diesem Fall schalten sich die Threads, deren Befehl ein anderer ist, ab und nehmen nicht an den Befehlen teil. Da die Zuordnung zwischen Rechenkernen und den Threads eines Warps bei der Befehlsausführung statisch ist, bleiben die entsprechenden Rechenkerne unbenutzt. Dadurch geht Rechenleistung verloren. Die verlorene Rechenleistung lässt sich über die Warp-Ausführungseffizienz bestimmen, die nach folgender Formel definiert ist:

$$\text{Warp-Ausführungseffizienz} = \frac{\text{An einer Instruktion teilnehmende Threads}}{\text{Warp-Größe}}$$

Dabei ist die erzielte Rechenleistung direkt proportional zur Warp-Ausführungseffizienz. Deshalb stellt die Warp-Ausführungseffizienz oft einen limitierenden Faktor in der Performance von nicht perfekt datenparallelen Algorithmen, wie einer SPH-Simulation, dar. So beschäftigen sich viele Optimierungstechniken damit, wie sie die Warp-Ausführungseffizienz erhöhen können.

2.1.5 Instruktionen zur Intra-Warp-Kommunikation

Als Nächstes sollen die Instruktionen in CUDA zur Intra-Warp-Kommunikation beschrieben werden. Sie teilen sich in Warp-Vote-Functions und Warp-Shuffle-Functions auf. Die Kommunikation durch diese Funktionen ist im Vergleich zu einer Kommunikation über den Shared-Memory performanter und sie benötigt keine Synchronisationsbefehle. Allerdings kann das Kernel sie im Vergleich zum Shared-Memory nur für die Kommunikation innerhalb eines Warps und nicht für die Kommunikation innerhalb eines Thread-Blocks verwenden.

Bei den Warp-Vote-Functions gibt es zwei Befehle:

- **__all(bool Predicate):** Die Funktion liefert wahr zurück, falls das Predicate in allen Warpthreads wahr ist.
- **__any(bool Predicate):** Die Funktion liefert wahr zurück, falls das Predicate in mindestens einem Warpthread wahr ist.

Über diese beiden Instruktionen kann ein Kernel leicht einen Entscheidungsprozess auf Warp-Basis verwirklichen. Ein solcher Entscheidungsprozess ist oft für Erhöhung des Warp-Ausführungseffizienz sinnvoll.

Die Warp-Shuffle-Instruktionen teilen sich ebenfalls in zwei Instruktionstypen auf:

- **__shfl(int/float Value, int LaneID):** Hierbei erhält jeder Warpthread denjenigen Wert als Rückgabewert, welchen der Warpthread mit der Nummer „LaneID“ innerhalb des Warps als Parameter „Value“ hineingegeben hat.
- **__shfl_up(int/float Value, int Delta) und __shfl_down(int/float Value, int Delta):** Beide sind analog zur __shfl-Instruktion definiert, abgesehen davon, dass „Delta“ einen Warpthread relativ zur Position des Threads innerhalb des Warps beschreibt.

Warp-Shuffle-Funktionen sind zum Beispiel dafür vorteilhaft, wenn ein Kernel eine parallele Reduktion oder eine parallele Prefix-Sum innerhalb des Warps durchführen muss.

2.1.6 Speichermodell

2.1.6.1 Allgemeines

Als Nächstes sollen die Speicherbereiche in CUDA vorgestellt werden. So gibt es in CUDA folgende Speicherbereiche:

- **Registerspeicher:** Für automatische Variablen in den Registern
- **Lokaler Speicher:** Für automatische Variablen auf dem Stack
- **Globaler Speicher:** Für die dauerhafte Abspeicherung von Daten auf der GPU
- **Texturenspeicher:** Für die dauerhafte Abspeicherung von Texturen auf der GPU
- **Shared-Memory:** Für den Shared-Memory, bei welchem es sich um den Scratch-Pad-Speicher handelt
- **Konstantenspeicher:** Für die Konstanten, welche als Argumente beim Kernelaufruf übergeben werden.

Die soeben genannten Speicherbereiche sollen im Folgenden genauer beschrieben werden. Da der Registerspeicher, der lokale und der globale Speicher in dieser Arbeit von großer Bedeutung sind, werden sie ausführlicher erklärt. Dem Texturenspeicher, dem Shared-Memory und dem Konstantenspeicher kommen jedoch kaum Bedeutung zu, weshalb sie nur kurz am Rande erläutert werden.

Der Texturenspeicher wird verwendet um Texturen abzuspeichern. Zugriffe auf den Speicher geschehen nicht über Zeiger sondern über Texturobjekte. Dadurch ist der Speicher bufferbasiert. Bei einem Texturzugriff führen die Texture-Units automatisch die Adressberechnung und die Interpolation durch.

So erlaubt der Shared-Memory die Kommunikation der Threads innerhalb des selben Thread-Blocks. Besonders am Shared-Memory ist neben seiner hohen Bandbreite, dass er in Bänke unterteilt ist. Die Bänke erfordern ein spezielles Zugriffsmuster durch die einzelnen Warpthreads, da es sonst zu Bank-Conflicts kommt und eine Sequentialisierung entsteht.

Der Konstantenspeicher ist in der Größe auf 64 kiByte begrenzt. Zudem kann er vom Kernel aus nur gelesen werden und wird durch den performanten Konstanten-Cache zwischengespeichert.

Auch kann der Konstantenspeicher direkt als Operand für Befehle verwendet werden. Fordern mehrere Threads eines Warps unterschiedliche Adressen an so tritt eine Sequentialisierung auf.

2.1.6.2 Registerspeicher und lokaler Speicher

Folgender Punkt soll den Registerspeicher und den lokalen Speicher näher erläutern. Beide werden für automatische Variablen verwendet.

So legt der Compiler die automatischen Variablen eines Threads zu nächst in den Registern eines Multiprozessors ab. Dabei sind die Register der mit Abstand performanteste Speicher der GPU, da sie mehrere 10 TByte/s an Bandbreite besitzen. Im Gegensatz zu herkömmlichen Prozessoren ist die Anzahl der Register pro Thread nicht fest, sondern sie wird vom Compiler je nach Komplexität des Kernels gewählt. Muss ein Kernel viele automatische Variablen abspeichern, so gibt der Compiler jedem GPU-Thread des Kernels viele Register, während wenn ein Kernel wenig automatische Variablen abspeichern muss, jeder Thread nur wenige Register erhält. In vielen Anwendungen reichen dabei weniger als 32 Register pro Thread aus. Zudem lagert der Compiler erst automatische Variablen auf den Stack aus, wenn die maximale Registeranzahl von 255 Registern pro Thread erreicht ist. Der Vorgang wird im allgemeinen als Registerspilling bezeichnet.

Der lokale Speicher wird für automatische Variablen verwendet, die auf dem Stack liegen. Die automatischen Variablen werden dort wegen Registerspilling abgelegt. Zusätzlich werden dort Arrays von automatischen Variablen abgelegt, welche das Kernel dynamisch indexiert. Denn ein indexierter Zugriff ist bei Registern auch auf GPUs nicht möglich. Der lokale Speicher befindet sich im DRAM der GPU und wird durch den L2-Cache und den L1-Cache zwischengespeichert. Der L1-Cache ist allerdings pro Thread extrem klein, so dass ein jeder Thread nur wenige Byte an L1-Cache-Speicher besitzt. Im schlimmsten Fall sind es bei 16 kiByte L1-Cache-Größe nur 8 Byte an L1-Cache-Speicherplatz pro Thread. Dadurch gehen bereits bei kleinen Stacks viele lokale Speicherzugriffe auf den L2-Cache, der mit schlimmstenfalls 52 Byte pro GPU-Thread ebenfalls klein ist, oder den langsamen DRAM über. Aus diesem Grund ist der lokale Speicher verglichen mit dem Stack auf einer CPU vergleichsweise unperformant.

Des Weiteren gibt es die Möglichkeit die Register pro Thread künstlich per Compilerbefehl zu limitieren. Das Begrenzen erzielt zwar den Vorteil, dass ein Multiprozessor dadurch mehr Threads und damit mehr Warps gleichzeitig bearbeiten kann. Damit kann der Multiprozessor auch tendenziell besser die Latenzen überbrücken. Allerdings gibt es auf diese Weise auch mehr Registerspilling in den langsameren lokalen Speicher. Somit kann das Registerspilling die Performance wiederum verschlechtern. Deshalb ist die Begrenzung der Register immer ein zweischneidiges Schwert. Diese Problematik wird noch einmal in Kombination mit der Occupancy näher im Punkt 2.1.7 erläutert.

2.1.6.3 Globaler Speicher

In diesem Punkt soll der sogenannte globale Speicher näher erläutert werden. Der globale Speicher liegt im Device-Memory. Ein Host-Programm verwendet ihn für Daten, welche längerfristig über mehrere Kernel-Ausführungen hinweg im DRAM der GPU abgespeichert werden sollen. Somit ist der globale Speicher ähnlich zum Heap-Speicher bei der CPU-Programmierung.

Den globalen Speicher kann ein Host-Programm durch Befehle der CUDA-API allozieren. Zudem kann es mit Befehlen der CUDA-API Daten in den globalen Speicher und aus dem globa-

len Speicher einer GPU kopieren. Ebenfalls ist ein Kopieren zwischen dem globalen Speicher unterschiedlicher GPUs möglich. Des Weiteren wird der globale Speicher sowohl im Host-Code als auch im Kernel-Code über Zeiger repräsentiert. Die Zeiger verhalten sich weitgehend genauso wie herkömmliche C++-Zeiger. So können sie beim Zugriff vom Kernel aus dereferenziert werden. Lediglich ein Dereferenzieren vom Host aus ist nicht möglich. Ein Beispiel für die Zeigernatur des globalen Speichers ist in Abbildung 2.3 zu sehen. Speicherzugriffe auf diesen Bereich werden von dem L2-Cache und optional bei der Verwendung der __ldg-Intrinsic zusätzlich durch den Texture-Cache zwischengespeichert. Da der Texture-Cache ein Read-Only-Cache ist, darf die Intrinsic nur verwendet werden, wenn sich der entsprechende Speicherbereich während der Kernelausführung nicht verändert. Ein Speicherzugriff auf den globalen Speicher wird über 1-, 2-, 4-, 8- und 16-Byte-Operationen abgearbeitet. Für die Abarbeitung verwendet der Multiprozessor ohne Texture-Caching die Load-Store-Units und mit Texture-Caching die Texture-Units.

Hierbei arbeiten die Load-Store-Units den Speicherzugriff ebenfalls auf Warp-Basis ab. Dafür besitzen sie spezielle Coalescing-Hardware. Greifen mehrere Warpthreads auf die selben Adressen innerhalb des selben 32-Byte großen L2-Cache-Transaktionsblocks zu, so werden die Zugriffe zu einer einzigen Transaktion zusammengefasst. Dadurch werden die Daten nur einmal übertragen, wodurch im Endeffekt etwas Bandbreite eingespart wird. Deshalb sollte ein Kernel entweder mit den Warpthreads sequentiell auf den globalen Speicher zugreifen, oder alternativ bei chaotischen Zugriffen möglichst große Ladeinstruktionen verwenden. Auch lädt die GPU, sofern der Speicherzugriff über die Load-Store-Units abgearbeitet wird, bei einem L2-Cache-Miss nicht die komplette Cache-Line in den L2-Cache sondern nur diejenigen 32-Byte großen Transaktionsblöcke, die auch tatsächlich von dem Warp angefordert werden.

Wie genau ein Zugriff über den Texture-Cache beziehungsweise über die Texture-Units abgearbeitet wird, wird von NVIDIA nicht spezifiziert. Deshalb führte die Arbeit einige kleinere Benchmarks durch. Die Benchmarks ergaben, dass der Cache 16-Byte-Ladeoperationen benötigt, um seine maximale Performance zu erreichen. Das Texture-Caching hat zudem den Nachteil, dass bei einem Texture-Cache-Miss die GPU die entsprechende 128-Byte große Cache-Line komplett von dem L2-Cache in den Texture-Cache lädt. Ist die Cache-Line im L2-Cache nicht vorhanden, so fordert die GPU sie komplett aus dem DRAM an. Auf diese Weise erhöht sich durch das Texture-Caching auch der Overfetch bei chaotischen Speicherzugriffen.

Zudem können atomare Operationen auf den globalen Speicher ausgeführt werden. Sie sind durch Hardwarebeschleunigung performant und werden parallel durch mehrere sogenannte Atomic-Units abgearbeitet. Ihre Peak-Performance beträgt 64 Operationen pro Takt, wenn ein Kernel sie auf unterschiedliche Speicheradressen anwendet. Wendet ein Kernel die atomaren Operationen jedoch alle auf die ein und die selbe Adresse an so tritt eine Sequentialisierung auf. Dadurch sinkt die Performance auf eine Operation pro Takt ab. Auch unterstützen die Atomic-Units nur wenige atomare Operationen.

Der globale Speicher ist zudem ein Teil eines virtuellen Speicherraum, welcher unter anderem den Device-Memory der GPU selbst, den Device-Memory anderer GPUs und den Pinned-Memory der CPU umfasst. Zeiger auf diesen Bereich sind sowohl bei allen beteiligten GPUs als auch beim Host identisch. Dabei können GPUs prinzipiell vom Kernel aus nicht nur Zeiger auf ihren eigenen Device-Memory sondern auch auf den Device-Memory anderer GPUs dereferenzieren. Zusätzlich können die Copy-Engines den Device-Memory direkt von GPU zu GPU über den PCI-E kopieren. Dadurch bilden mehrere GPUs einen NUMA-Rechner. Dieses Feature sperrte NVIDIA allerdings auf nicht Tesla- und Quadro-Karten aus kommerziellen

Gründen. Selbst der Kopiervorgang zwischen zwei GPUs läuft wegen des Sperrens immer über den Host ab.

2.1.7 Occupancy

Nachdem die Speicherbereiche in CUDA erklärt worden sind, soll in diesem Punkt das Konzept der Occupancy genauer vorgestellt werden. So verbraucht ein Thread-Block je nach Kernel und Thread-Block-Größe unterschiedlich viele Ressourcen, nämlich Register, Shared-Memory und Verwaltungsressourcen. Deshalb kann ein Multiprozessor je nach Ressourcenverbrauch eines Thread-Blocks unterschiedlich viele Warps gleichzeitig beherbergen. Diejenige Anzahl an Warps, welche er bei einem gegebenen Kernel maximal beherbergen kann, wird als Occupancy bezeichnet. Da ein Multiprozessor viele Warps gleichzeitig benötigt, damit er gut anfallende Latenzen überbrücken kann (siehe Punkt 2.1.3), ist eine hohe Occupancy an sich meistens erstrebenswert. So gibt es für die Occupancy eines Kernels insgesamt unter Vernachlässigung des Shared-Memorys folgende Regeln:

- Ein Multiprozessor besitzt 65536 32-Bit Register
- Ein Thread muss ein Vielfaches von 8 Registern allozieren
- Ein Thread darf maximal 255 Register allozieren
- Ein Kernel muss - wegen den vier Warp-Schedulern eines Multiprozessors - immer ein Vielfaches von 4 Warps allozieren
- Ein Multiprozessor kann maximal 16 Thread-Blocks gleichzeitig bearbeiten
- Ein Multiprozessor kann maximal 64 Warps gleichzeitig bearbeiten

In den meisten Fällen limitiert der Registerverbrauch die Occupancy eines Kernels. Dementsprechend ist es sinnvoll die Occupancy in Abhängigkeit vom Registerverbrauch zu berechnen. Dies ist in folgender Tabelle zu sehen:

Register	0 bis 32	33 bis 40	41 bis 48	49 bis 56	57 bis 64	65 bis 72	73 bis 80	81 bis 96	97 bis 128	129 bis 168	169 bis 255
Occupancy	64	48	40	36	32	28	24	20	16	12	8

Aus der Tabelle folgt, dass es nur wenige sinnvolle Registeranzahlen gibt, welche ein Kernel allozieren sollte. Die Erfahrung lehrt jedoch, dass der CUDA-Compiler diese Tabelle nicht beachtetet und willkürlich Registeranzahlen wählt. So entschließt er sich oft 33 Register zu wählen obwohl dies die Occupancy von 64 auf 48 Warps und damit wahrscheinlich auch die Performance deutlich reduziert. Unter anderem deshalb ist die Begrenzung der Register in vielen Fällen lohnenswert. Jedoch muss ein Programmierer immer berücksichtigen, dass durch die Begrenzung der Register auch Registerspilling entsteht und dann der langsame lokale Speicher verwendet wird.

Problematisch ist es ebenfalls, dass der Compilerbefehl zur Begrenzung der Register im Quelltext bei der Kerneldefinition durch die sogenannten Launch-Bounds angegeben werden muss. Bei den Launch-Bounds handelt es sich um einen Funktions-Specifier, welcher

„__launch_bounds__(MAX_THREADS_PER_BLOCK, MIN_BLOCKS_PER_MP)“ lautet. Dadurch kann der Quelltext die gewünschte Occupancy nur indirekt auswählen. Insbesondere erzielt der Quelltext über diese Beschränkung nur, dass das Kernel mindestens die gewünschte Occupancy besitzt. Besaß das Kernel vor der Beschränkung bereits eine höhere Occupancy, so haben die Launch-Bounds keinerlei Auswirkung.

Letztendlich gilt es anzumerken, dass eine höhere oder hohe Occupancy nicht immer gut für die Performance ist. So erhöht sie meist auch den Working-Set für globale Speicherzugriffe, das der Cache-Hitrate schadet. Zudem gibt es neben der Occupancy noch die Möglichkeit ein Kernel so umzugestalten, dass es einen hohen Instruction-Level-Parallelism (ILP) oder einen höheren Memory-Level-Parallelism (MLP) besitzt. Dadurch zögert das Kernel das Stallen eines Warps hinaus und es kann somit Latenzen besser überbrücken. Ein höherer ILP und MLP benötigt aber oft auch mehr Register, wodurch wiederum die Occupancy reduziert wird. Auch gilt es anzumerken, dass der Quelltext die Register per Compilerbefehl nur begrenzen kann. Jedoch kann ein Programmierer in CUDA nicht die Register durch einen Compilerbefehl für einen höheren ILP und MLP erhöhen.

Somit ergeben sich drei Ansätze, wie ein Programmierer die Latenzen innerhalb eines Kernel optimieren kann: Das Optimieren oder Begrenzen des Registerverbrauchs für eine bessere Occupancy oder das Erhöhen des ILPs und MLPs. Welche dieser Ansätze genau zum Erfolg führen, ist von Fall zu Fall verschieden.

2.1.8 Tail-Effekt

Als Nächstes soll kurz der Tail-Effekt erläutert werden. Ein generelles Problem ist, dass GPUs viele Thread-Blocks beziehungsweise Threads benötigen um gut ausgelastet zu sein. Werden bei einem Kernel zu wenig Threads gestartet oder sind am Ende einer Kernelausführung nur noch wenige aktive Threads übrig, so ist die GPU schlecht ausgelastet. Dabei werden die wenigen verbleibenden aktiven Threads als Tail bezeichnet, woher auch der Name des Tail-Effekts stammt. Die Problematik des Tail-Effekts lässt sich am besten an einem Beispiel verdeutlichen. So kann eine GK-110-GPU im Vollausbau und maximaler Occupancy 30 720 Threads gleichzeitig bearbeiten. Angenommen es werden auf einer solchen GPU 30 721 Threads gestartet und jeder Thread würde eine Zeiteinheit rechnen müssen bis er terminiert. So würde die GPUs zuerst 30 720 der 30 721 Threads gleichzeitig bearbeiten und dafür eine Zeiteinheit benötigen. In dieser Zeit wäre sie vollkommen ausgelastet. Dann würde sie den einzigen verbleibenden Thread berechnen und ebenfalls eine Zeiteinheit benötigen. Die Auslastung in dieser Zeit wäre sehr gering. Da die GPU eine Zeiteinheit komplett und eine Zeiteinheit fast gar nicht ausgelastet ist, ergibt sich in diesem Beispiel eine mittlere Auslastung von in etwa 50%. Dadurch wird deutlich, dass der Tail-Effekt gerade bei kleineren Problemgrößen sich stark auf die Performance auswirken kann.

2.1.9 CUDA-Streams

Im Folgenden Punkt sollen die Streams innerhalb von CUDA vorgestellt werden. So kann ein Host-Programm für jede CUDA-GPU mehrere Streams erstellen. In die Streams kann der Host Kopieroperationen und Kernelaufrufe einreihen, welche die GPU dann asynchron abarbeitet. Ein Host-Programm kann sich selbst mit einem solchen Stream synchronisieren. Mehrere Streams untereinander kann der Host mit CUDA-Events synchronisieren. Sofern es die Abhängigkeiten in den Streams der GPU erlauben, so besitzt die GPU die Fähigkeit mehrere unterschied-

liche Kernel gleichzeitig zu berechnen. Auf diese Weise kann ein Host-Programm den Tail-Effekt vermeiden. Zudem kann die GPU Kopieroperationen mit ihren Copy-Engines ausführen, während sie ein Kernel mit ihren Multiprozessoren berechnet. Wenn während der Laufzeit viele langandauernde Kopieroperationen zu der GPU hin oder von der GPU aus stattfinden so ist es für die Performance wichtig, dass ein Programm diese Fähigkeit ausnutzt.

2.1.10 Programmierinterface

In diesem Punkt soll kurz das Programmierinterface von CUDA erläutert werden. Prinzipiell gibt es mehrere Möglichkeiten mit CUDA zu programmieren, wie eine Fortran-Anbindung, die CUDA-Runtime-API und die CUDA-Driver-API. Am komfortabelsten ist es jedoch die CUDA-Runtime-API zu verwenden, weshalb sie auch in der Masterarbeit verwendet wird.

Ein CUDA-Beispielprogramm der komponentenweisen Quadratur eines Vektors ist in 2.3 zu sehen. Dabei zeichnet sich die CUDA-Runtime-API dadurch aus, dass sowohl der Host-Code als auch der Device-Code in ein und die selben Quelltextdateien geschrieben werden. Bei dem Quelltext kann es sich sowohl beim Host-Code als auch beim Device-Code um C++ oder um C mit einigen CUDA spezifischen Erweiterungen handeln. Deshalb bezeichnet NVIDIA diese Sprache als CUDA C beziehungsweise CUDA C++. Während des Kompilierungsprozesses zieht der CUDA-Compiler den Device-spezifischen Code heraus und generiert aus den Quelltextdateien gewöhnlichen C++-Code beziehungsweise C-Code für den Host.

Da CUDA im Device-Code den C++-Standard abgesehen von einigen kleineren Restriktionen fast vollkommen unterstützt, kann ein Programmierer sich in Kernels sämtliche C++-Konstrukte wie Klassen oder Templates zu Nutze machen. Ein weiterer Vorteil ist, dass sich der Device-Code sämtliche Klassen und Structures mit dem Host-Code teilt. Dabei wird sichergestellt, dass die Klassen und Structures das selbe Alignement und Padding besitzen. Dieses identische Packing erleichtert einem Programmierer es stark die Kommunikation zwischen dem Host und dem Device zu programmieren. Zudem ist es möglich ein und die selbe Funktion sowohl von einem Kernel als auch vom Host aus aufzurufen, wodurch ein Programmierer redundanten Quelltext einsparen kann. Schließlich ist es ebenfalls vorteilhaft, dass die Speicherbereiche in CUDA weitestgehend zeigerbasiert sind. Lediglich der Texturspeicher ist bufferbasiert. Dabei verhalten sich die Zeiger weitgehend so wie normale C++-Zeiger.

2.1.11 Alternativen zu CUDA

Folgender Punkt soll kurz Alternativen zu CUDA vorstellen. Zudem begründet er, weshalb CUDA für die Arbeit verwendet wurde. Neben CUDA gibt es als GPGPU-Alternative hauptsächlich noch OpenCL, OpenGL-Compute-Shader, C++-AMP, OpenACC, und Direct-Compute. Da OpenGL-Compute-Shader sowie Direct-Compute nur einen geringen Funktionsumfang besitzen und C++-AMP sowie Direct-Compute an Windows gebunden sind, bleiben als GPGPU-Alternativen nur noch OpenCL und OpenACC übrig.

Da OpenCL sehr beliebt ist, sollen zuerst die Vor- und Nachteile von OpenCL im Vergleich zu CUDA herausgearbeitet werden. OpenCL besitzt als Vorteil, dass es offen ist und nicht nur NVIDIA-GPUs sondern auch AMD-GPUs und CPUs für Berechnungen unterstützt. Nachteilig ist jedoch, dass es die Hardware nicht oder nur kaum spezifiziert und ein Programm damit diverse Funktionalitäten von NVIDIA-GPUs nicht verwenden kann. Dadurch wird die hardwarenahe Programmierung, welche in dieser Arbeit ein wichtiger Optimierungsansatz ist, er-

```
//Kernelfunktion
__global__ void CalculateSquareOfVectorKernel(float* VectorGPU, int VectorSize)
{
  //Bestimme welche Komponente Thread quadrieren soll
  int ThreadID = blockIdx.x * blockDim.x + threadIdx.x;
  //Abbruch falls die Komponente des Threads nicht mehr im Vektor
  if(ThreadID >= VectorSize)
    return;
  //Führt Quadratur aus
  VectorGPU[ThreadID] = VectorGPU[ThreadID] * VectorGPU[ThreadID];
}

//Hostfunktion zum Aufruf des Kernels
void CalculateSquareOfVector(float* VectorHost, int VectorSize)
{
  //Alloziere Vektor auf der GPU
  float* VectorGPU;
  cudaMalloc(&VectorGPU, VectorSize*sizeof(float));
  //Kopiere Vektor zur GPU
  cudaMemcpy(VectorGPU,VectorHost, sizeof(float)*VectorSize, cudaMemcpyDefault);

  //Bestimme Zahl der zu startenden Thread-Blocks
  //Dabei ist ein Aufrunden erforderlich, damit genügend Threads gestartet werden
  int ThreadBlockSize = 256;
  int ThreadBlockCount = VectorSize/ThreadBlockSize+1;
  //Starte Kernel
  VectorSquareKernel<<<ThreadBlockCount,ThreadBlockSize>>>(VectorGPU,VectorSize);

  //Kopiere Vektor zurück zu Host
  cudaMemcpy(VectorHost,VectorGPU, sizeof(float)*VectorSize, cudaMemcpyDefault);
  //Gebe Speicher auf der GPU wieder frei
  cudaFree(VectorGPU);
}
```

Abbildung 2.3: Beispiel eines CUDA-Programms: Hierbei wird ein Vektor im Speicher des Hosts zuerst auf die GPU hochgeladen, dann dort komponentenweise quadriert, und anschließend wieder zurück zum Host kopiert

schwert. Des Weiteren unterstützt es im Vergleich zu CUDA keine Zeiger sondern nur Buffer-Objekte. Zudem sind bei OpenCL die Host-Programme und die Device-Programme logisch gesehen stark voneinander getrennt, wodurch sich die Programmierung stark erschwert. Auch beherrscht OpenCL nur einen C-Dialekt für Kernels, weshalb hier der C++-Dialekt von CUDA vorteilhafter ist.

Weiter erschwerend kommt hinzu, dass NVIDIA zum Verfassungszeitpunkt seine OpenCL-Implementierung nicht mehr weiterentwickeln zu scheint. Dementsprechend unterstützt NVIDIA statt der seit dem Jahr 2013 aktuellen OpenCL-Version 2.0 oder der seit dem Jahr 2011 spezifizierten OpenCL-Version 1.2 nur die OpenCL-Version 1.1 aus dem Jahr 2010. Auch beherrscht NVIDIAs OpenCL-Compiler neuere GPU-Features nicht, wie zum Beispiel das Caching von globalen Speicherzugriffen im Texture-Cache. Ebenfalls funktionieren neuere Versionen von NVIDIAs Visual-Profiler, der ein sehr mächtiges Profiling-Tool ist, nur noch unter

CUDA. Zudem sind alle primären Test-GPUs NVIDIA-GPUs. Deshalb scheidet OpenCL im Vergleich zu CUDA und OpenACC aus.

OpenACC besitzt den Vorteil, dass es offen ist und nicht nur auf NVIDIA-GPUs lauffähig ist. Jedoch besitzt es im Ausgleich dafür wieder weniger Hardware-Nähe. Da sich die folgende Arbeit mit hardwarenahen Optimierungen beschäftigt, scheidet es ebenfalls aus. Somit bleibt nur CUDA als einzige Möglichkeit für diese Arbeit übrig.

2.2 OpenGL-Grundlagen

Nachdem soeben ein Überblick über CUDA gegeben worden ist, soll dieser Punkt kurz die benötigten OpenGL-Grundlagen erläutern. Da OpenGL jedoch sehr komplex, chaotisch, und umfangreich ist - wesentlich umfangreicher als CUDA - wird hier nur das allernötigste erklärt. Für eine ausführlichere Erklärung kann der interessierte Leser zum Beispiel das Buch [DW] verwenden. Prinzipiell gibt es für OpenGL nur DirectX als Alternative. DirectX konnte in dieser Arbeit nicht verwendet werden, weil die Implementierung der Arbeit unter Linux lauffähig sein muss und DirectX nur unter Windows funktioniert.

So ist OpenGL eine Rasterisierungs-API. Sie beschäftigt sich im Wesentlichen mit der Bildsynthese einer 3D-Szene durch einen Rasterisierungsprozess. Dafür werden dreidimensionale geometrische Primitive, also Punkte, Linien und Dreiecke, auf einen Bildschirm projiziert und anschließend rasterisiert. Für diese Projektion ist es nötig, die Eckpunkte der geometrischen Primitive per affiner Transformation vom sogenannten World-Space-Koordinatensystem ins sogenannte Clip-Space-Koordinatensystem zu überführen. Damit die Zeichenreihenfolge der einzelnen Primitive für das Endergebnis beinahe und meistens egal ist, verwendet OpenGL zusätzlich einen Tiefentest mit einem Tiefenbuffer.

Da die GPU konzeptbedingt beim Rasterisierungsprozess keinen wahlfreien Zugriff auf die Szenengeometrie besitzt, muss eine Visualisierung sich bei vielen Spezialeffekten, wie Transparenz oder Schattenwurf, diverser Tricks bedienen. Für die Tricks zeichnet eine Visualisierung die 3D-Szene mehrmals in unterschiedliche Bilder beziehungsweise Framebuffer. Auch finden diese Zeichenvorgänge oft mit mehreren unterschiedlichen Kameras statt. In die Framebuffer speichert sie dabei nicht nur normale Farbwerte, sondern je nach Anwendungsfall unterschiedliche Daten, wie Tiefenwerte, Dickewerte, Normalen, oder Materialwerte ab. Dadurch zeichnet eine Visualisierung von einem bestimmten Betrachtungspunkt aus gesehen eine Approximationen der Szene. Die Approximation kann sie dann weiter für die Spezialeffekte verwenden, indem sie die Framebuffer als Texturen bindet und anschließend bei den weiteren Zeichenpässen von der GPU aus auf diese Texturen zugreift.

Der Rasterisierungsprozess von OpenGL wird durch die sogenannte Rasterpipeline beschrieben. Manche Stufen der Rasterpipeline sind konfigurierbar, während andere Stufen durch sogenannte Shader-Programme programmierbar sind. Dabei führen die Raster-Engines der GPU die konfigurierbaren Stufen hardwarebeschleunigt aus, während die Multiprozessoren die Shader-Programme analog wie CUDA-Kernel berechnen. Zudem sind viele Stufen optional. Deshalb wird im Folgenden speziell die Pipeline beschrieben, wie sie bei der Visualisierung der SPH-Simulation benötigt wird. So definiert eine Visualisierung zuerst, in welche Bilder sie zeichnen möchte, indem sie die Bilder in OpenGL als Framebuffer bindet. Des Weiteren gibt die Visualisierung an, welche Vertex-Arrays mit Vertex-Daten sie verwenden möchte. Zudem kann sie Texturen definieren, aus welchen die GPU bei jedem Shader wahlfrei lesen kann. Auch muss sie

die Werte der Konstanten für die Shader setzen. Abschließend startet die Visualisierung einen OpenGL-Zeichenaufruf mit Anzahl und Art der zu zeichnenden Primitive, wie Punkte oder Dreiecke. Durch den Zeichenaufruf wird auf der GPU die Rasterpipeline abgearbeitet, welche in Abbildung 2.4 zu sehen ist. Sie setzt sich aus folgenden Stufen zusammen:

- **Vertex-Shader:** Die GPU ruft den Vertex-Shader für jeden Vertex der zu zeichnenden Primitive einmal auf. Er erhält die entsprechenden Einträge seines Vertexes aus den Vertex-Arrays als Eingabedaten. Anhand dieser kann er beliebige Berechnungen ausführen, wie zum Beispiel die Position des Vertexes zu transformieren. Anschließend reicht er seine Endergebnisse an den Geometry-Shader weiter.
- **Geometry-Shader:** Der Geometry-Shader wird einmal pro zu zeichnenden Primitiv aufgerufen. Dabei hat er bei jeden Eckpunkt seines Primitivs Zugriffs auf die Endergebnisse des Vertex-Shaders. Dadurch kann er ebenfalls beliebige Berechnungen ausführen. Anhand der Berechnungen muss er ausgehende Primitive erzeugen (Punkte, Linien oder Dreiecke). Für jeden Eckpunkt dieser Primitive muss er festlegen, welche Position im Clip-Space der Eckpunkt später haben soll. Zudem können für jeden Eckpunkt weitere Attribute definiert werden. Sowohl die Positionen im Clip-Space auch die weiteren Attribute werden der Rasterization-Stufe übergeben.
- **Rasterization-Stufe:** Als Nächstes wird die Rasterization-Stufe aufgerufen. Diese arbeitet jedes aus dem Geometry-Shader eingehende Primitiv ab. Dabei bestimmt sie, ausgehend von den Eckpunkten im Clip-Space, welche Pixel auf dem Bildschirm das Primitiv belegt. Diese Pixel werden als Fragmente bezeichnet. Zudem wird, je nach Konfiguration des Fragment-Shaders, bereits ein früher Tiefentest ausgeführt. Der frühe Tiefentest wird in der Literatur als Early-Z-Rejection bezeichnet. Besteht das Fragment den Tiefentest nicht, so kann es verworfen werden. Dadurch kann die Rasterization-Stufe im Optimalfall ein Großteil aller Fragmente aussortieren und viel Rechenleistung einsparen. Besteht das Fragment den Tiefentest, so werden die aus dem Geometry-Shader eingehenden Attribute des Primitivs innerhalb des Primitivs an der Stelle des Fragments perspektivisch korrekt interpoliert.
- **Fragment-Shader:** Die GPU ruft den Fragment-Shader einmal pro Fragment auf. Der Fragment-Shader erhält die perspektivisch korrekt interpolierten Attribute als Eingabedaten von der Rasterization-Stufe. Anhand der Eingabe kann der Fragment-Shader mehrere Endergebnisse errechnen, wie zum Beispiel die Farbe des Pixels. Die Endergebnisse muss er dann an die Per-Fragment-Operations-Stufe zum Zurückschreiben in den Framebuffer weiterreichen. Während der Pixel des Fragments fest ist, so kann der Fragment-Shader noch einmal den Tiefenwert seines Fragments frei verändern. Zusätzlich kann er sein Fragment verwerfen. Durch das Verwerfen werden seine Endergebnisse nicht an die Per-Fragment-Operations-Stufe weitergereicht.
- **Per-Fragment-Operations-Stufe:** Diese führen für jedes Fragment noch einmal einen Tiefentest aus. Besteht das Fragment den Tiefentest, dann wird der Tiefenbuffer aktualisiert und die Endergebnisse des Fragment-Shaders werden nach einem erfolgten Blending in den Framebuffer zurückgeschrieben. Dabei gibt das Blending an, wie die Endergebnisse des Fragment-Shaders mit den Werten, welche an der Stelle des Fragments bereits in dem Framebuffer stehen, kombiniert werden. Eine mögliche Kombinationsmöglichkeit ist die Addition, bei der der neue Wert vor dem Zurückschreiben zu dem alten Wert addiert wird.

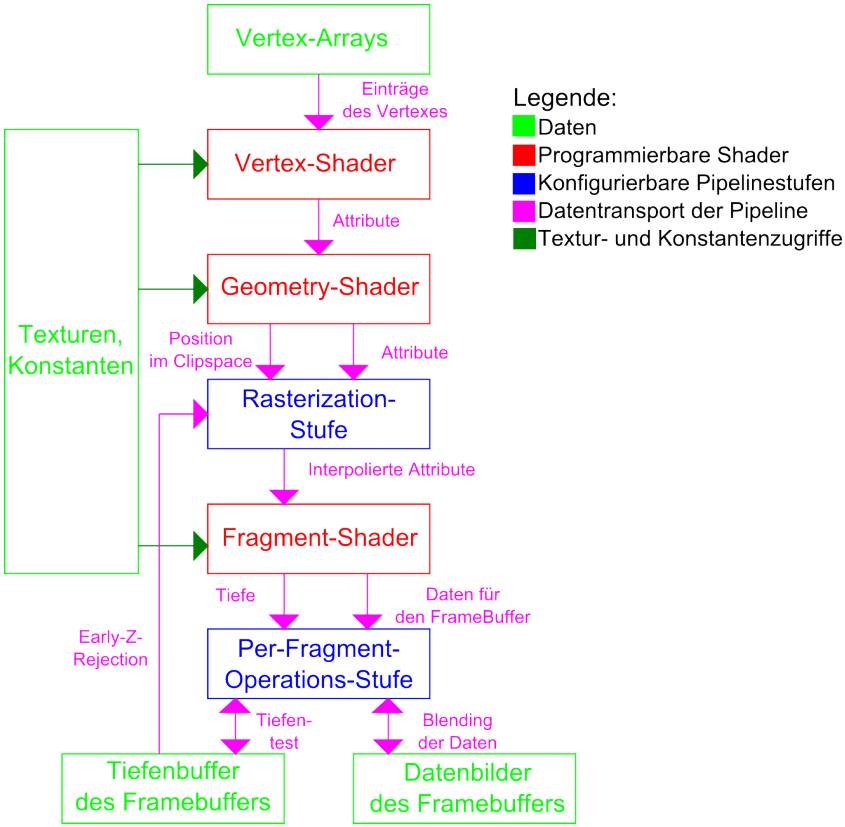


Abbildung 2.4: Überblick über die OpenGL-Pipeline

Nach diesem Überblick soll noch einmal die Interpolation der Attribute durch die Rasterization-Stufe näher erläutert werden. Durch die perspektivisch korrekte Interpolation werden alle Attribute auf dem Bildschirm korrekt interpoliert, welche innerhalb des Primitivs in dem Clip-Space und damit in dem World-Space gesehen linear verlaufen. Denn würde die GPU die Perspektive beziehungsweise die Tiefe bei der Interpolation nicht berücksichtigen, und einfach auf dem Bildschirm gesehen linear interpolieren, so würden sich störende Verzerrungen ergeben. Dies ist noch einmal in Abbildung 2.5 zu sehen. Aus dem Grund kann ein Programmierer sämtliche Berechnungen in einem Shader-Programm, welche innerhalb der Fragmente eines Primitivs im Clip- oder World-Space gesehen linear verlaufen, in den Geometry-Shader hochziehen. Anschließend muss er die Rechenergebnisse im Shader-Programm der Rasterization-Stufe zur Interpolation für die Fragmente übergeben. Da ein Primitiv in den allermeisten Fällen aus wesentlich mehr Fragmenten als aus Eckpunkten besteht und die Interpolation hardwarebeschleu-

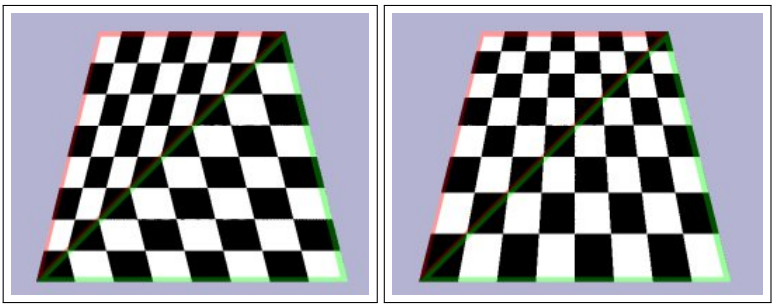


Abbildung 2.5: Unterschied zwischen linearer und perspektivisch korrekter Interpolation: Zu sehen ist ein gerendertes Schachbrett bestehend aus zwei rasterisierten Dreiecken mit einer Schachbretttextur. Auf dem linken Bild wurden die Texturkoordinaten als Fragmentattribut auf dem Bildschirm gesehen linear interpoliert. Dadurch sind deutliche Verzerrungen zu erkennen. Durch die perspektivisch korrekte Interpolation auf dem rechten Bild lassen sich die Verzerrungen vermeiden. Quelle des Bildes ist [WP].

nigt ist, kann eine Visualisierung sich dadurch viel Rechenleistung der GPU einsparen. Ein gutes Beispiel hierfür ist der Vektor von der Position der Kamera zur Position des Fragments im World-Space. Entweder könnte ein Shader-Programm, um diesen zu berechnen, die Position der Eckpunkte des Primitivs im World-Space zur Interpolation an die Rasterization-Stufe übergeben. Anschließend müsste es dann im Fragment-Shader den Vektor per Differenz zwischen Kameraposition und Fragmentposition berechnen. Da der Vektor allerdings innerhalb der Fragmente eines Primitivs immer linear verläuft könnte das Shader-Programm die Differenz ebenfalls bereits im Geometry-Shader berechnen. Anschließend müsste es die Differenz durch die Rasterization-Stufe interpolieren lassen. Dadurch könnte das Shader-Programm sich drei FLOPs pro Fragment einsparen. Würde das Shader-Programm aber den normalisierten Vektor zwischen Kamera und Fragment im Fragment-Shader benötigen, so wäre durch die Normalisierung der Verlauf des Vektors im World-Space nicht mehr linear. Deshalb darf hier ein Programmierer die Normalisierung nicht mehr in den Geometry-Shader hochziehen.

Als Nächstes soll noch kurz auf die Early-Z-Rejection eingegangen werden. Damit die Early-Z-Rejection der GPU stattfinden kann, darf der Fragment-Shader den Tiefenwert seines Fragments nicht ändern. Allerdings gibt es in neueren OpenGL-Versionen auch die Möglichkeit den Fragment-Shader im Quelltext angeben zu lassen, dass die Tiefenwerte aller Fragmente durch die Änderung gegenüber ihren ursprünglichen Werten nur vergrößert werden. Unter dieser Voraussetzung kann dann die GPU die Early-Z-Rejection dennoch verwenden. Die Effizienz der Early-Z-Rejection kann dadurch jedoch gemindert werden. Zudem muss die Visualisierung die Geometrie der Szene immer von vorne nach hinten zeichnen, damit die Early-Z-Rejection gut funktionieren kann.

Des Weiteren gilt es immer zu berücksichtigen, dass die Rasterpipeline durch die Hardwarebeschleunigung an sich performant ist. Jedoch kann jedes ihrer einzelnen Glieder die Performance limitieren. So besitzen GPUs neben dem Rechendurchsatz der Shader (gemessen in FLOPS) auch noch einen Dreiecksdurchsatz (gemessen in Dreiecke pro Sekunde) und eine Füllrate für

die Pixel im Framebuffer (gemessen in Pixel oder Fragmenten pro Sekunde). Dadurch kann ein Programmierer bei einer Visualisierung zum Beispiel immer mit Performanceeinbußen rechnen, wenn die Visualisierung mit einfachen Shadern sehr viele kleine Dreiecke oder sehr viele große Dreiecke zeichnet.

Letztendlich gilt anzumerken, dass der Geometry-Shader ein Bestandteil der Pipeline ist, welches nur selten verwendet wird. Wird er nicht verwendet so steuert der Vertex-Shader direkt die Rasterization-Stufe an.

2.3 Interoperabilität zwischen OpenGL und CUDA

Nachdem CUDA und OpenGL vorgestellt wurden, soll kurz die Interoperabilität zwischen beiden näher erläutert werden. So existiert die Möglichkeit von CUDA aus auf OpenGL-Speicherobjekte, wie Texturen oder auf Arrays mit Vertex-Daten zuzugreifen. Dafür muss das Host-Programm die OpenGL-Objekte zunächst für CUDA beanspruchen. Währenddessen darf nur CUDA auf die Objekte zugreifen. Der Zugriff erfolgt dabei direkt, weshalb bei der Interoperabilität keine unnötigen Kopieroperationen entstehen. Anschließend muss das Programm die Objekte frei geben, damit OpenGL wieder auf die Objekte zugreifen darf. Diese Art der Interoperabilität ist vom Software-Design in vielen Fällen nachteilig, da dadurch oft CUDA-Programmteile, welche allgemeine Berechnungen durchführen, von OpenGL und damit von der Visualisierung abhängig werden. Dies verstößt gegen den Grundsatz, dass die Visualisierung und das restliche Programm logisch gesehen immer getrennt sein sollten. Aus dem Grund wäre für das Software-Design eine Art der Interoperabilität sinnvoller, bei welcher OpenGL auf CUDA-Objekte zugreifen könnte. Diese Art der Interoperabilität existiert jedoch nicht.

2.4 Beschränkungen durch Consumer-GPUs

In diesem Punkt sollen nun diejenigen Einschränkungen diskutiert werden, die sich durch die Verwendung von Consumer-GPUs für ein GPGPU-Projekt wie diese Masterarbeit ergeben. So verkauft NVIDIA ein und die selben GPU-DIEs in mehreren GPU-Serien: Einer Consumer-Serie, welche sie als Geforce bezeichnen, eine Workstation-Serie, die Quadro heißt, und eine GPU-Computing-Serie, die als Tesla vermarktet wird. Aus wirtschaftlichen Gründen sind die Workstation-Serie und die GPU-Computing-Serie trotz identischer DIEs um ein Vielfaches teurer als die Consumer-Serie.

Damit NVIDIA diese teureren GPUs dennoch verkaufen kann, baut es in seine Consumer-GPUs diverse zum Teil künstliche Limitierungen ein. Die Wesentlichen davon sind:

- Wie bereits im Punkt 2.1.6.3 erwähnt so bilden mehrere NVIDIA-GPUs innerhalb eines einzigen Rechners prinzipiell ein NUMA-System und können gegenseitig auf ihren globalen Speicher zugreifen. Zudem besitzen NVIDIA-GPUs die Möglichkeit direkt über einen Netzwerkadapter mit anderen NVIDIA-GPUs zu kommunizieren. Beide Features sind jedoch auf Geforce-Karten deaktiviert.
- Eine der beiden Copy-Engines ist bei Geforce-Karten deaktiviert.
- Bei den meisten Geforce-Karten ist die DPFP-Performance künstlich stark reduziert.
- Es gibt keinen ECC-DRAM auf Geforce-Karten.

- Da OpenGL zum Teil archaisch ist und aus vor Zeiten mit weit verbreiteten 3D-GPUs stammt, kennt OpenGL an sich keine GPUs. Dadurch ist die Kontexterstellung Plattformspezifisch. Hierbei tritt jedoch das Problem auf, dass ein Programm unter Windows nur einen OpenGL-Kontext für die primäre NVIDIA-GPU erstellen kann. Es existiert zwar prinzipiell eine NVIDIA spezifische OpenGL-Extension, welche auch ein Zeichnen mit mehreren GPUs unter OpenGL mit Windows ermöglicht. Diese ist jedoch nur für Quadro- und Tesla-Karten verfügbar. Deshalb kann ein Programm, wenn es mehrere Geforce-Karten in einem Windows-Rechner verwendet, nur die primäre Geforce-Karte für OpenGL Zeichenaufrufe verwenden.
- Unter Windows ist es wegen dem Windows Display Driver Model (WDDM) nicht möglich mit einem Prozess, der nicht lokal sondern über MPI von einem anderem Rechner gestartet worden ist, auf die Display-Adapter zuzugreifen. Damit ist auch kein Zugriff auf OpenGL und CUDA möglich. NVIDIA hat für dieses Problem einen speziellen Treiber entwickelt, den sogenannten TCC-Treiber. Dieser nimmt die GPUs aus der WDDM-Umgebung heraus, so dass ihre Ansteuerung auch unter Windows von einem MPI-Prozess aus möglich ist. Der TCC-Treiber ist nur auf Tesla-Karten verfügbar. Deshalb kann ein Programm unter Windows Geforce-Karten nicht in Kombination mit MPI verwenden.

Da in dieser Arbeit aus finanziellen Gründen nur herkömmliche Geforce-Consumer-GPUs zur Verfügung standen, ist die Implementierung auch den soeben genannten Einschränkungen unterworfen. Dabei war das Fehlen von ECC-DRAM egal, da in der Arbeit nur einfachere Tests mit kürzerer Laufzeit ausgeführt werden. Ebenso ist die reduzierte DPFP-Performance nicht von Bedeutung, da in folgender Arbeit die GPUs alle Berechnungen mit einfacher Genauigkeit durchführen. Das Fehlen der NUMA-Unterstützung und des direkten Netzwerkzugriffs durch die GPUs war lediglich etwas ärgerlich. Denn beides erlaubt elegantere und eventuell etwas performantere Lösungen bei der Cluster-Programmierung. Das gleiche gilt für das Fehlen der zweiten Copy-Engine. Die beiden Windows-spezifischen Restriktionen waren am problematischsten. Somit würde in folgender Arbeit viel Rechenleistung dadurch verloren gehen, dass ein Programm unter Windows und OpenGL nur eine GPU pro Rechner für das Rendering verwenden kann. Am problematischsten ist jedoch, dass unter Windows ein Programm Consumer-GPUs nicht in Kombination mit MPI über das Netzwerk ansteuern kann. Deshalb ist eine SPH-Simulation für einen GPU-Cluster nicht unter Windows mit Consumer-GPUs realisierbar. Da diese Limitierungen nicht zu Beginn des Projekts bekannt waren, führten sie dazu, dass die Implementierung der Arbeit während ihrer Entwicklung nach Linux portiert werden musste.

3 SPH-Grundlagen

3.1 Übersicht

In diesem Kapitel sollen nun die benötigten SPH-Grundlagen vorgestellt werden. Die Erläuterungen aus folgendem Kapitel wurden weitestgehend aus den Quellen [CH], [GGa] und [DV] übernommen. Dabei sind sie primär sehr stark an [CH] angelehnt.

Wie bereits in der Motivation vorgestellt, so diskretisiert eine SPH-Simulation die Flüssigkeit durch einzelne Partikel, welche sich mit dem Geschwindigkeitsfeld der Flüssigkeit mitbewegen. Für die Diskretisierung werden hierbei nur Partikel innerhalb des Volumens der Flüssigkeit benötigt, die dort auch unregelmäßig verteilt sein dürfen. Jedem Partikel werden an seiner Position $\vec{r}$ die dortige Geschwindigkeit $\vec{v}$ und Dichte ρ der Flüssigkeit als Strömungsgrößen zugewiesen. Des Weiteren besitzt jedes Partikel eine zeitlich konstante Masse m ausgehend vom Flüssigkeitsvolumen, welches das Partikel repräsentiert. Zunächst muss die Ortsdiskretisierung derjenigen partiellen Differentialgleichungen, die die Flüssigkeit beschreiben, ausgeführt werden. Bei den partiellen Differentialgleichungen handelt es sich in dieser Arbeit um die Kontinuitätsgleichung der Dichte und um die Impulsgleichung. Die Ortsdiskretisierung diskretisiert beziehungsweise eliminiert dabei die räumlichen Ableitungen in den partiellen Differentialgleichungen mit Hilfe der SPH-Interpolation. Dafür interpoliert und diskretisiert die SPH-Interpolation die Skalar-, Vektor- und Gradientenfelder der partiellen Differentialgleichungen im Raum an den Schwerpunkten eines jeden Partikels. Für die Interpolation dienen die Partikel in der näheren Umgebung mit ihren Strömungsgrößen als Stützstellen. Auf diese Weise berechnet die Ortsdiskretisierung bei einer SPH-Simulation die zeitlichen Änderungen der Strömungsgrößen $\frac{D\vec{v}}{Dt}$ und $\frac{D\rho}{Dt}$ für jedes Partikel. Somit transformiert die Ortsdiskretisierung die partiellen Differentialgleichungen in die gewöhnlichen Differentialgleichungen der zeitlichen Ableitungen. Anschließend können die gewöhnlichen Differentialgleichungen durch die Zeitdiskretisierung beziehungsweise die Zeitintegration für jedes Partikel gelöst werden. Dies soll nun alles im Detail erläutert werden.

3.2 Strömungsmechanische Grundlagen

Zuerst sollen hierfür die Differentialgleichungen vorgestellt werden, welche zur Beschreibung einer Flüssigkeit benötigt werden. So besitzt eine Flüssigkeit in der einfachsten Form zwei Erhaltungsgrößen, nämlich die Massenerhaltung und die Impulserhaltung. Während die Massenerhaltung durch die Kontinuitätsgleichung der Dichte beschrieben wird, wird die Impulserhaltung durch die Impulsgleichung beschrieben. Bei beiden handelt es sich um partielle Differentialgleichungen. Zusätzlich wird die gewöhnliche Differentialgleichung der Geschwindigkeitsdefinition für die Bewegung der Partikel benötigt. Alle drei Differentialgleichungen sind miteinander gekoppelt. Weitere Flüssigkeitseigenschaften, wie die innere Energie, ließen sich durch weitere Differentialgleichungen ebenfalls beschreiben und mit SPH simulieren. Darauf wird jedoch in dieser Arbeit verzichtet. Da sich die Partikel mit der Flüssigkeit mitbewegen, benötigt eine SPH-Simulation für ihre Differentialgleichungen die Lagrang'sche Betrachtungsweise. Für diese Betrachtungsweise werden die Gleichungen so umgeformt, dass die linke Seite der Gleichung der materiellen Ableitung der benötigten Größe entspricht. Die materielle Ableitung lautet wie folgt:

$$\frac{D\Phi}{Dt} = \frac{\delta\Phi}{\delta t} + \vec{v} \cdot \nabla\Phi \tag{3.1}$$

Sie beschreibt durch $\frac{D\Phi}{Dt}$ die zeitliche Änderung der skalaren oder vektoriellen Feldgröße Φ für einen masselosen Beobachter, welcher sich innerhalb des Feldes mit dem Geschwindigkeitsfeld $\vec{v}$ bewegt. Dabei ist $\frac{\delta\Phi}{\delta t}$ die lokale zeitliche Änderung und $\vec{v} \cdot \nabla\Phi$ die Advektion durch die Bewegung des Beobachters.

3.2.1 Geschwindigkeitsdefinition

Zunächst wird die zeitliche Positionsänderung eines Beobachters in Lagrang'scher Betrachtungsweise, also für einen masselosen Beobachter der sich mit dem Geschwindigkeitsfeld der Flüssigkeit bewegt, benötigt. Sie folgt aus der Geschwindigkeitsdefinition und lautet:

$$\frac{D\vec{r}}{Dt} = \vec{v} \tag{3.2}$$

Sie beschreibt durch $\frac{D\vec{r}}{Dt}$ die zeitliche Positionsänderung für die Partikel der Flüssigkeitssimulation.

3.2.2 Impulsgleichung

Die Teilchen innerhalb einer Flüssigkeit wirken auf sich gegenseitig Druckkräfte und Reibungskräfte aus. Zusätzlich wirken von außen externe Kräfte auf die Flüssigkeit, wie die Gravitationskraft oder die Corioliskraft. Durch diese Kräfte werden die Teilchen der Flüssigkeit beschleunigt und abgebremst. Diese Geschwindigkeitsänderungen werden durch die Impulsgleichung beschrieben. Die Darstellung der Impulsgleichung in der Lagrang'schen Betrachtungsweise lautet:

$$\frac{D\vec{v}}{Dt} = -\frac{1}{\rho}\nabla P + \Gamma + \vec{g} \tag{3.3}$$

Die Impulsgleichung in der Lagrang'schen Betrachtungsweise basiert auf dem zweiten newtonschen Gesetz $a = \frac{F}{m}$ nur dass sie die Beschleunigung für ein infinitesimales Massenelement der Flüssigkeit berechnet. Die Impulsgleichung beschreibt durch $\frac{D\vec{v}}{Dt}$ die Beschleunigung für die Partikel der Flüssigkeitssimulation. Bei ihr ist $-\frac{1}{\rho}\nabla P$ die Beschleunigung, die das Partikel durch das Druckgefälle ∇P innerhalb der Flüssigkeit erfährt, $\vec{g}$ eine Beschleunigung durch externe Kräfte, wobei in dieser Arbeit nur die Gravitationskraft verwendet wird, und Γ die Beschleunigung durch einen dissipativen Term, welcher für die Viskosität beziehungsweise die Reibung innerhalb der Flüssigkeit verantwortlich ist.

Der Druck einer nur schwer komprimierbaren Flüssigkeit wie Wasser lässt sich über die Tait-Zustandsgleichung berechnen:

$$P = B\left(\left(\frac{\rho}{\rho_0}\right)^{\gamma} - 1\right) \tag{3.4}$$

Dabei sind B und ρ_0 die Referenzwerte für Druck und Dichte. B lässt sich wiederum aus $\frac{c^2\rho_0}{\gamma}$ berechnen, wobei c die Schallgeschwindigkeit bei der Referenzdichte ist. Für Wasser gilt $\rho_0 = 1000\frac{kg}{m^3}$, $\gamma = 7$ und $c = 1484\frac{m}{s}$. Bei der Tait-Zustandsgleichung können bereits kleinere Dichteschwankungen in Bezug zur Referenzdichte ρ_0 hohe Druckunterschiede und damit

starke Rückstellkräfte erzeugen. Die Rückstellkräfte bewirken wiederum, dass die Dichte nur kaum von ρ_0 abweicht. Durch diese Art von Rückstellkräften ist die Simulation einer solchen Flüssigkeit ein steifes Problem, welches nur sehr kleine Zeitschritte erlaubt. Um die Steifheit deutlich zu reduzieren und die Zeitschritte stark zu erhöhen nehmen SPH-Simulationen deshalb meist nicht die eigentliche sehr große Schallgeschwindigkeit, sondern eine deutlich kleinere numerische Schallgeschwindigkeit. Diese wird im Vorneherein auf mindestens das zehnfache der maximal zu erwartenden Geschwindigkeit der Flüssigkeit gesetzt. Auf diese Art bleiben die Dichteschwankungen kleiner als 1%. In folgender Arbeit wird immer ein Wert von $c = 100\frac{m}{s}$ als numerische Schallgeschwindigkeit verwendet.

Es ist hierbei interessant anzumerken, dass der Druck gemäß der Tait-Zustandsgleichung nur von der Dichte beziehungsweise der infinitesimalen Masse an einem Punkt abhängig ist. Aus diesem Grund ist die Kraft, die der Druckgradient erzeugt, ebenfalls kurzreichweitig. Deshalb muss eine Flüssigkeitssimulation für die Berechnung des Druckgradienten an einem gegebenen Punkt auch nur die Flüssigkeit in der näheren Umgebung betrachten. Dadurch wird erst wiederum die kurzreichweitige SPH-Interpolation ermöglicht. In dieser Hinsicht unterscheiden sich Flüssigkeitssimulationen von N-Körper-Simulationen. Denn die Gravitationskraft besitzt eine unendlich große Reichweite. Deshalb muss wiederum eine N-Körper-Simulation die Massenverteilung der gesamten Simulation betrachten um das Gravitationsfeld an einem gegebenen Punkt zu berechnen.

3.2.3 Kontinuitätsgleichung

Durch das Geschwindigkeitsfeld findet auch ein Massentransport innerhalb der Flüssigkeit statt, der wiederum die Dichteverteilung innerhalb der Flüssigkeit ändert. Diese Dichteänderung wird durch die Kontinuitätsgleichung beschrieben, die aus der Massenerhaltung folgt. Ihre Darstellung in der Lagrang'schen Betrachtungsweise ist:

$$\frac{D\rho}{Dt} = -\rho\nabla \cdot \vec{v} \tag{3.5}$$

Sie beschreibt durch $\frac{D\rho}{Dt}$ die zeitliche Dichteänderung für die Partikel der Flüssigkeitssimulation.

3.3 SPH-Interpolation

Als Nächstes wird die Grundidee hinter SPH, die Interpolation einer skalaren oder vektoriellen Funktion und des Gradienten einer skalaren Funktion im Raum anhand von Stützstellen, vorgestellt. So lässt sich jede skalare oder vektorielle Funktion im Raum $f(\vec{r})$ als Faltung mit der Dirac-Funktion δ exakt wiedergeben:

$$f(\vec{r}) = \int f(\vec{r}')\delta(\vec{r} - \vec{r}')d\vec{r}' \tag{3.6}$$

Hierbei tritt dasjenige Problem auf, dass die Dirac-Funktion wegen ihres Grenzwertcharakters beziehungsweise wegen ihrer infinitesimalen Glättungslänge für die Interpolation einer Funktion anhand von Stützstellen ungeeignet ist. Deshalb approximiert die SPH-Methode die Faltung

mit der Dirac-Funktion durch eine Faltung mit einer kompakten Funktion, dem sogenannten Smoothing-Kernel W:

$$f_i(\vec{r}) \cong \int f(\vec{r}')W(\vec{r}-\vec{r}',h)d\vec{r}' \tag{3.7}$$

Dabei ist h die nicht infinitesimale Glättungslänge des Kernels, welche auch indirekt den maximalen und endlichen Einflussradius des Kernels r_{max} über die Formel $r_{max} = q_{max}h$ beschreibt. SPH-Simulationen können als Smoothing-Kernels verschiedene Funktionen, welche unterschiedliche numerische Vorteile besitzen, hernehmen. Dabei ist q_{max} von Kernel zu Kernel verschieden. Die Kernels müssen aber folgende mathematische Eigenschaften erfüllen:

- Normierung:

$$\int W(\vec{r}-\vec{r}',h)d\vec{r}' = 1 \tag{3.8}$$

- Kompaktheit:

$$W(\vec{r}-\vec{r}',h) = 0 \text{ für } |\vec{r}-\vec{r}'| \geq q_{max}h \tag{3.9}$$

- Grenzwertbetrachtung:

$$\lim_{h\to 0} W(\vec{r}-\vec{r}',h) = \delta(\vec{r}-\vec{r}') \tag{3.10}$$

Hinzu kommen Stetigkeit, Symmetrie, Positivität, und eine streng fallende Monotonie. In dieser Arbeit wurde als Smoothing-Kernel stets das sogenannte quintische Wendland-Kernel verwendet. Es lautet:

$$W(\vec{r},h) = \begin{cases} \frac{12}{256\pi h^3}(2-q)^4(2q+1) & \text{falls } q < 2 \\ 0 & \text{falls } q \geq 2 \end{cases} \tag{3.11}$$

Dabei ist $q = \frac{|\vec{r}|}{h}$ und $q_{max} = 2$. Zur Veranschaulichung wurde der Graph des Wendland-Kernels in Abbildung 3.1 eingezeichnet.

Schließlich führt das SPH-Verfahren für die numerische Berechnung das Integral aus Gleichung 3.7 in eine Summenapproximation an der Stelle eines Partikelschwerpunkts über:

$$f_s(\vec{r}_a) \cong \sum_b V_b f(\vec{r}_b)W(\vec{r}_a-\vec{r}_b,h) \tag{3.12}$$

Dabei bezeichnet a das Zentralpartikel, an dessen Schwerpunkt die Funktion interpoliert wird, und b alle Nachbarpartikel, die für die Interpolation verwendet werden. Zur kompakteren Schreibweise dieser Gleichung, wird eine spezielle Nomenklatur verwendet. So wird bei einer Differenz einer Größe zwischen Zentralpartikel und Nachbarpartikel stets das Subskript ab verwendet, wie zum Beispiel $\vec{r}_a - \vec{r}_b = \vec{r}_{ab}$. Analog gilt $W(\vec{r}_a-\vec{r}_b,h) = W_{ab}$. Des Weiteren ist folgende Schreibweise für die Approximation einer Funktion an einer gegebenen Stelle in der Literatur verbreitet:

$$f_s(\vec{r}_a) \cong \langle f \rangle_a \tag{3.13}$$

Somit ergibt sich durch die vereinfachte Schreibweise bei Gleichung 3.12:

$$\langle f \rangle_a = \sum_b V_b f_b W_{ab} \tag{3.14}$$

Das Volumen einer Stützstelle lässt sich über die Masse und Dichte desjenigen Nachbarpartikels berechnen, welches als Stützstelle verwendet wird:

$$\langle f \rangle_a = \sum_b \frac{m_b}{\rho_b} f_b W_{ab} \tag{3.15}$$

Da Masse und Dichte eines Partikels gegeben sind, kann diese Gleichung direkt für die Interpolation einer Funktion verwendet werden. Es ist dabei wichtig anzumerken, dass in SPH die einzelnen Partikel einer Simulation gemäß obiger Formel theoretisch eine unterschiedliche Masse besitzen können. Bei vielen SPH-Simulationen, wie bei der SPH-Simulation in dieser Arbeit, ist aber die Masse aller Partikel gleich. Deshalb ließe sich die Masse aus obiger Summe herausziehen. Auf analoge Weise ließen sich auch viele folgende Gleichungen algebraisch vereinfachen. Damit folgende Darstellungen mit der Literatur übereinstimmen, wird diese Vereinfachung nicht durchgeführt.

Des Weiteren lässt sich aus dem Volumen der mittlere Partikelabstand Δx berechnen:

$$\Delta x = V^{\frac{1}{3}} \tag{3.16}$$

Dabei ist das Verhältnis zwischen diesem mittlerem Partikelabstand Δx und der Glättungslänge des Kernels h wichtig. Denn es bestimmt, wie viele benachbarte Partikel das Kernel bei der Interpolation mit einbezieht. Je mehr Partikel mit einbezogen werden, desto höher ist die numerische Genauigkeit der Interpolation. Jedoch nimmt auch der Rechenaufwand zu. Als Kompromiss zwischen Rechenaufwand und Genauigkeit wird ein Verhältnis $\frac{h}{\Delta x}$ von 1 bis 2 empfohlen. Folgende Arbeit verwendet immer einen Wert von 1.5 für dieses Verhältnis. Auf diese Weise werden für die Interpolation in etwa 110 benachbarte Partikel innerhalb der Cutoff-Distance verwendet.

Letztendlich benötigt eine SPH-Simulation auch immer den interpolierten Gradienten von skalaren Funktionen. Eine einfache mögliche Approximation für den Gradienten lautet:

$$\langle \nabla f \rangle_a = \nabla \sum_b \frac{m_b}{\rho_b} f_b W_{ab} = \sum_b \frac{m_b}{\rho_b} f_b \nabla W_{ab} \tag{3.17}$$

Somit lässt sich der Gradient der skalaren Funktion mit Hilfe des Gradienten des Wendland-Kernels berechnen. Der Gradient des Wendland-Kernels berechnet sich wie folgt:

$$\nabla W(\vec{r}, h) = \begin{cases} \frac{21}{256\pi h^4} (-10q) (2-q)^3 \frac{\vec{r}}{|\vec{r}|} & \text{falls } q < 2 \\ 0 & \text{falls } q \geq 2 \end{cases} \tag{3.18}$$

Dabei gilt wieder $q = \frac{|\vec{r}|}{h}$. Zur Veranschaulichung wurde der Gradient des Wendland-Kernels ebenfalls in Abbildung 3.1 eingetragen.

Die so eben genannte Approximation hat jedoch den Nachteil, dass der Beitrag von zwei Partikeln zur jeweiligen Interpolationssumme nicht symmetrisch ist:

$$f_b \nabla W_{ab} \neq f_a \nabla W_{ba} \tag{3.19}$$

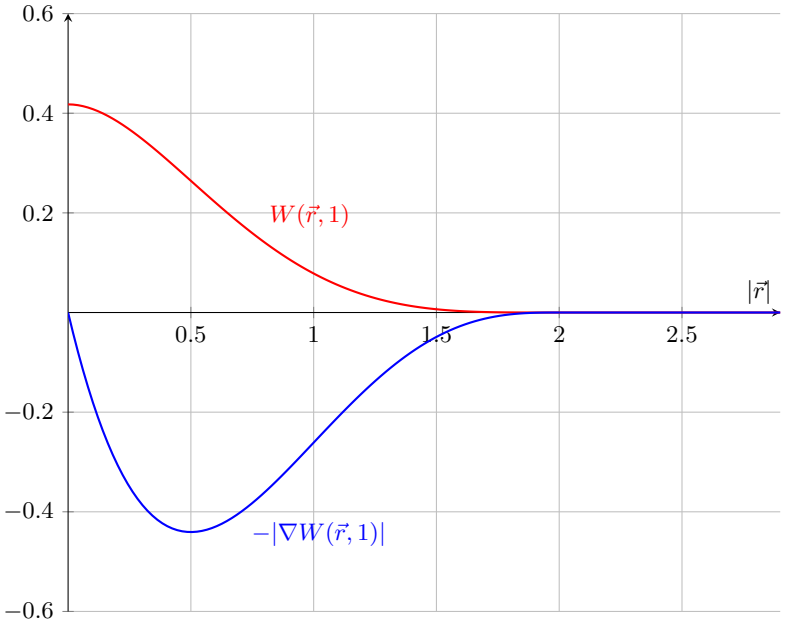


Abbildung 3.1: Graph des Wendlandkernels und Graph des Gradientens des Wendlandkernels

Das Partikel b trägt also einen anderen Wert zur Interpolationssumme des Partikels a bei als das Partikel a zur Interpolationssumme des Partikels b beiträgt.

Eine solche Symmetrie ist jedoch zum Beispiel bei der Kraftberechnung zwischen zwei Partikeln über einen solchen Gradienten vorteilhaft. Denn in dem Fall wäre das dritte newtonsche Axiom von Actio und Reactio erfüllt. Deshalb gibt es mehrere symmetrische Approximationen für den Gradienten. Die in der folgenden Arbeit benötigten symmetrischen Approximationen lauten:

$$\left\langle \frac{\nabla f}{\rho} \right\rangle_a = \sum_b m_b \left(\frac{f_a}{\rho_a^2} + \frac{f_b}{\rho_b^2} \right) \nabla W_{ab} \tag{3.20}$$

$$\langle \rho \nabla f \rangle_a = \sum_b \frac{m_b}{\rho_b} (f_b - f_a) \nabla W_{ab} \tag{3.21}$$

Die etwas aufwändigere Herleitung davon ist in [CH] zu finden.

3.4 Fixed-Radius-Near-Neighbors-Problem

Algorithmisch gesehen handelt es sich bei der SPH-Interpolation um ein Fixed-Radius-Near-Neighbors-Problem. Deshalb soll das Fixed-Radius-Near-Neighbors-Problem kurz vorgestellt werden.

So muss die SPH-Interpolation für jedes Zentralpartikel nur diejenigen Nachbarpartikel betrachten, deren Abstand zum Zentralpartikel kleiner als der maximale Einflussradius r_{max} des Smoothing-Kernels ist. Denn für die Nachbarpartikel außerhalb der Entfernung besitzt das Kernel wegen dessen Kompaktheit den Wert null. Dementsprechend sind die Summanden der entsprechenden Nachbarpartikel in den Gleichungen der Interpolation 3.15, 3.17, 3.20, und 3.21 ebenfalls null. Aus dem Grund können sie nichts mehr zum Ergebnis der Summe beitragen. Eben diese Problematik ist algorithmisch gesehen ein Fixed-Radius-Near-Neighbors-Problem. Bei einem Fixed-Radius-Near-Neighbors-Problem geht es allgemein darum möglichst effizient für jedes Zentralpartikel diejenigen Nachbarpartikel, die innerhalb der Cutoff-Distance um das Zentralpartikel herum liegen, zu finden und anhand den gefundenen Nachbarpartikeln problemspezifische Berechnungen auszuführen. Im Falle einer SPH-Simulation ist die Cutoff-Distance der maximale Einflussradius r_{max} des Smoothing-Kernels. Bei den problemspezifischen Berechnungen handelt es nicht nur bei SPH sondern auch bei vielen anderen Problemstellungen um eine einfache Superposition. Jedes Nachbarpartikel trägt einen bestimmten Wert zum Ergebnis des Zentralpartikels bei. Dabei werden die Beiträge der Nachbarpartikel aufsummiert. Ein solches Fixed-Radius-Near-Neighbors-Problem tritt in ähnlicher Form bei vielen weiteren partikelbasierten Simulationsmethoden auf, wie bei diversen Festkörpersimulationen, Simulationen von deformierbaren Körpern, oder bei Moleküldynamik-Simulationen. Wegen den vielen Anwendungsfällen ist die Berechnung eines Fixed-Radius-Near-Neighbors-Problem algorithmisch gesehen von großer Bedeutung.

3.5 Berechnung der zeitlichen Positionsänderung

Als Nächstes soll vorgestellt werden, wie eine SPH-Simulation die zeitliche Positionsänderung, also die Bewegung, eines Partikels $\frac{D\vec{r}}{Dt}$ bestimmen kann. Hierfür existieren zwei Ansätze. Die erste und einfachere Möglichkeit ist es die Positionsänderung eines Partikels über dessen Geschwindigkeit $\vec{v}$ gemäß der Geschwindigkeitsdefinition aus Gleichung 3.2 zu bestimmen:

$$\left\langle \frac{D\vec{r}}{Dt} \right\rangle_a = \vec{v}_a \tag{3.22}$$

Alternativ gibt es die Möglichkeit der XSPH-Geschwindigkeit. So kann eine SPH-Simulation die Partikel auch mit einer korrigierten Geschwindigkeit $\vec{v}_{korr}$ bewegen, indem sie die Geschwindigkeit des Partikels mit den Geschwindigkeiten der Nachbarpartikel glättet:

$$\left\langle \frac{D\vec{r}}{Dt} \right\rangle_a = \vec{v}_{a,korr} = \vec{v}_a + \epsilon \sum_b \frac{m_b}{0.5(\rho_a + \rho_b)} \vec{v}_{ba} W_{ab} \tag{3.23}$$

Dabei ist ϵ eine Konstante, welche die Stärke der Glättung angibt, und im Bereich von 0 bis 1 liegt. Somit muss die SPH-Simulation für die Berechnung der XSPH-Geschwindigkeit ein Fixed-Radius-Near-Neighbors-Problem lösen. Durch das Glätten vermeidet die SPH-Simulation, dass sich die Partikel zu schnell aufeinander zu oder auseinander bewegen können.

Dadurch wird eine gleichmäßigere Verteilung der Stützstellen erreicht, wodurch die numerische Stabilität erhöht werden kann. Allerdings geschieht dies auf Kosten einer zusätzlichen numerischen Viskosität. Letztlich ist es wichtig anzumerken, dass trotz der Fortbewegung durch die korrigierte Geschwindigkeit $\vec{v}_{korr}$ jedes Partikel seine Geschwindigkeit $\vec{v}$ als Attribut beibehält.

3.6 Ortsdiskretisierung der Beschleunigung

In diesem Punkt soll erläutert werden, wie ein SPH-Verfahren die Beschleunigung $\frac{D\vec{v}}{Dt}$ eines Partikels berechnet. Hierfür muss die Impulsgleichung 3.3 ortsdiskretisiert werden. Für die Diskretisierung der Beschleunigung durch den Druckgradienten $-\frac{\nabla P}{\rho}$ kann eine SPH-Simulation die symmetrische Approximation des Gradienten aus Gleichung 3.20 verwenden. Dadurch ergibt sich:

$$-\left\langle \frac{\nabla P}{\rho} \right\rangle_a = -\sum_b m_b \left(\frac{P_a}{\rho_a^2} + \frac{P_b}{\rho_b^2} \right) \nabla W_{ab} \tag{3.24}$$

Es gibt verschiedene Arten von Viskosität, die eine SPH-Simulation als dissipativen Term Γ verwenden kann, wie zum Beispiel die die laminare Viskosität oder die künstliche Viskosität. Letztere wird in dieser Arbeit verwendet. Sie lässt sich berechnen durch:

$$\langle \Gamma \rangle_a = -\sum_b m_b \Pi_{ab} \nabla W_{ab} \tag{3.25}$$

Hierbei gilt für Π_{ab} :

$$\Pi_{ab} = \begin{cases} \frac{-\alpha c \mu_{ab}}{0.5(\rho_a + \rho_b)} & \text{falls } \vec{v}_{ab}\vec{r}_{ab} < 0 \\ 0 & \text{falls } \vec{v}_{ab}\vec{r}_{ab} > 0 \end{cases} \tag{3.26}$$

Dabei ist α eine Konstante, welche abhängig vom Problem zu wählen ist. In der restlichen Arbeit ist $\alpha = 0.01$. Die Variable μ_{ab} ist wiederum definiert als:

$$\mu_{ab} = \frac{h\vec{v}_{ab}\vec{r}_{ab}}{\vec{r}_{ab}^2 + 0.01h^2} \tag{3.27}$$

Verwendet die Simulation für die Beschleunigung der Flüssigkeit durch externe Kräfte die Erdbeschleunigung mit $\vec{g} = (0, -9.8, 0)^T m/s^2$ und summiert diese mit der Beschleunigung durch den Druckgradienten und durch die künstliche Viskosität auf, so ergibt sich für die Beschleunigung $\frac{D\vec{v}}{Dt}$ eines Partikels insgesamt:

$$\left\langle \frac{Dv}{Dt} \right\rangle_a = -\sum_b m_b \left(\frac{P_a}{\rho_a^2} + \frac{P_b}{\rho_b^2} + \Pi \right) \nabla W_{ab} + \vec{g} \tag{3.28}$$

Dadurch muss die SPH-Simulation für die Berechnung der Beschleunigung wieder ein Fixed-Radius-Near-Neighbors-Problem lösen.

3.7 Ortsdiskretisierung der zeitlichen Dichteänderung

Im Folgenden soll vorgestellt werden, wie ein SPH-Verfahren die zeitliche Dichteänderung $\frac{D\rho}{Dt}$ eines Partikels berechnen kann. Eine Möglichkeit hierfür ist die Ortsdiskretisierung der Kontinuitätsgleichung 3.5 mit der symmetrischen Approximation des Gradienten aus Gleichung 3.21:

$$\left\langle \frac{D\rho}{Dt} \right\rangle_a = -\left\langle \rho \nabla \vec{v} \right\rangle_a = \sum_b m_b \vec{v}_{ab} \nabla W_{ab} \tag{3.29}$$

Dabei ist diese Gleichung algorithmisch gesehen wieder ein Fixed-Radius-Near-Neighbors-Problem. Sofern die SPH-Simulation die XSPH-Geschwindigkeit verwendet, muss sie in dieser Gleichung aus Konsistenzgründen ebenfalls den durch die XSPH-Geschwindigkeit korrigierten Wert $\vec{v}_{korr}$ verwenden. Durch die Integration der Dichte entstehen jedoch große Druckunterschiede wegen Schallwellen innerhalb der Flüssigkeit. Des Weiteren lässt sich die Normbedingung des Smoothing-Kernels aus Gleichung 3.8 in diskreter Form wie folgt darstellen:

$$\sum_b \frac{m_b}{\rho_b} W_{ab} = 1 \tag{3.30}$$

Dies ist durch eine zeitliche Integration der Dichte ebenfalls nicht mehr erfüllt. Deshalb muss eine SPH-Simulation die Dichte in etwa alle 30 Zeitschritte durch einen sogenannten Shepard-Filter normieren:

$$\langle \rho \rangle_{a,\text{norm}} = \frac{\sum_b m_b W_{ab}}{\sum_b \frac{m_b}{\rho_b} W_{ab}} \tag{3.31}$$

Somit handelt es sich beim Shepard-Filter ebenfalls um ein Fixed-Radius-Near-Neighbors-Problem. Nach der Normierung wird die Dichte ρ der Partikel durch die normierte Dichte ρ_{norm} ersetzt.

Neben der zeitlichen Integration der Dichte gibt es für die Berechnung der Dichte noch einen weiteren Ansatz. Denn alternativ kann eine SPH-Simulation die Dichte eines Partikels direkt per SPH-Interpolation berechnen, indem sie die Dichte in Gleichung 3.15 einsetzt. Dadurch ergibt sich folgendes Fixed-Radius-Near-Neighbors-Problem:

$$\langle \rho \rangle_a = \sum_b m_b \frac{\rho_b}{\rho_b} W_{ab} = \sum_b m_b W_{ab} \tag{3.32}$$

Die Möglichkeit besitzt jedoch den Nachteil, dass nahe an den Rändern der Flüssigkeit die Zahl der Stützstellen für die Interpolation der Dichte abnimmt. Aus diesem Grund nimmt die Dichte ebenfalls zum Rand hin ab. Dies ist allerdings nicht physikalisch und erzeugt dort ein Druckgefälle, welches ähnlich wie eine Oberflächenspannung die Partikel nach Innen drückt. Deshalb muss eine SPH-Simulation bei der direkten Berechnung der Dichte die Dichte anschließend ebenfalls durch den Shepard-Filter normieren.

3.8 Schrittweitenkontrolle

Für die numerische Stabilität ist bei einer SPH-Simulation eine Schrittweitenkontrolle für die Zeitintegration notwendig. Die Schrittweitenkontrolle wählt eine maximalen Zeitschrittgröße

Δt_{max} und setzt sich aus mehreren Teilbedingungen zusammen. So darf gemäß der künstlichen Viskosität und der CFL-Bedingung bei SPH der Zeitschritt maximal folgendes Δt_{cv} betragen:

$$\Delta t_{cv} = \min_a \left(\frac{h}{c + \max_b |\mu_{ab}|} \right) \tag{3.33}$$

Des Weiteren muss bei der Wahl des Zeitschrittes berücksichtigt werden, dass wegen der Beschleunigung $\frac{D\vec{v}}{Dt}$, die ein Partikel erfährt, nur eine folgende maximale Zeitschrittweite von Δt_f verwendet werden kann:

$$\Delta t_f = \min_a \left(\sqrt{\frac{h}{|\frac{D\vec{v}}{Dt}|}} \right) \tag{3.34}$$

Diese Bedingungen lassen sich wie folgt zum maximalen Zeitschritt Δt_{max} kombinieren:

$$\Delta t_{max} = \min \left(0.3\Delta t_{cv}, 0.3\Delta t_f \right) \tag{3.35}$$

3.9 Zeitdiskretisierung

Als Nächstes soll auf die Zeitdiskretisierung beziehungsweise auf die Zeitintegration eingegangen werden. Obwohl die SPH-Simulation wegen der Tait-Zustandsgleichung ein steifes Problem ist, so sind implizite Verfahren bei großen Partikelanzahlen zu zeitaufwändig. Deshalb empfiehlt die Literatur häufig das Prädiktor-Korrektor-Verfahren zweiter Ordnung, welches wegen der Empfehlung ebenfalls in dieser Arbeit verwendet wird. Bei ihm handelt es sich um ein explizites Einschrittverfahren mit folgendem Butcher-Tableau:

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ \hline & 0 & 1 \end{array} \tag{3.36}$$

Dieses Butcher-Tableau lässt sich nun auf das System der gewöhnlichen Differentialgleichungen anwenden, das durch die Ortsdiskretisierung entstanden ist. Im Folgenden wird die Anwendung des Butcher-Tableaus exemplarisch für eine SPH-Simulation, die die Dichte zeitlich integriert, erläutert. Berechnet eine SPH-Simulation die Dichte direkt durch Interpolation, so läuft das Prädiktor-Korrektor-Verfahren abgesehen davon, dass die Integration der Dichte entfällt, analog ab. Gemäß des Butcher-Tableaus teilt sich das Prädiktor-Korrektor-Verfahren in zwei Teilschritte auf. Im ersten Teilschritt werden die Strömungsgrößen der Partikels zum Halbzeitschritt $t + \frac{\Delta t}{2}$ mit ihren Änderungen zu t berechnet:

$$\vec{r}_{t+\frac{\Delta t}{2}} = \vec{r}_t + \frac{\Delta t}{2} \left(\frac{D\vec{r}}{Dt} \right)_t \tag{3.37}$$

$$\vec{v}_{t+\frac{\Delta t}{2}} = \vec{v}_t + \frac{\Delta t}{2} \left(\frac{D\vec{v}}{Dt} \right)_t \tag{3.38}$$

$$\rho_{t+\frac{\Delta t}{2}} = \rho_t + \frac{\Delta t}{2} \left(\frac{D\rho}{Dt} \right)_t \tag{3.39}$$

Als Nächstes findet der zweite Teilschritt statt. Dabei wird der Zustand der Simulation zum Halbzeitschritt $t + \frac{\Delta t}{2}$ nun benutzt um die zeitlichen Änderungen zu diesem Zeitpunkt zu berechnen. Diese zeitlichen Änderungen zu $t + \frac{\Delta t}{2}$ werden dann anschließend verwendet, um ausgehend von dem Zustand der Simulationsdomäne zu Beginn des Zeitschritts t, den Zustand der Simulationsdomäne zum Ende des Zeitschritts $t + \Delta t$ zu berechnen:

$$\vec{r}_{t+\Delta t} = \vec{r}_t + \Delta t \left(\frac{D\vec{r}}{Dt} \right)_{t+\frac{\Delta t}{2}} \tag{3.40}$$

$$\vec{v}_{t+\Delta t} = \vec{v}_t + \Delta t \left(\frac{D\vec{v}}{Dt} \right)_{t+\frac{\Delta t}{2}} \tag{3.41}$$

$$\rho_{t+\Delta t} = \rho_t + \Delta t \left(\frac{D\rho}{Dt} \right)_{t+\frac{\Delta t}{2}} \tag{3.42}$$

3.10 Randbedingungen

In diesem Punkt sollen kurz die Randbedingungen bei einer SPH-Simulation vorgestellt werden. So gibt es bei einer Flüssigkeitssimulation mehrere Arten von Randbedingungen, die auftreten können:

- **Freie Oberflächen:** Flüssigkeitsoberfläche grenzt an Luft an
- **Am Rande des simulierten Raums:** Periodische Randbedingungen, reflektierende Randbedingungen, Einlassrandbedingungen und Auslassrandbedingungen
- **Festkörperrandbedingungen:** Flüssigkeitsoberfläche grenzt an einen Festkörper an

Da freie Oberflächen bei einer SPH-Flüssigkeitssimulation nicht explizit modelliert werden müssen, und besondere Randbedingungen am Rand des simulierten Raums für einfache Testfälle nicht erforderlich sind, werden in dieser Arbeit nur Festkörperrandbedingungen modelliert. Hierfür gibt es zwei Möglichkeiten:

- Die Oberflächen des Festkörpers werden ebenfalls per Partikel diskretisiert. Kommt ein Partikel der Flüssigkeit einem Partikel der Oberfläche zu nahe, so werden abstoßende Kräfte gemäß eines Lennard-Jones-Potentials simuliert. Diese Kräfte führen dazu, dass das Partikel sich wieder vom Festkörper weg bewegt.
- Die Simulation definiert den Festkörper über ein Volumen. Bewegt sich ein Partikel der Flüssigkeit in das Volumen hinein, so kann die Simulation es entweder direkt an der Oberfläche des Volumens reflektieren oder es alternativ ebenfalls per abstoßender Kraft aus dem Volumen hinausbewegen.

Implementiert die Simulation die Festkörper-Randbedingungen per Kraft, so wird die so berechnete Kraft zu $\frac{D\vec{v}}{Dt}$ hinzuaddiert. Um ein Aufschaukeln durch die abstoßende Kraft zu vermeiden kann es sinnvoll sein, die Partikel zusätzlich durch eine ihrer Geschwindigkeit entgegenwirkende Reibungskraft leicht abzubremsen.

3.11 Anfangsbedingungen

Als Nächstes soll vorgestellt werden, wie die Anfangsbedingungen bei einer solchen Simulation gewählt werden. Dabei ist die Flüssigkeitsverteilung im einfachsten Fall als räumliche Funktion mit Geschwindigkeitsverteilung und ohne Dichteverteilung gegeben. Diese wird nun mit einem kartesischem Gitter abgetastet. Ist der Abtastpunkt innerhalb der Flüssigkeit so wird ein Partikel erstellt. Dabei ist der Gitterzellenabstand die gewünschte Auflösung der Simulation und zudem der mittlere Partikelabstand Δx. Aus Δx und der Referenzdichte ρ_0, welche ein jedes Partikel zu Beginn besitzt, kann die Simulation leicht die Masse eines jeden einzelnen Partikels berechnen:

$$m = \rho_0 \Delta x^3 \tag{3.43}$$

Des Weiteren besitzt jedes Partikel am Anfang die Geschwindigkeit $\vec{v}$ der volumetrischen Funktion an seinem Abtastpunkt. Da die Dichte überall gleich ist, kollabiert die Flüssigkeit zu Beginn wegen der Erdbeschleunigung ein wenig. Dies wird in der Literatur allerdings wahrscheinlich wegen der Geringfügigkeit des Effekts auf Grund der schweren Kompressibilität meist nicht besonders behandelt.

4 Single-GPU-Implementierung einer SPH-Simulation

4.1 Aktuelle Ansätze für effiziente GPU-SPH-Implementierungen

Nachdem im letzten Kapitel die benötigten SPH-Grundlagen beschrieben worden sind, soll in diesem Kapitel zunächst eine SPH-Implementierung für eine einzige GPU beschrieben, optimiert und untersucht werden. Denn auf diese Weise kann die Arbeit leicht die grundlegenden Algorithmen erklären und untersuchen, ohne an vielen Stellen die Eigenschaften des Clusters berücksichtigen zu müssen. Zudem kommen die Algorithmen größtenteils in identischer Weise in der Cluster-Version vor. Deshalb sind die Ansätze, die Diskussion und die Untersuchungsergebnisse dieses Kapitels weiterhin für die Cluster-Version gültig. Für diese Untersuchungen und Erklärungen der Single-GPU-SPH-Implementierung sollen einleitend zuerst in diesem Punkt allgemein aktuelle Lösungsansätze aus der Literatur für eine GPU-SPH-Implementierung vorgestellt werden.

Wie in dem Kapitel 3 der SPH-Grundlagen erklärt, so muss eine SPH-Simulation in jedem Zeitschritt mehrmals ein Fixed-Radius-Near-Neighbors-Problem, zum Beispiel für die Berechnung der zeitlichen Änderungen der Dichte und der Geschwindigkeit, berechnen und danach die Zeitintegration ausführen. Bei einer SPH-Simulation kostet das Lösen der Fixed-Radius-Near-Neighbors-Probleme immer einen großen Teil der Rechenzeit, und ist zudem so komplex, dass es viele Optimierungsansätze erlaubt. Deshalb sollen im Folgenden Optimierungen für das Fixed-Radius-Near-Neighbors-Problems für GPUs aus der Literatur vorgestellt werden. Da das Fixed-Radius-Near-Neighbors-Problem nicht nur in SPH vorkommt, sind die Literaturquellen im Folgenden auch nicht auf SPH beschränkt.

Um bei diesem Problem die Komplexität beim Finden der nächsten Nachbarn von $O(n^2)$ auf $O(n)$ zu reduzieren benötigt die SPH-Simulation zuerst eine Space-Partitioning-Datenstruktur. Hierfür eignet sich auf GPUs in der Regel ein kartesisches Gitter als Datenstruktur, in welches die SPH-Simulation die Partikel einsortiert, am besten. Denn vorteilhaft an einem solchen Gitter ist, dass es eine GPU leicht bauen und traversieren kann. Jedoch benötigt das Gitter verglichen mit vielen anderen Datenstrukturen, gerade bei Simulationen mit weit verteilten Partikeln, ein Mehr an Speicherplatz. In solchen Fällen können beispielhaft Baumdatenstrukturen, wie die Baumdatenstruktur aus [CH], vorteilhaft sein. Der Nachteil des Gitters wird jedoch dadurch relativiert, dass die Flüssigkeit in vielen Anwendungsfällen sich nicht über ein weites Gebiet verteilt, sondern kompakt ist. Dadurch verbraucht das Gitter an sich nur ein Bruchteil des gesamten Speicherplatzes der Simulation. In einer Testszene aus [JDa] waren es zum Beispiel nur in etwa drei Prozent. Zudem gibt es auch Ansätze, wie die SPH-Simulation nur noch diejenigen Gitterteile, in denen sich Flüssigkeit befindet, abspeichern muss. Ein solcher Ansatz ist beispielhaft in dem Artikel [TH] vorgestellt. Dadurch kann eine SPH-Simulation den Speicherverbrauch eines solchen Gitters effizient reduzieren. Deshalb und wegen der überlegenen Performance eines Gitters verwendet die Literatur bei SPH-Simulationen auf GPUs meist ein Gitter.

Allerdings gibt es beim Gitter viele Unterschiede, wie das Gitter auf die Partikel verweist, wie das Gitter gebaut wird, wie die Gitterzellen im Speicher der GPU angeordnet sind, oder wie das Gitter traversiert wird. Zuerst soll darauf eingegangen werden, wie das Gitter auf die Partikel verweist. So werden die Partikel nie direkt im Gitter sondern immer in einem eigenen Partikelarray abgespeichert. Das Gitter verweist dann über Indexe auf das Partikelarray. Für die Verweise gibt es folgende zwei Möglichkeiten (siehe zum Beispiel [SG]):

- **Static-Matrix:** (Name stammt aus [JDb]) Jede Gitterzelle enthält eine Liste der Indexe von denjenigen Partikeln, welche sich in ihr befinden. Da dynamische Datenstrukturen auf GPUs nur schlecht zu implementieren sind, wird die Liste als statisch alloziertes Array implementiert.
- **Dynamic-Vector:** (Name stammt aus [JDb]) Das Partikelarray wird nach den Gitterzellen sortiert. Durch die Sortierung muss jede Gitterzelle nur noch den Index des ersten und des letzten Partikels, die sich in ihr befinden, abspeichern. Sämtliche Partikel deren Index zwischen den beiden Indexen liegt, befinden sich wegen der Sortierung ebenfalls in der Gitterzelle.

Das Static-Matrix-Gitter ist laut [SG] jedoch deutlich langsamer, da die nach Gitterzellen sortierten Partikel im zweiten Verfahren wesentlich lokalere Speicherzugriffe als die unsortierten Listen erlauben. Zusätzlich ist es bei dem Static-Matrix-Gitter für dessen Speicherplatzverbrauch ungünstig, dass jede Gitterzelle ein Array an Indexen abspeichern muss. Deshalb verwendet die Literatur meist nur das Dynamic-Vector-Gitter. Bei dem Dynamic-Vector-Gitter muss die SPH-Simulation für das Bauen des Gitters jeweils die Partikel nach Gitterzellen sortieren. Hierfür verwendet die Literatur für GPUs zwei Ansätze: Das Counting-Sort-Verfahren und das Radix-Sort-Verfahren (siehe die Artikel [SG] und [RHb]). Beide wurden in dieser Arbeit implementiert und werden im Punkt 4.8 genauer beschrieben. Bei dem Dynamic-Vector-Gitter kann die SPH-Simulation die Gitterzellen entweder linear oder auf eine Weise anordnen, welche bei der Traversierung des Gitters für Cache-Blocking sorgt, wenn die GPU die Zentralpartikel ihrer Reihenfolge nach berechnet. Für eine solche Blocking-Anordnung verwendet zum Beispiel der Artikel [PG] die Z-Kurve. Da die Partikeldaten nach Gitterzellen sortiert sind, hilft dieses Blocking auch bei den Speicherzugriffen auf diese Daten.

Anschließend soll auf die Optimierungen für das Traversieren des Gitters beziehungsweise für die Berechnungen, welche mit Hilfe des Gitters ausgeführt werden, eingegangen werden. Dabei führt die SPH-Simulation bei den allermeisten Ansätzen die Berechnungen direkt während der Traversierung aus. Dieser Ansatz wird in der Literatur als Linked-Cell-Ansatz bezeichnet. Neben kleineren Optimierungen wie der Schleifenstruktur (siehe Punkt 4.9.2 dieser Arbeit oder den Quelltext von [DSPHb]) und der Verwendung des Texture-Caches (siehe Punkt 4.9.5 oder beispielhaft Artikel [JA]), gibt es in der Literatur beim Traversieren des Gitters und beim Ausführen der Berechnungen drei größere Optimierungsansätze:

- **Shared-Memory-Blocking:** Eine Optimierung durch Blocking über den Shared-Memory ist im Artikel [PG] zu finden. In diesem Paper werden, stark vereinfacht dargestellt, um die Berechnungen durchzuführen zuerst die Partikel einer Gitterzelle und alle Partikel in den benachbarten Gitterzellen in den Shared-Memory der GPU geladen. Anschließend werden die Berechnungen selbst für alle Partikel in dieser Gitterzelle durchgeführt. Da alle dafür benötigten Daten bereits im Shared-Memory liegen, wird der L2-Cache und der DRAM entlastet sowie ein latenzarmer Zugriff auf die Daten ermöglicht. Dieses Verfahren wurde jedoch im Artikel nur für ältere GPUs ohne L2-Cache entwickelt und konnte dort einen Performancegewinn erzielen. Als es für den SPH-Simulator Fluids v.3 [RHa] implementiert und auf moderneren GPUs getestet wurde, war die Performance deutlich schlechter als bei einer Durchführung der Traversierung und Berechnungen ohne Shared-Memory-Blocking. Der Autor dieser Masterarbeit vermutet, dass durch das Kopieren in den Shared-Memory lediglich ein Overhead entsteht, da moderne GPU-Caches

für SPH-Simulationen genügend schnell und genügend groß sind (siehe Punkt 4.9.12 dieser Arbeit). Der Overhead schlägt sich dann letztendlich auch auf die Performance nieder.

- **Ausnutzen der Symmetrie:** Ein weiterer Ansatz ist es auszunutzen, dass die Berechnungen zwischen zwei Partikeln komplett identisch oder bis auf das Vorzeichen identisch sind. Diese Symmetrie kann die SPH-Simulation nutzen, indem sie die Berechnung nur für eins der beiden Partikel ausführt und anschließend für das andere Partikel abspeichert. Wie eine SPH-Simulation dies effizient auf der CPU ausnutzen kann, ist zum Beispiel in [GGb] beschrieben. Allerdings fällt es schwer die Symmetrie auf GPUs auszunutzen, da die SPH-Simulation dafür wegen der Nebenläufigkeit der GPUs viele atomare Operationen benötigen würde. Deshalb wird in der Literatur, wie beispielhaft in [GGb], bei GPUs davon abgeraten. Auf CPUs kann die Optimierung gemäß [JA] allerdings einen deutlichen Performancegewinn erzielen. So wurde dort in etwa ein 1.4-facher Speedup auf einem modernen 3.0 GHz 80546K Intel-Xeon-Prozessor gemessen. Maximal wäre ein Speedup von zwei zu erwarten gewesen. Wie lange die Optimierung noch die Laufzeit reduzieren kann, ist jedoch wegen der zunehmenden Breite der SIMD-Einheiten von modernen CPUs und der problematischen Vektorisierung dieser Optimierung fraglich.
- **Verlet-Listen:** Neben dem bereits vorgestellten Linked-Cell-Ansatz, bei welchem die Berechnungen direkt beim Traversieren ausgeführt werden, gibt es noch den Ansatz der Verlet-Liste. Er wurde zum Beispiel im Artikel [JA] auf GPUs implementiert und genauer beschrieben. Der Ansatz erstellt beim Traversieren des Gitters für jedes Partikel zunächst nur eine Nachbarschaftsliste, welche als Verlet-Liste bezeichnet wird. Die Verlet-Liste beinhaltet alle benachbarten Partikel innerhalb der Cutoff-Distance. Bei jedem darauf folgenden Berechnungsschritt muss dann nicht mehr das Gitter sondern nur noch die Verlet-Liste traversiert werden. Die Verlet-Liste hat den Vorteil einer einfacheren und datenparalleleren Traversierung. Zudem kann die SPH-Simulation die Liste über mehrere Zeitschritte hinweg verwenden, sofern sie kleinere algorithmische Fehler beim Finden der Nachbarpartikel toleriert. Die Fehler lassen sich reduzieren oder auch komplett vermeiden, wenn die SPH-Simulation die Verlet-Liste mit einer etwas größeren Cutoff-Distance als der eigentlichen Cutoff-Distance baut. Allerdings hat die Liste den entscheidenden Nachteil, dass sie den Speicherplatzverbrauch der gesamten Simulation in etwa verdoppelt oder verdreifacht.

4.2 Übersicht

In diesem Punkt soll eine Übersicht über dieses Kapitel gegeben werden. So verwendet die Single-GPU-Implementierung, welche dieses Kapitel erläutert wird, wegen der hohen Performance ein kartesisches Dynamic-Vector-Gitter. Des Weiteren wird für die Berechnungen der Fixed-Radius-Near-Neighbors-Probleme weitestgehend der Linked-Cell-Ansatz gewählt. So befasst sich das Kapitel primär damit, wie das Bauen des Gitters und die Berechnungen der Fixed-Radius-Near-Neighbors-Probleme implementiert und optimiert worden sind. Für diese Optimierungen ist ebenfalls eine Untersuchung von beiden nötig. Auch wird kurz die Berechnung der Zeitintegration vorgestellt und untersucht. Für die Single-GPU-Implementierung wird zuerst vorgestellt, wie die SPH-Verfahren aus dem Kapitel 3 sich zu Algorithmen kombinieren lassen. Zudem wird die Wahl der Rechengenauigkeit und die Abspeicherung der Daten des Gitters sowie der Partikel erläutert. Des Weiteren werden für die Untersuchungen in dem Ka-

pitel das Testsystem, die Testszene und die Messmethoden vorgestellt. Zudem wird ein erster Performanceüberblick gegeben. So lässt sich dieses Kapitel in folgende Unterpunkte gliedern:

- **Punkt 4.3:** Die Implementierung der SPH-Algorithmen
- **Punkt 4.4:** Die Wahl der Rechengenauigkeit
- **Punkt 4.5:** Die Abspeicherung der Partikeldaten
- **Punkt 4.6:** Die Abspeicherung der Gitterdaten
- **Punkt 4.7:** Das Testsystem, Messmethoden, die Testszene und ein erster Performanceüberblick
- **Punkt 4.8:** Das Bauen des Gitters
- **Punkt 4.9:** Die Berechnungen und die Optimierungen des Fixed-Radius-Near-Neighbors-Problems
- **Punkt 4.10:** Die Berechnung der Zeitintegration

4.3 SPH-Algorithmen

In diesem Punkt sollen die Formeln der SPH-Grundlagen aus Kapitel 3 zu Algorithmen kombiniert werden. Denn es ergeben sich verschiedene SPH-Verfahren je nachdem wie die SPH-Simulation die Formeln aus dem Kapitel 3 der SPH-Grundlagen zusammensetzt. Von den SPH-Verfahren wurden für diese Arbeit zwei implementiert.

Als Festkörperrandbedingungen verwenden in der Arbeit beide Verfahren ein Höhenfeldterrain. Zusätzlich werden am Rand des Volumens des Gitters Wände als Festkörperrandbedingung eingebaut, damit die Flüssigkeit nicht aus dem Gitter herausfließen kann. Beide Festkörperrandbedingungen werden als Volumen definiert, das eine Kraft auf diejenigen Partikel auswirkt, welche sich in das Volumen hereinbewegen.

Das erste SPH-Verfahren integriert die Dichte zeitlich und verwendet dabei keine XSPH-Geschwindigkeit. Es fasst die Berechnungen aus Kapitel 3 sofern möglich zu einem einzigen Fixed-Radius-Near-Neighbors-Pass zusammen, wodurch sich folgende Berechnungsschritte ergeben:

- **Bauen des Gitters zu t**
- **1. Fixed-Radius-Near-Neighbors-Problem: Shepard-Filter.** Dieser Schritt, der nur alle 30 Zeitschritte ausgeführt wird, normiert die Dichten der Partikel mit Hilfe des Shepard-Filters gemäß Gleichung 3.31.
- **2. Fixed-Radius-Near-Neighbors-Problem: Zeitliche Änderungen zu t.** Hier berechnet das Verfahren die zeitlichen Änderung zum Zeitpunkt t. Dabei bestimmt es die Änderung der Dichte $\frac{D\rho}{Dt}$ gemäß Gleichung 3.29 und die Beschleunigung $\frac{D\vec{v}}{Dt}$ gemäß Gleichung 3.28. Zudem wird für die Beschleunigung $\frac{D\vec{v}}{Dt}$ ebenfalls die Interaktion mit dem Höhenfeld-Terrain berücksichtigt. Des Weiteren berechnet das Verfahren die maximale Zeitschrittweite.

- **Erster Teilschritt des Prädiktor-Korrektor-Verfahrens:** Hierbei führt das Verfahren die Zeitintegration der Strömungsgrößen $\vec{r}$, $\vec{v}$ und ρ von t zu $t + 0.5\Delta t$ mit Hilfe der Änderungen zu t aus.
- **Bauen des Gitters zu** $t + 0.5\Delta t$
- **3. Fixed-Radius-Near-Neighbors-Problem: Zeitliche Änderungen zu** $t + 0.5\Delta t$**.** Nun berechnet das Verfahren die zeitlichen Änderung zum Zeitpunkt $t + 0.5\Delta t$. Der Schritt läuft analog zur Berechnung des 2. Problems ab. Nur diesmal findet die Bestimmung der maximalen Zeitschrittweite nicht statt.
- **Zweiter Teilschritt des Prädiktor-Korrektor-Verfahrens:** Bei diesem Schritt findet die Zeitintegration der Strömungsgrößen $\vec{r}$, $\vec{v}$ und ρ von t zu $t+\Delta t$ mit Hilfe der Änderungen zu $t + 0.5\Delta t$ statt.

Bei den Beschreibungen und Untersuchungen der restlichen Arbeit wird stets dieses erste SPH-Verfahren verwendet. Bei ihm wurde die XSPH-Geschwindigkeit nicht verwendet, da sie einen weiteren Fixed-Radius-Near-Neighbors-Pass benötigen würde, wodurch sich die Performance drastisch reduzieren würde. Des Weiteren gibt es in der Literatur widersprüchliche Meinungen in wie weit die Verwendung der XSPH-Geschwindigkeit sinnvoll ist. Da der Pass der XSPH-Geschwindigkeit zudem algorithmisch gesehen für die Untersuchungen weniger interessant ist, wird auf die Verwendung der XSPH-Geschwindigkeit verzichtet. Wegen der Modularität der Implementierung ließe sich allerdings die XSPH-Geschwindigkeit sehr leicht einbauen. Auch gilt anzumerken, dass eine reine Single-GPU-Implementierung des Verfahrens in der finalen Version dieser Arbeit nicht vorhanden ist. Deshalb wird im Folgenden die finale GPU-Cluster-Implementierung untersucht. Im Falle einer einzigen GPU verhält sie sich jedoch weitestgehend so wie eine reine Single-GPU-Implementierung.

Bei dem ebenfalls implementierten zweiten Verfahren wird die Dichte nicht zeitlich integriert, sondern die SPH-Interpolation für ihre Berechnung verwendet. Deshalb lautet es wie folgt:

- **Bauen des Gitters zu** t
- **1. Fixed-Radius-Near-Neighbors-Problem: SPH-Interpolation der Dichte zu** t**.** Hier wird die Dichte ρ zum Zeitpunkt t gemäß Gleichung 3.32 berechnet.
- **2. Fixed-Radius-Near-Neighbors-Problem: Shepard-Filter zu** t**.** Nun normiert das Verfahren die soeben berechnete Dichte ρ durch den Shepard-Filter aus Gleichung 3.31.
- **3. Fixed-Radius-Near-Neighbors-Problem: Zeitliche Änderungen zu** t**.** Als Nächstes werden die zeitlichen Änderung der Positionen $\frac{D\vec{r}}{Dt}$ mit XSPH-Geschwindigkeit gemäß Gleichung 3.23 und die Beschleunigung $\frac{D\vec{v}}{Dt}$ gemäß Gleichung 3.28 zum Zeitpunkt t berechnet. Ebenfalls wird hier der Einfluss der Randbedingungen auf die Beschleunigung und der maximale Zeitschritt ermittelt.
- **Erster Teilschritt des Prädiktor-Korrektor-Verfahrens:** Hierbei berechnet das Verfahren die Zeitintegration der Strömungsgrößen $\vec{r}$ und $\vec{v}$ von t zu $t + 0.5\Delta t$ mit Hilfe der Änderungen zu t.
- **Bauen des Gitters zu** $t + 0.5\Delta t$

- **4. Fixed-Radius-Near-Neighbors-Problem: SPH-Interpolation der Dichte zu $t + 0.5\Delta t$.** Jetzt bestimmt das Verfahren die Dichte zu $t + 0.5\Delta t$ analog zum ersten Fixed-Radius-Near-Neighbors-Problem.
- **5. Fixed-Radius-Near-Neighbors-Problem: Shepard-Filter zu $t + 0.5\Delta t$.** Bei diesem Problem werden die Dichten wieder durch den Shepard-Filter normiert.
- **6. Fixed-Radius-Near-Neighbors-Problem: Zeitliche Änderungen zu $t + 0.5\Delta t$.** Schließlich werden die zeitlichen Änderung zu $t + 0.5\Delta t$ analog zum 3. Fixed-Radius-Near-Neighbors-Problem berechnet, nur dass diesmal wieder kein maximaler Zeitschritt bestimmt wird.
- **Zweiter Teilschritt des Prädiktor-Korrektor-Verfahrens:** Bei dem Schritt berechnet das Verfahren die Zeitintegration der Strömungsgrößen $\vec{r}$ und $\vec{v}$ von t zu $t + \Delta t$ mit Hilfe der Änderungen zu $t + 0.5\Delta t$.

4.4 Rechengenauigkeit

In diesem Punkt soll kurz auf die in der Implementierung verwendete Gleitkommagenauigkeit eingegangen werden. So verwendet die Arbeit die einfache Gleitkommagenauigkeit für die Berechnungen. Dies geschah aus zwei Gründen:

- **Ausreichende SP-Genauigkeit:** So reicht gemäß dem Artikel [ACa] im Dual-SPHysics-Simulator die einfache Rechengenauigkeit für die SPH-Simulation der dort untersuchten Testprobleme aus. Denn eine experimentelle Überprüfung zeigte, dass die experimentelle Unsicherheit größer als der Unterschied zwischen einfacher und doppelter Genauigkeit ist. Allerdings ist es bei dem Simulator ebenfalls geplant für die Zukunft die Berechnungen optional in doppelter Genauigkeit zu implementieren.
- **Niedrige DP-Performance auf GPUs:** Viele NVIDIA Graphikkarten besitzen nur eine extrem niedrige DP-Performance. Die Gründe hierfür sind eine künstliche Limitierung bei Consumer-Graphikkarten oder eine Platzeinsparung durch weniger DP-Recheneinheiten auf dem GPU-DIE. Daraus folgt, dass bei vielen NVIDIA-Graphikkarten die DP-Performance nur in etwa $\frac{1}{24}$ (Geforce 780 GTX) bis $\frac{1}{32}$ (Geforce 980 GTX) der SP-Performance beträgt. Dadurch sind performante DP-Berechnungen auf diesen Karten nicht durchführbar. Dies ist zwar auf den Geforce Titans und Geforce Titan Blacks, welche als primäre Test-Graphikkarten verwendet wurden, kein Problem, da das Verhältnis dort $\frac{1}{3}$ beträgt. Sie sind zudem die bislang einzige NVIDA Consumer-Graphikkarten-Serie mit guter DP-Performance. Jedoch war weder der Typ der Test-Graphikkarten im Vorneherein bekannt noch scheint eine solche zusätzliche Einschränkung auf wenige NVIDIA-Graphikkarten sinnvoll.

Somit ist für folgende Arbeit die einfache Genauigkeit ausreichend. Zusätzlich würde sich die Implementierung der Arbeit bei der Verwendung von doppelter Genauigkeit auf sehr wenige Graphikkarten beschränken. Deshalb verwendet die Arbeit stets die einfache Rechengenauigkeit.

4.5 Abspeicherung der Partikeldaten

Folgende Punkte sollen kurz erläutern, wie die Daten der Partikel und des Gitters im globalen Speicher der GPU abgespeichert werden. Hierfür geht dieser Punkt zuerst auf die Abspeicherung der Partikeldaten ein.

So ist es bei SPH-Simulationen vorteilhaft, wenn die Partikeldaten nicht im Gitter selbst sondern in einer separaten Array-Datenstruktur abgespeichert werden. Da die Arbeit zudem ein Dynamic-Vector-Gitters verwendet, sind die Partikel innerhalb dieser Array-Datenstruktur stets nach Gitterzellen sortiert. Eine SPH-Simulation kann für diese Array-Datenstruktur entweder eine Structure aus Arrays oder ein Array aus Structures verwenden. Dabei ist das Array aus Structures meist nachteilig für die Speicherzugriffe. Denn eine SPH-Simulation benötigt bei einem bestimmten Fixed-Radius-Near-Neighbors-Pass meist nicht alle Daten eines Partikels, wodurch die einzelnen Speicherzugriffe auf das Array einen sehr großen Stride besitzen. Der Stride wirkt sich katastrophal auf die Speicherbandbreite und Cache-Hitrate aus. Zudem kann die SPH-Simulation bei dem Array aus Structures nicht leicht für alle Partikel jeweils einen einzigen Member der Structure kopieren. Das einfache Kopieren ist jedoch für die Cluster-Simulation stark von Vorteil. Deshalb verwendet die Implementierung dieser Arbeit für die Abspeicherung die Structure aus Arrays.

Da die implementierte SPH-Simulation mit der einfachen Genauigkeit rechnet, verwendet sie zur Abspeicherung der Partikeldaten ebenfalls nur die einfache Genauigkeit. So speichert sie die Partikeldaten im globalen Speicher der GPU als eine Structure, die aus mehreren 4-Byte- und 16-Byte-Arrays besteht, ab. Dabei setzen sich die Daten des $n$ten Partikels der Simulation aus den $n$ten Einträgen aller Arrays zusammen. Die Simulation verwendet die 16-Byte-Arrays meist dafür, die dreidimensionalen Partikeldaten, wie die Geschwindigkeiten oder die Positionen, per Float4 abzuspeichern. Dadurch kann die GPU wegen dem Coalescing chaotischere Zugriffe auf die Partikeldaten, wie sie bei der Berechnung des Fixed-Radius-Near-Neighbors-Problems auftreten, wiederum performanter abarbeiten. Zusätzlich ist eine 16-Byte-Ladeoperation für den Texture-Cache deutlich performanter als eine 4-Byte-Ladeoperation. Allerdings geht dadurch die vierte Komponente des Float4 verloren, wodurch die Simulation wiederum Speicherplatz, Speicherbandbreite und Platz im Cache verschwendet. So setzt sich die Structure aus Arrays aus folgenden 16-Byte-Arrays zusammen:

1. **16-Byte-Array:** Für die Positionen der Partikel zu Beginn des Zeitschritts
2. **16-Byte-Array:** Für die Positionen der Partikel zum Halbzeitschritt
3. **16-Byte-Array:** Für die Geschwindigkeiten der Partikel zu Beginn des Zeitschritts
4. **16-Byte-Array:** Für die Geschwindigkeiten der Partikel zum Halbzeitschritt
5. **16-Byte-Array:** Durch Zeiger-Aliasing wird es für mehrere Zwecke verwendet:
   - Für das Float4-Array der zeitlichen Änderungen der Dichten und der Geschwindigkeiten
   - Für das temporäre Float4-Array bei der Float4-Permutation (siehe Punkt 4.8.1.3)
   - Als temporäres Uint-Array für die Indexe der Partikel in ihrer Gitterzelle, die bei der Konstruktion des Gitters mit Counting-Sort benötigt werden

- Für die gewichteten Mittel, die bei der Berechnung der Ellipsoid-Hauptachsen für die Visualisierung (siehe Punkt 5.3) benötigt werden

6. 16-Byte-Array: Für die ersten Hauptachsen der Ellipsoide der Visualisierung und die X-Komponente von deren Positionen (siehe Punkt 5.3)

7. 16-Byte-Array: Für die zweiten Hauptachsen der Ellipsoide der Visualisierung und die Y-Komponente von deren Positionen

8. 16-Byte-Array: Für die dritten Hauptachsen der Ellipsoide der Visualisierung und die Z-Komponente von deren Positionen

Zusätzlich besitzt die Structure aus Arrays folgende 4-Byte-Arrays:

1. 4-Byte-Array: Für die Permutationsindexe (siehe Punkt 4.8.1.3)

2. 4-Byte-Array: Für die Dichten der Partikel zu Beginn des Zeitschritts oder alternativ für die Dichten vor dem Shepard-Filter

3. 4-Byte-Array: Für die Dichten der Partikel zum Halbzeitschritt oder alternativ für die Dichten nach dem Shepard-Filter

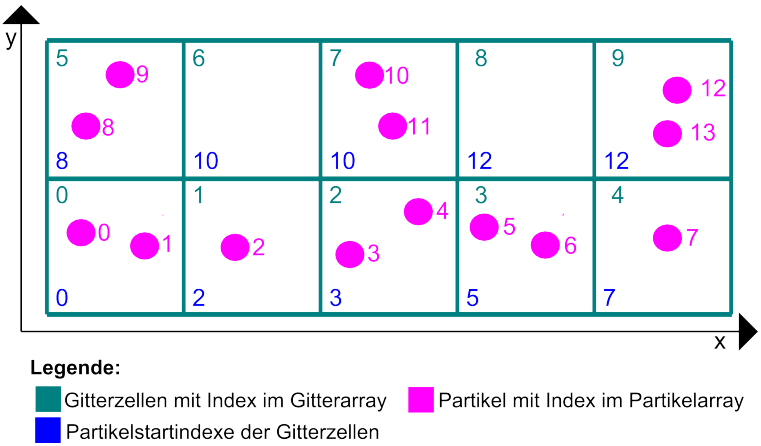
4. 4-Byte-Array: Für das temporäre Float-Array bei der Float-Permutation (siehe Punkt 4.8.1.3)

Dieser Speicherplatzverbrauch wird im Laufe der Arbeit jedoch noch etwas reduziert. So wird im Punkt 4.9.9 die Performance untersucht, wenn die Dichten der Partikel in die nicht genutzte vierte Komponente des Float4 der Positionen gepackt werden. Durch die Optimierung entfallen das zweite, dritte und vierte Float-Array. Bis zu diesem Punkt 4.9.9 wird die Optimierung des Packings jedoch nicht verwendet. Des Weiteren benötigt die SPH-Simulation zusätzliche 4 Byte Speicherplatz pro Partikel, sofern die Blocking-Optimierung aus Punkt 4.9.6 verwendet wird. Diese Optimierung wurde jedoch in der finalen Version der Arbeit verworfen.

Aus den Gründen benötigt ein Partikel in der finalen Version der Arbeit mit Visualisierung, Packing der Postionen und Dichten sowie ohne Blocking insgesamt 132 Byte an Speicherplatz. Unter der Annahme, dass das Gitter nur sehr wenig globalen Speicherplatz kostet und deshalb alle 6 GiByte des globalen Speicher der GPU für Partikel verwendet werden können, ergeben sich insgesamt 48.8 Millionen Partikel. Dieser Wert scheint zunächst groß zu sein. Jedoch ergibt er wegen der Dreidimensionalität der Simulation lediglich einen Würfel mit einer Kantenlänge von 365 Partikeln. Dies verdeutlicht noch einmal, dass für eine fein aufgelöste dreidimensionale Flüssigkeitssimulation sehr viel Speicherplatz benötigt wird.

Der zusätzliche Speicherplatzverbrauch durch die Visualisierung ließe sich jedoch durch für die Arbeit nicht mehr durchgeführte Optimierungen komplett vermeiden. Diese Optimierungen sind in Punkt 5.3 erläutert. Würde die Arbeit zusätzlich diese Optimierung verwenden, so würden pro Partikel nur noch 84 Byte an Speicherplatz anfallen. Auch würden pro Partikel nur noch 8 Byte an Speicherplatz, nämlich für die vierte Komponente der beiden Float4 der Geschwindigkeiten, verloren gehen.

Schematische Darstellung eines Dynamic-Vector-Gitters:



Aus dem Gitter resultierende Array-Datenstruktur:

Indexe der Gitterzellen im Gitterarray	0	1	2	3	4	5	6	7	8	9
Partikelstartindexe	0	2	3	5	7	8	10	10	12	12

Abbildung 4.1: Beispiel für ein zweidimensionales Dynamic-Vector-Gitter

4.6 Abspeicherung des Gitters

Folgender Punkt soll die Abspeicherung des Gitters vorstellen. Die Implementierung verwendet als Gitter ein kartesisches Dynamic-Vector-Gitter, das sie als eine Structure aus Uint-Arrays im globalen Speicher der GPU abspeichert. In einem Array wird für jede Gitterzelle der Index des ersten Partikels, das sich in der Gitterzelle befindet, abgespeichert. Dieser Index referenziert den entsprechenden Eintrag des Partikels in der Structure aus Arrays für Partikeldaten und wird im Folgenden Partikelstartindex genannt. Zudem wird beim Bauen des Gitters in dieser Arbeit stets sicher gestellt, dass die Partikelstartindexe monoton ansteigend sind. Wegen der Sortierung der Partikel nach Gitterzellen und der Monotonie lässt sich die Anzahl beziehungsweise die Menge der Partikel in einer Gitterzelle aus der Differenz zwischen dem Partikelstartindex der nächsten Gitterzelle und dem Partikelstartindex der Gitterzelle selbst berechnen. Ein weiteres Array, in welchem das Gitter die Indexe des letzten Partikels innerhalb der Gitterzelle abspeichert, entfällt dadurch. Somit lässt sich das Gitter alleine durch das Array mit den Partikelstartindexen vollständig beschreiben. Ein solches Dynamic-Vector-Gitter ist beispielhaft für den zweidimensionalen Fall in Abbildung 4.1 gezeigt. Hinzu kommen bei dieser Structure fallabhängig weitere temporäre Arrays, welche für das Bauen des Gitters mit Radix-Sort (siehe Punkt 4.8.2) oder für das Cache-Blocking (siehe Punkt 4.9.6) benötigt werden. Da beides in der finalen Version der Arbeit nicht verwendet wird, verbraucht das Gitter dort nur 4 Byte pro Gitterzelle.

Innerhalb dieser Arrays ist das dreidimensionale Gitter als linearisiertes dreidimensionales Ar-

ray im Speicher der GPU angeordnet. Dadurch kann die Implementierung den Index I einer Gitterzelle mit den Koordinaten X, Y und Z im Gitter wie folgt errechnen:

$$I = \mathrm{Dim}_x \mathrm{Dim}_y Z + \mathrm{Dim}_x Y + X$$

4.7 Testsystem, Messmethoden, Testszene und Performanceüberblick

Damit die Untersuchungen in dem Kapitel nachvollziehbar bleiben werden diesem Punkt das verwendete Testsystem vorgestellt und die Messmethoden beschrieben. Anschließend wird die Testszene vorgestellt und ein erster grober Performanceüberblick gegeben.

Zuerst wird hier auf das Testsystem eingegangen. So wurde für die Tests eine Geforce Titan der Compute-Capability 3.5 mit folgenden Eigenschaften verwendet:

- **GPU-Takt:**
  - Ohne Boost: 837 MHZ
  - Mit Boost und ohne "Double Precision"-Option: 990 MHZ
  - Mit Boost und "Double Precision"-Option: 849 MHZ
- **Multiprozessoren:** 14
- **SP-FLOPS:**
  - Ohne Boost: 4500 GFLOPS
  - Mit Boost und ohne "Double Precision"-Option: 5322 GFLOPS
  - Mit Boost und "Double Precision"-Option: 4565 GFLOPS
- **DRAM:**
  - Größe: 6 GiByte GDDR5
  - Anbindung: 384-Bit
  - Takt: 6004 MHZ
  - Bandbreite: 288 GByte/s

Des Weiteren wurde Ubuntu-Linux mit der Graphiktreiberversion 340.29 und das CUDA-Toolkit mit der Version 6.5 verwendet. Bei dem GPU-Takt der Geforce Titan traten jedoch Probleme auf. So kann die Geforce Titan ihren GPU-Takt ausgehend von diversen Kriterien, wie Stromverbrauch, Temperatur und Siliziumqualität des GPU-DIEs, durch eine Boost-Funktion dynamisch um bis zu 150 MHZ erhöhen. Dieser Boost ist unter Windows nicht direkt sondern nur über Workarounds deaktivierbar. Allerdings wurde keine solche Möglichkeit unter Linux gefunden. Der Stromverbrauch und die Temperatur können aber zeitlich schwanken. Deshalb ergibt sich durch die Boost-Funktion ein nur schwer kompensierbarer Messfehler. Der Fehler lässt sich etwas reduzieren, indem die "Double Precision"-Option im Treiber aktiviert wird. Diese Option führt dazu, dass die GPU ihren DPFP-Durchsatz erhöht. Zusätzlich bewirkt die Option den Nebeneffekt, dass die GPU die potentiell höhere Verlustleistung durch einen deutlich niedrigeren Boost-Takt kompensiert. Dieser Effekt wird ausgenutzt, um den Messfehler

durch den Boost von maximal 18% auf maximal 1.4% zu reduzieren. Da die Fixed-Radius-Near-Neighbors-Berechnungen in dieser Arbeit keinen Shared-Memory aber lokalen Speicher benötigen, konfiguriert die Implementierung den L1-Cache auf dessen maximale Größe von 48 kiByte.

Für die Messungen selbst wurde der Visual-Profiler verwendet. Er ist ein mächtiges Programm des NVIDIA-Toolkits. Er kann neben den Kernellaufzeiten viele Hardware-Metriken, wie die erreichten FLOPS oder die verwendete Speicherbandbreite, messen. Da es im Visual-Profiler sehr viele Metriken gibt führt im Folgenden die Arbeit nur diejenigen Metriken auf, die für die jeweilige Diskussion von Bedeutung sind. Bei den Messungen traten allerdings zusätzliche Probleme auf:

- **Nicht näher definierte und widersprüchliche Metriken:** Der Visual-Profiler misst Hardware-Events, welche die GPU durch Hardware-Zähler abzählt, und berechnet daraus Metriken. Wie sich diese Metriken aus den Events aber genau berechnen ist in neueren Visual-Profiler-Versionen nicht mehr dokumentiert. So gibt es bei der Compute-Capability 3.5 zwei Metriken für die Hitrate bei Lesezugriffen auf den L2-Cache: L2-Hitrate durch L1-Reads und L2-Hitrate durch Texture-Reads. Obwohl im L1-Cache bei der Compute-Capability 3.5 nur der lokale Speicher zwischengespeichert wird, so scheint die L2-Hitrate durch L1-Reads auch Lesezugriffe auf den globalen Speicher, die ohne Texture-Caching ausgeführt werden, zu umfassen. Bei der Compute-Capability 3.5 gehen Zugriffe auf den globalen Speicher, die kein Texture-Caching verwenden, aber ohne den L1-Cache abzufragen direkt auf den L2-Cache über. Interessanterweise gibt es ebenfalls eine Metrik für die L1-Hitrate durch globale Speicherzugriffe, welche im L1-Cache nicht zwischengespeichert werden. Des Weiteren umfasst die L2-Hitrate durch Texture-Reads auch diejenigen L2-Cache-Zugriffe, welche durch den im Texture-Cache zwischengespeicherten globalen Speicher entstehen.
- **Qualitative Ergebnisse bei Metriken:** Einige Metriken, welche die Auslastung beschreiben, gibt der Visual-Profiler nur qualitativ aus. Die Ausgabe besitzt die Form <Adjektiv>(<Ziffer>), wie zum Beispiel „Mid(6)“. Die Vermutung liegt nahe, dass die Ziffer den Zehnerprozentsatz der Auslastung angibt. Somit gibt der Visual-Profiler die Metriken nur ungenau aus. Zumindest bei der Auslastung der DRAM-Bandbreite besteht jedoch die Möglichkeit diese genau aus dem Verhältnis zwischen der gemessenen Lese-DRAM-Bandbreite und Schreib-DRAM-Bandbreite zur Peak-DRAM-Bandbreite zu berechnen.
- **Keine Metrik für Auslastung der CUDA-Cores:** Es gibt zwar eine Metrik für die Auslastung der arithmetischen Ausführungseinheiten. Wie der Visual-Profiler diese jedoch genau aus den Auslastungen der einzelnen unterschiedlichen arithmetischen Ausführungseinheiten berechnet ist unklar. Deshalb wird die Auslastung der CUDA-Cores in folgender Arbeit durch das Verhältnis zwischen den ausgeführten IPC und den sechs IPC, welche maximal durch die CUDA-Cores verarbeitet werden können, geschätzt. Da meist ein Großteil der Befehle in den Kernels CUDA-Core-Befehle sind, ist der Fehler dieser Schätzung gering.

Als Nächstes soll auf die Testszene eingegangen werden. So wurde als Testszene ein Küstenabschnitt mit einem Startabstand von 0.55 m verwendet. Daraus ergaben sich 3972734 Flüssigkeitspartikel. Um für eine Bewegung der Partikel zu sorgen wird eine der Festkörper-Randebenen durch eine Sinus-Funktion entlang ihrer Normalen in Bewegung versetzt. Dadurch entstehen

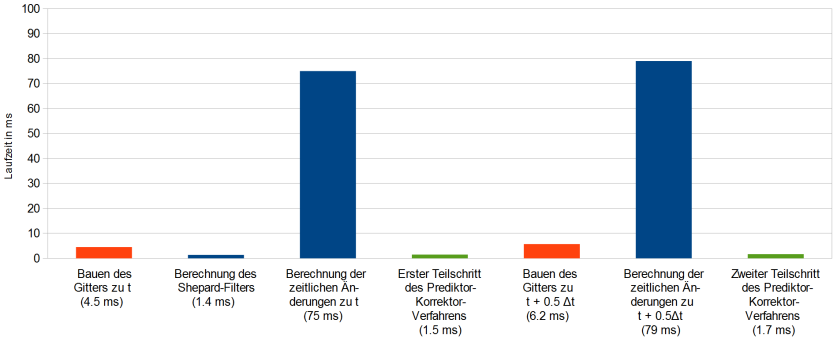


Abbildung 4.2: Übersicht über die Laufzeiten der einzelnen Schritte der SPH-Algorithmus

wiederum Wellen in der Flüssigkeit. Zu Beginn der Simulation befinden sich die Partikel jedoch wegen der Wahl der Anfangsbedingungen aus Punkt 3.11 auf einem regelmäßigen Gitter. Diese regelmäßige Anordnung würde jedoch künstlich die Datenparallelität erhöhen und damit die Messungen verfälschen. Deshalb wartete diese Arbeit für die Messung 10000 Zeitschritte bis die Partikel in Unordnung waren. Für die Kernellaufzeiten wird die Laufzeit von 100 Schritten gemittelt. Da das Profiling im Visual-Profiler bis zu eine Stunde pro Zeitschritt dauert, werden für die Metriken nur 10 Zeitschritte gemittelt.

Nachdem Testsystem, Testszene und Messmethoden vorgestellt worden sind, lässt sich gemäß dieser ein grober Performanceüberblick erstellen. Der Performanceüberblick wurde durch die fertig optimierte Version des Programms erstellt und ist im Diagramm der Abbildung 4.2 gezeigt. Da der Shepard-Filter nur alle 30 Zeitschritte berechnet wird, wurden seine Kosten in dem Diagramm auch über diese 30 Zeitschritte gemittelt. Bei dem Diagramm wird deutlich, dass das Bauen des Gitters, die Zeitintegration, und der Shepard-Filter extrem günstig sind. Am teuersten sind die Berechnungen der zeitlichen Änderungen. Deshalb sollte für das Optimieren dieses SPH-Verfahrens das meiste Augenmerk auch darauf gelegt werden, wie die Berechnungen der zeitlichen Änderungen optimiert werden können.

4.8 Bauen des Gitters

4.8.1 Counting-Sort

4.8.1.1 Übersicht

Zuerst muss die SPH-Simulation zu Beginn jedes Zeitschritts oder Halbzeitschritts das Gitter bauen. Dieses Bauen soll im Folgenden näher erklärt und untersucht werden. Hierfür implementiert die Arbeit zwei Möglichkeiten: Radix-Sort und Counting-Sort. Als Erstes wird nun der Counting-Sort-Ansatz erklärt, welcher aus [SG] und [RHb] stammt.

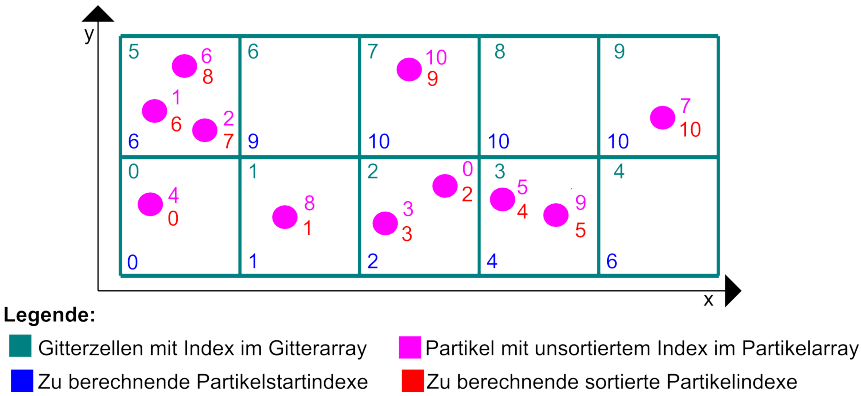
Bei diesem implementierten Ansatz wird das Gitter-Array der Partikelstartindexe zuerst dafür

verwendet, um die Anzahl der Partikel in jeder Gitterzelle abzuspeichern. Anschließend wendet die SPH-Simulation die in-place arbeitende Prefix-Sum auf dieses Array an um die Partikelstartindexe im selben Array zu erhalten. So lässt sich das Counting-Sort-Verfahren auf in folgende Schritte, die noch einmal in Abbildung 4.3 anhand eines Beispiels veranschaulicht werden, unterteilen:

- **Initialisierung:** Zuerst setzt die SPH-Simulation die Anzahl der Partikel in jeder Gitterzelle per cudaMemset auf null.
- **Einfügen der Partikel in die Gitterzellen:** Als Nächstes findet der Einfügen-Schritt statt. Die SPH-Simulation berechnet ihn in einem CUDA-Kernel, welches in Abbildung 4.4 zu sehen ist. Für das Kernel wird zunächst ein GPU-Thread pro Partikel gestartet. Das Partikel des Threads besitzt zunächst einen unsortierten Index. Der Thread bestimmt anhand der aktuellen Position des Partikels, in welcher Gitterzelle das Partikel ist. Dann erhöht er per atomarer Addition die Anzahl der Partikel in dieser Gitterzelle um eins. Der Rückgabewert der atomaren Addition ist der sortierte Index des Partikels innerhalb der Gitterzelle. Zuletzt schreibt der Thread den Wert in das entsprechende Array zurück.
- **Prefix-Sum über die Anzahl der Partikel in den Gitterzellen:** Anschließend führt die SPH-Simulation eine Prefix-Sum über die Anzahl der Partikel in den Gitterzellen aus. Dadurch erhält die SPH-Simulation für jede Gitterzelle einen sortierten Partikelstartindex.
- **Berechnung der Permutationsindexe:** Nun kann jedes Partikel aus der Addition von dem Partikelstartindex seiner Gitterzelle und seinem Partikelindex innerhalb seiner Gitterzelle, seinen neuen sortierten Partikelindex berechnen. Aus den sortierten Partikelindexen werden nun die Permutationsindexe entsprechend der Permutationsmethode erstellt.
- **Permutation der Partikeldaten:** Nun werden anschließend noch die Partikeldaten anhand des Permutationsindexes so permutiert, dass sie ebenfalls nach Gitterzellen sortiert sind.

Diese Konstruktionsmethode hat durch die Prefix-Sum für die Gittertraversierung die vorteilhafte Eigenschaft, dass die Partikelstartindexe der Gitterzellen monoton ansteigend sind. Interessanterweise ist dieses Counting-Sort-Verfahren, dadurch dass die Reihenfolge der nebenläufigen atomaren Operationen nicht definiert ist, weder stabil noch deterministisch. Das ist für Counting-Sort unüblich. Da die Reihenfolge der Partikel innerhalb einer Gitterzelle bei einer SPH-Simulation prinzipiell egal ist, hat dies hier keine direkten negativen Auswirkungen. Allerdings führt die zufällige Reihenfolge der Partikel bei den Berechnungen des Fixed-Radius-Near-Neighbors-Problems dazu, dass die Reihenfolge der Summanden in den Gleichungen der SPH-Interpolation vertauscht werden. Da das Kommutativgesetz bei Gleitkommaberechnungen nicht gilt führt dies wiederum dazu, dass die gesamte Simulation nicht mehr deterministisch ist. Erschwerend kommt hinzu, dass Flüssigkeitssimulationen chaotische Systeme sind. Deshalb können diese kleinen zufälligen Fehler nach kürzerer Zeit zu großen Schwankungen im Ergebnis führen. Dieses Problem könnte die SPH-Simulation jedoch leicht kompensieren, indem sie die Partikel noch einmal in jeder Gitterzelle sortiert. Da es nur wenige Partikel pro Gitterzelle gibt, wäre der Aufwand dafür auch niedrig.

Darstellung des zu bauenden Gitters:



Einfügen der Partikel in die Gitterzellen:
Partikel:

Unsortierter Partikelindex	0	1	2	3	4	5	6	7	8	9	10
Index der Gitterzelle des Partikels	2	5	5	2	0	3	5	9	1	3	7
Index des Partikels innerhalb der Gitterzelle	0	0	1	1	0	0	2	0	0	1	0

Gitter:

Indexe der Gitterzellen im Gitterarray	0	1	2	3	4	5	6	7	8	9
Anzahl der Partikel in Gitterzelle	1	1	2	2	0	3	0	1	0	1

Prefix-Sum über die Anzahl der Partikel in den Gitterzellen:

Indexe der Gitterzellen im Gitterarray	0	1	2	3	4	5	6	7	8	9
Anzahl der Partikel in Gitterzelle	1	1	2	2	0	3	0	1	0	1
Partikelstartindexe	0	1	2	4	6	6	9	9	10	10

Berechnung der sortierten Partikelindexe:

Unsortierter Partikelindex	0	1	2	3	4	5	6	7	8	9	10
Index der Gitterzelle des Partikels	2	5	5	2	0	3	5	9	1	3	7
Partikelstartindex der Gitterzelle des Partikels	2	6	6	2	0	4	6	10	1	4	9
Index des Partikels innerhalb der Gitterzelle	0	0	1	1	0	0	2	0	0	1	0
Sortierter Partikelindex = Partstartind. + Ind. in Zelle	2	6	7	3	0	4	8	10	1	5	9

Abbildung 4.3: Beispiel für die Konstruktion des Gitters durch Counting-Sort

```
__global__ void InsertParticles(ParticleData Particles, ParticleGrid Grid)
{
  int ThreadID = blockIdx.x * blockDim.x + threadIdx.x;
  float3 Position = make_float3(Particles.Positions[ThreadID]);
  uint CellIndex = Grid.GetCellIndex(Position);
  uint PartIndexInCell= atomicAdd(&Grid.ParticleCountInCell[CellIndex], 1);
  Particles.IndicesInCell[PartID] = PartIndexInCell;
}
```

Abbildung 4.4: Kernel für das Einfügen der Partikel in die Gitterzellen

In den Folgenden Punkten soll kurz das Vorgehen der Prefix-Sum und Permutation erläutert werden. Anschließend soll die Konstruktion des Gitters auf ihre Performance untersucht werden. Bei den dafür durchgeführten Benchmarks wird die in Punkt 4.9.3 ermittelte optimale Gitterzellengröße verwendet, welche ein Drittel der Cutoff-Distance beträgt. Dadurch hat das Gitter in der Testszene die Dimensionen von 364 auf 201 auf 364, wodurch sich insgesamt 26 631 696 Gitterzellen ergeben.

4.8.1.2 Prefix-Sum

Als Prefix-Sum implementiert die Arbeit selbst ein einfaches Verfahren, welches sich in drei Schritte beziehungsweise CUDA-Kernel unterteilen lässt:

- **1. Prefix-Schritt: Aufsummieren der Partikelanzahl innerhalb der Gitterzellen eines Thread-Blocks.** Das Prefix-Sum-Verfahren startet ein Kernel mit jeweils einem Thread für vier Gitterzellen. Der Thread lädt zuerst per 16-Byte-Ladeoperation die Anzahlen der Partikel innerhalb seiner vier Gitterzellen. Anschließend summiert er sie auf. Diese Summe wird nun ebenfalls innerhalb des Thread-Blocks aufsummiert, wodurch das Kernel die gesamte Anzahl von Partikeln innerhalb der Gitterzellen des gesamten Thread-Blocks erhält.
- **2. Prefix-Schritt: Prefix-Sum über die Anzahl der Partikel der Thread-Blocks.** Es wird ein Kernel mit einem einzigen Thread-Block gestartet, welcher ausgehend von der gesamten Anzahl der Partikel pro Thread-Block aus dem letzten Schritt für jeden dieser Thread-Blocks einen Partikelstartindex berechnet.
- **3. Prefix-Schritt: Prefix-Sum über die Partikel innerhalb eines Thread-Blocks.** Zuletzt startet das Prefix-Sum-Verfahren wieder ein Kernel mit einem Thread pro vier Gitterzellen. Jeder Thread berechnet zusammen mit seinem Thread-Block ausgehend vom Partikelstartindex des Thread-Blocks aus dem letzten Schritt durch die Prefix-Sum innerhalb des Thread-Blocks den Startindex jeder seiner vier Gitterzellen. Zuletzt schreibt er sie als Endergebnis der Prefix-Sum zurück.

Dabei implementiert die Arbeit allgemein das Aufsummieren von Werten von unterschiedlichen Threads innerhalb eines Thread-Blocks beziehungsweise die Prefix-Sum innerhalb eines Thread-Blocks hauptsächlich durch Warp-Shuffle-Funktionen.

	Laufzeit	DRAM-Auslastung	CUDA-Core-Auslastung
1. Prefix-Schritt	0.81 ms	48%	23%
2. Prefix-Schritt	0.07 ms	1.5%	1.5%
3. Prefix-Schritt	0.92 ms	81%	28%

Abbildung 4.5: Benchmark für die Prefix-Sum

Die Messungen in Abbildung 4.5 zeigen, dass das erste Kernel und das dritte Kernel der Prefix-Sum einen Großteil der Performance kosten, während das zweite Kernel nur noch wenig Performance kostet. Zudem sind das ist dritte Kernel durch die Speicherbandbreite limitiert, während das zweite Kernel mit seinem einzigen Thread-Block die GPU nur schlecht auslastet. Deshalb wurde ebenfalls versucht ein Verfahren zu schreiben, welches die Prefix-Sum mit einem einzigen Pass beziehungsweise Kernel berechnen kann. Dadurch sollte vermieden werden, dass die Daten mehrmals gelesen werden müssen. Auf diese Weise sollte wiederum die Limitierung durch die Speicherbandbreite vermeiden werden. Zudem würde auch das zweite Kernel, welches die GPU nur schlecht auslastet, wegfallen. Da nun der Schritt des obigen zweiten Kernels On-The-Fly berechnet werden musste, wurde eine Synchronisation zwischen unterschiedlichen Thread-Blocks benötigt. Diese Synchronisation führte nun dazu, dass die Auslastung der GPU gering war und das Verfahren in etwa dreimal so lange gedauert hat. Deshalb wurde dieser Ansatz verworfen. Letztlich ist die Prefix-Sum ein komplexes Thema, weshalb die Implementierung dieser Arbeit noch viele Verbesserungsansätze besitzt. Somit wäre es versuchenswert die Prefix-Sum weiter mit den Vorschlägen aus der Literatur wie zum Beispiel mit denjenigen aus dem Artikel [MaH] zu verbessern.

4.8.1.3 Permutation der Partikeldaten

Als Nächstes soll auf die Permutation der Partikeldaten eingegangen werden. Dafür werden zunächst die Permutationsindexe je nach Permutationsmethode in einem Kernel berechnet. Anschließend findet die Permutation der Datenarrays mit den so eben berechneten Permutationsindexen statt. Die Permutation wird dabei für jedes einzelne der gerade benötigten Float4- und Float-Datenarrays nacheinander mit jeweils einem eigenen Kernelaufruf ausgeführt. Die Kernel schreiben für jedes Partikeldatenarray die permutierten Daten in das temporäre Float- beziehungsweise Float4-Datenarray. Anschließend vertauscht die Permutation die Zeiger auf die Arrays. Für das Berechnen der Permutationsindexe und der Permutation selbst gibt es zwei unterschiedliche Permutationsmethoden:

- **„Vorwärtspermutation“:** Die Permutationsmethode berechnet als Erstes in einem Kernel, welches in Abbildung 4.6 gezeigt ist, die Permutationsindexe. Dabei speichert es die neuen permutierten Partikelindexe als Permutationsindexe an ihrer ursprünglichen Position ab. Anschließend wird ein Permutationskernel mit einem Thread pro Partikel gestartet. In dem Kernel liest ein jeder Thread den Permutationsindex und die zu permutierenden Partikeldaten an der Stelle seiner Thread-ID ein. Anschließend schreibt er die Partikeldaten ins temporäre Array an die Stelle des Permutationsindex zurück.

```
__global__ void CalcForwardPermutationIndicesKernel(ParticleData Particles, ParticleGrid Grid)
{
  uint ThreadID = blockIdx.x * blockDim.x + threadIdx.x;
  float3 Position = make_float3(Particles.Positions[ThreadID]);
  uint CellIndex = Grid.GetCellIndex(Position);
  uint CellBegin = Grid.ParticleStart[CellIndex];
  uint IndexNew = CellBegin + Particles.IndicesInCell[PartID];
  Particles.PerumteIndices[ThreadID] = IndexNew;
}

template<typename PermuteType>
__global__ void PermuteForwardKernel(PermuteType* In, PermuteType* Out, uint* PerumteIndices)
{
  uint ThreadID = blockIdx.x * blockDim.x + threadIdx.x;
  uint PerumteIndex = PerumteIndices[ThreadID];
  //coalesced read
  PermuteType Temp = In[ThreadID];
  //chaotic write
  Out[PerumteIndex] = Temp;
}
```

Abbildung 4.6: Kernel für die Vorwärtspermutation

- **„Rückwärtspermutation“:** Bei dieser Permutationsmethode werden als erstes in einem Kernel, wie es in Abbildung 4.7 zu sehen ist, die Permutationsindexe berechnet. Es speichert die ursprünglichen Partikelindexe als Permutationsindexe an ihrer permutierten Position ab. Nun wird wieder ein Permutationskernel mit einem Thread pro Partikel gestartet. In dem Kernel liest jeder Thread zunächst den Permutationsindex an der Stelle seiner Thread-ID ein. Als Nächstes lädt er die zu permutierenden Partikeldaten an der Stelle des Permutationsindexes und schreibt sie in das temporäre Array an die Stelle seiner Thread-ID zurück. Der Quelltext dieses Kernels ist in Abbildung 4.7 zu lesen.

Dadurch werden bei der Vorwärtspermutation die Partikeldaten sequentiell beziehungsweise coalesced aus dem globalen Speicher eingelesen, während sie chaotisch in den globalen Speicher zurückgeschrieben werden. Bei der Rückwärtspermutation findet aber das Einlesen chaotisch und das Zurückschreiben sequentiell statt. Je nach genauem Caching-Verhalten der GPU können die Unterschiede dazu führen, dass eine der beiden Permutationsmethoden schneller oder langsamer ist. Deshalb wurden beide Permutationsmethoden implementiert und untersucht. Hierfür wurde jeweils die Laufzeit für die Permutation eines einzigen Float4-Arrays und Float-Arrays gemessen. Bei den Messergebnissen in Abbildung 4.8 zeigt sich, dass sowohl die Vorwärtspermutation als auch die Rückwärtspermutation eines Float4-Arrays mit je 0.61 ms in etwa gleich schnell sind. Lediglich die Float-Rückwärtspermutation ist mit 0.22 ms gegenüber 0.24 ms minimal langsamer. Das ist vermutlich auf ihren geringeren Memory-Level-Parallelism zurückzuführen. Somit sind beide Methoden in etwa gleichwertig. In Folgender Arbeit wird aus historischen Gründen die Rückwärtspermutation weiter verwendet.

```
__global__ void CalcBackwardPermutationIndicesKernel(ParticleData Particles, ParticleGrid Grid)
{
  uint ThreadID = blockIdx.x * blockDim.x + threadIdx.x;
  float3 Position = make_float3(Particles.Positions[ThreadID]);
  uint CellIndex = Grid.GetCellIndex(Position);
  uint CellBegin = Grid.ParticleStart[CellIndex];
  uint IndexNew = CellBegin + Particles.IndicesInCell[PartID];
  Particles.PerumteIndices[IndexNew] = ThreadID;
}

template<typename PermuteType>
__global__ void PermuteBackwardKernel(PermuteType* In, PermuteType* Out, uint* PerumteIndices)
{
  uint ThreadID = blockIdx.x * blockDim.x + threadIdx.x;
  uint PerumteIndex = PerumteIndices[ThreadID];
  //chaotic read
  PermuteType Temp = In[PerumteIndex];
  //coalesced write
  Out[ThreadID] = Temp;
}
```

Abbildung 4.7: Kernel für die Rückwärtspermutation

	Float	Float4
Laufzeit Vorwärtspermutation	0.22 ms	0.61 ms
Laufzeit Rückwärtspermutation	0.24 ms	0.61 ms

Abbildung 4.8: Benchmark der Permutationsmethoden

4.8.1.4 Untersuchung

Abschließend soll das Bauen des Gitters mit Counting-Sort untersucht werden. Hierfür wurden alle Schritte, welche für das Bauen benötigt werden, im Visual-Profiler gebenchmarkt und in Abbildung 4.9 eingetragen. Da die GPU beim Bauen zum Halbzeitschritt mehr Daten permutieren muss als beim Bauen zu Beginn eines Zeitschritts, wurde die Dauer der Float- und Float4-Permutation über beide Bauvorgänge gemittelt. Letztlich gilt es anzumerken, dass mit der Optimierung des Packings aus Punkt 4.9.9 die Float-Permutation entfällt, wodurch die Konstruktion etwas günstiger wird.

Bei fast allen Schritten gilt, dass sie durch die Speicherbandbreite limitiert sind. Denn die in etwa gemessene Auslastung der Speicherbandbreite von 80% ist je nach Zugriffsmuster und Memory-Level-Parallelism in etwa das Maximum, welches die GPU bei unoptimierten Speicherzugriffen in einem Kernel erreichen kann. Eine Ausnahme hierfür ist die Float-Permutation, welche für eine hohe Auslastung der Speicherbandbreite einen zu niedrigen Memory-Level-Parallelism besitzt. Ebenso ist der erste und zweite Schritt der Prefix-Sum nicht durch die Speicherbandbreite limitiert.

Werden beim Einfügen-Kernel die atomaren Operationen pro Takt berechnet, so ergibt sich ein

	Laufzeit	DRAM-Auslastung	CUDA-Core-Auslastung
Initialisierung	0.44 ms	89%	0%
Einfügen	0.51 ms	79%	20%
1. Prefix-Schritt	0.81 ms	48%	23%
2. Prefix-Schritt	0.07 ms	1.5%	1.5%
3. Prefix-Schritt	0.92 ms	81%	28%
Permutationsindexe	0.50 ms	79%	22%
Float-Permutation	0.37 ms	67%	13%
Float4-Permutation	1.85 ms	82%	5%
Insgesamt	5.5 ms	71%	17 %

Abbildung 4.9: Benchmark für die Konstruktion des Gitters:

Wert von in etwa 10 Operationen pro Takt. Dieser Wert ist deutlich niedriger als der Maximalwert von 64 Operationen pro Takt, welche die GPU maximal ausführen kann. Dies verdeutlicht noch einmal die Performanz der Atomic-Units der GPU. Interessanterweise nimmt die Laufzeit des Einfügen-Schritts deutlich von 0.78 ms auf 0.98 ms ab, wenn die Gitterzellen von einem Drittel der Cutoff-Distance auf die Cutoff-Distance vergrößert werden. Auf diese Weise muss das Kernel mehr Gitterdaten aus dem DRAM laden und zurückschreiben, weshalb zunächst eine längere Laufzeit zu erwarten wäre. Der Performancegewinn ist aber wahrscheinlich darauf zurückzuführen, dass die atomaren Speicherzugriffe des Kernels bei kleinerer Gitterzellengröße chaotischer werden. Auf diese Weise kann das Kernel wiederum die Atomic-Units besser auslasten.

Die Auslastung der CUDA-Cores ist bei allen Schritten hauptsächlich wegen der Limitierung durch die Speicherbandbreite mit 5% bis 28% schlecht. Insgesamt ergibt dies eine mittlere Auslastung von 17%. Da diese Auslastung nur eine optimistische Schätzung ist, ist die tatsächliche Auslastung wahrscheinlich noch geringer. Zudem beinhalten die Kernel für die Konstruktion des Gitters an diversen Stellen Modifikationen für die Cluster-Version. Deswegen wäre in einer auf eine einzige GPU optimierten Version diese Auslastung ebenfalls noch einmal geringer.

Zusammenfassend kann gesagt werden, dass die Konstruktion des Gitters, abgesehen von den ersten beiden Schritten der Prefix-Sum und der Float-Permutation, durch die Speicherbandbreite limitiert wird. Aus dem Grund ist es der wichtigste Verbesserungsansatz die Auslastung und Ausnutzung der Speicherbandbreite beim Bauen des Gitters zu optimieren. Hierfür könnte die Implementierung die Techniken aus dem Artikel [VV] verwenden. Die Techniken versuchen den Memory-Level-Parallelism zu optimieren, damit die GPU ihre Speicherbandbreite besser auslasten kann. Zudem entsteht bei jeder Float4- und Float-Permutation ein Overhead, da das Kernel jedes mal erneut die Permutationsindexe aus dem DRAM laden muss. Dies ließe sich in einem modifizierten Kernel etwas vermeiden, welches mehrere Float4- und Float-Permutationen gleichzeitig durchführt. Da eine parallele Permutation immer Double-Buffered durchgeführt werden muss, würden allerdings wiederum weitere temporäre Float- und Float4-Arrays benötigt werden. Auch könnten die ersten Permutationen bereits in demjenigen Kernel durchgeführt werden, welches die Permutationsindexe berechnet. Da die GPU dadurch die Per-

mutationsindexe einmal weniger komplett aus dem DRAM laden müsste, würde wiederum etwas Speicherbandbreite eingespart werden. Da die gesamte Konstruktion des Gitters mit circa 6% nur einen Bruchteil der Gesamtlaufzeit benötigt, besitzen die Optimierungen allesamt nur eine geringe Priorität.

4.8.2 Radix-Sort

Alternativ zur Konstruktion des Gitters per Counting-Sort kann eine SPH-Simulation für GPUs ebenfalls den Radix-Sort-Ansatz aus dem Artikel [SG] verwenden. Die Konstruktion durch Radix-Sort ist aber allgemein langsamer als die Konstruktion durch Counting-Sort. Deswegen ist die Konstruktion durch Radix-Sort nur noch aus historischen Gründen in der Implementierung dieser Arbeit vorhanden.

Für das Bauen mit Radix-Sort wird ein temporäres Uint-Array für das Gitter benötigt, in welches die SPH-Simulation die temporären Indexe des letzten Partikels in der entsprechenden Gitterzellen abspeichert. Die Indexe werden im Folgenden als Partikelendindexe bezeichnet. Zudem werden zwei temporäre Uint-Arrays für die Partikeldaten benötigt. In das eine Uint-Array der Partikeldaten werden die Indexe der entsprechenden Partikel als Schlüssel für die Radix-Sortierung abgespeichert. In dem anderen Uint-Array speichert die SPH-Simulation die Indexe derjenigen Zellen in denen sich die Partikel befinden als Werte für die Radix-Sortierung ab. Das Radix-Sort-Verfahren teilt sich in folgende Schritte auf:

- **Initialisierung:** Die Partikelstartindexe werden auf den höchst möglichen Wert und die Partikelendindexe auf null gesetzt.
- **Erstellung der Schlüssel-Werte-Paare:** Hier berechnet jedes Partikel den Index derjenigen Gitterzelle, in der es sich befindet. Dieser Index dient als Wert für den Radix-Sort, während der Index des Partikels im Partikel-Array als Schlüssel dient. Beide Indexe werden entsprechend in die beiden temporären Uint-Arrays der Partikeldaten geschrieben.
- **Radix-Sort:** Hier werden nun die beiden Arrays mit den Schlüsseln und den Werten nach dem Array mit den Werten sortiert.
- **Aktualisierung der Partikelstartindexe und Partikelendindexe:** Anschließend wird über die sortierten Zellenindexe der Partikel iteriert. Unterscheidet sich der Zellenindex des vorherigen Partikels, so beginnt die Zelle bei diesem Partikel. Deshalb wird der Partikelstartindex dieser Gitterzelle entsprechend aktualisiert. Unterscheidet sich der Zellenindex des nächsten Partikels, so endet die Zelle bei diesem Partikel. In diesem Fall wird der Partikelendindex dieser Gitterzelle ebenfalls entsprechend aktualisiert. Da bei leeren Zellen ohne ein Partikel keine Aktualisierung stattfindet, besitzen sie immer noch den höchst möglichen Wert für den Partikelstartindex und null für den Partikelendindex.
- **Herstellen der Monotonie der Partikelstartindexe:** Nun wird das Gitter noch so transformiert, dass die Partikelstartindexe wie beim Counting-Sort monoton ansteigend sind, da dies bei der Traversierung des Gitters extrem vorteilhaft ist. Hierfür muss im wesentlichen eine Prefix-Sum berechnet werden.
- **Permutation der Partikeldaten:** Zu Letzt werden die Partikeldaten analog wie beim Counting-Sort permutiert.

Ein Großteil dieser Laufzeit wird allerdings durch den Radix-Sort selbst verursacht. Dabei verwendet diese Arbeit die Implementierung von Radix-Sort aus der Thrust-Bibliothek. Die Implementierung scheint jedoch schlecht optimiert zu sein, da sie intern für sich mehrmals temporären globalen Speicher per cudamalloc alloziert und mehrmals teure GPU-CPU-Synchronisationsbefehle verwendet. Aus den so eben genannten Gründen dauert die Konstruktion durch Radix-Sort mit 11 ms deutlich länger als die Konstruktion durch Counting-Sort mit 5.5 ms. Jedoch benötigt das Radix-Sort-Verfahren keine atomaren Operationen auf den globalen Speicher. Dadurch ist es eventuell performanter auf GPUs, bei welchen atomare Operationen auf den globalen Speicher teurer sind. Zudem gibt es ebenfalls ältere GPUs, die atomare Operationen auf den globalen Speicher nicht unterstützen. In diesem Fall ist der Radix-Sort die einzige Möglichkeit das Gitter auf der GPU zu bauen.

Nachtrag: Wenige Tage vor der Fertigstellung dieser Arbeit veröffentlichte NVIDIA die Version 7.0 des CUDA-Toolkits. In dieser Version hat es laut eigenen Angaben die Performance von Thrust-Radix-Sort um bis zu 50% für primitive Datentypen erhöht. Diesbezüglich wären weitere Untersuchungen sinnvoll, welche wegen der späten Veröffentlichung nicht mehr durchgeführt werden konnten.

4.9 Berechnungen des Fixed-Radius-Near-Neighbors-Problems

4.9.1 First-Approach

Als Nächstes soll vorgestellt werden, wie die Berechnungen des Fixed-Radius-Near-Neighbors-Problems implementiert und anschließend verbessert werden. Dafür werden nacheinander verschiedene Versionen vorgestellt. Die Versionen verwenden hierbei weitestgehend den Linked-Cell-Ansatz, welcher die Fixed-Radius-Near-Neighbors-Berechnungen bereits während der Gittertraversierung ausführt. Als Beispielprobleme werden die Fixed-Radius-Near-Neighbors-Probleme desjenigen SPH-Verfahrens aus Punkt 4.3 verwendet, bei welchem die Dichte über die Zeit integriert wird. So wird hier die Berechnung des Shepard-Filters, also die Berechnung des 1. Fixed-Radius-Near-Neighbors-Problems dieses Verfahrens, untersucht. Ebenfalls untersucht diese Arbeit die Berechnung der zeitlichen Änderungen mit Bestimmung der maximalen Zeitschrittgröße zu Beginn des Zeitschritts, welche das 2. Fixed-Radius-Near-Neighbors-Problems dieses Verfahrens ist. Die Berechnung des Shepard-Filters trägt dabei allerdings nur wenig zur Gesamtrechenzeit bei, während das Berechnen der zeitlichen Änderungen insgesamt einen Großteil der Laufzeit ausmacht. Allerdings ist eine Untersuchung des Shepard-Filters exemplarisch als Beispielproblem dennoch interessant, da es sich hierbei um ein Fixed-Radius-Near-Neighbors-Problem handelt, welches nur wenige Recheninstruktionen pro Nachbarpartikel enthält. Dies steht im Gegensatz zur Berechnung der zeitlichen Änderungen, die viele Recheninstruktionen pro Nachbarpartikel enthält. Folgender Punkt soll zunächst denjenigen Ansatz für die Berechnung der Fixed-Radius-Near-Neighbors-Probleme, der als erstes für die Arbeit implementiert wurde, vorstellen. Der Ansatz wird im Folgenden als First-Approach-Implementierung bezeichnet.

Generell lässt sich die Berechnung des Problems leicht per Template abstrahieren, welches dem Visitor-Pattern ähnelt. Vorteilhaft von solchen Templates ist neben der enormen Code-Ersparnis, dass der Compiler sie expandieren und dadurch stark optimieren kann. Eine solche Abstraktion für das Kernel der First-Approach-Implementierung ist in Abbildung 4.10 zu sehen. So wird in der First-Approach-Implementierung immer ein Template-Kernel mit einem

```
template<class ProblemStruct>
__global__ void FirstApproachFRNNKernel(ParticleGrid Grid, ParticleData Particles)
{
  int CentralPartID = blockIdx.x * blockDim.x + threadIdx.x;
  ProblemStruct PS;
  PS.Init(Particles, CentralPartID);
  int3 CellIDCenter = Grid.CalculateCellID(PS.CentralPosition);

  for(int z = -1; z <= 1; z++)
  for(int y = -1; y <= 1; y++)
  for(int x = -1; x <= 1; x++)
  {
    int3 CellIDTrav = CellIDCenter + make_int3(x, y, z);
    uint CellIndex = Grid.GetCellIndex(CellIDTrav);
    uint CellBegin = Grid.ParticleStart[CellIndex];
    uint CellEnd = Grid.ParticleStart[CellIndex + 1];
    for(uint i = CellBegin; i < CellEnd; i++)
        PS.Evaluate(Particles, i);
  }
  PS.FinalizeAndWriteBack(Particles, CentralPartID);
}
```

Abbildung 4.10: First-Approach-Kernel der Berechnung des Fixed-Radius-Near-Neighbors-Problems

Thread pro Partikel auf der GPU gestartet. Das Partikel dient hierbei als Zentralpartikel, wodurch der Thread über alle benachbarten Partikel iterieren und die entsprechenden problemspezifischen Berechnungen ausführen muss. Als Template-Argument nimmt dieses Kernel eine problemspezifische Structure entgegen. Diese Structure speichert dabei Zwischenergebnisse ab und gibt diejenigen Berechnung vor, welche gemäß des spezifischen Fixed-Radius-Near-Neighbors-Problems ausgeführt werden müssen. Auf diese Weise werden ebenfalls die Traversierung der Datenstruktur und die eigentlichen problemspezifischen Berechnungen voneinander abgekapselt. In Abbildung 4.11 ist eine solche Structure beispielhaft für den Shepard-Filter aus Gleichung 3.31 zu sehen. Anhand ihr soll das Kernel des First-Approach-Verfahrens in Abbildung 4.10 erläutert werden.

In dem First-Approach-Verfahren initialisiert ein jeder Thread in einem ersten Schritt die problemspezifische Structure. Dabei lädt der Thread sämtliche benötigten Daten des Zentralpartikels des Threads aus dem globalen Speicher und initialisiert weitere Werte. So setzt er zum Beispiel beim Shepard-Filter den Nenner $\sum_b \frac{m_b}{\rho_b} W_{ab}$ und Zähler $\sum_b m_b W_{ab}$ aus Gleichung 3.31 auf Null.

Nun bestimmt der Thread ausgehend von der geladenen Position des Partikels, in welcher Gitterzelle das Partikel sich befindet. Um alle Nachbarpartikel innerhalb der Cutoff-Distance zu finden, werden als Nächstes die Gitterzelle selbst und all ihre direkten Nachbarn traversiert. Dadurch ergeben sich 27 traversierte Gitterzellen. Damit das Kernel mit dieser Traversierung alle Partikel innerhalb der Cutoff-Distance effizient findet, muss die Kantenlänge einer Gitterzelle in diesem Fall gleich der Cutoff-Distance sein. Diese Traversierung wird in Abbildung 4.12 gezeigt. Dadurch ergeben sich drei ineinander verschachtelte For-Schleifen. Für jede dieser

```
struct ShepardProblemStruct
{
    float3 CentralPosition;
    float CentralShepardNominator;
    float CentralShepardDenominator;

    void Init(ParticleData &Particles, uint CentralPartID)
    {
     CentralPosition = make_float3(Particles.Positions[CentralPartID]);
     CentralShepardNominator = 0.f;
     CentralShepardDenominator = 0.f;
    }

   void Evaluate(ParticleData &Particles, uint NeighborID)
   {
     float3 NeighborPosition = make_float3(Particles.Positions[NeighborID]);
     float PartDistance = distance(CentralPosition, NeighborPosition);
     if(PartDistance >= CutOffDistance)
         return;

     float NeighborDensity = Particles.Denseties[NeighborID];
     float KernelValue = WendlandKernel(PartDistance);
     CentralShepardNominator += KernelValue;
     CentralShepardDenominator += 1.f / NeighborDensity * KernelValue;
   }

   void FinalizeAndWriteBack(ParticleData &Particles, uint CentralPartID)
   {
     CentralShepardDenominator *= ParticleMass;
     float CentralResultDensity = CentralShepardNominator/CentralShepardDenominator;
     Particles.DensitiesCorrected[CentralPartID] = CentralResultDensity;
   }
};
```

Abbildung 4.11: Problemspezifische Structure für die Berechnung des Shepard-Filters

Gitterzellen lädt der Thread nun den Partikelstartindex der Gitterzelle selbst und den Partikelstartindex der nächsten Gitterzelle. Dadurch kann er die Menge der Partikel in der Gitterzelle bestimmen.

Anschließend iteriert der Thread in einer weiteren verschachtelten inneren For-Schleife über die Partikel in der Gitterzelle selbst. Für jedes Nachbarpartikel wird gemäß des Linked-Cell-Ansatzes eine problemspezifische Auswertungsfunktion aufgerufen. Dabei lädt die Auswertungsfunktion zunächst die Position des Nachbarpartikels. Anschließend überprüft sie ob sich das Nachbarpartikel innerhalb der Cutoff-Distance vom Zentralpartikels befindet. Wenn nicht so kann hier bereits mit den Berechnungen für das Nachbarpartikel abgebrochen und die Auswertungsfunktion verlassen werden. Andernfalls werden die eigentlichen Berechnungen ausgeführt. So wird im Falle des Shepard-Filters die Dichte des Nachbarpartikels ρ_b geladen und ausgehend von dem Abstand der beiden Partikel der Wert des Wendlandkernels W_{ab} berechnet. Schließlich inkrementiert die Funktion den Zähler um W_{ab} und den Nenner um $\frac{1}{\rho_b} W_{ab}$.

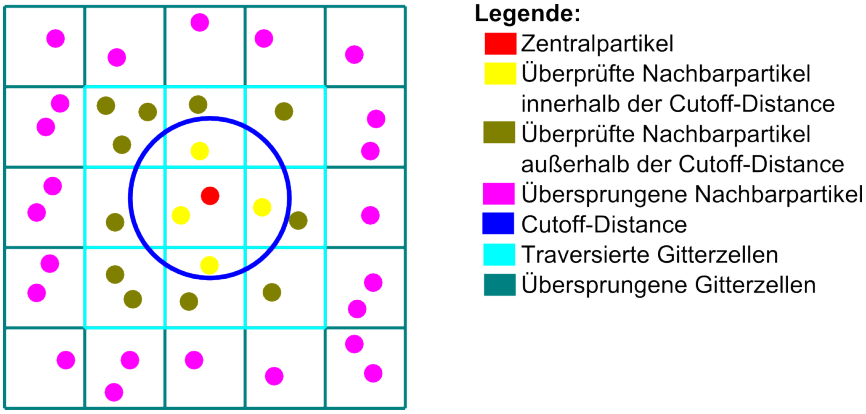


Abbildung 4.12: Übersicht über die Traversierung des Gitters

Nachdem der Thread alle Gitterzellen und damit alle Nachbarpartikel traversiert hat, ruft er eine weitere Funktion der problemspezifischen Structure auf. Diese ist dafür verantwortlich finale Berechnungen auszuführen. Im Falle des Shepard-Filters wird der Nenner noch mit der Partikelmasse m multipliziert und der Quotient zwischen Zähler und Nenner gebildet. Der Quotient entsprecht der normierten Dichte und ist das Endergebnis des Filters. Anschließend muss diese Funktion die Endergebnisse des Problems, wie beim Shepard-Filter die soeben berechnete normierte Dichte, zurückschreiben.

Die Berechnungen der zeitlichen Änderungen mit Bestimmung der maximalen Zeitschrittweite lassen sich analog per problemspezifischer Structure berechnen. Die Structure hierfür ist in Abbildung 4.13 gezeigt. Dabei führt die Auswertungsfunktion die Berechnungen der Dichteänderung gemäß Gleichung 3.28 und der Beschleunigung gemäß Gleichung 3.29 durch. Ebenfalls berechnet diese Structure die maximale Zeitschrittweite aus Gleichung 3.35. Hierfür sucht die Auswertungsfunktion bereits das maximale $|\mu_{ab}|$, welches bei der Berechnung der Viskosität entsteht. Schließlich wird bei den finalen Berechnungen noch die Beschleunigung durch die Randbedingungen ermittelt und zur Beschleunigung des Zentralpartikels hinzuaddiert. Aus der Beschleunigung und dem maximalen μ_{ab} lässt sich nun der maximale Zeitschritt für das Zentralpartikel bestimmen. Damit die SPH-Simulation nicht noch zusätzlich eine parallele Reduktion über die maximalen Zeitschritte aller Partikel benötigt, wird in diesem Kernel bereits der maximale Zeitschritt über alle Partikel durch atomare Operationen ermittelt. Dafür wird der maximale Zeitschritt eines jeden Partikels zunächst per Warp-Shuffle-Funktionen auf den maximalen Zeitschritt des Warps reduziert. Denn dadurch muss das Kernel nur noch eine atomare Operation pro Warp durchführen, wodurch wiederum etwas Last von den atomaren Units der GPU genommen wird. Allerdings kann die Structure diesen so ermittelten maximalen Zeitschritt nicht direkt in den globalen Speicher der GPU zurückschreiben, da die Atomic-Units keine Min-Operation für Gleitkommazahlen besitzen. Hier verwendet die Arbeit den Trick aus [MH]. Der Trick transformiert einen Float per bijektiver Abbildung in einen Uint. Die Transformation ist so definiert, dass der Float-Vergleich auf zwei Floats immer das gleiche Ergebnis wie der Uint-Vergleich auf die beiden zu Uints transformierten Floats liefert. Deshalb

```
template <bool UpdateDT>
struct ChangesProblemStruct
{
    float3 CentralPosition; float3 CentralVelocity; float CentralDensity;
    float CentralPDivRho2; float3 CentralAcceleration; float CentralDensityChange; float CentralMuMax;

    void Init(ParticleData &Particles, uint CentralPartID)
    {
     CentralPosition = make_float3(Particles.Positions[CentralPartID]);
     CentralVelocity = make_float3(Particles.Velocities[CentralPartID]);
     CentralDensity = Particles.Densities[CentralPartID];
     CentralPDivRho2 = TaitEquation(CentralDensity) / (CentralDensity * CentralDensity);
     CentralMuMax = -FloatMax;
     CentralAcceleration = make_float3(0.f);
     CentralDensityChange = 0.f;
    }

   void Evaluate(ParticleData &Particles, uint NeighborID)
   {
     float3 NeighborPosition = make_float3(Particles.Positions[NeighborID]);
     float PartDistance = distance(CentralPosition, NeighborPosition);
     if(PartDistance >= CutOffDistance)
       return;

     float3 NeighborVelocity = make_float3(Particles.Velocities[NeighborID]);
     float NeighborDensity = make_float3(Particles.Densities[NeighborID]);
     float3 DistanceVec = CentralPosition - NeighborPosition;
     float3 VelocityDif = CentralVelocity - NeighborVelocity;

     float3 KernelGradientValue = GradientWendLandKernel(DistanceVec);
     float Mu = EvaluateMu(DistanceVec, VelocityDif);
     float ViscosityTerm = EvaluateViscosityTerm(Mu, CentralDensity, NeighborDensity, DistanceVec, VelocityDif);
     float NeighborPDivRho2 = TaitEquation(NeighborDensity) / (NeighborDensity * NeighborDensity);
     CentralAcceleration -= (ViscosityTerm + NeighborPDivRho2 + CentralPDivRho2) * KernelGradientValue;
     CentralMuMax = fmaxf(CentralMuMax, abs(Mu));
     CentralDensityChange += VelocityDif * KernelGradientValue;
   }

   void FinalizeAndWriteBack(ParticleData &Particles, uint CentralPartID)
   {
     CentralAcceleration *= ParticleMass;
     CentralAcceleration += EvaluateBoundaryAcceleration(CentralPosition,CentralVelocity) + Gravitation;
     CentralDensityChange *= ParticleMass;
     if(UpdateDT)
       EvaluateAndUpdateDT(CentralAcceleration, CentralMuMax);

     Particles.Changes[CentralPartID] = make_float4(CentralAcceleration, CentralDensityChange);
   }
};
```

Abbildung 4.13: Problemspezifische Structure für die Berechnung der zeitlichen Änderungen

wendet die Structure zuerst diese Transformation auf den maximalen Zeitschritt an. Anschließend wird der maximale Zeitschritt als Uint per atomarer Operation zurückgeschrieben.

Schließlich muss das SPH-Verfahren auch die zeitlichen Änderungen zum Halbzeitschritt ohne die Bestimmung des maximalen Zeitschritts als 3. Fixed-Radius-Near-Neighbors-Problem berechnen. Diese Berechnungen lassen sich leicht dadurch definieren, indem die Bestimmung des maximalen Zeitschritts über ein Template-Argument der Structure aus der Abbildung 4.13 deaktiviert wird. Dadurch sollte das Kernel ein wenig schneller werden. In der Wirklichkeit ist es jedoch meist etwas langsamer. Die Ursachen hierfür sind wahrscheinlich, wie im Punkt 4.9.12 diskutiert wird, unterschiedliche Optimierungen durch den Compiler. Da beide Kernel allerdings von der Komplexität gesehen extrem ähnlich sind, wird die Berechnung der zeitlichen Änderungen ohne die Bestimmung des maximalen Zeitschritts im Folgenden bei den einzelnen Optimierungen nicht besonders untersucht. Lediglich bei den abschließenden Untersuchungen im Punkt 4.9.12 wird dieses Kernel näher betrachtet.

Abschließend soll der First-Approach kurz untersucht werden, um Verbesserungsansätze zu finden. Hierfür wurde er für den Shepard-Filter und für die Berechnung der zeitlichen Änderungen mit der Bestimmung des maximalen Zeitschritts in Abbildung 4.14 gebenchmarkt. Die Messungen zeigen, dass die Berechnung der zeitlichen Änderungen mit 217 ms deutlich länger dauert als die Berechnung des Shepard-Filters mit 94 ms. Dies ist darauf zurückzuführen, dass die Berechnungen der Auswertungsfunktion beim Shepard-Filter deutlich weniger komplex sind als bei den zeitlichen Änderungen (siehe Quelltext in den Abbildung 4.13 und 4.11). Die Warp-Ausführungseffizienz ist mit 42% bei der Berechnung der zeitlichen Änderungen im Vergleich zum Shepard-Filter mit 95% ebenfalls relativ niedrig. Dies ist auf die Warpdivergenz innerhalb der im Vergleich zum Shepard-Filter deutlich komplexeren Auswertungsfunktion zurückzuführen. Dadurch geht bei der Berechnung der zeitlichen Änderungen viel Rechenleistung verloren. Deshalb ist es sinnvoll zu versuchen die Datenparallelität des Programms zu erhöhen. Die L2-Cache-Hitrate ist mit 99% und 97% ebenfalls bereits sehr gut, weshalb der Workingset wahrscheinlich gut in den L2-Cache passt. Aus diesem Grund ist die Auslastung der DRAM-Bandbreite mit 0.9% und 1.1% sehr niedrig. Die Auslastung der L2-Cache-Bandbreite ist ebenfalls niedrig. Daraus kann insgesamt gefolgert werden, dass die niedrige Auslastung der CUDA-Cores sehr wahrscheinlich durch Latenzen verursacht wird. Somit geht primär durch die Warp-Ausführungseffizienz und durch die Latenzen Performance verloren. Deshalb sollte es im Folgenden neben allgemeinen algorithmischen Verbesserungen das Ziel sein die Warp-Ausführungseffizienz zu verbessern und die Latenzen besser zu verbergen. Diese Ansätze dienen im Folgenden dazu das First-Approach-Kernel weiter zu verbessern.

4.9.2 Herausoptimierung der X-Schleife der Gittertraversierung

In diesem Punkt soll die Herausoptimierung der X-Schleife der Gittertraversierung aus dem Kernel vorgestellt werden. Da das Gitter ein nach $\mathrm{Dim}_x\mathrm{Dim}_y Z + \mathrm{Dim}_x Y + X$ linearisiertes dreidimensionales Array ist, liegen die Gitterzellen entlang der X-Achse sequentiell im Speicher. Des Weiteren sind die Partikel nach Gitterzellen sortiert. Deshalb liegen die Partikel einer Gruppe von Gitterzellen, welche entlang der X-Achse aufeinanderfolgenden, ebenfalls sequentiell im Speicher. Eine solche Gruppe von Gitterzellen wird im Folgenden als X-Gruppe bezeichnet. Da die Partikelstartindexe der Gitterzellen ebenfalls monoton ansteigend sind, kann die SPH-Simulation die Menge derjenigen Partikel einer X-Gruppe in zwei Speicherzugriffen auf das Gitter bestimmen. Das so modifizierte Kernel ist in Abbildung 4.15 zu sehen. Durch die Entfernung der X-Schleife lassen sich folgende Vorteile erzielen:

	First-Approach-Shepard-Kernel	First-Approach-Änderungen-Kernel
Laufzeit	94 ms	217 ms
Warp-Ausführungseffizienz	94%	42%
DRAM-Auslastung	0.9%	1.1%
L2-Auslastung	Low(2)	Low(2)
L2-Hitrate durch L1-Reads	99%	97%
Auslastung der CUDA-Cores	41%	45%

Abbildung 4.14: Benchmark für den First-Approach

- **Reduktion des Overheads der Gittertraversierung:** Das Kernel muss statt mehrere Gitterzellen entlang der X-Achse zu traversieren nur noch die Partikelstartindexe zweier Gitterzellen aus dem Speicher laden. Deshalb besitzt diese Art der Traversierung einen geringeren Overhead.
- **Reduktion der Warpdivergenz:** Die Anzahl der Partikel innerhalb einer Gitterzelle schwankt, selbst wenn die Gitterzelle vollkommen von der Flüssigkeit aufgefüllt ist, stark. Deshalb besitzt sie eine große Standardabweichung im Vergleich zu ihrem Erwartungswert. Das Verhältnis von beiden wird in der Mathematik auch als Variationskoeffizient bezeichnet. Durch den großen Variationskoeffizienten ergibt sich ebenfalls eine hohe Warpdivergenz in der innersten For-Schleife, sofern die SPH-Simulation nur über die Partikel einer Gitterzelle iteriert. Fasst die SPH-Simulation die Iteration über die Partikel von n Gitterzellen in einer einzigen For-Schleife zusammen, so erhöht sich der Erwartungswert um den Faktor n, die Standardabweichung und damit der Variationskoeffizient jedoch nur um $\sqrt{n}$. Auf diese Weise kann die Simulation die Warpdivergenz in dieser For-Schleife etwas reduzieren.

Das so modifizierte Kernel wurde in Abbildung 4.16 gebenchmarkt. Das Benchmark zeigt, dass sich durch die Optimierung die Warp-Ausführungseffizienz beim Kernel der zeitlichen Änderungen deutlich von 44% auf 49% und die Performance von 217 ms auf 165 ms erhöhen. Bei dem Shepard-Kernel fällt der Performancegewinn von 94 ms auf 91 ms deutlich geringer aus. Dies ist wahrscheinlich darauf zurückzuführen, dass es bereits ohne diese Optimierung wegen der einfacheren Auswertungsfunktion eine wesentlich höhere Warp-Ausführungseffizienz von 94% besaß, welche sich durch die Optimierung nur noch kaum auf 97% erhöht. Somit stellt das Herausoptimieren der X-Schleife eine einfache Möglichkeit dar, um unnötige Befehle einzusparen und die Warp-Ausführungseffizienz zu verbessern. Das bewirkt wiederum eine Performanceerhöhung. Deshalb wird diese Optimierung im Folgenden weiter verwendet.

4.9.3 Änderung der Gitterzellengröße

In diesem Punkt soll darauf eingegangen werden, wie die SPH-Simulation ihre Performance durch die Änderung der Gitterzellen-Größe optimieren kann. Denn bislang war die Gitterzellengröße gleich der Cutoff-Distance, wodurch die SPH-Simulation stets die Nachbarpartikel aus 27 Gitterzellen überprüfen musste. Dieser Wert kann geändert werden, indem die SPH-Simulation die Gitterzellengröße und damit die Traversierungsdistanz n_{Trav} ändert. Setzt die

```
template<class ProblemStruct>
__global__ void NoXLoopFRNNKernel(ParticleGrid Grid, ParticleData Particles)
{
  uint CentralPartID = blockIdx.x * blockDim.x + threadIdx.x;
  ProblemStruct PS;
  PS.Init(Particles, CentralPartID);
  int3 CellIDCenter = Grid.CalculateCellID(PS.CentralPosition);

  for(int z = -1; z <= 1; z++)
  for(int y = -1; y <= 1; y++)
  {
    int3 CellIDTravXMin = CellIDCenter + make_int3(-1,y,z);
    uint CellIndexXMin = Grid.GetCellIndex(CellIDTravXMin);
    uint PartBegin = Grid.ParticleStart[CellIndexXMin];
    uint PartEnd = Grid.ParticleStart[CellIndexXMin+3];
    for(uint i = PartBegin; i < PartEnd; i++)
      PS.Evaluate(Particles, i);
  }
  PS.FinalizeAndWriteBack(Particles,CentralPartID);
}
```

Abbildung 4.15: Fixed-Radius-Near-Neighbors-Kernel mit herausoptimierter X-Schleife

	First-Approach-Shepard-Kernel	Shepard-Kernel ohne X-Schleife	First-Approach-Änderungen-Kernel	Änderungen-Kernel ohne X-Schleife
Laufzeit	94 ms	91 ms	217 ms	165 ms
Warp-Ausführungseffizienz	94%	97%	44%	49%

Abbildung 4.16: Benchmark für das Kernel mit herausoptimierter X-Schleife

SPH-Simulation die Gitterzellengröße auf $\frac{1}{n_{\text{Trav}}}$tel der Cutoff-Distance, so erhöht sich die Zahl der für ein Zentralpartikel zu betrachtenden Gitterzellen auf $(2n_{\text{Trav}}+1)^3$. Dementsprechend müssen auch $(2n_{\text{Trav}}+1)^2$ X-Gruppen von Gitterzellen traversiert werden. Beispielhaft ergeben sich bei einer Traversierungsdistanz von 1 und 2 insgesamt 27 beziehungsweise 125 untersuchte Gitterzellen und 9 beziehungsweise 25 traversierte X-Gruppen. Die so entstehende Traversierung ist für eine Traversierungsdistanz von 1 und 2 in der Abbildung 4.17 gezeigt. Das auf diese Weise modifizierte Kernel ist in Abbildung 4.18 zu sehen.

Durch die Verkleinerung der Gitterzellengröße entstehen folgende Vorteile:

- **Weniger Overhead in der Auswertungsfunktion:** Je feiner das Gitter aufgelöst wird, desto mehr nähert sich der Würfel der traversierten Gitterzellen einer optimalen Bounding-Box um diejenige Kugel, welche durch die Position des Zentralpartikels und die Cutoff-Distance beschrieben wird, an. Durch diese Optimalität müssen, wie in Ab-

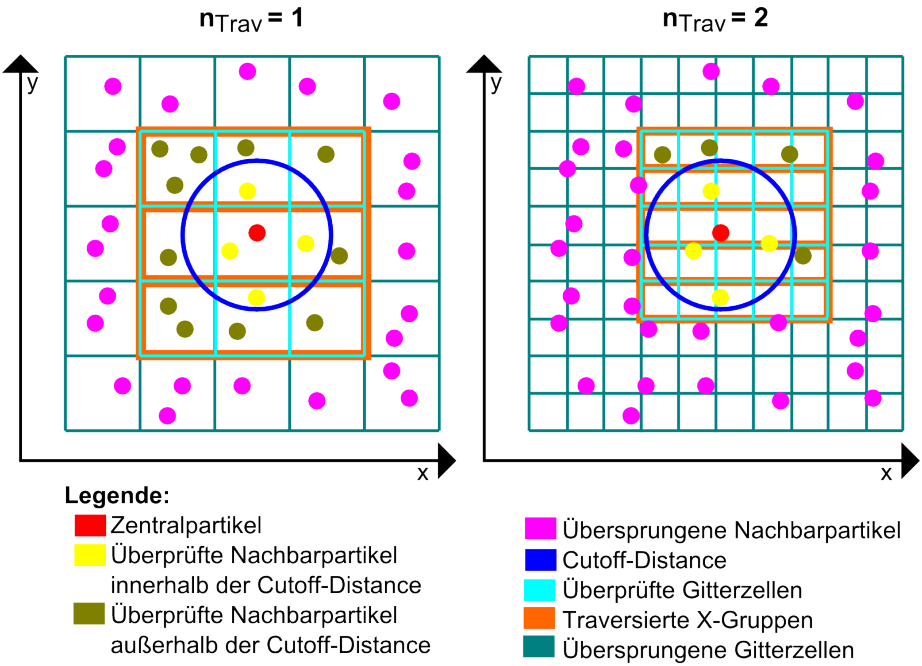


Abbildung 4.17: Traversierung des Gitters bei verkleinerter Gitterzellengröße

bildung 4.17 gezeigt, weniger Nachbarpartikel, die sich außerhalb der Cutoff-Distance befinden, überprüft werden.

- **Weniger Warpdivergenz in der Auswertungsfunktion:** Durch die feinere Auflösung befindet sich ein höherer Anteil der überprüften Partikel innerhalb der Cutoff-Distance. Deshalb muss seltener mit der Auswertungsfunktion abgebrochen werden, wodurch die Optimierung die Warpdivergenz erhöht.

Allerdings sind kleinere Gitterzellen auch nachteilig:

- **Größerer Overhead bei der Gittertraversierung:** Es müssen mehr X-Gruppen von Gitterzellen traversiert werden. Dadurch wächst der Aufwand der Traversierung mit $(2n_{\text{Trav}} + 1)^2$ an. Hier zeigt sich ebenfalls der Vorteil der Herausoptimierung der X-Schleife. Ohne die Optimierung müsste die SPH-Simulation jede einzelne Gitterzelle traversieren, wodurch der Aufwand mit $(2n_{\text{Trav}} + 1)^3$ anwachsen würde.
- **Mehr Kosten beim Bauen des Gitters:** Das Bauen des Gitters wird teurer, da die Laufzeit der Prefix-Sum von der Anzahl der Gitterzellen abhängig ist. Zudem treten bei der

```
template<class ProblemStruct>
__global__ void VariableCellSizeFRNNKernel(ParticleGrid Grid, ParticleData Particles)
{
  uint CentralPartID = blockIdx.x * blockDim.x + threadIdx.x;
  ProblemStruct PS;
  PS.Init(Particles, CentralPartID);
  int3 CellIDCenter = Grid.CalculateCellID(PS.CentralPosition);

  for(int z = -GridTravDist; z <= GridTravDist; z++)
  for(int y = -GridTravDist; y <= GridTravDist; y++)
  {
    int3 CellIDTravXMin = CellIDCenter + make_int3(-GridTravDist,y,z);
    uint CellIndexXMin = Grid.GetCellIndex(CellIDTravXMin);
    uint PartBegin = Grid.ParticleStart[CellIndexXMin];
    uint PartEnd = Grid.ParticleStart[CellIndexXMin+2*GridTravDist+1];
    for(uint i = PartBegin; i < PartEnd; i++)
      PS.Evaluate(Particles, i);
  }
  PS.FinalizeAndWriteBack(Particles,CentralPartID);
}
```

Abbildung 4.18: Fixed-Radius-Near-Neighbors-Kernel mit verkleinerter Gitterzellengröße

Konstruktion des Gitters chaotischere Speicherzugriffe auf, welche schlecht für das Caching sind.

- **Mehr Speicherplatzverbrauch durch das Gitter:** Der Speicherplatzverbrauch des Gitters wächst mit n_{Trav}^3 an. Dadurch führt bereits eine Erhöhung der Traversierungsdistanz von eins auf zwei zu einer Verachtfachung des Speicherplatzverbrauchs.
- **Mehr Warpdivergenz durch weniger Partikel in den Gitterzellen:** Hier ist der gegenteilige Effekt aus dem letzten Punkt zu beobachten. Das Volumen einer X-Gruppe wird verkleinert. Dadurch erhöht sich jedoch wiederum der Variationskoeffizient der Partikel in der X-Gruppe, der zu einer höheren Warpdivergenz führt.

In Abbildung 4.19 wurde die Performance für verschiedene Werte der Traversierungsdistanz n_{Trav} gemessen. Bei den Messungen zeigt sich, dass beim Shepard-Kernel ein Wert von 2 für die Traversierungsdistanz mit 72 ms Laufzeit optimal ist. Des Weiteren nimmt beim Shepard-Kernel die Warp-Ausführungseffizienz mit zunehmender Gitterzellengröße ab. Die Ursache hierfür ist wahrscheinlich, dass die Zahl der Partikel in einer X-Gruppe mit zunehmender Traversierungsdistanz stärker schwankt.

Beim Kernel der zeitlichen Änderungen ist ein Wert von 3 für die Traversierungsdistanz mit 115 ms Laufzeit optimal. Interessant ist, dass die Warp-Ausführungseffizienz zuerst mit größeren Werten für die Traversierungsdistanz zunimmt und anschließend wieder abnimmt. Dies ist darauf zurückzuführen, dass zuerst weniger Partikel außerhalb der Cutoff-Distance überprüft werden müssen, wodurch die Warpdivergenz in der Auswertungsfunktion abnimmt. Dieser Effekt ist im Shepard-Kernel wegen der einfacheren Auswertungsfunktion nicht zu beobachten.

n_{Trav}	1	2	3	4
Überprüfte Gitterzellen	27	125	343	729
Traversierte X-Gruppen	9	25	49	81
Shepard-Kernel				
Laufzeit	91 ms	72 ms	76 ms	88 ms
Warp-Ausführungseffizienz	97%	88%	80%	70%
Änderungen-Kernel				
Laufzeit	165 ms	121 ms	115 ms	130 ms
Warp-Ausführungseffizienz	49%	53%	53%	48.7%
Gitter				
Laufzeit beim Bauen	3.6 ms	3.9 ms	5.4 ms	8.8 ms
Speicherplatzverbrauch	4 MB	32 MB	106 MB	252 MB
Anteil am gesamten Speicherplatzverbrauch	0.7%	6%	17%	32%

Abbildung 4.19: Benchmark für die Kernel bei Änderung der Gitterzellengröße

Anschließend nimmt die Warp-Ausführungseffizienz jedoch wieder ab, da die Zahl der Partikel in einer X-Gruppe wieder stärker schwankt, wodurch wiederum mehr Warpdivergenz entsteht.

Die Laufzeit beim Bauen des Gitters nimmt zuerst bei einer Erhöhung der Traversierungsdistanz von 1 auf 2 und 3 nur kaum von 3.6 ms auf 3.9 ms beziehungsweise 5.4 ms zu. Erst bei einer Traversierungsdistanz von 4 steigt sie stark auf 8.8 ms an. Dies ist darauf zurückzuführen, dass bei einem Großteil der Schritte des Bauens die Laufzeit nicht oder nur kaum von der Gittergröße abhängt. Beim Speicherplatzverbrauch des Gitters gilt, dass es in Relation zum Speicherplatzverbrauch der Partikeldaten von 530 MB beziehungsweise zur gesamten Simulation zu betrachten ist. So macht der Speicherplatzverbrauch des Gitters bei einer Traversierungsdistanz von 1 und 2 mit 4 MB und 32 MB nur einen Anteil von 0.7% und 6% des gesamten Speicherplatzes der Simulation aus. Erst bei einer Traversierungsdistanz von 3 nimmt er mit 106 MB einen größeren Anteil von 17% ein. Bei einer Traversierungsdistanz von 4 kostet er mit 252 MB und 32% einen großen Anteil.

Somit sind die Werte von 2 und 3 für die Traversierungsdistanz bei den Berechnungen der Fixed-Radius-Near-Neighbors-Probleme in etwa optimal, während die Werte von 1 und 4 deutlich schlechter sind. Aus dem Grund lässt sich die Performance deutlich erhöhen, wenn die SPH-Simulation die Gitterzellengröße von der Cutoff-Distance auf die Hälfte oder ein Drittel der Cutoff-Distance ändert. Werden alle Laufzeiten aufsummiert, so ist eine Traversierungsdistanz von 3 um 9 ms pro Zeitschritt schneller als ein Wert von 2. Deshalb wird im Folgenden weiter eine Traversierungsdistanz von 3 verwendet. Wenn allerdings sich bei einer SPH-Simulation wenige Partikel über ein großes Volumen verstreuen oder der Speicherplatzverbrauch der Simulation allgemein ein Problem darstellt, dann kann eine Traversierungsdistanz von 2 ebenfalls gute Ergebnisse bei einem deutlich geringeren Speicherplatzverbrauch für das Gitter liefern.

```
template<class ProblemStruct, int MAX_THREADS_PER_BLOCK, int MIN_BLOCKS_PER_MP>
__launch_bounds__(MAX_THREADS_PER_BLOCK, MIN_BLOCKS_PER_MP)
__global__ void AutoOccupancyFRNNKernel(ParticleGrid Grid, ParticleData Particles)
{
  /* ..... */
}
```

Abbildung 4.20: Erweiterung des Fixed-Radius-Near-Neighbors-Kernels durch eine wahlfreie Occupancy

4.9.4 Autotuning der Occupancy

Als Nächstes soll auf die Idee des Autotunings der Occupancy bei der Lösung der Fixed-Radius-Near-Neighbors-Probleme eingegangen werden. So zeigte es sich bei der Diskussion der First-Approach-Kernels, dass die Berechnungen des Fixed-Radius-Near-Neighbors-Problems wahrscheinlich durch Latenzen limitiert werden. In diesem Fall ist wie in dem Punkt 2.1.7 beschrieben eine Erhöhung der Occupancy durch die Begrenzung der Register oft für die Performance vorteilhaft. Jedoch kann das Begrenzen auch oft nachteilig sein. Zudem gibt es keine einfachen Regeln dafür, wie sich der optimale Wert der Occupancy ermitteln lässt. Denn dieser Wert ist sowohl von der Art des Fixed-Radius-Near-Neighbors-Problems als auch von der GPU abhängig. Um dennoch die optimale Occupancy für jedes Fixed-Radius-Near-Neighbors-Problem zu ermitteln, wählte die Arbeit einen einfachen Autotuning-Ansatz. Die Grundidee des Autotunings der Occupancy wird in der Literatur bei der GPGPU-Programmierung oft verwendet. Beispielhaft beschreibt [SGG] einen ähnlichen Autotuning-Ansatz für die Occupancy wie diese Masterarbeit. Allerdings versucht [SGG] durch das Autotuning nicht die Performance eines Fixed-Radius-Near-Neighbors-Problems sondern die Performance eines Belief-Propagation-Algorithmus, der zum Beispiel für die Bilderkennung benötigt wird, zu verbessern. Beim Autotuning der Occupancy geht die für diese Masterarbeit gewählte Implementierung wie folgt vor: Zunächst wird für jede der Occupancies eine Version des Fixed-Radius-Near-Neighbors-Kernels zur Compile-Zeit erzeugt. Anschließend werden die Versionen zur Laufzeit gebenchmarkt, um die beste Version auszuwählen. Dies soll nun näher erläutert werden.

So wird zunächst die Funktion des Kernels mit Launch-Bounds versehen. Um viele verschiedene Versionen des Kernels mit unterschiedlicher Occupancy zu erzeugen wird ausgenutzt, dass sich die Argumente der Launch-Bounds über ein Template, da dies ebenfalls eine Compile-Zeit-Konstante ist, setzen lassen. Dadurch ergibt sich für die Lösung des Fixed-Radius-Near-Neighbors-Problems das in Abbildung 4.20 gezeigte Kernel. Über das Template erzeugt nun die Implementierung für jeder der möglichen Occupancies der Test-GPU aus Punkt 2.1.7 eine Version der Kernels. Die Launch-Bounds begrenzen die Zahl der Register aber nur auf einen maximalen Wert. Besitzt das Kernel ohne Begrenzung bereits einen niedrigeren Registerverbrauch, so bleibt die Begrenzung ohne Auswirkung. Deshalb erzeugen viele der Begrenzungen identische Versionen vom Kernel. Wegen der automatischen Erstellung ist es jedoch schwer diese identischen Versionen zu vermeiden. Des Weiteren ist an der gewählten Vorgehensweise problematisch, dass GPUs aus unterschiedlichen Compute-Capabilities unterschiedliche Regeln für ihre Occupancy besitzen. Beim Template-Ansatz muss die Implementierung aber die Launch-Bounds entsprechend den Occupancy-Regeln der verwendeten GPUs zur Compile-Zeit

Occupancy	64	48	40
Shepard-Kernel (23 Register)			
Laufzeit	76 ms	-	-
Übertragenes DRAM-Speichervolumen	365 MB	-	-
Änderungen-Kernel (40 Register)			
Laufzeit	105 ms	115 ms	-
Übertragenes DRAM-Speichervolumen	995 MB	1001 MB	-

Abbildung 4.21: Benchmark für das Autotuning der Occupancy

festlegen. Dadurch erfordert das Hinzufügen neuer GPUs immer auch eine Änderung im Quelltext.

Zur Laufzeit benchmarkt nun die SPH-Simulation die verschiedenen Versionen eines jeden Fixed-Radius-Near-Neighbors-Kernels. Um die identischen Versionen herauszufiltern wird mit der CUDA Occupancy-API überprüft, ob die Version des Kernels die gewünschte Occupancy besitzt. Wenn nicht, so braucht die Simulation die Version nicht zu benchmarken, da die Registerbegrenzung auf den gegebenen Wert beim Kernel keine Auswirkungen hatte. Beim Benchmarken ist es des Weiteren problematisch, dass zu Beginn der Simulation die Partikel in einem regelmäßigen Gitter angeordnet sind. Durch die Simulation geraten die einzelnen Partikel allerdings zunehmend in Unordnung. Die Regelmäßigkeit und die Unregelmäßigkeit haben jedoch starke Auswirkungen auf den Code-Fluss innerhalb des Kernels, weshalb ein einmaliges Benchmarken zu Beginn die Ergebnisse des Benchmarks verfälschen könnte. Da es stark fallabhängig ist, wie viele Zeitschritte die Simulation bis zu einer unregelmäßigen Partikelanordnung benötigt, wird das Benchmark alle paar tausend Zeitschritte ausgeführt. Das erzeugt zwar einen zusätzlichen Overhead durch das Benchmarken. Der Overhead ist aber vernachlässigbar gering, weil die Simulation das Benchmark nur sehr selten ausführt.

Die Auswirkungen des Autotunings wurden im Benchmark, dessen Ergebnisse in Abbildung 4.21 eingetragen wurden, untersucht. Da das Shepard-Kernel nur 23 Register benötigt, besitzt es bereits ohne das Autotuning die maximale Occupancy von 64 Warps, wodurch das Autotuning dort keine Auswirkungen auf die Performance erzielt. Das Kernel der zeitlichen Änderungen benötigt jedoch ohne Autotuning 40 Register, wodurch es nur noch eine Occupancy von 48 Warps erzielt. Deshalb überprüft das Autotuning die Registerzahlen 32 und 40, welche die Occupancy von 48 beziehungsweise 64 Warps ermöglichen. Dabei zeigt es sich, dass bei der starken Beschränkung der Register von 40 auf 32 sich die Performance deutlich von 115 ms auf 105 ms erhöht, weshalb das Autotuning für die folgenden Berechnungen die schnellere Version verwendet. Der Performancegewinn durch das Begrenzen zeigt, dass die Performance des Kernels bei einer Occupancy von 48 noch stark durch Latenzen limitiert war. Interessanterweise nimmt das übertragene DRAM-Speichervolumen nicht zu, sondern es werden unabhängig von der Occupancy in etwa 1 GB an Daten mit dem DRAM ausgetauscht. Das zeigt wiederum, dass das durch die Limitierung wahrscheinlich entstandene Registerspilling noch in den L1-Cache der GPU passt.

So zeigt sich insgesamt, dass das Autotuning der Occupancy eine einfache Möglichkeit ist um die Performance deutlich zu erhöhen. Prinzipiell kann jedes beliebige Kernel diese Idee zur

Performancesteigerung verwenden, sofern es nicht bereits ohne Autotuning die maximale Occupancy erzielt. Diese Voraussetzung war in dieser Arbeit allerdings nur bei den Fixed-Radius-Near-Neighbors-Kerneln gegeben. Wegen der Performancesteigerung durch den Autotuning-Ansatz wird im folgenden immer diejenige Occupancy-Version der beiden Kernel weiter untersucht, welche die maximale Performance besitzt.

1. Nachtrag: NVIDIA baute ebenfalls in der neuen Version 7.0 des CUDA-Toolkits Funktionalitäten ein, mit welchen ein Host-Programm zu seiner Laufzeit Kernels aus Quelltextdateien kompilieren kann. Auf diese Weise kann ein Host-Programm unter anderem Defines von Kernels zu seiner Laufzeit ändern. Dadurch würde sich das Autotuning der Occupancy wahrscheinlich etwas eleganter gestalten lassen. Die späte Veröffentlichung dieses CUDA-Toolkits kurz vor der Fertigstellung dieser Arbeit machte eine Implementierung davon allerdings unmöglich.

4.9.5 Verwendung des Texture-Caches

Als Nächstes soll darauf eingegangen werden, wie die SPH-Simulation mit Hilfe des Texture-Caches die Performance bei den Fixed-Radius-Near-Neighbors-Problemen optimieren kann. So ist das bisherige Fixed-Radius-Near-Neighbors-Verfahren mit einer L2-Cache-Auslastung von Mid(6) beim Shepard-Kernel und beim zeitlichen Änderungen-Kernel mittelmäßig L2-Cache-Bandbreiten intensiv. Dadurch kann die Performance bereits durch die Auslastung der L2-Cache-Bandbreite etwas reduziert werden. Deshalb kann es für die SPH-Simulation vorteilhaft sein den Texture-Cache zu verwenden um damit den L2-Cache zu entlasten. Denn der Texture-Cache besitzt eine deutlich höhere Bandbreite. Des Weiteren ist die Auslastung der CUDA-Cores mit 33% beim Shepard-Kernel und 46% beim zeitlichen Änderungen-Kernel weiterhin gering, weshalb wahrscheinlich immer noch eine Limitierung durch Latenzen vorliegt. Diese werden vermutlich hauptsächlich in der Auswertungsfunktion verursacht. Denn hier lädt ein jeder Thread zuerst die Position des Nachbarpartikels aus dem globalen Speicher. Während er die Position lädt kann er keine weitere Berechnungen ausführen. Aus dem Grund ist er währenddessen untätig. Nachdem die Position geladen und die Überprüfung der Nachbarschaft abgeschlossen ist, muss der Thread beim Shepard-Kernel die Dichte und beim Änderungen-Kernel die Dichte und die Geschwindigkeit des Nachbarpartikels laden. Hier kann der Thread in der Zwischenzeit nur wenige Berechnungen durchführen. Da keine oder nur wenige Berechnungen während Speicherzugriffen durchgeführt werden können, führt dies nun wieder dazu, dass die GPU die Latenzen nur schlecht verbergen kann. Ein analoges Problem besteht bei der Gittertraversierung in der Y-Schleife. Aus diesem Grund kann es vorteilhaft sein den Texture-Cache durch die __ldg-Intrinisc zu verwenden, welcher geringere Latenzen als der L2-Cache besitzt.

Der Texture-Cache ist jedoch mit 48 kiByte sehr klein. Da kommt es dem Cache sehr entgegen, dass die Zentralpartikel der Threads eines Multiprozessors sich nahe beisammen befinden. Dadurch verursachen sie beim Iterieren über das Gitter und die sortierten Partikeldaten ebenfalls lokale Speicherzugriffe. Dennoch kann es vorkommen, dass die Threads eines Multiprozessors ein größeres Workingset als die Cache-Größe besitzen. Deshalb scheint es sinnvoll zu sein nicht alle Partikeldaten sondern nur einen Teil der Partikeldaten im Texture-Cache zwischenzuspeichern. Dadurch scheidet das Caching des Gitters selbst im Vorneherein aus. Denn die Daten des Gitters werden im Vergleich zu den Partikeldaten wegen der herausoptimierten X-Schleife und der Traversierungsdistanz von 3 verglichen mit dem Partikeldaten nur relativ selten angefordert. Des Weiteren ist es am Wichtigsten die Positionen der Nachbarpartikel zwischenzuspeichern, da sie am häufigsten abgefragt wird.

	Kein Texture-Caching	Texture-Caching: $\vec{r}$	Texture-Caching: $\vec{r}$, ρ	Texture-Caching: $\vec{r}$, $\vec{v}$	Texture-Caching: $\vec{r}$,ρ, $\vec{v}$
Shepard-Kernel					
Laufzeit	76 ms	54 ms	56 ms	-	-
L2-Auslastung	Mid(6)	Low(3)	Low(3)	-	-
Texture-Cache-Auslastung	-	Low(3)	Mid(4)	-	-
Texture-Cache-Hitrate	-	85%	78%	-	-
CUDA-Core-Auslastung	33%	48%	47%	-	-
Änderungen-Kernel					
Laufzeit	105 ms	82 ms	96 ms	86 ms	84 ms
L2-Auslastung	Mid(6)	Mid(4)	Low(3)	Mid(4)	Mid(6)
Texture-Cache-Auslastung	-	Low(2)	Low(2)	Low(3)	Mid(4)
Texture-Cache-Hitrate	-	84%	75%	56%	43%
CUDA-Core-Auslastung	46%	58%	50%	58%	62%

Abbildung 4.22: Benchmark für die Kernel mit dem unterschiedlichen Texture-Caching

So wurde die Performance des Shepard-Kernels und des zeitlichen Änderungen-Kernels in Abbildung 4.22 in Abhängigkeit davon, welche Partikeldaten im Texture-Cache zwischengespeichert werden, gebenchmarkt. Dabei zeigt sich, dass es sowohl beim Shepard-Kernel als beim Kernel der zeitlichen Änderungen performantesten ist, wenn die SPH-Simulation nur die Position im Texture-Cache zwischenspeichert. Dadurch lässt sich die Performance allerdings deutlich von 76 ms auf 54 ms beim Shepard-Kernel und von 105 ms auf 82 ms beim Änderungen-Kernel erhöhen. Auch nimmt die CUDA-Core Auslastung durch die Optimierung deutlich von 33% auf 48% beim Shepard-Kernel und von 46% auf 58% beim Änderungen-Kernel zu. Die deutliche Zunahme und die nur mittelmäßige Auslastung der L2-Cache-Bandbreite zeigen, dass mit Wahrscheinlichkeit immer noch eine Limitierung der Performance durch Latenzen vorlag. Die Hitrate des Texture-Caches ist in demjenigen Fall, dass die SPH-Simulation nur die Positionen zwischenspeichert, mit in etwa 85% sehr gut, woraus folgt, dass das Workingset in etwa in den Texture-Cache passt. Speichert die SPH-Simulation mehr als die Positionen im Texture-Cache zwischen, so nimmt die Texture-Cache-Hitrate stark auf bis zu 43% ab. Die Performance wird zwar ebenfalls etwas schlechter, bleibt aber bei beiden Kernels in etwa konstant. Das ist wahrscheinlich darauf zurückzuführen, dass sich der Performancegewinn durch das vermehrte Texture-Caching sowie die kleiner werdende Texture-Cache-Hitrate in etwa kompensieren. Lediglich die Performance des Änderungen-Kernels wird deutlich schlechter, wenn die SPH-Simulation zusätzlich zur Position die Dichte im Texture-Cache zwischenspeichert. Die genaue Ursache hierfür ist unklar. Eine Möglichkeit wäre aber, dass der Compiler in diesem Fall das Kernel etwas schlechter optimiert. Des Weiteren nimmt die Auslastung des L2-Caches ebenfalls zu, je mehr Variablen im Texture-Cache zwischengespeichert werden. Dies ist ebenfalls auf die kleiner werdende Hitrate des Texture-Caches zurückzuführen, da bei einem Miss eine komplette Cache-Line aus dem L2-Cache geholt werden muss. Interessanterweise nimmt die Schätzung

der CUDA-Core-Auslastung beim Änderungen-Kernel trotz abnehmender Performance mit der Anzahl der im Texture-Cache zwischengespeicherten Größen zu. Dies ist wahrscheinlich darauf zurückzuführen, dass der Compiler durch das Einfügen der __ldg-Intrinisc etwas anderen Maschinencode erzeugt sowie dass die Auslastung der CUDA-Cores nur geschätzt ist.

Somit lässt sich sagen, dass sich durch das Texture-Caching die Performance stark erhöhen ließ. Allerdings ist es wegen der kleinen Größe des Texture-Caches für die Performance vorteilhaft nicht alle sondern nur einen Teil aller Partikeldaten dort zwischenzuspeichern. Auch lässt sich durch das Texture-Caching die Performance am meisten erhöhen, wenn ausschließlich die Positionen dort zwischengespeichert werden. Aus dem Grund wird das ausschließliche Zwischenspeichern der Positionen im Folgenden bei beiden Kerneln weiter verwendet.

4.9.6 L2-Cache-Blocking

Als Nächstes sollen die Berechnungen des Fixed-Radius-Near-Neighbors-Problems weiter durch L2-Cache-Blocking verbessert werden. Auf diese Weise soll die L2-Cache-Hitrate erhöht werden. Dadurch sollen wiederum die Auswirkungen der DRAM-Latenzen reduziert und die DRAM-Bandbreite eingespart werden.

Denn momentan gibt es eine direkte Zuweisung der Thread-ID auf die ID desjenigen Zentralpartikels, welches der Thread bearbeiten soll. Die GPU erstellt und bearbeitet die einzelnen Threads beziehungsweise Thread-Blocks zwar parallel aber linear aufsteigend. Deswegen arbeitet sie die Partikel ebenfalls gemäß dieser Reihenfolge ab. So werden beispielhaft und etwas vereinfacht dargestellt bei einer maximalen Occupancy von 64 Warps zuerst die Partikel mit den Nummern 0 bis 28 671 von den 14 Multiprozessoren der GPU bearbeitet. Dann bearbeitet die GPU die Partikel mit den Nummern 28 672 bis 57 343 und so weiter. Zudem besitzt jedes Zentralpartikel bei maximaler Occupancy im L2-Cache Platz für 14 32-Bit-Werte. Aus diesem Grund würden die Daten der Zentralpartikel, welche die GPU zu einem gegebenen Zeitpunkt bearbeitet, Platz im L2-Cache finden. Arbeitet die GPU die Partikel jedoch linear ab, so benötigt ein jedes Zentralpartikel die Daten der Nachbarpartikel. Da diese Nachbarpartikel wegen der linearen Abarbeitung der GPU momentan zum Großteil nicht als Zentralpartikel dienen, erhöht sich der Cache-Working-Set dadurch drastisch. Deshalb könnte es vorteilhaft sein, wenn die GPU die Partikel in einer Cache-Blocking freundlicheren Reihenfolge abarbeiten würde.

Eine Möglichkeit hierfür ist es, wie im Punkt 6.3.1 der aktuellen Ansätze kurz vorgestellt, die Gitterzellen selbst und damit die Partikel und ihre Partikeldaten im Speicher nicht linear sondern zum Beispiel durch eine Z-Kurve anzuordnen. Allerdings würde dies die Traversierung des Gitters deutlich teurer machen, da die Implementierung in diesem Fall nicht mehr die X-Schleife herausoptimieren könnte. Zudem wäre später kein einfaches Kopieren der Daten im Cluster, wie im Punkt 4.1 erläutert, mehr möglich.

Aus diesem Grund wird in dieser Arbeit ein anderer Ansatz gewählt. Bei dem Ansatz ändert die SPH-Simulation die Zuweisung von den IDs der Threads auf die IDs der Partikel, so dass die GPU die Partikel in einer räumlich lokalen Reihenfolge abarbeitet. Dadurch behält das Verfahren die Anordnung des Gitters und der Partikel im Speicher bei. Dafür muss das Kernel lediglich so modifiziert werden, dass ein jeder Thread zu Beginn des Kernels die ID seines Zentralpartikels per einfachen Lookup in einem Array bestimmt, in welchem diese Zuweisungen abgespeichert werden. Das so modifizierte Kernel ist in Abbildung 4.23 zu sehen.

Für diesen Lookup müssen allerdings vor dem Starten des Fixed-Radius-Near-Neighbors-Kernels die Einträge des Lookup-Arrays erstellt werden. Dieses Erstellen geschieht beim Bauen

```
template<class ProblemStruct>
__global__ void BlockingFRNNKernel(ParticleGrid Grid, ParticleData Particles)
{
  uint ThreadID = blockIdx.x * blockDim.x + threadIdx.x;
  uint CentralPartID = Particles.PartIDLuT[ThreadID];
  ProblemStruct PS;
  PS.Init(Particles, CentralPartID);
  /* ..... */
}
```

Abbildung 4.23: Fixed-Radius-Near-Neighbors-Kernel mit L2-Cache-Blocking

des Gitters, indem die SPH-Simulation nur für das Erstellen des Lookup-Arrays die Reihenfolge der Gitterzellen vertauscht. Hierfür werden die beim Bauen des Gitters im Einfügen-Schritt berechneten Anzahlen der Partikel in den Gitterzellen zuerst gemäß der Z-Kurve permutiert und in einem weiteren temporären Uint-Array des Gitters abgespeichert. Deshalb benötigt das Blocking 4 Byte mehr Speicherplatz pro Gitterzelle. Darauf wird nun die Prefix-Sum angewendet, wodurch die SPH-Simulation die permutierten Partikelstartindexe erhält. Anschließend werden aus den permutierten Partikelstartindexen in einem weiteren Kernel die permutierten Partikelindexe berechnet und in einem 4-Byte-Array für Partikeldaten abgespeichert. Auf diese Weise wird nun letztendlich erreicht, dass in den Fixed-Radius-Near-Neighbors-Kernels die Partikel räumlich gesehen in der Reihenfolge der Z-Kurve abgearbeitet werden.

Damit dieses Blocking einen Speedup erzielt, darf allerdings ein Gitterblock mit den Dimensionen $(\mathrm{Dim}_x, \mathrm{Dim}_y, 2n_{\mathrm{Trav}} + 1)^T$ und die darinnen enthaltenen Partikel nicht in den L2-Cache passen. Denn in diesem Fall würde das Workingset der naiven Abarbeitung ohne Blocking bereits komplett in dem L2-Cache passen. Dies ist wahrscheinlich bei der gewählten Test-Szene der Fall, da diese eine L2-Cache-Hitrate von circa 96% besitzt. Deshalb wird die Szene auf 20 500 000 Partikel durch einen Startabstand von 0.32 m vergrößert. Des Weiteren zeigte es sich bei ersten Tests, dass die Z-Kurve extrem ungünstig für den Texture-Cache ist. Deshalb wurde eine zweite Version des Blockings geschaffen, bei welcher die Gitterzellen zuerst linear zu einem 32 auf 32 auf 32 großen Würfel gruppiert werden. Diese so entstandenen Würfel werden nun wiederum in der Z-Kurve angeordnet. Dadurch ergibt sich eine Kombination aus linearer Anordnung und Z-Anordnung.

Auf diese Weise wurde nun das in Abbildung 4.24 gezeigte Benchmark angefertigt. Aus den Laufzeiten folgt, dass diejenige Version mit der Kombination aus linearer Anordnung und Z-Anordnung am performantesten ist. So erhöht die Kombination aus linearer Anordnung und Z-Anordnung beim Shepard-Kernel die Performance von 333 auf 319 ms und beim Kernel der zeitlichen Änderungen die Performance von 500 ms auf 485 ms gegenüber der Version ohne Blocking. Ebenfalls erhöht sie beim Shepard-Kernel die L2-Cache-Hitrates von 90% auf 99% und senkt das übertragene DRAM-Volumen von 3 GB auf 1.2 GB deutlich. Beim Kernel der Änderung fällt die Senkung von 6.5 GB auf 4.3 GB geringer aus. Ebenso verbessern sich die L2-Cache-Hitrates von 91% auf 94% beziehungsweise von 95% auf 96% deutlich weniger. Die Vermutung liegt nahe, dass das auf das Registerspilling zurückzuführen ist, da hier weiterhin die Version mit einer von 40 auf 32 stark beschränkten Registerzahl verwendet wird. Eine Überprüfung der Version mit 40 Registern bestätigt die Vermutung, da bei ihr nur in etwa 1.9 GB

	Ohne Blocking	Mit kompletter Z-Anordnung	Kombination aus linearer Anordnung und Z-Anordnung
Shepard-Kernel			
Laufzeit	333 ms	364 ms	319 ms
DRAM-Auslastung	3%	1.3%	1.3%
Übertragenes DRAM-Speichervolumen	3 GB	1.4 GB	1.2 GB
Texture L2-Hitrate	90%	99%	99%
L2-Hitrate durch L1-Reads	96%	98%	98%
Texture-Cache-Hitrate	87%	64%	86%
Änderungen-Kernel (64er Occupancy)			
Laufzeit	500 ms	599 ms	485 ms
DRAM-Auslastung	5%	5%	4%
Übertragenes DRAM-Speichervolumen	6.5 GB	7.8 GB	4.3 GB
Texture L2-Hitrate	91%	95%	94%
L2-Hitrate durch L1-Reads	95%	94%	96%
Texture-Cache-Hitrate	87%	64%	86%

Abbildung 4.24: Benchmark für die Kernel mit L2-Cache-Blocking

an DRAM-Speichervolumen übertragen werden. Dies steht im Gegensatz zu der Messung aus Punkt 4.9.4, bei welcher sich das übertragene DRAM-Speichervolumen durch die Begrenzung der Register nicht geändert hat. Eine mögliche Ursache für den Gegensatz wäre jedoch, dass das Workingset bei der kleineren Problemgröße ebenfalls kleiner ist, wodurch im L2-Cache mehr Platz für das Registerspilling ist. Eine alternativer Grund wäre, dass der Compiler wegen den kleineren weiteren Änderungen einen etwas anderen Maschinenode mit mehr Registerspilling erzeugt, wodurch sich wiederum die benötigte DRAM-Bandbreite stark erhöht. Die Auslastung der DRAM-Bandbreite ist mit 1.3% bis 5% bei beiden Kerneln immer noch niedrig. Deshalb ist der Performancegewinn wahrscheinlich bei der Kombination aus der linearen Anordnung und Z-Anordnung auf weniger Latenzen durch Speicherzugriffe zurückzuführen. Dies ist wiederum ein weiteres Anzeichen dafür, dass das Kernel, trotz der bislang hohen Cache-Hitrate von 90% bis 95%, immer noch durch Latenzen limitiert ist.

Des Weiteren zeigt die Tabelle, dass sich die Performance durch das komplette Z-Blocking stark von 333 ms auf 364 ms beim Shepard-Kernel und von 500 ms auf 599 ms beim Kernel der zeitlichen Änderungen reduziert. Die Vermutung liegt nahe, dass dieser Performanceverlust auf die stark von 87% auf 64% gesunkene Texture-Cache-Hitrate zurückzuführen ist. Interessanterweise erhöht sich beim Kernel der zeitlichen Änderungen das übertragene DRAM-Volumen durch das komplette Z-Blocking sogar leicht verglichen mit der Version ohne Blocking. Wahrscheinlich ist dies darauf zurückzuführen, dass die zusätzlichen Texture-Cache-Misses wiederum L2-Cache-Misses verursachen.

Somit lässt sich insgesamt durch das Blocking das direkte Ziel des Blockings, nämlich die Erhöhung der Performance der Fixed-Radius-Near-Neighbors-Kernels durch die Erhöhung der Cache-Hitrate und die Reduktion des übertragenen DRAM-Volumens, erreichen. Durch die Optimierung des Blockings ließ sich bei der vergrößerten Problemgröße die Performance vom Shepard-Kernel um 14 ms und die Performance des Änderungen-Kernel um 15 ms erhöhen. Allerdings gilt es zusätzlich zu berücksichtigen, dass die SPH-Simulation beim Blocking ein weiteres 4-Byte-Partikel-Array als Lookup-Array und zusätzlich ein weiteres 4-Byte-Gitter-Array für die Konstruktion des Lookup-Arrays benötigt. Zusätzlich kostet die Konstruktion des Lookup-Arrays insgesamt pro Zeitschritt in etwa 10 ms. Deshalb scheint der Nutzen dieses Blockings gering zu sein. Aus dem Grund verwendet es die Arbeit im Folgenden nicht weiter. Deshalb wird ebenfalls weiterhin die kleinere Problemgröße von 3972734 Partikeln verwendet.

Jedoch wären weitere Untersuchungen bezüglich dieser Optimierung interessant. So benötigte die getestete Szene nur in etwa 45% der gesamten 6 GiByte des DRAMs der GPU, da das Profiling im Visual-Profiler ansonsten wegen des Auslagern des globalen Speichers zu lange gedauert hätte. Des Weiteren existieren Tesla-Graphikkarten mit dem selben GK-110-GPU-DIE, welche bereits 12 GiByte an DRAM besitzen. Somit könnte eine GK-110-GPU wesentlich größere Szenen als die getestete Szene bearbeiten. Solche Szenen würden auch durch eine Blocking-Optimierung stärker profitieren. Ebenso wäre es interessant weitere Blocking-Anordnungen, mit Hilfe derer das Lookup-Array konstruiert wird, zu untersuchen. So wäre zum Beispiel eine Anordnung interessant, bei der mehrere Gitterzellen zuerst linear zu einem Würfel gruppiert werden. Anschließend würden die Würfel wiederum linear im Speicher angeordnet sein. Denn bei einer solche Anordnung wäre die Konstruktion des Lookup-Arrays leichter. Allerdings würde sie die Cache-Oblivious-Eigenschaften der Z-Kurve verlieren. Zudem wäre die Untersuchung auf anderen GPUs interessant, da sich GPUs stark in ihrem Hardwareaufbau unterscheiden. Beispielhaft gilt bei AMD-GPUs, dass sie eine wesentlich höhere Nebenläufigkeit bezüglich ihrer Cache-Größe besitzen. So besitzt beispielhaft eine AMD R290x nur 1 MiByte L2-Cache und kann maximal 112 640 Threads gleichzeitig bearbeiten. Bei solchen GPUs wäre ein Blocking vermutlich ebenfalls vorteilhafter.

4.9.7 Verlet-Listen On-The-Fly

In diesem Punkt soll die Idee der Verlet-Listen On-The-Fly vorgestellt werden. Bei der Idee wird das Fixed-Radius-Near-Neighbors-Kernel aus Abbildung 4.9.4 um eine Verlet-Liste, welche das Kernel dort On-The-Fly erstellt und danach auswertet, erweitert, um die Warp-Divergenz des Kernels zu reduzieren.

So treten in der bisherigen besten Version des Fixed-Radius-Near-Neighbors-Kernel aus dem Punkt 4.9.3 vornehmlich an zwei Stellen Warpdivergenz aus:

- In der innersten For-Schleife, bei welcher das Kernel über die Partikel in einer X-Gruppe iteriert.
- Bei dem Return-Befehl-Befehl in der problemspezifischen Structure, durch welchen abgefragt wird, ob das Nachbarpartikel innerhalb der Cutoff-Distance liegt.

Die Warpdivergenz führt dazu, dass bei den problemspezifischen Berechnungen nicht mehr alle Warp-Threads teilnehmen. Sind die Berechnungen bei einem gegebenen Problem sehr günstig,

```
template<class ProblemStruct>
__global__ void VerletListFRNNKernel(ParticleGrid Grid, ParticleData Particles)
{
  uint CentralPartID = blockIdx.x * blockDim.x + threadIdx.x;
  uint VerletList[MaxParticlesInNeighborHood];
  uint ParticlesInList = 0;
  float3 CentralPosition = PS.Positions[CentralPartID]
  int3 CellIDCenter = Grid.CalculateCellID(PS.CentralPosition);

  for(int z = -GridTravDist; z <= GridTravDist; z++)
  for(int y = -GridTravDist; y <= GridTravDist; y++)
  {
    int3 CellIDTravXMin = CellIDCenter + make_int3(-GridTravDist,y,z);
    uint CellIndexXMin = Grid.GetCellIndex(CellIDTravXMin);
    uint PartBegin = Grid.ParticleStart[CellIndexXMin];
    uint PartEnd = Grid.ParticleStart[CellIndexXMin+2*GridTravDist+1];
    for(int i = CellBegin; i < CellEnd; i++)
      if(distance(CentralPosition, make_float3(Particles.Positions[i])) <= CutOffDistance);
      {
        VerletList[ParticlesInList] = i;
        ParticlesInList++;
      }
  }
  ProblemStruct PS;
  PS.Init(Particles, CentralPartID);
  uint NextIndex = VerletList[0];
  for(uint i = 0; i < ParticlesInList; i++)
  {
    uint CurrentIndex = NextIndex;
    NextIndex = VerletList[i+1];
    PS.Evaluate(Particles, CurrentIndex);
  }
  PS.FinalizeAndWriteBack(Particles,CentralPartID);
}
```

Abbildung 4.25: Verlet-Liste-On-The-Fly-Kernel für der Berechnung des Fixed-Radius-Near-Neighbors-Problems

so sind die Auswirkungen der Warp-Divergenz auf die Performance eher gering. Sind die Berechnungen jedoch teuer, so sollte eine SPH-Simulation die Warp-Divergenz optimieren. Eine einfache Möglichkeit hierfür ist es, eine Verlet-Liste zu verwenden. Die Verlet-Liste wird dabei On-The-Fly im Kernel innerhalb des lokalen Speichers erstellt und anschließend auch noch im selben Kernel ausgewertet. Da die SPH-Simulation dafür den temporären lokalen Speicher verwendet, muss sie nur noch eine Verlet-Liste für jeden Thread im DRAM der GPU abspeichern und nicht eine Verlet-Liste für jedes Partikel. Auf diese Weise entfällt der höhere Speicherplatzverbrauch der Verlet-Liste.

Das mit der Verlet-Liste modifizierte Kernel ist in Abbildung 4.25 zu sehen. Das Kernel alloziert zuerst statisch ein Array für die Partikelindexe der Verlet-Liste. Im Folgenden findet aus Performancegründen keine Überprüfung auf einen Überlauf der Liste statt. Deshalb muss das Kernel, um eine Speicherschutzverletzung zu vermeiden, die Größe des Arrays so groß wählen,

dass das Kernel auch im schlimmsten Fall alle Nachbarpartikel innerhalb der Cutoff-Distance eines jeden Zentralpartikels in die Liste aufnehmen kann. Als Nächstes wird in dem Kernel analog wie bisher das Gitter traversiert. Bei dem Iterieren über die Partikel ruft das Kernel allerdings nicht die problemspezifische Auswertungsfunktion auf. Es überprüft lediglich, ob sich das Nachbarpartikel innerhalb der Cutoff-Distance des Zentralpartikels befindet. Ist das der Fall, so wird der Index des Partikels in die Verlet-Liste eingefügt. Nachdem das Kernel alle Gitterzellen traversiert hat, traversiert es über die Nachbarpartikel in der Verlet-Liste in einer weiteren For-Schleife. Für jedes Partikel in dieser Liste ruft es nun die Auswertungsfunktion der problemspezifischen Structure auf. Bei der Verlet-Liste zeigte sich eine einfache Software-Pipelining-Technik stark performancesteigernd. Bei ihr lädt das Kernel denjenigen Partikelindex, den es in einem bestimmten Schleifendurchlauf benötigt, immer bereits verfrüht zu Beginn des vorhergehenden Schleifendurchlaufs aus der Verlet-Liste. Denn die Ladeoperationen aus dem lokalen Speicher der Liste verursachen große Latenzen, die die GPU so besser überbrücken kann. Da jedes Zentralpartikel in etwa gleich viele Nachbarpartikel besitzt, unterscheiden sich die Zahl der Schleifendurchläufe beim Leeren der Liste innerhalb eines Warps kaum. Zudem befinden sich in der Verlet-Liste alle Nachbarpartikel innerhalb der Cutoff-Distance. Durch beides wird letztendlich die Warpdivergenz minimiert. Aus dem Grund kann die SPH-Simulation die Überprüfung in der Auswertungsfunktion, ob sich das Nachbarpartikel innerhalb der Cutoff-Distance befindet, entfernen, um die Performance etwas zu erhöhen.

Generell besitzt das Verfahren der Verlet-Liste On-The-Fly aber zwei Nachteile:

- **Overhead:** Durch das Einfügen und Entfernen in beziehungsweise aus der Verlet-Liste entsteht ein Overhead.
- **Größerer Cache-Workingset:** Die Verlet-Listen aller GPU-Threads insgesamt sind bei maximaler Occupancy und 120 Einträgen pro Partikel 14 Megabyte groß. Der L2-Cache ist jedoch nur 1.5 Megabyte groß. Dadurch verdrängen sich die Daten der Verlet-Liste nicht nur selbst, sondern sie verdrängen auch die Gitterdaten und die Partikeldaten. Beide werden jedoch sehr häufig gelesen, weshalb durch ihr Verdrängen hohe Kosten entstehen.

Dieser Workingset lässt sich allerdings etwas verkleinern. Dafür muss das Kernel die Verlet-Liste nicht nur einmalig nach der Traversierung des Gitters leeren. Deshalb wurden noch zwei weitere Versionen des Verlet-Liste-Kernels geschaffen, welche die Verlet-Liste einmalig in jedem Schleifendurchlauf der Z-Schleife beziehungsweise der Y-Schleife bei der Traversierung des Gitters leeren. Je öfters das Kernel die Liste leert, desto kleiner wird auch ihr benötigter Workingset. Allerdings kann sie auch die Warp-Ausführungseffizienz weniger verbessern. Des Weiteren muss das Kernel bei diesen Modifikationen zudem die Daten der problemspezifischen Structure auch während der Traversierung vorrätig halten, wodurch die benötigte Registerzahl während der Traversierung wieder etwas vergrößert wird. Des Weiteren ist es fraglich, ob dieser Ansatz überhaupt noch mit dem Begriff Verlet-Liste beschrieben werden kann.

Die verschiedenen Möglichkeiten die Verlet-Liste zu implementieren wurden in Abbildung 4.26 untersucht. Die Messungen zeigen, dass bei beiden Kernels die beste Version der Verlet-Liste diejenige ist, welche die SPH-Simulation in jedem Z-Schleifendurchlauf leert. Dennoch brauchen das Shepard-Kernel mit 72 ms gegenüber 52 ms und das Kernel der zeitlichen Änderungen mit 89 ms gegenüber 82 ms länger als ohne Verlet-Liste. Dadurch ist die Verlet-Liste im jetzigen Zustand nicht als Erfolg zu werten. Die niedrigere Performance ist sehr wahrscheinlich auf den Overhead der Verlet-Liste zurückzuführen. Aus den Messungen folgt zudem, dass desto

	Keine Verlet-Liste	Verlet-Liste nach der Traversierung	Verlet-Liste in der Z-Schleife	Verlet-Liste in der Y-Schleife
Shepard-Kernel				
Laufzeit	54 ms	190 ms	72 ms	88 ms
Max. Register	23	29	31	32
Beste Occupancy	64	64	64	64
Warp-Ausführ-ungseffizienz	79%	80%	74%	65%
DRAM-Auslastung	3%	31%	43%	18%
L2-Hitrate durch L1-Reads	98%	80%	72%	90%
Texture L2-Hitrate	95%	83%	96%	91%
Änderungen-Kernel				
Laufzeit	82 ms	214 ms	89 ms	113ms
Max. Register	40	36	41	41
Beste Occupancy	64	48	64	64
Warp-Ausführ-ungseffizienz	54%	84%	75%	64%
DRAM-Auslastung	4%	33%	40%	8%
L2-Hitrate durch L1-Reads	97%	83%	96%	95%
Texture L2-Hitrate	94%	85%	83%	90%

Abbildung 4.26: Benchmark für die Fixed-Radius-Near-Neighbors-Kernel mit den verschiedenen Verlet-Listen

häufiger das Kernel die Verlet-Liste leert, umso höher wird auch dessen L2-Cache-Hitrate und umso geringer dessen DRAM-Auslastung. Dies ist auf den kleiner werdenden Workingset der Verlet-Listen zurückzuführen, welcher zunehmend besser in den L2-Cache der GPU passt. Die Messung der Warp-Ausführungseffizienz zeigt, dass die Verlet-Liste beim Kernel der zeitlichen Änderungen die Warp-Ausführungseffizienz stark von 54% auf bis zu 84% erhöhen konnte. Eine Erhöhung beim Shepard-Kernel fiel jedoch wegen dessen einfacheren Auswertungsfunktion von 79% auf bis zu 80% deutlich geringer aus. Somit konnte die Verlet-Liste zumindest das Ziel der Erhöhung der Warp-Ausführungseffizienz erreichen. Je häufiger die Verlet-Liste geleert wird, desto geringer wird jedoch ihr Nutzen bezüglich der Warp-Ausführungseffizienz. Leeren die Kernel sie in jeder Y-Schleife so sinkt die Warp-Ausführungseffizienz deutlich von 84% auf 64% beim Kernel der zeitlichen Änderungen und von 80% auf 65% beim Shepard-Kernel. Das ist auf die zunehmend stärker schwankende Zahl der Partikel innerhalb der Verlet-Listen eines Warps zurückzuführen. Interessanterweise sind die performantesten Versionen der Verlet-Listen-Kernels diejenigen, welche ihre Verlet-Liste immer in der Z-Schleife leeren. Dies wird vermutlich dadurch verursacht, dass hier der Workingset noch nicht zu groß ist und zudem

die Warp-Ausführungseffizienz nicht zu niedrig ist. Da die DRAM-Auslastung mit bis zu 40% nie deutlich limitiert, ist der Performance-Gewinn durch den kleineren Workingset wahrscheinlich auf die besser werdenden Cache-Hitrates zurückzuführen, welche die Latenzen verringern. Ebenfalls ist zu beobachten, dass wenn die Verlet-Liste nach der Traversierung des Gitters geleert wird, sich wie prognostiziert ihre benötigten Register reduzieren.

Aus diesen Gründen gibt es noch weitere interessante Verbesserungsansätze, um die Verlet-Liste noch etwas umzustrukturieren. Das Ziel sollte es bei den Verbesserungen sein, dass die Latenzen bei der Verlet-Liste weniger ins Gewicht fallen. Auf diese Weise könnte die Verlet-Liste doch noch für die gewählten Fixed-Radius-Near-Neighbors-Probleme Performance-steigernd wirkt. Da die aktuelle Version der Verlet-Liste jedoch für die gewählten Fixed-Radius-Near-Neighbors-Probleme schlechter ist, wird im Folgenden weiterhin das Kernel ohne Verlet-Liste verwendet. Allerdings könnte die Verlet-Liste bereits in ihrer aktuellen Form für komplexere Fixed-Radius-Near-Neighbors-Probleme besser sein, bei welchen eine bessere Warp-Ausführungseffizienz wichtiger als der Overhead und die Latenzen der Verlet-Liste sind. Ebenfalls interessant wäre es die Verlet-Liste auf einer SIMD-Architektur zu testen, die eine geringere Nebenläufigkeit und größere Caches besitzt. Hierunter fällt zum Beispiel der Xeon-PHI-Prozessor von Intel.

4.9.8 Warp-ausführungseffizientes Umstrukturieren der Nachbarpartikel-For-Schleife durch Überspringen von Nachbarpartikeln

In diesem Punkt soll ein Ansatz vorgestellt werden, wie beim Kernel aus Punkt 4.9.3 die innerste For-Schleife, welche über die Nachbarpartikel iteriert, umgestaltet werden kann. Dabei sollen Nachbarpartikel außerhalb der Cutoff-Distance so übersprungen werden, dass die Warp-Ausführungseffizienz erhöht wird.

So waren bisherig die Kernel aus Punkt 4.9.3 am besten, bei welchen das Gitter ohne Blocking per Linked-Cell mit einer Traversierungsdistanz von 3 traversiert worden ist. Die Kernel rufen in der Auswertungsfunktion, welche in Abbildungen 4.11 und 4.13 dargestellt sind, immer der Return-Befehl auf, wenn sich ein Partikel außerhalb der Cutoff-Distance befindet. Auf diese Weise verursachen Nachbarpartikel, welche sich außerhalb der Cutoff-Distance befinden, viel Warp-Divergenz, wodurch wiederum viel Rechenleistung verloren geht. Deshalb sollte es das Ziel sein, dass die Auswertungsfunktion nur noch mit Nachbarpartikeln aufgerufen wird, die sich innerhalb der Cutoff-Distance befinden. Ein interessanter Ansatz um das zu erreichen ist es, die innerste For-Schleife, welche über die Partikel in einer X-Gruppe iteriert, in den Kernels aus Punkt 4.9.3 umzugestalten. Das so modifizierte Kernel ist in Abbildung 4.27 zu sehen. Bei dem modifizierten Kernel werden zu Beginn jedes innersten For-Schleifendurchlaufs Positionen von den zwei nächsten Nachbarpartikeln geladen. Ist das erste Nachbarpartikel innerhalb der Cutoff-Distance, so ruft das Kernel am Ende der For-Schleife die Auswertungsfunktion mit dem ersten Nachbarpartikel auf. Ansonsten erhöht es den Iterator der For-Schleife um eins erhöht und führt, falls sich das zweite Nachbarpartikel innerhalb der Cutoff-Distance befindet, den selben Auswertungsfunktionsaufruf am Ende der For-Schleife mit dem zweiten Nachbarpartikel aus. Dadurch überprüft das Kernel bis zu zwei Partikel bevor es die Auswertungsfunktion aufruft. Auf diese Weise soll wieder derjenige Anteil der Threads in einem Warp, die bei einem Schleifendurchlauf ein Nachbarpartikel innerhalb der Cutoff-Distance mit der Auswertungsfunktion bearbeiten, erhöht werden. Somit soll wiederum die Warp-Ausführungseffizienz

gesteigert werden. Allerdings verursachen die zusätzlichen Sprungbefehle und das Laden des zweiten Nachbarpartikels auch einen zusätzlichen Overhead. Um den Overhead etwas zu reduzieren ist es wichtig die Auswertungsfunktion etwas anzupassen, so dass Speicherzugriffe und Abstandsberechnungen nicht doppelt ausgeführt werden. Dadurch, dass die Positionen von zwei Nachbarpartikeln zu Beginn der For-Schleife geladen werden, wird zwar etwas L2-Cache und Texture-Cache-Bandbreite verschwendet, wenn die Position des zweiten Nachbarpartikels überhaupt nicht benötigt werden sollte. Allerdings führt das verfrühte Laden dazu, dass die Position sehr wahrscheinlich in den Registern zur Verfügung steht, wenn sie benötigt wird. Somit handelt es sich bei diesem verfrühten Laden um eine Form des Software-Pipelinings. Auf diese Weise ließ sich in kleineren Benchmarks die Performance des Kernels etwas steigern.

Analog implementierte diese Arbeit eine Version des Kernels, bei welchem die Threads nicht nur eins sondern bis zu zwei Partikel überspringen. Die Version besitzt zwar einen größeren Overhead, kann aber potentiell die Warp-Ausführungseffizienz weiter steigern.

Schließlich wurden die mit diesem Ansatz modifizierten Kernel in Abbildung 4.28 untersucht. Die Untersuchungen zeigen, dass diese Verbesserung beim Shepard-Kernel die Laufzeit von 54 ms auf 65 ms deutlich verschlechterte, während sie beim Kernel der zeitlichen Änderungen die Laufzeit von 82 ms auf 79 ms minimal verbesserte. Dieser Unterschied ist wahrscheinlich wieder auf die komplexere Auswertungsfunktion des Kernels der zeitlichen Änderungen zurückzuführen. Auch ist diejenige Version, bei welcher nur ein Nachbarpartikel übersprungen wird, wesentlich performanter als diejenige Version, bei welcher bis zu zwei Nachbarpartikel übersprungen werden. Interessanterweise verschlechtert sich die Warp-Ausführungseffizienz durch die Umgestaltung beim Shepard-Kernel von 79% auf bis zu 55% deutlich, während sie sich beim Kernel der zeitlichen Änderungen nur kaum von 54% auf 56% und 55% verbesserte. Mit Wahrscheinlichkeit ist dies auf die zusätzliche Warpdivergenz zurückzuführen, welche bei der Berechnung, ob ein Partikel zu überspringen ist oder nicht, auftritt. Aus dem Grund ist auch die kleine Performancesteigerung nicht auf das eigentliche Ziel der Optimierung, nämlich die Erhöhung der Warp-Ausführungseffizienz, zurückzuführen. Die Ursache für die Performancesteigerung ist wahrscheinlich das Software-Pipelining, welches der GPU beim Laden der Positionen der Nachbarpartikel hilft, die Latenzen besser zu überbrücken. Die Messungen zeigen deutlich, dass durch diese Optimierung die Texture-Cache-Auslastung zunimmt, je mehr Positionen der Nachbarpartikel das Kernel im Voraus lädt. Die zunehmende Auslastung rührt daher, dass die zusätzlichen Positionen umsonst geladen wurden, wenn ein Partikel nicht übersprungen wird. Wegen der nur mittelmäßigen Texture-Cache-Auslastung ist eine Limitierung deshalb allerdings unwahrscheinlich.

Da die Optimierungen beim Shepard-Kernel keinen Erfolg zeigte, wird für das Shepard-Kernel weiterhin das bislang beste Kernel ohne diese Optimierung aus Punkt 4.9.3 verwendet. Beim Änderungen-Kernel sind sowohl die Version mit dem Überspringen eines Nachbarpartikels als auch die bisherig beste Version ohne Überspringen aus Punkt 4.9.3 in etwa gleichwertig. Deshalb werden beide weiter untersucht.

4.9.9 Packing der Dichte und der Position in einen Float4

In diesem Punkt soll vorgestellt werden, wie durch das Packing der Partikeldaten die Performance der Fixed-Radius-Near-Neighbors-Kernel weiter erhöht wird. Für das Packing speichert die SPH-Simulation sowohl die Dichten als auch die Positionen der Partikel zusammen in einem einzigen Float4-Array ab.

```
template<class ProblemStruct>
__global__ void VariableCellSizeFRNNKernel(ParticleGrid Grid, ParticleData Particles)
{
  uint CentralPartID = blockIdx.x * blockDim.x + threadIdx.x;
  ProblemStruct PS;
  PS.Init(Particles, CentralPartID);
  int3 CellIDCenter = Grid.CalculateCellID(PS.CentralPosition);

  for(int z = -GridTravDist; z <= GridTravDist; z++)
  for(int y = -GridTravDist; y <= GridTravDist; y++)
  {
    int3 CellIDTravXMin = CellIDCenter + make_int3(-GridTravDist, y, z);
    uint CellIndexXMin = Grid.GetCellIndex(CellIDTravXMin);
    uint PartBegin = Grid.ParticleStart[CellIndexXMin];
    uint PartEnd = Grid.ParticleStart[CellIndexXMin+2*GridTravDist+1];
    for(uint i = PartBegin; i < PartEnd; i++)
    {
      float3 NPosCur = make_float3(__ldg(&Particles.Positions[i]));
      float3 NPosNex = make_float3(__ldg(&Particles.Positions[i+1]));
      float DistCur = distance(NPosCur, S.Position);
      if(DistCur >= CutOffDistance)
      {
        i++;
        NPosCur = NPosNex;
        DistCur = distance(NPosCur, S.Position);
        if(DistCur >= CutOffDistance || i >= PartEnd)
          continue;
      }
      PS.Evaluate(Particles, NPosCur, DistCur, i);
    }
  }

  PS.FinalizeAndWriteBack(Particles, CentralPartID);
}
```

Abbildung 4.27: Fixed-Radius-Near-Neighbors-Kernel mit der Warp-ausführungseffizienten Umgestaltung der Nachbarpartikel-For-Schleife durch Überspringen von Nachbarpartikeln

	Ohne Überspringen	Überspringen von bis zu einem Partikel	Überspringen von bis zu zwei Partikeln
Shepard-Kernel			
Laufzeit	54 ms	65 ms	67 ms
Warp-Ausführungseffizienz	79%	60%	55%
L2-Auslastung	Low(3)	Low(3)	Low(3)
Texture-Cache-Auslastung	Low(2)	Mid(4)	Mid(5)
Änderungen-Kernel			
Laufzeit	82 ms	79 ms	94 ms
Warp-Ausführungseffizienz	54%	56%	55%
L2-Auslastung	Mid(4)	Mid(4)	Mid(4)
Texture-Cache-Auslastung	Low(2)	Low(3)	Mid(4)

Abbildung 4.28: Benchmark für die Fixed-Radius-Near-Neighbors-Kernel mit der Warpausführungseffizienten Umgestaltung der Nachbarpartikel-For-Schleife durch Überspringen von Nachbarpartikeln

So speicherte bislang die SPH-Simulation die Dichten und die Positionen der Partikel jeweils in einem eigenen Float- beziehungsweise einem Float4-Array ab. Da aber das verwendete SPH-Verfahren bei jedem Fixed-Radius-Near-Neighbors-Problem sowohl die Dichten als auch die Positionen der Nachbarpartikel benötigt scheint es lohnenswert zu sein beides in einem einzigen Float4-Array abzuspeichern. Durch das Packing werden die ersten drei Komponenten des Float4 weiterhin von der Position und die vierte Komponente von der Dichte belegt. Auf diese Weise können die Fixed-Radius-Near-Neighbors-Kernel sowohl die Dichte als auch die Position eines jeden Nachbarpartikels mit einer einzigen 16-Byte-Ladeoperation laden. Zusätzlich werden, da die vierte Komponente des Float4-Arrays der Positionen bislang nicht benötigt wurde, 8 Byte Speicherplatz pro Partikel gespart. Die Einsparung kommt ebenfalls der Cache-Effizienz und der benötigten Cache- und Speicherbandbreite zu gute. Des Weiteren lassen sich auf diese Weise die Kosten für das Bauen des Gitters, da die GPU weniger Daten permutieren muss, und die Kosten für die Zeitintegration, da die GPU weniger Daten einlesen und zurückschreiben muss, reduzieren. Auch wird bei der Konstruktion des Gitters kein Float-Permutationsarray mehr benötigt, weshalb noch einmal 4 Byte an Speicherplatz eingespart werden können.

Die Auswirkungen auf das Änderungen-Kernel und das Shepard-Kernel wurden in Abbildung 4.29 gebenchmarkt. Da beim Änderungen-Kernel bislang die Versionen mit dem Überspringen aus Punkt 4.9.8 als auch ohne das Überspringen aus Punkt 4.9.3 in etwa gleich performant waren, werden beide Versionen mit dem Packing gebenchmarkt. Beim Shepard-Kernel wurde nur die performantere Version ohne Überspringen aus Punkt 4.9.3 untersucht. Dabei zeigte sich,

	Ohne Packing von Position und Dichte	Mit Packing von Position und Dichte
Shepard-Kernel ohne Überspringen		
Laufzeit	54 ms	44 ms
CUDA-Core-Auslastung	48%	53%
Änderungen-Kernel ohne Überspringen		
Laufzeit	82 ms	80 ms
CUDA-Core-Auslastung	58%	58%
Änderungen-Kernel mit Überspringen		
Laufzeit	79 ms	80 ms
CUDA-Core-Auslastung	62%	61%

Abbildung 4.29: Benchmark für die Fixed-Radius-Near-Neighbors-Kernel mit Packing von Position und Dichte in einen Float4

dass sich die Laufzeit vom Shepard-Kernel deutlich von 54 ms auf 44 ms steigern lassen konnte. Die Auslastung der CUDA-Cores nimmt ebenfalls von 48% auf 53% zu. Die große Zunahme ist vermutlich auf einen Nebeneffekt dieser Optimierung zurückzuführen. So lädt das Kernel die Dichte des Nachbarpartikels jetzt früher, weshalb es nun die Dichte auch früher als Operand für Rechenoperationen verwenden kann. Der Performancegewinn dieser Optimierung fällt beim Kernel der zeitlichen Änderungen deutlich geringer aus. So verbesserte sich die Laufzeit bei der Version ohne Überspringen von 82 ms auf 80 ms nur unwesentlich und blieb bei der Version mit Überspringen von 79 ms auf 80 ms in etwa konstant. Die Ursache hierfür ist vermutlich, dass die Latenzen beim Laden der Dichte nur eine geringe Rolle auf die Performance des Änderungen-Kernel besitzen. Somit ist das Packing der Postionen und Dichten der Partikel in einen Float4 vorteilhaft, weshalb es im Folgenden weiter verwendet wird

4.9.10 Software-Pipelining durch verfrühtes Laden der Position

In diesem Punkt soll kurz eine weitere einfache Versionen vom Software-Pipelining vorgestellt werden, um die Performance des Fixed-Radius-Near-Neighbors-Problems weiter zu verbessern. Denn bislang zeigte es sich, dass die Performance der Kernel wahrscheinlich immer noch durch Latenzen limitiert sind. Solche Latenzen lassen sich mit der Hilfe von Software-Pipelining besser überbrücken. Für das Software-Pipelining ist es sinnvoll die For-Schleife, in welcher das Kernel über die Partikel in einer X-Gruppe iteriert, in die Auswertungsfunktion hineinzuziehen, da somit das Software-Pipelining spezifisch für die Auswertungsfunktion programmiert werden kann.

Für den ersten Ansatz des Software-Pipelinings wurde das Kernel aus Abbildung 4.9.3, bei welchen noch nicht die innerste For-Schleife für das Überspringen umstrukturiert wurde, umgeformt. Nach der Umformung lädt das Kernel nun zu Beginn eines jeden innersten For-

```
void SheppardPoblemStruct::Evaluate(ParticleData &Particles, uint Start, uint End)
{
  float4 NextPositionAndDensity = Particles.PositionsAndDensities[NeighborID];
  for(uint NeighborID = Start; NeighborID < End; NeighborID++)
  {
    float3 NeighborPosition = make_float3(NextPositionAndDensity);
    float NeighborDensity = NextPositionAndDensity.w;
    float PartDistance = distance(CentralPosition,NeighborPosition);
    NextPositionAndDensity = Particles.PositionsAndDensities[NeighborID+1];

    if(PartDistance >= CutOffDistance)
      continue;

    float KernelValue = WendlandKernel(PartDistance);
    CentralShepardNominator += KernelValue;
    CentralShepardDenominator += 1.f / NeighborDensity * KernelValue;
  }
}
```

Abbildung 4.30: Auswertungsfunktion des Shepard-Kernels mit Software-Pipelining durch verfrühtes Laden der Postionen und Dichten

	Kernel ohne verfrühtem Laden	Kernel mit verfrühtem Laden
Shepard-Kernel		
Laufzeit	44 ms	41 ms
Maximale Register	22	29
Optimale Occupancy	64	64
CUDA-Core-Auslastung	53%	68%
Änderungen-Kernel ohne Überspringen		
Laufzeit	80 ms	100 ms
Maximale Register	40	46
Optimale Occupancy	64	48
CUDA-Core-Auslastung	61%	55%

Abbildung 4.31: Benchmark für die Fixed-Radius-Near-Neighbors-Kernel mit Software-Pipelining durch verfrühtes Laden der Postionen und Dichten

Schleifendurchlaufs bereits den Float4 der Position und der Dichte, welche erst im nächsten Schleifendurchlauf benötigt wird. Dies führt wiederum dazu, dass beide zu Beginn des nächsten Schleifendurchlaufs wahrscheinlich bereits in den Registern zur Verfügung stehen. Die Auswertungsfunktion des so modifizierten Shepard-Kernels ist beispielhaft in Abbildung 4.30 zu sehen.

Die modifizierten Kernel wurden auch in Abbildung 4.31 gebenchmarkt. Dabei zeigte sich, dass

sich durch das Software-Pipelining im Shepard-Kernel die Laufzeit noch einmal um 3 ms von 44 ms auf 41 ms senkte und die CUDA-Core-Auslastung von 53% auf 68% erhöhte. Deutlich ist zudem zu sehen, dass die Zahl der Register beim Shepard-Kernel von 22 auf 29 ansteigt. Von den sieben Registern werden vier für den höheren Memory-Level-Parallelism benötigt. Dadurch kann das Kernel die Position des nächsten Nachbarpartikels bereits im Hintergrund laden. Es wäre interessant weiter zu untersuchen, wofür der Compiler die übrigen drei zusätzlichen Register benötigt. Da das Shepard-Kernel mit 29 Registern immer noch die maximale Occupancy von 64 Warps erzielt, hat das Ansteigen keine negativen Auswirkungen auf die Performance.

Beim Kernel der zeitlichen Änderungen ergibt diese Verbesserung nur bei derjenigen Version Sinn, bei welcher keine Nachbarpartikel außerhalb der Cutoff-Distance übersprungen werden. Deshalb wurde auch nur die Version ohne Überspringen dementsprechend modifiziert. Aber auch dort führte dieses Software-Pipelining zu keinem Erfolg. So verschlechterte das verfrühte Laden die Laufzeit deutlich von 80 ms auf 100 ms. Die CUDA-Core Auslastung sank ebenfalls stark von 61% auf 55%. Die Verschlechterung ist diesmal auf den höheren Registerverbrauch, der für den höheren Memory-Level-Parallelism benötigt wird, zurückzuführen. Denn dadurch erhöhte sich wiederum der maximale Registerverbrauch des Kernels von 40 Register auf 46 Register. Deshalb sinkt die durch das Autotuning ermittelte optimale Occupancy von 64 auf 48 Warps, wodurch sich die Fähigkeit des Kernels Latenzen zu überbrücken verschlechtert. Die Verschlechterung wirkt dem eigentlichen Ziel des Software-Pipelinings, nämlich dass das Kernel Latenzen besser überbrücken kann, entgegen.

Somit ist diese Art des Software-Pipelining nur für das Shepard-Kernel zur Performancesteigerung lohnenswert und wird dort weiter verwendet. Für das Änderungen-Kernel müssen andere Versionen des Software-Pipelinings probiert werden, damit es Latenzen besser überbrücken kann.

4.9.11 Software-Pipelining durch verfrühtes Laden der Geschwindigkeit im Kernel der zeitlichen Änderungen

Aus dem Grund soll in diesem Punkt ein weiteres Software-Pipelining spezifisch für das Kernel der zeitlichen Änderungen vorgestellt werden. Bei dem Ansatz wird in der Auswertungsfunktion das Laden der Geschwindigkeit vor die If-Abfrage, ob das Nachbarpartikel innerhalb der Cutoff-Distance liegt, gezogen. Auf diese Weise steht die Geschwindigkeit wiederum schneller in den Registern zur Verfügung. Dies geschieht allerdings wiederum auf Kosten einer erhöhten Cache-Bandbreite, da nun die Geschwindigkeit auf jeden Fall geladen wird, und nicht nur, wenn sich das Nachbarpartikel innerhalb der Cutoff-Distance befindet. Beides wurde in Abbildung 4.32 mit den bisher performantesten Versionen des Änderungen-Kernels, nämlich dasjenige Änderungen-Kernel, bei welchen bis zu ein Nachbarpartikel übersprungen wird, und das Änderungen-Kernel ohne Überspringen, vermessen. Zusätzlich wurde dort das Texture-Caching der Geschwindigkeit neu evaluiert, da nun die Geschwindigkeit wieder häufiger benötigt wird.

Bei der Betrachtung der Messungen in Abbildung 4.32 zeigt es sich, dass sich bei allen Versionen durch das verfrühte Laden der Geschwindigkeit in Kombination mit dem Texture-Caching der Geschwindigkeit die Performance erhöht. So ließ sich die Laufzeit durch die Optimierung beim Änderungen-Kernel ohne Überspringen von 80 ms auf 75 ms und beim Änderungen-Kernel mit Überspringen von 80 ms auf 79 ms erhöhen. Dadurch ist insgesamt diejenige Version mit verfrühtem Laden des Geschwindigkeit, Texture-Caching der Geschwindigkeit und ohne Überspringen mit 75 ms am performantesten.

	Ohne verfrühtem Laden der Geschwindigkeit	Mit verfrühtem der Laden und ohne Texture-Caching der Geschwindigkeit	Mit verfrühtem der Laden und mit Texture-Caching der Geschwindigkeit
Änderungen-Kernel ohne Überspringen			
Laufzeit	80 ms	84 ms	75 ms
CUDA-Core-Auslastung	58%	55%	62%
L2-Auslastung	Mid(4)	Mid(6)	Mid(6)
Texture-Cache Auslastung	Low(2)	Low(2)	Mid(4)
Texture-Cache-Hitrate	85%	85%	46%
Änderungen-Kernel mit Überspringen			
Laufzeit	80 ms	99 ms	79 ms
CUDA-Core-Auslastung	61%	53%	68%
L2-Auslastung	Low(3)	Mid(6)	Mid(6)
Texture-Cache-Auslastung	Low(3)	Low(3)	Mid(6)
Texture-Cache-Hitrate	88%	89%	62%

Abbildung 4.32: Benchmark für das Änderungen-Kernel mit Software-Pipelining durch verfrühtes Laden der Geschwindigkeit

Interessanterweise wird das Texture-Caching bei beiden Versionen für eine Erhöhung der Performance benötigt. Da die L2-Cache-Auslastung mit und ohne Texture-Caching konstant den Wert Mid(6) besitzt und somit sich nicht deutlich reduziert, ist der große Performancegewinn durch das Texture-Caching wahrscheinlich weiterhin auf die Reduktion der Latenzen zurückzuführen. Ebenso reduziert das Texture-Caching der Geschwindigkeit die Texture-Cache-Hitrate wiederum von 85% auf 46% beziehungsweise von 88% auf 62% deutlich.

Ebenfalls erhöhte sich sowohl bei der Version ohne Überspringen als auch bei der Version mit Überspringen durch die hier vorgestellte Optimierung die geschätzte CUDA-Core-Auslastung von 61% auf 68% beziehungsweise von 58% auf 62%. Da sie mit 62% und 68% immer noch mittelmäßig ist und zudem auch alle relevanten Bandbreiten nur mittelmäßig ausgelastet werden, ist die Performance sehr wahrscheinlich weiterhin durch Latenzen limitiert.

Somit lässt sich abschließend sagen, dass sich durch die Optimierung des verfrühtem Laden der Geschwindigkeit mit Texture-Caching die Performance des Änderungen-Kernel deutlich weiter auf 75 ms erhöhen ließ. Deshalb wird die in Kombination mit dieser Optimierung performanteste Version, nämlich die Version ohne Überspringen, im Folgenden weiter verwendet.

4.9.12 Abschließende Untersuchungen

Abschließend sollen sowohl die finalen Versionen des Shepard-Kernels als auch des Änderungen-Kernels ausführlich untersucht werden. Dabei werden die bisher performantesten Kernel-Versionen verwendet. Beim Änderungen-Kernel handelt es sich um die Version ohne Überspringen und mit verfrühtem Laden der Geschwindigkeit aus dem vorherigen Punkt 4.9.11. Des Weiteren wurde bislang für die einfacheren Benchmarks nur diejenige Version des Änderungen-Kernels mit der Berechnung der maximalen Zeitschrittgröße untersucht, da sie von der Komplexität her sehr ähnlich zur Version ohne die Berechnung der maximalen Zeitschrittgröße ist. Da letztere Version jedoch auch einen Großteil der Laufzeit verursacht, wird sie ebenfalls für die finalen Untersuchungen herangezogen. Beim Shepard-Kernel wird die performanteste Version aus dem Punkt 4.9.10 verwendet, bei welcher die Dichte und die Position des Nachbarpartikels verfrüht geladen werden. Die Messergebnisse der ausführlichen Benchmarks für diese drei Kernel wurde in Abbildung 4.33 eingetragen.

Zuerst soll auf die Laufzeiten eingegangen werden. So ist das Shepard-Kernel mit 41 ms wegen seiner einfacheren Auswertungsfunktion wesentlich günstiger als die beiden Versionen des Änderungen-Kernels mit 75 ms und 79 ms. Dabei ist interessant, dass das Änderungen-Kernel ohne die Berechnung der maximalen Zeitschrittweite langsamer als das Kernel mit dieser Berechnung ist, obwohl es weniger berechnen müsste. Auch besitzt das Kernel ohne die Berechnung der maximalen Zeitschrittgröße eine höhere Registerzahl, obwohl es theoretisch permanent eine automatische Variable, nämlich den Wert von CentralMuMax aus Abbildung 4.13, weniger abspeichern müsste. Des Weiteren besitzt es eine deutlich höhere DRAM-Schreibbandbreite von 2.9 GB/s gegenüber 1.4 GB/s, welche auf ein erhöhtes Registerspilling hindeutet. Aus dem selben Grund ist die L2-Cache-Hitrate durch L1-Reads bei der Version ohne die Berechnung mit 89% gegenüber 93% etwas niedriger. Somit liegt die Vermutung nahe, dass der Compiler beide Versionen etwas anders optimiert. Eine mögliche Ursache hierfür ist, dass die zusätzlichen Berechnungen in der Auswertungsfunktion dem Compiler dabei helfen, die Kosten der Schleifenstrukturen besser einzuschätzen, wodurch wiederum eine bessere Optimierung erfolgt.

Nachdem die Laufzeiten diskutiert worden sind, soll kurz auf den Registerverbrauch und die Occupancy eingegangen werden. So benötigt das Shepard-Kernel nur 29 Register. Da ein Thread im Minimalfall 32 Register allozieren muss, bleiben so 3 Register pro Thread ungenutzt. Diese Register könnten durch geschickteres Code-Design ebenfalls noch dazu verwendet werden, damit die GPU die limitierenden Latenzen besser überbrücken kann. Auch erreicht das Shepard-Kernel auf jeden Fall die maximale Occupancy eines Multiprozessors von 64 Warps, weshalb das Autotuning der Occupancy hier keinen Effekt erzielt. Die Änderungen-Kernel benötigen im unbeschränkten Fall mit 39 und 40 Register deutlich mehr als das Shepard-Kernel. Die Ursache hierfür ist wieder der größere Workingset für automatische Variablen der wesentlich komplexeren Auswertungsfunktion. Da die Occupancy nur 48 Warps beträgt kann hier das Autotuning verwendet werden, um die optimale Occupancy zu bestimmen. Es ermittelt einen optimalen Wert von 64 Warps, welcher wieder die maximalen Occupancy eines Multiprozessors ist. Dafür müssen die Register pro Thread auf 32 reduziert werden. Trotz des stark reduzierten Registerverbrauchs ist die Auslastung der DRAM-Bandbreite weiterhin mit 4% bis 5% niedrig. Daraus kann wiederum gefolgert werden, dass der durch das Registerspilling vergrößerte Workingset für automatische Variablen wahrscheinlich weiterhin in etwa in die Caches der GPU passt.

Als Nächstes soll auf die geschätzte Auslastung der CUDA-Cores eingegangen werden. Dabei

	Shepard-Kernel	Änderungen-Kernel mit der Berechnung von Δt	Änderungen-Kernel ohne die Berechnung von Δt
Allgemein			
Laufzeit	41 ms	75 ms	79 ms
Max. Register	29	39	40
Optimale Occupancy	64	64	64
Instruktionsmetriken			
Ausgeführte IPC	4.1	3.7	3.5
Warp-Ausführungs-effizienz	79%	57%	57%
Arithmetik-metriken			
Geschätzte Auslastung der CUDA-Cores	68%	62 %	58%
Auslastung der Arith-metischen Ausführ-ungseinheiten	Mid(6)	Mid(5)	Mid(5)
$\text{FLOPS}_{\text{gemessen}}$	380 GFLOPS	368 GLOPS	349 GFLOPS
$\frac{\text{FLOPS}_{\text{gemessen}}}{\text{FLOPS}_{\text{peak}}}$	8.3%	8.1%	7.7%
Speichermetriken			
DRAM-Lese-bandbreite	6.5 GB/s	11 GB/s	12 GB/s
DRAM-Schreib-bandbreite	2.0 GB/s	1.4 GB/s	2.9 GB/s
DRAM-Auslastung	3%	4%	5%
L2-Auslastung	Low(3)	Mid(6)	Mid(6)
L2-Hitrate durch L1-Reads	96%	93%	89%
L2-Hitrate durch Texture-Reads	96%	98%	98%
Texture-Cache-Auslastung	Mid(4)	Mid(4)	Mid(4)
Texture-Cache-Hitrate	87%	46%	47%
Load-Store-Units Auslastung	Low(1)	Low(1)	Low(1)
Texture-Units Auslastung	Low(3)	Low(3)	Low(3)

Abbildung 4.33: Finales Benchmark für die Fixed-Radius-Near-Neighbors-Kernel

wird deren Auslastung wieder über das Verhältnis zwischen den ausgeführten IPC und denjenigen IPC, welchen die CUDA-Cores maximal verarbeiten können, geschätzt. Auf diese Weise ergibt sich je nach Kernel ein Wert von 58% bis 68%. Der Wert stimmt in etwa mit der Visual-Profiler-Metrik für die Auslastung der arithmetischen Ausführungseinheiten von Mid(5) bis Mid(6) überein. Die Auslastung aller anderen relevanten Ausführungseinheiten der GPU, nämlich den Texture-Units und den Load-Store Units, ist jedoch nur niedrig. Auch ist die Auslastung der GPU-internen Bandbreiten nur mittelmäßig und die Auslastung der DRAM-Bandbreite mit 3% bis 5% gering. Deshalb ist die niedrige Auslastung der CUDA-Cores wahrscheinlich wieder hauptsächlich auf eine Limitierung durch Latenzen zurückzuführen. Somit gehen durch Latenzen je nach Kernel in etwa 42% bis 32% an Performance verloren.

In diesem Absatz soll kurz die Warp-Ausführungseffizienz eingegangen werden. So ist die Warp-Ausführungseffizienz beim Shepard-Kernel mit 79% relativ gut, weshalb deswegen dort nur 21% der Performance verloren gehen. Die Warp-Ausführungseffizienz bei den Änderungen-Kerneln ist mit 58% deutlich schlechter, weswegen dort 42% der Performance verloren gehen. Die Unterschiede zwischen beiden Kerneln sind wieder auf die komplexere Auswertungsfunktion des Änderungen-Kernels zurückzuführen, wodurch die Warpdivergenz in der Auswertungsfunktion auch die Warp-Ausführungseffizienz stärker reduziert.

Nun soll auf die erreichten FLOPS eingegangen werden. So erreicht das Shepard-Kernel 380 GFLOPS, während die beiden Versionen des Änderungen-Kernels 368 GFLOPS und 349 GLOPS erreichen. Das entspricht beim Shepard-Kernel 8.3% der Peak-Performance, während es bei den Änderungen-Kernels 8.1% und 7.7% sind. Dieser niedrige Wert ist höchstwahrscheinlich darauf zurückzuführen, dass durch die Warp-Ausführungseffizienz weiterhin trotz deren Optimierung beim Shepard-Kernel 21% und bei den Änderungen-Kernels 43% Performance verloren gehen. Zusätzlich gehen gemäß der Auslastungsabschätzung der CUDA-Cores wegen einer Limitierung durch Latenzen weiterhin je nach Kernel in etwa 32% bis 42% der Performance verloren. Zudem werden die FLOPS standardmäßig in FMA berechnet, wobei FMA-Operationen doppelt zählen. Ein weiterer Anteil geht deshalb verloren, dass nicht alle FLOPs im Programm FMA sind. Des Weiteren benötigt die Traversierung des Gitters selbst nur Integer-Rechenleistung, welche in den FLOPS nicht enthalten ist. Aus diesem Grund eignet sich die FLOPS-Metrik bei diesem Verfahren nicht sehr gut, um die gesamte erreichte Rechenleistung zu bestimmen. Jedoch ist es auch diskussionswürdig, ob eine solche Metrik überhaupt den Overhead, der durch die Traversierung einer Space-Partitioning-Datenstruktur entsteht, erfassen sollte. Wird zusätzlich berücksichtigt, dass beide Kernels die Symmetrie der Berechnungen nicht ausnutzen, so verschwenden sie in etwa 50% dieser erreichten FLOPS noch einmal. Wird dieser Prozentsatz von den erreichten FLOPS abgezogen, so bleiben bei allen Kerneln gerade noch einmal in etwa 4% der Peak-Performance übrig. Auch verschwendet die GPU wegen der beschränkten Auflösung des Gitters ebenfalls Rechenleistung dafür, um Nachbarpartikel außerhalb der Cutoff-Distance zu überprüfen. Der hierfür verschwendete Anteil fällt besonders beim Shepard-Kernel ins Gewicht, da dessen Auswertungsfunktion sehr einfach ist. Allerdings ist der Anteil bei beiden Kernels nicht ohne weitere Untersuchungen zu bestimmen.

Ebenfalls von Interesse sind die Speichermetriken. Dabei fällt zuerst auf, dass die Auslastung der DRAM-Bandbreite mit 3% bis 5% je nach Kernel weiterhin niedrig ist. Aus diesem Grund kann die Speicherbandbreite die Performance nur sehr geringfügig reduzieren. Die niedrige Auslastung ist wiederum auf die gute L2-Cache-Hitrate zurückzuführen, welche für L1-Reads 89% bis 96% und für Texture-Reads 96% bis 98% beträgt. Die Hitrate des Texture-Caches beträgt beim Shepard-Kernel 87% und bei den Änderungen-Kernels nur noch 46%. Der Un-

terschied ist darauf zurückzuführen, dass das Shepard-Kernel nur den Float4-Wert für die Positionen und Dichten dort zwischenspeichert, während die finalen Versionen des Änderungen-Kernels auch den Float4-Wert der Geschwindigkeit dort zwischenspeichern. Deshalb hat auch das Änderungen-Kernel einen doppelt so großen Texture-Cache-Workingset, der sich in der deutlich niedrigeren Hitrate äußert. Wegen der niedrigeren Texture-Cache-Hitrate benötigen die Änderungen-Kernels auch mit Mid(6) deutlich mehr L2-Cache-Bandbreite als das Shepard-Kernel mit Low(3). Dementsprechend besitzt die Auslastung der L2-Cache-Bandbreite wahrscheinlich beim Shepard-Kernel einen vernachlässigbaren und beim Änderungen-Kernel nur einen geringen Einfluss auf die Performance. Ebenso ist die Auslastung der Texture-Cache-Bandbreite mit Low(3) bei allen Kerneln nur gering, weshalb sie die Performance nur kaum reduzieren kann. Zudem ist die Auslastung der Load-Store-Units bei allen Kerneln mit Low(1) gering. Die niedrige Auslastung ist darauf zurückzuführen, dass die Load-Store-Units bei den finalen Versionen der Kernels hauptsächlich nur noch zum Laden des Gitters und für das Registerspilling verwendet werden. Letzteres tritt allerdings nur bei den Änderungen-Kernels auf. Wegen ihren niedrigen Wert kann die Auslastung der Load-Store-Units nur einen sehr geringen Einfluss auf die Laufzeit besitzen. Da bei allen Kerneln sämtliche Partikeldaten über die Texture-Units geladen werden, ist deren Auslastung mit Mid(4) deutlich höher. Da sie ebenfalls nur mittelmäßig ist, kann sie die Performance ebenfalls nicht stark reduzieren.

Zu Letzt soll das Fazit aus der Diskussion gezogen werden. So wird weiterhin die Performance durch die Warp-Ausführungseffizienz, die je nach Kernel zwischen 79% und 58% beträgt, limitiert. Zusätzlich kostet die Limitierung durch Latenzen 42% bis 32% an Performance. Da die Auslastung der DRAM-Bandbreite wegen der guten L2-Cache-Hitrate mit 3% bis 5% extrem gering ist, kann sie nicht die Performance limitieren. Des Weiteren sind die Auslastungen der GPU-internen Bandbreiten und der Ausführungseinheiten der GPU nur gering bis mittelmäßig. Dadurch können ihre Auslastungen keinen großen Einfluss auf die Performance besitzen. Die Limitierung durch Latenzen und die niedrige Warp-Ausführungseffizienz führen dazu, dass die GPU je nach Kernel nur in etwa 8% ihrer Peak-Performance erreichen kann. Dabei ist gerade der Performanceverlust durch die Warp-Ausführungseffizienz und durch die Latenzen unbefriedigend, da die Arbeit versucht hat eben diese beiden Probleme zu optimieren. Dennoch ließen sich durch die Optimierungen, die im diesem Kapitel durchgeführt wurden, die Laufzeit des Änderungen-Kernels von 217 ms auf 75 ms und die Laufzeit des Shepard-Kernels von 94 ms auf 41 ms deutlich reduzieren. Deshalb ist die Optimierung dennoch als Erfolg zu werten. Der deutliche Performancegewinn verdeutlicht noch einmal, dass eine SPH-Simulation mit GPGPU sehr optimiert sein muss, um eine gute Performance zu erreichen.

4.9.13 Ausblick

In diesem Punkt soll vorgestellt werden, welche weiteren Verbesserungen vorgenommen werden könnten um die Performance bei der Berechnung des Fixed-Radius-Near-Neighbors-Problems zu erhöhen. Neben kleineren Ansätzen harter Optimierung, wie dass mal hier ein Befehl und dort ein Register eingespart werden könnte, oder denjenigen kleineren Ansätzen, welche bereits bei den einzelnen Verbesserungen erwähnt worden sind, scheinen hauptsächlich folgende Ansätze vielversprechend zu sein:

- **Überspringen von Gitterzellen außerhalb der Cutoff-Distance:** Momentan wird immer ein Würfel von $(2n_{\text{Trav}} + 1)^3$ Gitterzellen traversiert. Einige von diesen Gitterzellen befinden sich jedoch komplett außerhalb der Cutoff-Distance, weshalb die SPH-

Simulation sie ebenfalls komplett überspringen könnte. Beim Überspringen gilt es zwei Fälle zu betrachten:

  - **Dynamischer Fall:** Ob sich die entsprechende Gitterzelle außerhalb der Cutoff-Distance befindet, ist von der Position des Zentralpartikels in seiner Gitterzelle abhängig. Um diese Gegebenheit zu überprüfen müsste das Kernel bei der Traversierung für die entsprechenden Gitterzellen eine Schnittpunktberechnung zwischen Gitterzellenwürfel und der Kugel der Cutoff-Distance durchführen. Da eine Schnittpunktberechnung zwischen Kugel und Würfel viele Rechenoperationen benötigt, könnte das Kernel sie konservativ als Schnittpunktberechnung zwischen Kugel und Bounding-Kugel um den Würfel der Gitterzelle approximieren.
  - **Statischer Fall:** Ob sich die Gitterzelle außerhalb der Cutoff-Distance befindet, ist von der Position des Zentralpartikels in seiner Gitterzelle unabhängig. Dieser Fall tritt nur bei größeren Werten für die Traversierungsdistanz n_{Trav} auf. Um ihn auszunutzen müssten lediglich die Start- und Endwerte in den For-Schleifen der Gittertraversierung angepasst werden.

  In Kombination mit diesen beiden Ansätze wäre es ebenfalls versuchenswert für das Verhältnis zwischen Gitterzellengröße und Cutoff-Distance keine natürliche sondern eine rationale Zahl zu wählen.

- **Höhere Auflösung des Gitters entlang der X-Achse:** Da die Gittertraversierung mit herausoptimierter X-Schleife stattfindet, sind die Kosten der Gittertraversierung selbst nur abhängig davon, wie viele Gitterzellen entlang der Y-Achse und Z-Achse traversiert werden. Dies könnte die SPH-Simulation wiederum ausnutzen indem sie für die X-Achse eine höhere Auflösung wählt als für die Y- und Z-Achse. Denn dadurch müsste sie wiederum weniger Nachbarpartikel außerhalb der Cutoff-Distance betrachten. Auf diese Weise könnte die SPH-Simulation sich Rechenleistung einsparen, ohne dass für die Gittertraversierung zusätzliche Kosten entstehen würden.

- **Herausoptimierung der Randbehandlung:** Im jetzigen Zustand der Arbeit werden bei der Traversierung des Gitters in der Y-Schleife einige Befehle zur Randbehandlung benötigt, auf welche in dieser Arbeit nicht näher eingegangen wurde. Würde keine Randbehandlung stattfinden, so würden diese Befehle ebenfalls entfallen. Zudem könnte die Implementierung ohne diese Randbehandlung die Gittertraversierung in eine einzige For-Schleife weiter vereinfachen. Auf diese Weise könnte sich die Implementierung zusätzlich einige Register einsparen. Ein Ansatz um die Randbehandlung komplett zu vermeiden wäre es das Gitter an jeder Seite um die Traversierungsdistanz n_{Trav} zu vergrößern. Zusätzlich muss die SPH-Simulation sicherstellen, dass sich in diesen Randzellen niemals Partikel befinden. Nun wäre bei der Gittertraversierung auch sicher gestellt, dass in den For-Schleifen niemals der Randfall eintritt. Aus diesem Grund würde die Randbehandlung ebenfalls nicht benötigt werden.

- **Blocking des Gitters:** Ein weiterer Ansatz für das Blocking wäre es das Gitter im Speicher zu sehr großen Blöcken anzuordnen, wobei das Gitter innerhalb eines solchen Blockes linear angeordnet ist. Dadurch wären die Partikel räumlich gesehen ebenfalls in Blöcken angeordnet. Neben dem Blocking-Effekt würde das Fixed-Radius-Near-Neighbors-Kernel weiterhin keine X-Schleife bei der Gittertraversierung benötigen. Zudem wäre die lineare Anordnung für das Texture-Caching vorteilhaft. Diese

Art des Blockings würde allerdings im Fixed-Radius-Near-Neighbors-Kernel zusätzliche Randbehandlungen an den Rändern eines solchen Blocks benötigen. Auch käme diese Blocking-Methode der Performance bei der Konstruktion des Gitters zu gute.

- **Weitere Optimierung der Warp-Ausführungseffizienz:** Es wurden bereits Ansätze wie die Verlet-Liste und das Warp-effiziente Überspringen von Nachbarpartikeln außerhalb der Cutoff-Distance unternommen um die Warp-Ausführungseffizienz zu steigern. Dennoch ist sie in den finalen Kernelversionen mit 79% bis 57% niedrig. Deshalb könnte es für die SPH-Simulation lohnenswert sein die Schleifen weiter umzugestalten, um eine etwas höhere Warp-Ausführungseffizienz zu erzielen. Insbesondere wäre es versuchenswert die Modifikationen in Abhängigkeit davon durchzuführen, ob sich die betrachtete Gitterzelle in der Mitte oder am Rand des traversierten Würfels befindet. Denn in der Mitte des Würfels ist die Warp-Ausführungseffizienz hoch, während sie am Rand niedrig ist. Ein weiterer interessanter Ansatz die Warp-Ausführungseffizienz bei den Fixed-Radius-Near-Neighbors-Berechnungen zu erhöhen wäre es die Partikel innerhalb einer Gitterzelle noch einmal entlang der X-Achse zu sortieren.

- **Software-Pipelining:** Es zeigte sich bei den finalen Untersuchungen, dass die Performance immer noch durch Latenzen limitiert ist. Ebenfalls zeigte sich in dieser Arbeit, dass sich durch das Software-Pipelining die Performance deutlich erhöhen ließ. Deshalb wäre es versuchenswert noch weitere Versionen vom Software-Pipelining zu probieren.

- **Weiteres Autotuning:** Bei dem Autotuning der Occupancy zeigte es sich, dass es eine einfach zu implementierende Möglichkeit ist, um die Performance zu erhöhen. Des Weiteren zeigte es sich in den vorhergegangenen Untersuchungen oft, dass der optimale Algorithmus und dessen optimale Parameter von dem Fixed-Radius-Near-Neighbors-Problem selbst abhängig sind. Zudem sind sie ebenfalls von der GPU abhängig. Deshalb könnte eine SPH-Simulation den Autotuning-Ansatz ausbauen um den besten Algorithmus und dessen Parameter automatisch zu bestimmen.

- **Weitere Untersuchungen mit komplexeren Fixed-Radius-Near-Neighbors-Problemen:** Die Untersuchungen in diesem Kapitel wurden nur mit dem einfachen Shepard-Kernel und mit dem etwas komplexeren Kernel für die Berechnung der zeitlichen Änderungen durchgeführt. Es gibt jedoch auch SPH-Simulationen wie der Dual-SPHysics-Simulator aus [JDa], die deutlich komplexere Berechnungen als Fixed-Radius-Near-Neighbors-Problem durchführen. Diese Probleme besitzen einen größeren Workingset für automatische Variablen, sowie mehr globale Speicherzugriffe und Recheninstruktionen in der problemspezifischen Auswertungsfunktion. Dadurch besitzen sie eine geringere optimale Occupancy, wodurch sie wieder stärker durch Latenzen gebunden sind. Zudem sind bei diesen Problemen Optimierungen der Warp-Ausführungseffizienz wichtiger.

- **Unterschiedliche Anordnung von den 3D-Vektoren der Partikeldaten im globalen Speicher:** Bisher wurde angenommen, dass ein Float4-Array für das Abspeichern von 3D-Vektoren am performantesten ist, da hierbei die GPU den Vektor mit einem einzigen 16-Byte-Ladebefehl aus dem DRAM laden kann. Alternativ könnte die SPH-Simulation die 3D-Vektoren als drei Float-Arrays, als ein Float3-Array oder als ein Float-Array und ein Float2-Array im globalen Speicher der GPU anordnen. Dabei wäre es interessant die genauen Auswirkungen dieser unterschiedlichen Anordnungen auf die Performance zu untersuchen.

- **Ausnutzen der Symmetrie:** Wie in Punkt 4.1 bereits angesprochen, so sind die Berechnungen der Fixed-Radius-Near-Neighbors-Probleme meist zwischen zwei Partikeln symmetrisch. Obwohl in der Literatur wie dem Artikel [GGb] von dem Ausnutzen dieser Symmetrie auf GPUs im Allgemeinen abgeraten wird, so scheinen Versuche diesbezüglich dennoch lohnenswert zu sein. Denn eine solche Optimierung könnte fast die Hälfte der Rechenzeit einsparen. Zudem sind die Atomic-Units auf modernen GPUs im Vergleich zu älteren GPUs extrem performant. Aus dem Grund könnten sie eventuell bei den atomaren Operationen, die eine solche Optimierung benötigen würde, nicht extrem stark limitieren. Letztendlich wäre es hier wiederum ein weiterer Verbesserungsansatz die Zahl der atomaren Operationen geschickt über den Informationsaustausch über Shared-Memory oder Warp-Shuffle-Funktionen zu reduzieren.
- **Untersuchung auf anderen GPUs:** GPUs unterscheiden sich stark in ihrem Hardwareaufbau. So gilt beispielhaft bei AMD-GPUs allgemein, dass sie mehr Register besitzen und mehr Threads gleichzeitig bearbeiten können als NVIDIA GPUs. Auf diese Weise können Latenzen prinzipiell besser überbrückt werden. Allerdings erhöht dies auch den Workingset des globalen Speichers deutlich, wodurch sich die Cache-Hitrate verschlechtert. Erschwerend kommt hinzu, dass AMD-GPUs kleinere Caches besitzen als NVIDIA-GPUs. Des Weiteren ist die Warp-Größe (Wavefront-Größe in der AMD-Nomenklatur) mit 64 Threads doppelt so groß. Aus dem Grund sind auf AMD-GPUs Optimierungen der Warp-Ausführungseffizienz wichtiger. Da die Performance der Fixed-Radius-Near-Neighbors-Berechnungen sowohl durch die Latenzen als auch durch die Warp-Ausführungseffizienz reduziert wird, wären Untersuchungen auf einer solchen GPU interessant.

4.10 Zeitintegration

Zuletzt soll auf die Zeitintegration eingegangen werden. Bei der Zeitintegration handelt es sich bei beiden Teilschritten des Prädiktor-Korrektor-Verfahrens algorithmisch gesehen um ein einfaches Streaming-Problem, weshalb es auch durch einen einfachen Streaming-Algorithmus auf der GPU implementiert worden ist. Deswegen wird in der Implementierung bei beiden Teilschritten jeweils ein Kernel mit einem Thread pro Partikel gestartet. Jeder Thread liest zunächst die Strömungsgrößen sowie die Änderungen seines Partikels aus dem globalen Speicher ein. Anschließend wird die Zeitintegration berechnet und das Ergebnis zurückgeschrieben. Das Einlesen und Zurückschreiben findet bezüglich der Thread-IDs komplett sequentiell statt, weshalb das Zugriffsmuster ein vollständiges Coalescing erlaubt. Das Kernel für den ersten Zeitschritt des Prädiktor-Korrektor-Verfahrens ist beispielhaft in Abbildung 4.34 zu sehen.

Die Kernel der Zeitintegration wurden in Abbildung 4.35 gebenchmarkt. Beide Kernel der Zeitintegration zeichnen sich dadurch aus, dass sie extrem wenige arithmetische Instruktionen im Vergleich zur benötigten Speicherbandbreite besitzen. Dadurch sind sie stark durch die Speicherbandbreite limitiert. Dies zeigt sich auch in den Messungen. Denn beide Kernel benötigen in etwa 83% der Speicherbandbreite. Dies stellt bei den regulären Zugriffsmustern mit kompletten Coalescing das Maximum dar, das die GPU erreichen kann. Aus diesen Gründen ist die Auslastung der CUDA-Cores mit 27% und 18% gering.

Um bei solchen durch die Speicherbandbreite limitierten Kernels die Performance zu erhöhen könnte eine Optimierung den Ansatz aus [VV] verfolgen. Dabei versucht der Ansatz die

```
__global__ void PredictorCorrectorFirstSubStep(ParticleGrid Grid, ParticleData Particles)
{

  int PartID = blockIdx.x * blockDim.x + threadIdx.x;

  float4 Changes = Particles.Changes[PartID];
  float3 Acceleration = makefloat3(Changes);
  float DensityChange = Changes.w;

  float3 PositionAndDensityT = Particles.PositionsAndDensitiesT[PartID];

  float3 PositionT = make_float3([PositionAndDensityT);
  float3 VelocityT = make_float3(Particles.Velocities[PartID]);
  float DensityT = PositionAndDensityT.w

  float3 PositionTHalf = PositionT + 0.5f*DT*VelocityT;
  float3 VelocityTHalf = VelocityT + 0.5f*DT*Acceleration;
  float DensityTHalf = DensityT + 0.5f*DT*DensityChange;

  Particles.PositionsAndDensitiesTHalf[PartID] = make_float4(PositionTHalf, DensityTHalf);
  Particles.VelocitiesTHalf[PartID] = make_float4(VelocityTHalf);
}
```

Abbildung 4.34: Kernel für den ersten Teilschritt des Prädiktor-Korrektor-Verfahrens

	Erster Prediktor-Korrektor-Teilschritt	Zweiter Prediktor-Korrektor-Teilschritt
Laufzeit	1.4 ms	1.7 ms
DRAM-Auslastung	83%	82%
CUDA-Core-Auslastung	27%	18%

Abbildung 4.35: Benchmark für die Kernel der Zeitintegration

Occupancy für einen höheren ILP zu reduzieren, um die Speicherbandbreite besser auszulasten. Dafür müsste die SPH-Simulation das Kernel so umgestalten, dass jeder Thread mehrere Partikel gleichzeitig verarbeitet. Eine weitere Möglichkeit wäre es den zweiten Teilschritt mit demjenigen Kernel, welches die zeitlichen Änderungen zum Halbzeitschritt berechnet, zu kombinieren. Denn dieses Kernel wäre selbst nach dem Hinzufügen der Zeitintegration nicht durch die Speicherbandbreite limitiert, wodurch die SPH-Simulation insgesamt die Rechenleistung der GPU besser ausnutzen könnte. Jedoch würde sich durch die Optimierung der Cache-Workingset des Kernels der zeitlichen Änderungen vergrößern, wodurch die Berechnungen des Kernels ebenfalls teurer werden könnten. Bei dem ersten Teilschritt wäre dieser Ansatz allerdings nicht möglich, weil zuerst der maximale Zeitschritt berechnet werden muss. Da die Zeitintegration mit 1.4 ms für den ersten Teilschritt und 1.7 ms für den zweiten Teilschritt nur circa 2% der Gesamtlaufzeit ausmacht, besitzen diese Optimierungen einen geringen Stellenwert.

4.11 Fazit

In diesem Punkt soll eben kurz das Fazit des Kapitels gezogen werden. So behandelte dieses Kapitel, wie eine effiziente SPH-Simulation für eine einzige GPU in dieser Arbeit realisiert worden ist. Dafür wurden zunächst die implementierten SPH-Algorithmen vorgestellt. Des Weiteren wurden die einzelnen Teilschritte der Algorithmen, nämlich das Bauen des Gitters, das Berechnen der Fixed-Radius-Near-Neighbors-Probleme und die Zeitintegration vorgestellt, optimiert und untersucht. Dabei zeigte es sich zunächst, dass die Berechnung der Fixed-Radius-Near-Neighbors-Probleme am zeitaufwändigsten ist, während das Bauen des Gitters oder die Berechnung der Zeitintegration nur einen Bruchteil der gesamten Laufzeit benötigen. Für das Bauen des verwendeten Dynamic-Vector-Gitters wurden zwei Verfahren vorgestellt, nämlich das Counting-Sort-Verfahren und das Radix-Sort-Verfahren. Bei den Untersuchungen zeigte sich, dass das Counting-Sort-Verfahren performanter ist. Zudem ist die Konstruktion des Gitters beim Counting-Sort-Verfahren stark durch die Speicherbandbreite limitiert. Für die Berechnungen der Fixed-Radius-Near-Neighbors-Probleme wurde weitestgehend der Linked-Cell-Ansatz verwendet und optimiert. Bei den Optimierungen war es das Ziel, die Laufzeit durch die Reduktion der benötigten Rechenoperationen, die Erhöhung der Warp-Ausführungseffizienz und durch eine Verringerung der Latenzen zu senken. Trotz der Optimierungsversuche ist die Performance weiterhin durch Latenzen limitiert und die Warp-Ausführungseffizienz immer noch niedrig. Deshalb kann die GPU ihre Rechenleistung nur schlecht ausnutzen. Dennoch ließ sich durch die Optimierungen die Laufzeit der Fixed-Radius-Near-Neighbors-Probleme deutlich reduzieren. Allerdings ist das Fixed-Radius-Near-Neighbors-Problem komplex, so dass noch viel Optimierungspotential besteht. Deshalb nannte das Kapitel auch viele Optimierungsansätze für die Berechnung des Fixed-Radius-Near-Neighbors-Problems. Insgesamt verdeutlicht das Kapitel, dass bei der GPU-Programmierung viel Optimierungspotential besteht. Letztendlich ist es wichtig anzumerken, dass ein Großteil der Diskussionen in diesem Kapitel bezüglich des Fixed-Radius-Near-Neighbors-Problems nicht nur für eine SPH-Flüssigkeitssimulation sondern für die anderen Anwendungsfälle dieses Problems, wie zum Beispiel für die Moleküldynamiksimulation, relevant ist.

Prinzipiell wäre es ebenfalls interessant, die finale Version der Arbeit mit einer anderen SPH-Simulation zu vergleichen. Als Vergleichsprogramm soll der Dual-SPHysics-Simulator, welcher eine SPH-Simulation mit CUDA durchführt, verwendet werden. Ein direkter Laufzeitvergleich scheint jedoch nicht sinnvoll, da der Vergleich hierfür die gleichen problemspezifischen Berechnungen auf der gleichen Hardware mit den gleichen Eingangsdaten und Parametern untersuchen müsste. Deshalb werden hier nur die Features verglichen. So besitzt dieser Dual-SPHysics-Simulator fortgeschrittenere physikalische SPH-Berechnungen für die SPH-Simulation, während die Simulation dieser Arbeit nur einfache grundlegende Berechnungen beherrscht. Ebenfalls unterstützt er fortgeschrittene Randbedingungen, welche in der Implementierung dieser Arbeit nicht möglich sind. Zudem ist die Eingabe und Ausgabe, wie das Laden und Abspeichern einer Szene aus Dateien, des Dual-SPHysics-Simulators deutlich weiter ausgebaut.

Als hierarchische Datenstruktur verwendet der Dual-SPHysics-Simulator ebenfalls ein Dynamic-Vector-Gitter. Für dessen Konstruktion besitzt der Dual-SPHysics-Simulator eine Implementierung des Sortierens mit dem Thrust-Radix-Sort, welche dem Counting-Sort dieser Arbeit unterlegen ist. Zudem stellt der Sortieralgorithmus vom Dual-SPHysics-Simulator keine Monotonie der Partikelstartindexe im Gitter sicher, wodurch die Gittertraversierung erschwert wird.

Für die Gittertraversierung verwendet der Dual-SPHysics-Simulator ebenfalls den Linked-Cell-Ansatz. Der Linked-Cell-Ansatz des Dual-SPHysics-Simulator besitzt eine Schleifenstruktur, welche ein Gemisch aus der First-Approach-Version dieser Arbeit und derjenigen Version mit herausoptimierter X-Schleife ist. So konnte der Dual-SPHysics-Simulator die X-Schleife bei der Gittertraversierung wegen der fehlenden Monotonie der Partikelstartindexe nicht heraus optimieren. Allerdings dient die X-Schleife in diesem Simulator nur dazu einen Partikelstartindex und einen Partikelendindex zu finden. Über die Partikel zwischen den beiden Indexen iteriert der Dual-SPHysics-Simulator dann auch in der Y-Schleife, wobei er für jedes Nachbarpartikel wieder problemspezifischen Berechnungen ausführt. Da die Gittertraversierung dieser Arbeit keine X-Schleife mehr besitzt, ist sie etwas performanter. Des Weiteren wird bei der Gittertraversierung im Dual-SPHysics-Simulator ebenfalls die Verkleinerung der Gitterzellengröße erlaubt. Jedoch fehlen dort sämtliche weitere in dieser Arbeit vorgestellten Optimierungen. Zudem ist die Gittertraversierung des Dual-SPHysics-Simulator im Gegensatz zu der Gittertraversierung in dieser Arbeit nicht modular gehalten. Deshalb gibt es dort sehr viel sich wiederholenden Quelltext.

Dadurch lässt sich insgesamt sagen, dass obwohl die Implementierung dieser Arbeit vom Umfang her geringer ist, sie diverse performancesteigernde Ansätze besitzt, welche in dem DualSPHysics-Simulator nicht vorhanden sind.

5 Visualisierung

5.1 Übersicht möglicher Visualisierungstechniken für eine SPH-Simulation

In diesem Kapitel soll auf die implementierte Visualisierung eingegangen werden. So gibt es prinzipiell viele verschiedene Möglichkeiten eine SPH-Simulation zu visualisieren. Deswegen wird zuerst einleitend eine Übersicht über diese Möglichkeiten gegeben. Für eine Visualisierung lässt sich die Flüssigkeit gemäß der SPH-Interpolation als räumliche Funktion der Flüssigkeitsattribute interpretieren. Aus diesem Grund bieten sich für eine SPH-Simulation prinzipiell sämtliche Volumenrendertechniken an. Diese Techniken können sowohl mit abstrakten wissenschaftlichen Shadingtechniken kombiniert werden, so dass der Betrachter die Areas of Interest möglichst gut erkennen kann, als auch mit realistischen Shadingtechniken, so dass die Flüssigkeit für einen Betrachter wie eine echte Flüssigkeit aussieht.

Anschließend soll auf diejenigen Techniken für eine SPH-Visualisierung eingegangen werden, die hauptsächlich in der Literatur vorgeschlagen werden:

- **Indirekte Techniken:** Diese Techniken, wovon eine zum Beispiel der Artikel [JY] vorstellt, extrahieren eine Oberflächengeometrie der Flüssigkeit. Dies erfolgt meist durch den Marching-Cubes-Algorithmus. Dann verwendet die Technik die Oberflächengeometrie um diese als solches durch Raytracing oder durch Rasterisierung zu zeichnen. Die Definition der Oberfläche der Flüssigkeit erfolgt in vielen Fällen über das normalisierte Dichtefeld. Da der Marching-Cubes-Algorithmus ein katharisches Gitter voraussetzt, muss die Technik die räumliche Funktion zuvor durch ein solches Gitter abtasten.
- **Direkte Techniken:** Sie zeichnen das Volumen direkt, ohne zuvor eine Oberflächengeometrie zu extrahieren. Sie lassen sich noch einmal unterteilen in:
  - **Volumetrisches Raytracing oder Raycasting:** Diese Verfahren, wovon eins beispielhaft im Artikel [RF] verwendet wird, senden ausgehend von der Kamera Strahlen durch die Simulationsdomäne aus und werten für das Shading in regelmäßigen Abständen entlang des Strahls die volumetrische Funktion aus. Um das Auswerten der Funktion zu vereinfachen, tasten die Techniken die volumetrische Funktion ebenfalls oft zuvor durch ein Gitter ab. Dadurch muss das Raytracing oder Raycasting nur noch die Attribute innerhalb einer Gitterzelle interpolieren, ähnlich des Particle-In-Cell Ansatzes.
  - **Splatting-Techniken:** Diesen Techniken, wovon eine zum Beispiel im Artikel [WL] vorgestellt wird, zeichnen für jedes Partikel per Rasterisierung eine Form wie einen Kreis, eine projizierte eine Kugel oder ein projiziertes Ellipsoid auf den Bildschirm.

5.2 Übersicht über die implementierte Visualisierung

Nach der Vorstellung der verschiedenen möglichen Visualisierungstechniken soll diejenige Visualisierung vorgestellt werden, die letztendlich in dieser Arbeit implementiert worden ist. So wird in dieser Arbeit eine Ellipsoid-Splatting-Technik als Visualisierung verwendet. Denn sie ist leicht zu implementieren. Zudem lassen sich ihre Berechnungen später leicht innerhalb eines Clusters aufteilen. Dabei stammt die Grundidee des Splattings aus den Artikeln [WL]

und [MM]. Des Weiteren soll die Visualisierung nicht nur die Flüssigkeit selbst, sondern auch das Höhenfeld-Terrain zeichnen, das als Randbedingung für die SPH-Simulation dient.

Um die Flüssigkeit zu visualisieren zeichnet die Ellipsoid-Splatting-Technik für jedes Partikel der SPH-Simulation ein projiziertes Ellipsoid auf den Bildschirm. Auf diese Weise erstellt sie von der Kamera aus gesehen ein Tiefenbild und ein Dickebild. Das Tiefenbild dient als Approximation der Oberfläche der Flüssigkeit anhand welcher später das Oberflächen-Shading durchgeführt wird. Das Dickebild wird später benötigt um die Durchsichtigkeit der Flüssigkeit zu berechnen. Allerdings muss die Visualisierung vor dem Splatting für jedes Partikel die Hauptachsen seines Ellipsoides durch eine Hauptkomponentenanalyse berechnen. Damit ein Betrachter die durchsichtige Flüssigkeit nicht durch das undurchsichtige Terrain hindurchsieht, muss vor dem Splatting zusätzlich das Terrain in einen Tiefenbuffer gezeichnet werden. Der Tiefenbuffer wird dann beim Splatting der Ellipsoide für einen Tiefentest verwendet. Gleichzeitig wird die Farbe des Terrains in ein weiteres Bild gezeichnet. Durch das Splatting entsteht im Tiefenbild jedoch keine glatte Oberfläche, weil jedes der Ellipsoide jeweils eine kleine Wölbung im Tiefenbild bewirkt. Damit der Betrachter der Visualisierung dennoch den Eindruck einer glatten Flüssigkeitsoberfläche erhält, muss das Tiefenbild vor dem Oberflächenshading geglättet werden. Hierfür verwendet die Visualisierung das sogenannte Screen-Space-Curvature-Flow-Verfahren aus dem Artikel [WL]. Schließlich kann die Visualisierung das geglättete Tiefenbild, das Dickebild, und das Farbbild des Terrains zu dem finalen Renderergebnis zusammensetzen. Dabei findet hier erst das eigentliche Shading der Flüssigkeit statt, weshalb es sich bei diesem Verfahren um eine Deferred-Shading-Technik handelt.

Eine Übersicht über die so eben erklärte Vorgehensweise dieser Technik wird in Abbildung 5.1 gezeigt. Demnach lässt sich diese Technik in folgende Unterpunkte gliedern:

- **Punkt 5.3:** Berechnung der Hauptachsen der Ellipsoide per Hauptkomponentenanalyse
- **Punkt 5.4:** Erstellen eines Tiefenbilds und Dickebilds durch das Splatting der Ellipsoide
- **Punkt 5.5:** Glättung des Tiefenbild durch den Screen-Space-Curvature-Flow
- **Punkt 5.6:** Zusammensetzen des geglätteten Tiefenbilds der Flüssigkeit, des Dickebilds der Flüssigkeit und des Farbbilds des Terrains zum Endergebnis beim finalen Shading

Hinzu kommt das Erstellen eines Farbbilds und eines Tiefenbilds durch das Zeichnen des Terrains. Für das Zeichnen des Terrains werden die Texturen aus [TexTer] verwendet. Da der Zeichenvorgang eines Höhenfeld-Terrains trivial ist, wird er in folgender Arbeit nicht weiter erklärt. Die übrigen Schritte sollen im Folgenden jedoch im Detail erläutert werden.

5.3 Bestimmen der Hauptachsen

In diesem Punkt soll vorgestellt werden, wie die Visualisierung die Hauptachsen der Ellipsoide für das Ellipsoid-Splatting durch eine Hauptkomponentenanalyse bestimmt. Hierfür wird das Verfahren aus [JY] verwendet. Das Verfahren berechnet zunächst die Anisotropie der Partikelverteilung um jedes Partikel herum. Für diese Berechnung wird eine gewichtete Version der

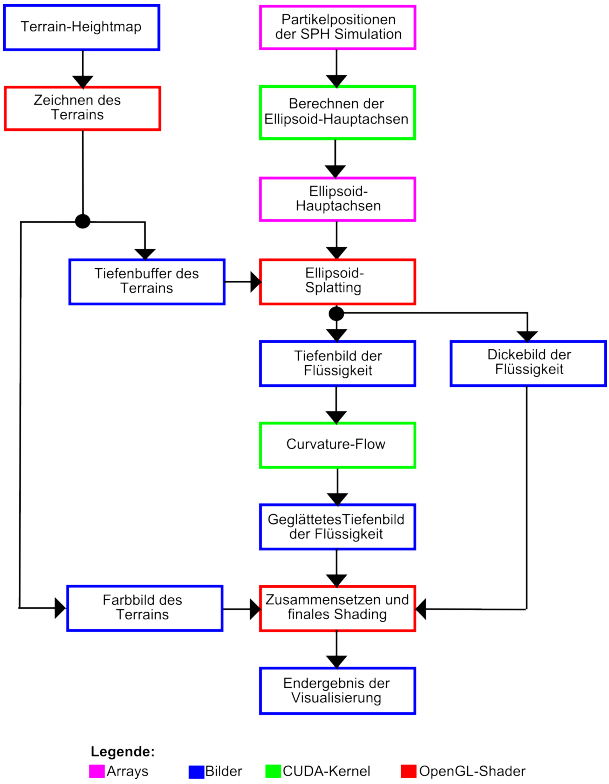


Abbildung 5.1: Überblick über die Visualisierung

Hauptkomponentenanalyse (WPCA) verwendet. Die WPCA bestimmt zunächst die gewichteten Mittel der Partikelpositionen $\vec{r}^w$:

$$\vec{r}_i^w = \frac{\sum_j w_{ij}\vec{r}_j}{\sum_j w_{ij}} \tag{5.1}$$

Dabei ist w_{ij} eine isotrope Gewichtungsfunktion mit dem maximalen Einflussradius $r_{\text{max,iso}}$:

$$w_{ij} = \begin{cases} 1 - \left(\frac{|\vec{r}_{ij}|}{r_{\text{max,iso}}}\right)^3 & \text{falls } |\vec{r}_{ij}| < r_{\text{max,iso}} \\ 0 & \text{falls } |\vec{r}_{ij}| \geq r_{\text{max,iso}} \end{cases} \tag{5.2}$$

Anschließend berechnet die WPCA eine Kovarianzmatrix C:

$$C_i = \frac{\sum_j w_{ij}(\vec{r}_j - \vec{r}_i^w)(\vec{r}_j - \vec{r}_i^w)^T}{\sum_j w_{ij}} \tag{5.3}$$

Schließlich zerlegt die WPCA die Kovarianzmatrix C durch die Singulärwertzerlegung wie folgt:

$$C = R\Sigma R^T \tag{5.4}$$

Dabei ist Σ eine Diagonalmatrix, die aus den der Größe nach sortierten Eigenwerten λ_1,λ_2 und λ_3 von C besteht. R ist eine Rotationsmatrix, welche die Eigenvektoren von C in ihren Spalten besitzt. Die Wurzeln Eigenwerte werden nun als Längenverhältnis der Hauptachsen eines Ellipsoids interpretiert. Dadurch kann die Visualisierung mit R^T einen Punkt oder Vektor vom Weltkoordinatensystem ins Hauptachsensystem des Ellipsoids transformieren. Die tatsächlichen Längen der Hauptachsen errechnet die Visualisierung dadurch, indem sie das Längenverhältnis mit einem konstanten Faktor multipliziert. Um die maximale Anisotropie zu beschränken wird zudem das maximale Längenverhältnis der Hauptachsen ebenfalls auf einen bestimmten Wert beschränkt. Dafür hält die Visualisierung die längste Hauptachse konstant und vergrößert die kleineren Hauptachsen dementsprechend. Des Weiteren kann es vorkommen, dass ein Partikel wenig Nachbarn besitzt, wodurch die WPCA kein zufriedenstellendes Ergebnis mehr erzielen kann. In diesem Fall setzt die Visualisierung R gleich der Identitätsmatrix und die Länge der Hauptachsen auf einen festen Wert. Dadurch wird das Partikel als Kugel gezeichnet.

Schließlich soll auf die Implementierung dieser Berechnungen eingegangen werden. So handelt es sich bei Gleichung 5.1 und 5.3 wegen des beschränkten Einflussradius der isotropen Gewichtungsfunktion jeweils um ein Fixed-Radius-Near-Neighbors-Problem, welche nach einander berechnet werden müssen. Beide Berechnungen wurden mit der generischen Form des Fixed-Radius-Near-Neighbors-Problems implementiert, welche in Punkt 4.9 vorgestellt wurde. Somit ergeben sich zwei Fixed-Radius-Near-Neighbors-Kernel. Auf weitere problemspezifische Optimierungen dieser beiden Fixed-Radius-Near-Neighbors-Kernel wurde jedoch verzichtet. Beide Kernels verwenden für den maximalen Einflussradius der isotropen Gewichtungsfunktion $r_{\text{max,iso}}$ den gleichen Wert wie für den maximalen Einflussradius r_{max} des Smoothing-Kernels der SPH-Simulation. Die Traversierung des Gitters muss deshalb mit der selben Traversierungsdistanz wie bei der SPH-Simulation erfolgen. Das erste Kernel berechnet die gewichteten Mittel aus Gleichung 5.1. Das nächste Kernel verwendet die gewichteten Mittel um die Kovarianzmatrizen C gemäß Gleichung 5.3 zu berechnen. Anschließend führt es eine Singulärwertzerlegung aus. Für die Singulärwertzerlegung wurde der Quelltext aus [CB] nach CUDA portiert. Die Singulärwertzerlegung verwendet das Kernel wiederum um für jedes Partikel eine Matrix K über die Formel $K = LR^T$ zu berechnen. Dabei ist die Matrix L eine Diagonalmatrix mit den inversen Längen der nach ihrer Größe sortierten Hauptachsen des Ellipsoids. Denn die Matrix K wird, wie im nächsten Punkt erklärt, beim Splatting benötigt, um einen Punkt beziehungsweise Vektor ins Einheitskugelsystem des Ellipsoids zu transformieren. Deshalb ist die Matrix K das Endergebnis des Kernels. Schließlich speichert das Kernel die Matrix K und die Position des Ellipsoids $\vec{r}$ in drei Float4 Arrays ab, die auf OpenGL-Buffer verweisen. Somit braucht das Ellipsoid-Splatting in der finalen Version der Implementierung 48 Byte an zusätzlichen Speicherplatz pro Partikel.

Der zusätzliche Speicherplatzverbrauch ließe sich jedoch noch einmal komplett durch Optimierung vermeiden. Diese Art der Optimierung wurde allerdings in der finalen Version der Arbeit nicht mehr implementiert. So lassen sich die Lage und die Länge der Hauptachsen aus der Matrix K auch über drei Eulerwinkel und drei Skalare beschreiben, die insgesamt nur 24 Byte groß sind. Dadurch könnte die Visualisierung hier bereits 12 Byte einsparen. In der aktuellen Version der Arbeit funktioniert die für die Konstruktion des Gitters benötigte Permutation im Punkt

4.8.1.3 mit CUDA-Zeiger auf OpenGL-Buffer nicht, da die Permutation die Zeiger vertauscht. Deshalb liegen die Partikelpositionen im globalen Speicher, der über CUDA alloziert wurde. Aus dem Grund kann OpenGL nicht direkt auf die Partikelpositionen zugreifen. Wäre ein direkter Zugriff in der aktuellen Implementierung möglich, dann würde die Visualisierung sich zusätzlich weitere 12 Byte pro Partikel für die Mittelpunkte der Ellipsoide einsparen. Des Weiteren berechnet die Implementierung dieser Arbeit immer die Visualisierung zu Beginn eines Zeitschritts. Deswegen wird zu dem Zeitpunkt der Visualisierung weder das Float4-Array der Positionen und Dichten noch das Float4-Array der Geschwindigkeit zum Halbzeitschritt benötigt. Würde die Implementierung der Arbeit für OpenGL-Objekte Zeigeraliasing unterstützen, so könnten für die Eulerwinkel und die Längen eben diese beiden Buffer verwendet werden. Jedoch unterstützt die aktuelle Implementierung auch kein Zeigeraliasing auf OpenGL-Buffer, da hier das Vertauschen bei der Permutation ebenfalls nicht mehr funktionieren würde. Durch diese Optimierungen würde der zusätzliche Speicherplatzverbrauch der Visualisierung insgesamt komplett entfallen.

5.4 Splatting der Ellipsoide

Nachdem die Hauptachsen der Ellipsoide berechnet worden sind, kann die Visualisierung mit dem Ellipsoid-Splatting selbst beginnen. Das Ziel des Splattings ist es, ein Tiefenbild und ein Dickebild von der Kamera aus gesehen zu erstellen, die die Visualisierung beide später für das Shading des Wassers benötigt. Der Vorteil vom Ellipsoid-Splatting gegenüber dem einfacheren Sphere-Splatting liegt darinnen, dass sich die Oberfläche der Flüssigkeit durch Ellipsoide besser approximieren lässt. Dies ist in Abbildung 5.2 zu sehen. Das Splatting selbst findet mit OpenGL statt.

Zunächst wird erläutert, wie die Visualisierung das Dickebild erstellt. Für das Zeichnen des Dickebilds müssen alle Fragmente, die sich vor dem Terrain befinden, passieren. Deshalb verwendet die Visualisierung für den OpenGL-Zeichenaufruf einen Tiefentest mit dem Tiefenbuffer des Terrains und deaktiviert das Hineinschreiben in diesen. Dann erfolgt der Zeichenaufruf, bei welchem für jedes Partikel ein Punkt mit einem speziellen Shader gezeichnet wird. Während in diesem Shader der Vertex-Shader die Position und die Hauptachsen des Partikels nur per Pass-Through an dem Geometry-Shader weiterreicht, erstellt der Geometry-Shader ein Quadrat beziehungsweise ein Bill-Board als Proxy-Geometry. Der Fragment-Shader führt dann per Raycasting eine Schnittpunktberechnung zwischen Ellipsoid und Strahl aus. Damit auch alle Fragmente, deren Strahl sich mit dem Ellipsoid schneidet, erstellt werden, muss das Quadrat eine Bounding-Box des Ellipsoids im Screen-Space sein. Dies wird erzielt indem der Geometry-Shader ein Quadrat gemäß Abbildung 5.3 aus der Position der Kamera, des Ellipsoids und der Länge der größten Hauptachse konstruiert.

Die Schnittpunktberechnung zwischen einem Strahl und Ellipsoid lässt sich am einfachsten dadurch berechnen, dass der Strahl zuerst ins Hauptachsensystem des Ellipsoids transformiert wird. Dafür muss er zuerst um $-\vec{r}$ verschoben und dann mit R^T rotiert werden. Wird der Strahl danach noch mit der inversen Länge der Hauptachsen skaliert, so ist in dem daraus resultierenden Koordinatensystem das Ellipsoid die Einheitskugel. Diese Rotation und Skalierung wird über eine Transformation mit der Matrix K erreicht. Schließlich muss nur noch die sehr einfache Schnittpunktberechnung zwischen dem transformierten Strahl und der Einheitskugel ausgeführt werden, welche im Wesentlichen das Lösen einer quadratischen Gleichung ist.

Da die Transformation des World-Space-Koordinatensystems ins „Einheitskugelsystem“ eine

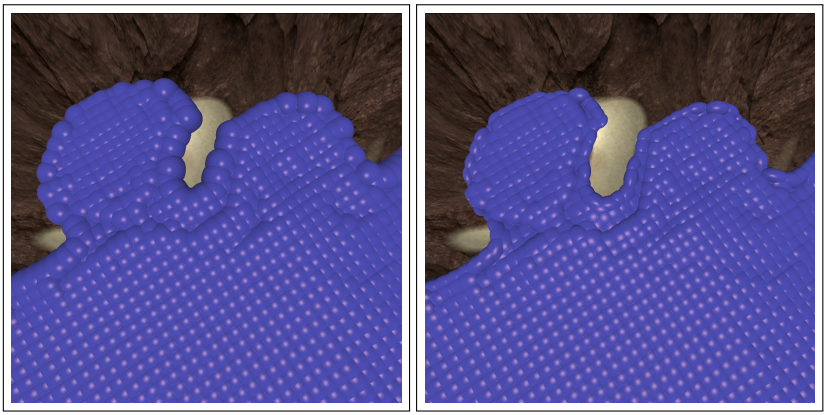


Abbildung 5.2: Unterschiede zwischen Sphere- und Ellipsoid-Splatting: Das linke Bild zeigt deutlich, dass die Kugeln den Rand der Flüssigkeit schlechter approximieren als die Ellipsoide auf dem rechten Bild.

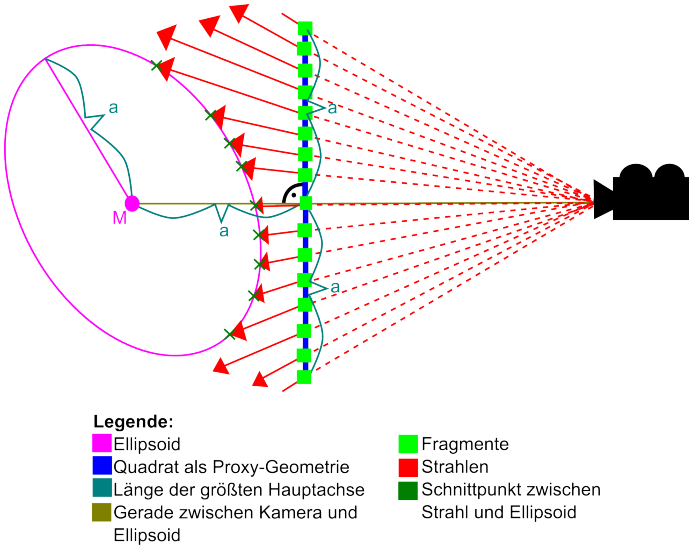


Abbildung 5.3: Zeichnen eines Ellipsoids per Rasterisierung

affine Abbildung ist, verläuft sie innerhalb des Quadrats linear. Deshalb kann sich die Visualisierung den im Punkt 2.2 erklärten Trick zu Nutze machen und den Strahl bereits im Geometry-Shader ins Einheitskugelsystem transformieren. Dann übergibt der Geometry-Shader den so transformierten Strahl der Rasterization-Stufe zur Interpolation. Ebenfalls übergibt der Geometry-Shader noch den ins Kamera-Koordinatensystem transformierten Strahl zur Interpolation an die Rasterization-Stufe. Durch diese Interpolationen muss der Fragment-Shader hauptsächlich nur noch die quadratische Gleichung lösen. Besitzt die quadratische Gleichung für ein Fragment keine Lösung, so verfehlt der Strahl das Ellipsoid. In diesem Fall verwirft der Fragment-Shader das Fragment. Ansonsten besitzt die quadratische Gleichung zwei Skalare als Lösung. Das kleinere von beiden Skalaren ist das Eintrittsskalar des Strahls in das Ellipsoid, während das größere von beiden das Austrittsskalar ist. Über die Differenz beider Skalare multipliziert mit der Länge der Strahlenrichtung im Kamerakoordinatensystem lässt sich die Dicke des Ellipsoids an der Stelle des Fragments bestimmen. Schließlich schreibt die GPU die so ermittelte Dicke per additiven Blending in das an den Framebuffer gebundene Dickebild zurück. Da das Fragment zunächst noch den zum Quadrat und nicht zum Ellipsoiden passenden Tiefenwert besitzt, muss der Fragment-Shader die Tiefe noch dementsprechend verändern. Dieses Verändern ist notwendig, damit der Tiefentest mit dem Terrain korrekt stattfinden kann. Der neue Tiefenwert lässt sich dabei leicht aus dem Produkt zwischen Eintrittsskalar und der Strahlenrichtung im Kamerakoordinatensystem berechnen.

Als Nächstes soll das Erstellen des Tiefenbilds vorgestellt werden. Dies läuft weitgehend analog ab, abgesehen davon, dass die GPU nun beim Zeichnen den Tiefenbuffer aktualisiert. Denn beim Tiefenbild dürfen nach dem Zeichenvorgang nur noch die vordersten Fragmente vorhanden sein. So sind der Vertex-Shader und der Geometry-Shader identisch. Ebenso löst der Fragment-Shader zuerst die quadratische Gleichung um ein Eintrittsskalar zu erhalten. Aus dem Eintrittskalar wird nun die Tiefe des Fragments berechnet. Danach verändert der Fragment-Shader die Tiefe seines Fragments dementsprechend und schreibt sie in das an den Framebuffer gebundene Tiefenbild zurück.

Schließlich sollen noch zwei kleinere weitere Optimierungen an diesem Splatting vorgestellt werden, welche diese Arbeit allerdings nur zum Teil implementierte. Beim Zeichnen von beiden Bildern gilt, dass der Fragment-Shader die Tiefe des Fragments verändert. Deshalb darf die GPU zunächst keine Early-Z-Rejection verwenden. Allerdings gilt wegen der Konstruktionsmethode der Proxy-Geometrie zusätzlich, dass der Fragment-Shader stets eine größere Tiefe als die ursprüngliche Tiefe des Quadrats zurückschreiben wird. Dies kann die Visualisierung ausnutzen, indem sie diese Tatsache im Quelltext des Fragment-Shaders angibt. Auf diese Weise darf die GPU dennoch die Early-Z-Rejection verwenden. Beim Dickebild ist der Nutzen davon jedoch beschränkt, da die GPU den Tiefenbuffer während des Zeichnens nicht aktualisiert. Deswegen kann die GPU sich nur Fragmente einsparen, welche hinter dem Terrain liegen. Beim Tiefenbild kann die GPU, da in diesem Fall der Tiefenbuffer beim Zeichnen aktualisiert wird, sich jedoch auch Fragmente einsparen, welche durch die Flüssigkeit verdeckt werden. Allerdings verlangt die Early-Z-Rejection um effizient zu funktionieren zusätzlich, dass die GPU die Objekte von vorne nach hinten zeichnet. Jedoch sind die Partikel nach dem Gitter sortiert, wodurch die GPU die Partikel auch in eben dieser Reihenfolge zeichnet. Deshalb ist diese Sortierung für die Early-Z-Rejection je nach Position der Kamera mal etwas besser, mal etwas schlechter, oder extrem schlecht. Die Visualisierung könnte dieses Problem vermeiden, indem sie die Partikel vor dem Zeichnen nach ihrem Tiefenwert sortiert oder zufällig permutiert. Letzteres wurde anhand einer statischen zufälligen Permutation kurzzeitig getestet, und erwies sich als gut Performance-steigernd. Da die Permutation allerdings nur statisch war, kam

sie nicht mit schwankenden Partikelanzahlen zurecht. Aus dem Grund wurde sie zunächst für die Cluster-Version verworfen und im Nachhinein nicht mehr in einer verbesserten Version implementiert.

Beim Erstellen des Dickebilds tritt dasjenige Problem auf, dass die GPU sehr viele Fragmente pro Pixel übereinander zeichnen muss. Dies lässt sich in diesem Fall auch nicht durch die Early-Z-Rejection vermeiden. Dadurch ist die benötigte Rechenzeit sehr stark von der Auflösung des Bildes abhängig. Deshalb spart sich die Visualisierung Rechenzeit ein, indem sie das Dickebild in einer niedrigeren Auflösung zeichnet. Hierfür muss allerdings zunächst der Tiefenbuffer des Terrains per Min- oder Max-Operation herunter skaliert werden. Da für das Dickebild eine grobe Approximation, wie in [WL] vorgeschlagen, reicht, wirkt sich die niedrigere Auflösung nur kaum auf die visuelle Qualität aber stark auf die Performance aus. Ebenso könnte die Visualisierung versuchen das Tiefenbild in einer niedrigeren Auflösung zu zeichnen. Würde die Visualisierung jedoch die Partikel umsortieren, damit die Early-Z-Rejection gut funktionieren könnte, so wäre der Nutzen beim Splatting selbst gering. Allerdings würden sich Vorteile beim Screen-Space-Curvature-Flow oder bei der Cluster-Version ergeben, da dort die Implementierung das Tiefenbild über das Netzwerk übertragen muss.

Zudem gilt anzumerken, dass das erstellte Bounding-Quadrat auf Grund dessen Konstruktionsmethode gemäß Abbildung 5.3 nicht optimal ist. Ist ein Ellipsoid nahe an der Kamera oder sind dessen Hauptachsen sehr anisotrop, so ist das erstellte Bounding-Quadrat schlecht. Deshalb werden wiederum zu viele Fragmente erstellt, die dann verworfen werden müssen. Der Nachteil wird aber dadurch relativiert, dass die meisten Ellipsoide sehr weit von der Kamera entfernt sind. Zusätzlich befinden sich die meisten Ellipsoide tief innerhalb der Flüssigkeit, weshalb sie in guter Näherung kugelförmig sind. Dennoch wäre es versuchenswert eine etwas bessere Konstruktionsmethode für das Bounding-Quadrat zu wählen, damit sich die GPU vielleicht ein paar Fragmente einsparen könnte.

Es wäre interessant Untersuchungen durchzuführen, wie das Splatting die Rechenleistung der GPU ausnutzt. Denn die Berechnungen in den Shadern sind einfach. Deshalb liegt die Vermutung nahe, dass die Performance des Splattings durch die Füllrate für Pixel von 33 Milliarden Pixel/s oder durch den Dreiecksdurchsatz von 5.9 Milliarden Dreiecke/s gebunden ist. Denn beide Werte sind im Vergleich zu den 4.5 Billionen SP-FLOPS vergleichsweise niedrig. Ob die Füllrate oder der Dreiecksdurchsatz limitieren wäre davon abhängig, wie groß die projizierten Ellipsoide auf dem Bildschirm gesehen sind oder wie viele Fragmente die Early-Z-Rejection passieren. Auf diese Weise scheint es insgesamt wahrscheinlich, dass die GPU beim Splatting ihre Rechenleistung nur schlecht ausnutzen kann. Es gibt aber keine bekannte Möglichkeit, ähnlich wie beim Visual-Profiler, für einen Shader ausführliche Metriken zu erstellen. Deshalb müsste eine solche Untersuchung mit diversen selbst geschriebenen Benchmarks ausgeführt werden. Wegen des beschränkten Umfangs der Arbeit sind diese Untersuchungen nicht mehr ausgeführt worden.

5.5 Berechnung des Screen-Space-Curvature-Flows

Anschließend muss die Visualisierung das Tiefenbild, das durch das Ellipsoiden-Splatting entstanden ist, glätten. Hierfür verwendet die Visualisierung das Verfahren des Screen-Space-Curvature-Flows aus dem Artikel [WL]. Da das Verfahren in dem Artikel bereits sehr ausführlich erklärt worden ist, sind folgende Erklärungen ebenfalls stark an dem Artikel angelehnt.

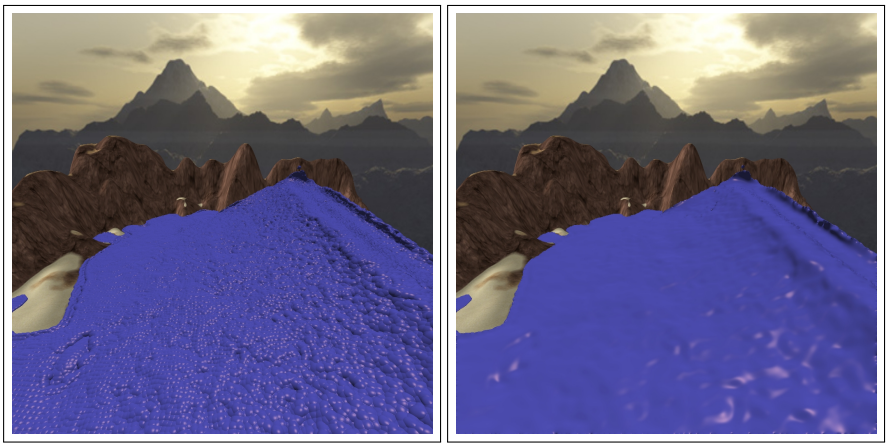


Abbildung 5.4: Glättung durch den Screen-Space-Curvature-Flow: Auf dem linken Bild wurde die Flüssigkeit ohne Screen-Space-Curvature-Flow mit einem Phong-Beleuchtungsmodell gerendert. Deutlich sind die Wölbungen der einzelnen Ellipsoide zu erkennen. Auf dem rechten Bild kann ein Betrachter, nachdem das Bild durch den Screen-Space-Curvature-Flow geglättet wurde, die Wölbungen der Ellipsoide nicht mehr erkennen.

So versucht eine echte Flüssigkeit auf Grund der Oberflächenspannung kleinere Wölbungen beziehungsweise Krümmungen in ihrer Oberfläche zu minimieren. Die Wölbungen der Flüssigkeitsoberfläche im Tiefenbild ergeben sich aber zwangsweise durch die Diskretisierung der Flüssigkeit durch Ellipsoide in Kombination mit dem Splatting. Deshalb versucht der Screen-Space-Curvature-Flow die Oberfläche der Flüssigkeit nachträglich zu glätten indem er die Krümmung im Tiefenbild minimiert. Der Effekt des Glättens ist in Abbildung 5.4 zu sehen. Für das Glätten bewegt der Screen-Space-Curvature-Flow die Tiefenwerte im Tiefenbild entlang der mittleren Krümmung H an deren Stelle nach vorne beziehungsweise nach hinten. Auf diese Weise entsteht folgende Differentialgleichung für die Tiefe z:

$$\frac{\delta z}{\delta t} = H \tag{5.5}$$

Die mittlere Krümmung ist über die Divergenz der Einheitsnormale $\vec{n}$ der Oberfläche definiert:

$$2H = \nabla \cdot \vec{n} \tag{5.6}$$

Des Weiteren lässt sich ein Pixel mit dem Index (x, y) und dem Tiefenwert z über die Invertierung der Projektion leicht in einen Punkt P im Kamerakoordinatensystem umrechnen:

$$P(x,y) = \begin{pmatrix} \frac{\frac{2x}{V_x} - 1}{F_x} \\ \frac{\frac{2y}{V_y} - 1}{F_y} \\ 1 \end{pmatrix} z(x,y) = \begin{pmatrix} W_x \\ W_y \\ 1 \end{pmatrix} z(x,y) \tag{5.7}$$

Dabei ist V_x die Auflösung entlang der X-Achse, V_y die Auflösung entlang der Y-Achse, und F_x sowie F_y die Brennweiten der Kamera bezüglich der X-Achse beziehungsweise der Y-Achse. Die Brennweiten lassen sich wiederum über das Frustrum der Kamera berechnen. So ist F_x die Steigung der rechten Frustrumebene $\frac{\delta x}{\delta z}$ im Kamerakoordinatensystem während F_y die Steigung der oberen Frustrumebene $\frac{\delta y}{\delta z}$ im Kamerakoordinatensystem ist.

Die Ableitungen von P im Bildschirmkoordinatensystem entlang der X- und entlang der Y-Achse beschreiben jeweils eine Tangente der durch das Tiefenbild definierten Flüssigkeitsoberfläche. Beide sind immer voneinander linear unabhängig wodurch sie eine Tangentialebene aufspannen. Dadurch lässt sich die Normale $\vec{n}$ selbst über das Kreuzprodukt der Ableitungen von P nach x und y nach berechnen:

$$\begin{aligned}\vec{n}(x,y) =& \frac{\delta P}{\delta x} \times \frac{\delta P}{\delta y} \\ =& \begin{pmatrix} C_x z + W_x \frac{\delta z}{\delta x} \\ W_y \frac{\delta z}{\delta x} \\ \frac{\delta z}{\delta x} \end{pmatrix} \times \begin{pmatrix} W_x \frac{\delta z}{\delta y} \\ C_y z + W_y \frac{\delta z}{\delta y} \\ \frac{\delta z}{\delta y} \end{pmatrix}\end{aligned} \tag{5.8}$$

Dabei ist $C_x = \frac{2}{V_x F_x}$ und $C_y = \frac{2}{V_y F_y}$. Das Ergebnis des Kreuzproduktes hängt nur über W_x und W_y von der Position des Pixels auf dem Bildschirm ab. Die Beiträge von W_x und W_y zum Endergebnis des Screen-Space-Curvature-Flows sind sehr klein und würden die Berechnungen sehr komplizierter machen. Deshalb setzt das Screen-Space-Curvature-Flow-Verfahren beides gleich null. Dadurch vereinfacht sich obige Gleichung zu:

$$\vec{n}(x,y) \approx \begin{pmatrix} C_x z \\ 0 \\ \frac{\delta z}{\delta x} \end{pmatrix} \times \begin{pmatrix} 0 \\ C_y z \\ \frac{\delta z}{\delta y} \end{pmatrix} = \begin{pmatrix} -C_y \frac{\delta z}{\delta x} \\ -C_x \frac{\delta z}{\delta y} \\ C_x C_y z \end{pmatrix} \tag{5.9}$$

Da die mittlere Krümmung über die Divergenz der Einheitsnormale definiert ist, muss das Verfahren die Normale schließlich normieren:

$$\vec{\tilde{n}} = \frac{\vec{n}(x,y)}{|\vec{n}(x,y)|} = \frac{(-C_y \frac{\delta z}{\delta x}, -C_x \frac{\delta z}{\delta y}, C_x C_y z)^T}{\sqrt{D}} \tag{5.10}$$

Mit folgendem D:

$$D = C_y^2 \left(\frac{\delta z}{\delta x}\right)^2 + C_x^2 \left(\frac{\delta z}{\delta y}\right)^2 + C_x^2 C_y^2 z^2 \tag{5.11}$$

Nun kann der Divergenz Operator auf die Einheitsnormale angewendet werden. Die Z-Komponente der Divergenz ist dabei immer Null, da z als Funktion von x und y gegeben ist. Deshalb verändert sie sich nicht, wenn x und y konstant gehalten werden. So gilt für die mittlere Krümmung:

$$2H = \frac{\delta \tilde{n}_x}{\delta x} + \frac{\delta \tilde{n}_y}{\delta y} = \frac{C_y E_x + C_x E_y}{D^{\frac{3}{2}}} \tag{5.12}$$

E_x und E_y sind:

$$E_x = \frac{1}{2}\frac{\delta z}{\delta x}\frac{\delta D}{\delta x} - \frac{\delta^2 z}{\delta x^2}D \tag{5.13}$$

$$E_y = \frac{1}{2}\frac{\delta z}{\delta y}\frac{\delta D}{\delta y} - \frac{\delta^2 z}{\delta y^2}D \tag{5.14}$$

Damit lässt sich H durch gegebene Größen ausdrücken. Letztendlich handelt es sich bei der Differentialgleichung des Screen-Space-Curvature-Flows $\frac{\delta z}{\delta t} = H$ um eine partielle Differentialgleichung. Deshalb müssen zunächst per Ortsdiskretisierung die partiellen Ableitungen der partiellen Differentialgleichung eliminiert werden. Anschließend kann die Visualisierung die nun gewöhnliche Differentialgleichung $\frac{\delta z}{\delta t}$ über ein Integrationsverfahren für solche gewöhnliche Differentialgleichungen lösen.

Für die Ortsdiskretisierung werden die ersten Ableitungen von $\frac{\delta z}{\delta y}$ und $\frac{\delta z}{\delta z}$ über den Central-Difference berechnet. Analog werden die zweiten Ableitungen $\frac{\delta^2 z}{\delta y^2}$ und $\frac{\delta^2 z}{\delta x^2}$ über die finite Differenzen der ersten Ordnung berechnet. Zudem berechnet die Visualisierung D gemäß Gleichung 5.11 und dann daraus $\frac{\delta D}{\delta x}$ und $\frac{\delta D}{\delta y}$ per Central-Difference.

Zudem muss die Visualisierung bei der Berechnung des Screen-Space-Curvature-Flows berücksichtigen, dass wenn im Tiefenbild sich die Tiefenwerte der Nachbarn zu sehr vom Tiefenwert eines Pixels unterscheiden, es sich hierbei wahrscheinlich um eine Diskontinuität in der Flüssigkeitsoberfläche handelt. Deshalb muss sie große Schwankungen im Tiefenwert bei der Berechnung des Screen-Space-Curvature-Flows besonders behandeln. Denn ansonsten würden auf dem fertig gerenderten Bild Teile der Flüssigkeit ineinander verschwimmen, welche auf Grund ihres Tiefenwerts weit voneinander entfernt sind. Für die Behandlung von Diskontinuitäten wird für jeden Pixel überprüft, ob der Tiefenunterschied zu einem benachbarten Pixel größer als ein gegebener Schwellwert ist. In diesem Fall setzen die Kernels diejenigen Ableitungen der zweiten Ordnung, welche diesen Nachbarpixel verwenden, an der Stelle des Pixels auf 0, weil die Ableitungen zweiter Ordnung dort nicht mehr definiert sind. Bei den Ableitungen erster Ordnung verwendet die Visualisierung dann statt dem Central-Difference den Forward- oder Backward-Difference, je nachdem in welcher Richtung die Diskontinuität auftritt.

Nach der Ortsdiskretisierung kann die Zeitintegration ausgeführt werden. Hierfür verwendet die Implementierung das explizite Euler-Verfahren. Allerdings muss bei der Integration die Schrittweite sehr klein sein. Denn bei zu großen Schritten beginnt die Lösung von $\frac{\delta z}{\delta t}$ zu oszillieren, wodurch im Tiefenbild ein deutlich sichtbares Rauschen entsteht. Die Ursache hierfür ist, dass die Berechnung des Screen-Space-Curvature-Flows ein steifes Problem ist. Aus diesem Grund braucht der Screen-Space-Curvature-Flow für eine qualitativ hochwertige Glättung des Tiefenbilds gerade bei größeren Auflösungen mehrere hundert Iterationen. Die große Anzahl der Iterationen machen den Screen-Space-Curvature-Flow teuer. Um die Rechenlast etwas zu reduzieren verwendet die Implementierung des Screen-Space-Curvature-Flows ein Empty-Space-Skipping Verfahren auf Thread-Block-Basis. Hier macht sie sich zunutze, dass sich die Flüssigkeitsverteilung im Tiefenbild durch den Screen-Space-Curvature-Flow nicht ändern kann. Beim Empty-Space-Skipping wird in einem vorbereitenden Pass zunächst überprüft, in welchen Bereichen des Tiefenbilds sich die Flüssigkeit befindet. Anschließend werden bei allen Screen-Space-Curvature-Flow-Iterationen nur Thread-Blocks für diejenigen Tiefenbild-Bereiche mit Flüssigkeit gestartet.

5.6 Finales Shading

In diesem Punkt soll das finale Shading vorgestellt werden. Für das finale Shading wird eine realistische Shading-Technik verwendet. Bei der Technik setzt die Visualisierung die bisher gerenderten Bilder zum fertig gerenderten Bild zusammen und schreibt es in den Frame-Buffer des Fensters zurück. Beim Zusammensetzen berechnet sie die Transparenz der Flüssigkeit anhand des Dickebilds und die Reflexion der Flüssigkeit anhand des geglätteten Tiefenbilds. Zusätzlich muss die Visualisierung in den Bereichen ohne Flüssigkeit das Terrain beziehungsweise einen Cube-Map-Himmel, dessen Textur aus [TexSky] stammt, zeichnen.

Für das finale Shading zeichnet die Visualisierung in OpenGL ein Bildschirm füllendes Rechteck. Der Fragment-Shader durchläuft dabei folgende Schritte, welche noch einmal in Abbildung 5.5 gezeigt werden:

- **Bestimmung der Hintergrundintensität $\vec{I}_B$:** Zuerst gilt es die Hintergrundintensität $\vec{I}_B$ des Pixels zu bestimmen. Bei ihr handelt es sich genauso wie bei allen folgenden Intensitäten um einen RGB-Vektor. Ist an der Stelle des Pixels Terrain vorhanden so wird die Farbe der Terrain-Farbtextur an dieser Stelle übernommen. Ansonsten berechnet der Fragment-Shader die Farbe des Himmels per Himmels-Cube-Map. Ist keine Flüssigkeit an der Position es Pixels vorhanden so kann der Fragment-Shader die Hintergrundintensität sofort als Endergebnis in den Framebuffer zurückschreiben und hier abbrechen.
- **Rekonstruktion der Oberflächennormale:** Der Fragment-Shader rekonstruiert die Oberflächennormale der Flüssigkeit aus dem Tiefenbild per „Central-Difference“-Filter. Dafür bildet er zuerst die Tiefenwerte der vier benachbarten Pixel ins Weltkoordinatensystem durch Invertierung der Kameratransformation ab. Aus den vier Positionen im Weltkoordinatensystem lassen sich nun zwei Tangentialvektoren berechnen, deren Kreuzprodukt die gesuchte Normale ist. Allerdings kann hier analog wie beim Screen-Space-Curvature-Flow eine Diskontinuität im Tiefenbild auftreten. In diesem Fall nimmt der Fragment-Shader fallabhängig für die Rekonstruktion der Normale den Forward-Difference oder den Backward-Difference.
- **Berechnung des Fresnelterms f:** Der Fresnelterm gibt an, wie gut die Flüssigkeit in Abhängigkeit von den Brechungsindexen und dem Blickwinkel das Licht reflektiert beziehungsweise transmittiert. Denn schaut ein Betrachter in einem flachen Winkel auf eine Flüssigkeit so ist der reflektierte Anteil niedrig und der transmittierte Anteil hoch. Bei einem steilen Blickwinkel jedoch ist dies genau anders herum. Dabei kann der Fragment-Shader den Blickwinkel und damit wiederum den Fresnelterm leicht über die Kameraposition, die Position des Fragments, und die Normale berechnen.
- **Bestimmung der transmittierten Intensität $\vec{I}_T$:** Als Nächstes berechnet der Fragment-Shader die transmittierte Intensität. Die transmittierte Intensität bewirkt, dass Objekte durch die Flüssigkeitsoberfläche hindurchscheinen oder dass tiefes Wasser blau erscheint. Bei der Transmission tritt jedoch Brechung auf. Eine allgemeine Brechungsberechnung ist aber nicht per Rasterisierung machbar. Aus dem Grund wird angenommen, dass bei der simulierten Flüssigkeit keine Brechung stattfindet. Dadurch bewegt sich die Hintergrundintensität entlang eines Strahls geradlinig Richtung Kamera, wodurch sie ihren Pixel beibehält. Allerdings tritt auf diesem Weg Absorption auf. Zusätzlich wird entlang des Strahls weitere Intensität durch die Flüssigkeit hineingestreut. Die hineingestreute

Intensität kann dadurch, dass sie ebenfalls durch die Flüssigkeit absorbiert wird, einen maximalen Wert von $\vec{I}_S$ annehmen. Sowohl das Hineinstreuen als auch die Absorption finden in Abhängigkeit von der Wellenlänge des Lichts und der durch die Flüssigkeit zurückgelegten Strecke gemäß des Absorptionsgesetzes statt. Die Strecke entlang der diese Effekte auftreten wird aus dem Wert d des Dickebilds an der Stelle des Pixels ermittelt. Kommt das Licht an der Oberfläche an, so wird gemäß des Fresnelterms nur ein Teil weiter zur Kamera transmittiert, der Rest wird in die Flüssigkeit zurück reflektiert. In Formeln gefasst gilt:

$$\vec{I}_T = (1 - f) \left(\vec{a} \circ \vec{I}_B + (\vec{1} - \vec{a}) \circ \vec{I}_S \right) \tag{5.15}$$

Dabei ist $\circ$ das Hadamard-Produkt. Das Ergebnis des Hadamard-Produkts ist ein Vektor, den das Hadamard-Produkts dadurch berechnet indem es seine beiden Operanden komponentenweise miteinander multipliziert. Des Weiteren gibt $\vec{a}$ den Grad der Absorption an und wird hier über das Absorptionsgesetz definiert als:

$$\vec{a} = \begin{pmatrix} e^{-\mu_r d} \\ e^{-\mu_g d} \\ e^{-\mu_b d} \end{pmatrix} \tag{5.16}$$

Die Absorptionskoeffizienten μ_r, μ_g und μ_b geben die Stärke der Absorption für die grüne, rote und blaue Intensität an. So absorbiert Wasser die rote Intensität am stärksten, die grüne weniger stark und die blaue Intensität am schwächsten.

- **Bestimmung der reflektierten Intensität $\vec{I}_R$:** Nun kann der Fragment-Shader die reflektierte Intensität bestimmen. Flüssigkeiten besitzen jedoch die Eigenschaft, dass an ihren Oberflächen eine gerichtete Reflexion auftritt. Eine allgemeine gerichtete Reflexionsberechnung ist jedoch nicht per Rasterisierung machbar. Deshalb wird hierfür angenommen, dass sich nur der Himmel in der Flüssigkeit spiegeln könne. Deshalb reflektiert der Fragment-Shader den Vektor zwischen der Kamera und der Position der Flüssigkeitsoberfläche an der Normale. Anschließend schlägt er die Intensität der Himmels-Cube-Map $\vec{I}_{Cube}$ an der Stelle des reflektierten Vektors nach. Optional kann er eine Glanzlicht-Reflexion $\vec{I}_{Specular}$ gemäß des Phong-Beleuchtungsmodells berechnen. Die reflektierte Intensität berechnet sich somit insgesamt wie folgt:

  $$\vec{I}_R = f\vec{I}_{Cube} + \vec{I}_{Specular} \tag{5.17}$$

- **Berechnen der Gesamtintensität $\vec{I}_G$:** Anschließend muss der Fragment-Shader die reflektierte und transmittierte Intensitäten zur Gesamtintensität des Pixels aufsummieren:

  $$\vec{I}_G = \vec{I}_T + \vec{I}_R \tag{5.18}$$

  Diese wird anschließend als Endergebnis der Visualisierung in den Framebuffer zurückgeschrieben.

Diese Art von realistischer Visualisierung besitzt jedoch dasjenige Problem, dass ein Betrachter wegen des Fresnelterms bei flachen Blickwinkel keine Oberflächendetails und bei steilen

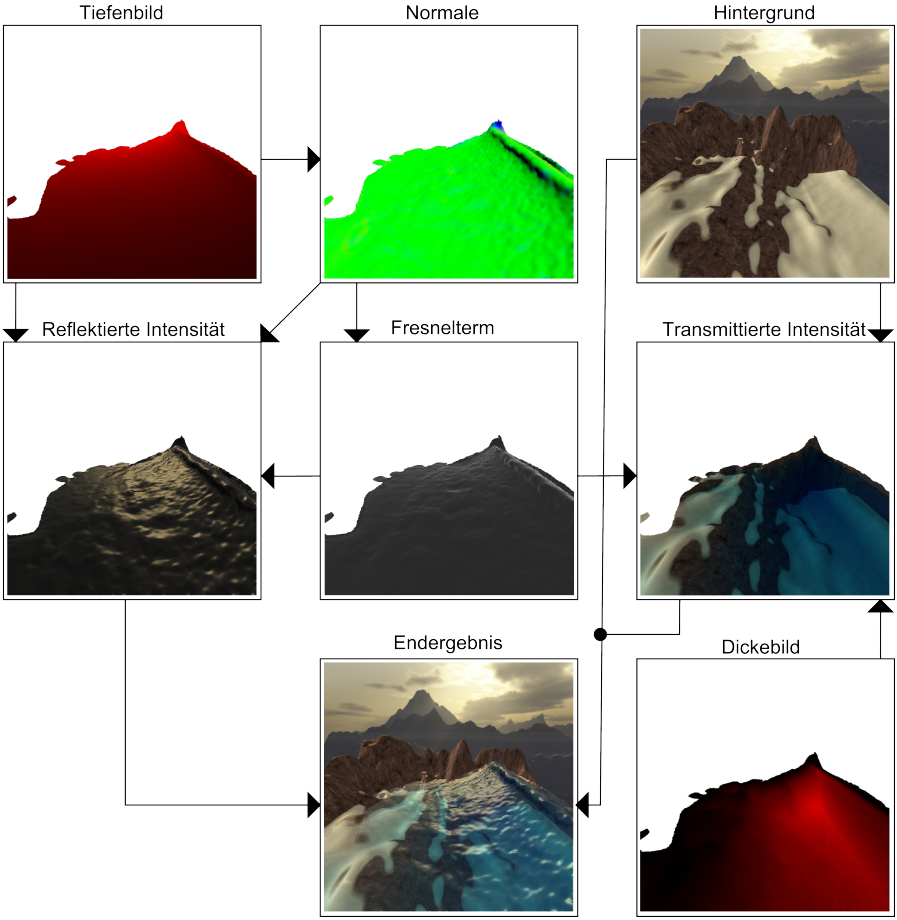


Abbildung 5.5: Zusammensetzung des Bildes beim finalen Shading

Blickwinkeln nicht die Dicke der Flüssigkeit erkennen kann. Deshalb kann es auch vorteilhaft sein eine wissenschaftliche Visualisierung zu verwenden, bei der das soeben genannte Problem nicht auftritt. Denn das Ziel einer solchen wissenschaftlichen Visualisierung sollte es immer sein, dass der Betrachter die gewünschten Areas of Interest möglichst gut erkennen kann. Prinzipiell ist es leicht möglich den Fragment-Shader auf eine wissenschaftlichere Visualisierung anzupassen, so lange diese das Dickebild und das Tiefenbild als Grundlage verwendet. Eine solche wissenschaftlichere Visualisierung ist beispielhaft in Abbildung 5.4 zu sehen, bei der die Flüssigkeit mit einem einfachen Phong-Beleuchtungsmodell gerendert worden ist. Bei einer wissenschaftlichen Visualisierung ist es jedoch problematisch, dass die optimale Shadingtechnik stark von den gewünschten Areas of Interest und damit vom Anwendungsfall abhängig ist. Denn je nach Anwendungsfall können unterschiedliche Gegebenheiten von Interesse sein, welche mit der Visualisierung leicht zu erkennen sein sollten. Deshalb fällt es schwer eine solche wissenschaftliche Visualisierung allgemein ohne einen spezifischen Anwendungsfall zu implementieren.

5.7 Bewertung der visuellen Qualität

Schließlich soll die visuelle Qualität der Visualisierung bewertet werden. So sieht die finale Visualisierung in Abbildung 5.6 bereits wirklichkeitsnah aus. Allerdings treten bei der visuellen Qualität noch folgende Probleme auf:

- **Keine allgemeine gerichtete Reflexion und Lichtbrechung:** Da die gerichtete Reflexion und Lichtbrechung nicht oder nur sehr eingeschränkt durch Rasterisierung machbar sind, müsste die Visualisierung hierfür eine Raytracing-Technik statt einer Splatting-Technik verwenden.
- **Nur Berücksichtigung der vordersten Flüssigkeitsoberfläche:** Durch die Approximation der Flüssigkeitsoberfläche durch ein Tiefenbild wird nur diejenige Flüssigkeitsoberfläche beim Oberflächenshading berücksichtigt, die der Kamera am nächsten ist. Wegen der Transparenz der Flüssigkeit könnten jedoch auch diejenigen Flüssigkeitsoberflächen, die weiter von der Kamera entfernt sind, etwas zur Farbe eines Pixels beitragen. Diese Beiträge können jedoch nicht mit der Splatting-Technik simuliert werden.
- **Das Fehlen von kleineren Unebenheiten oder Wellen:** Durch die beschränkte Auflösung der Simulation in Kombination mit dem Screen-Space-Curvature-Flow wirkt die Flüssigkeitsoberfläche unnatürlich glatt. Um dies zu vermeiden könnte die Visualisierung zum Beispiel wie in dem Artikel [SG] vorgeschlagen im Nachhinein das Tiefenbild noch mit einem Rauschen versehen. Dieses Rauschen würde zudem auf dem Bildschirm gesehen zu starken Schwankungen in der Oberflächennormale der Flüssigkeit und damit zu starken Schwankungen des Fresnelterms führen. Dadurch würde die Visualisierung auch bei einem flachen Blickwinkel auf die Flüssigkeit eine hohe Reflexivität erreichen, wodurch ein Betrachter dort die Oberfläche der Flüssigkeit besser erkennen könnte.
- **Bildfehler durch Screen-Space-Curvature-Flow:** Wie bei jeder Screen-Space-Technik so ist die visuelle Qualität des Screen-Space-Curvature-Flows stark von der Position und dem Blickwinkel der Kamera abhängig. Zudem treten bei Diskontinuitäten in der Oberfläche kleinere Bildfehler auf.

- **Keine fortgeschrittene Beleuchtungseffekte:** Es fehlen zum Beispiel Kaustiken, Schattenwurf durch die Flüssigkeit oder Strahlenbüschel.
- **Das Fehlen von Schaum:** Für die Simulation von Schaum gibt es mehrere verschiedene Ansätze. Manche davon färben die Oberfläche der Flüssigkeit zum Beispiel in Abhängigkeit der Geschwindigkeit weiß ein. Andere wie der Artikel [MM] verwenden sekundäre Schaum-Partikel, die von der Flüssigkeit erzeugt werden.
- **Das Fehlen von kleineren Spritzern und Gischt:** Hierfür wird ebenfalls in der Literatur wie zum Beispiel im Artikel [MM] ein sekundäres Partikelsystem implementiert.
- **Geringer Nutzen des Ellipsoid-Splattings bei hohen Flüssigkeitsauflösungen:** Je feiner die Flüssigkeit aufgelöst ist, desto besser wird die Approximation der Flüssigkeit durch Kugeln im Vergleich zu einer Approximation durch Ellipsoide. Zusätzlich ist ein Ellipsoid-Splatting jedoch deutlich teurer. Deshalb lohnt es sich bei hoch aufgelösten Flüssigkeiten im Vergleich zu einem Sphere-Splatting nur noch wenig.

5.8 Untersuchung

Als Nächstes soll die Performance der Visualisierung untersucht werden werden. Dabei fällt es schwer die Performance der Visualisierung zu bewerten oder in Relation zur Simulation selbst zu setzen. Denn die Kosten der Visualisierung sind von vielen Parametern abhängig:

- **Bildqualität:** Für eine hohe Bildqualität muss die Visualisierung sehr viele Screen-Space-Curvature-Flow-Iterationen ausführen.
- **Bildauflösung:** Je höher die Bildauflösung ist, umso mehr Iterationen benötigt der Screen-Space-Curvature-Flow um das Tiefenbild zu glätten. Zusätzlich ist eine Iteration teurer. Des Weiteren müssen beim Splatting mehr Fragmente berechnet werden.
- **Flüssigkeitsauflösung:** Wird die Flüssigkeitsauflösung erhöht, so werden die Wölbungen im Tiefenbild kleiner. Dadurch benötigt die Visualisierung auch weniger Screen-Space-Curvature-Flow-Iterationen um das Tiefenbild zu glätten.
- **Position der Kamera:** Umso mehr Flüssigkeit sich näher an der Kamera befindet, desto mehr Fragmente muss die Visualisierung erstellen und desto weniger gut kann das Empty-Space-Skipping des Screen-Space-Curvature-Flows funktionieren.
- **Partikelanzahl:** Die Kosten für das Splatting der Ellipsoide und die Berechnung der Ellipsoid-Hauptachsen sind stark von der Partikelanzahl abhängig.
- **Zahl der Kameras:** Selbst wenn es nicht in dieser Arbeit implementiert worden ist, so könnte ein Betrachter die Flüssigkeit von verschiedenen Standpunkten aus sehen wollen. Für jede Kamera müsste die Visualisierung den teuren Screen-Space-Curvature-Flow und das Splatting ausführen.
- **Anzahl von Simulationsschritten pro Zeichenvorgang:** Wegen der kleinen Zeitschrittgröße auf Grund der Steifheit der SPH-Simulation bewegen sich die Partikel in fein aufgelösten Simulationen pro Simulationsschritt kaum. Deshalb kann es sinnvoll sein nur alle paar Simulationsschritte die Simulation zu visualisieren.

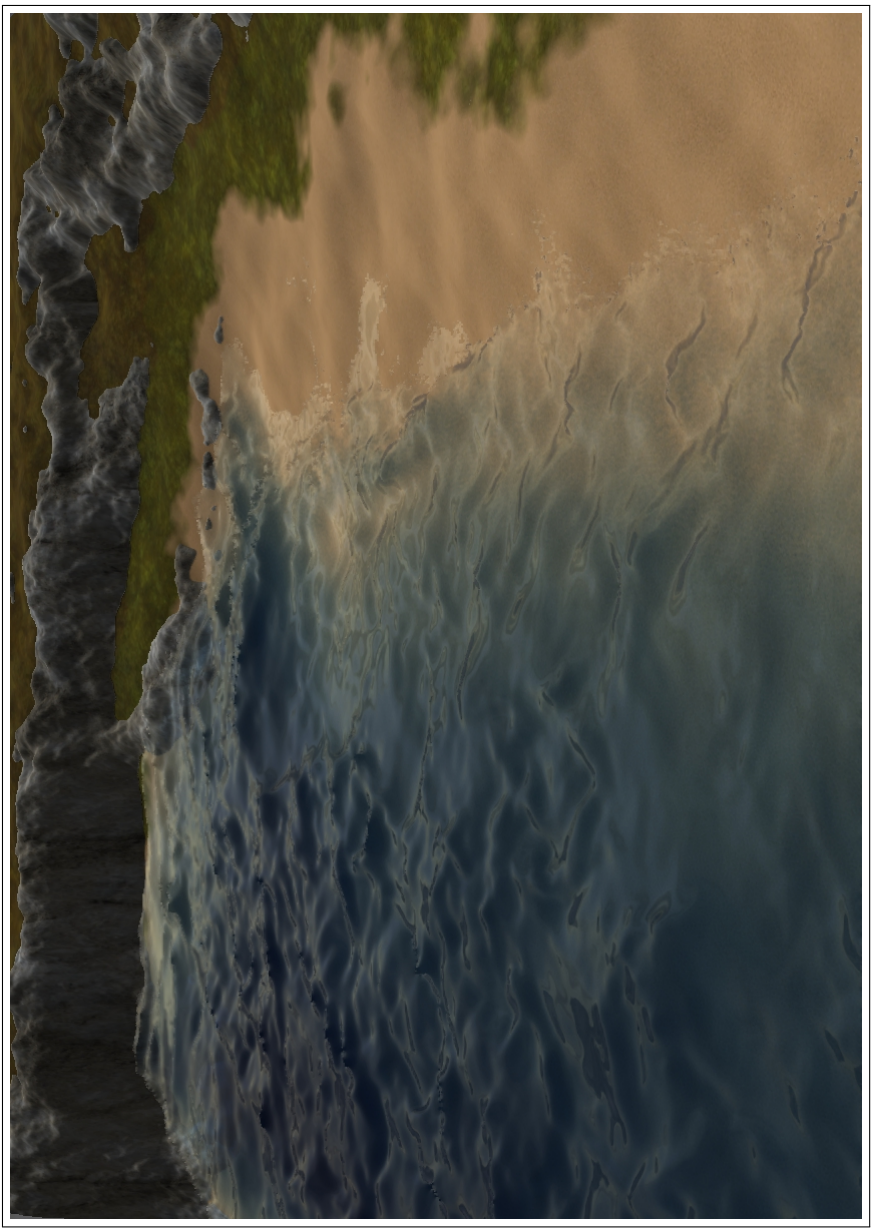


Abbildung 5.6: Finales Renderergebnis der Visualisierung

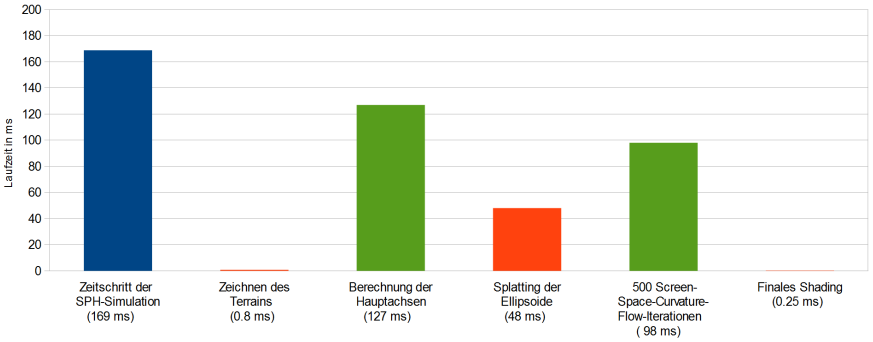


Abbildung 5.7: Übersicht über die Laufzeiten der einzelnen Schritte der Visualisierung

Dennoch wurden Benchmarks für einen einfachen Performance-Überblick durchgeführt, welcher in Abbildung 5.7 zu sehen ist. Für ihn wurde wieder die Testszene mit 3 972 734 Partikeln verwendet. Zudem wurde eine Bildauflösung von 1024 auf 1024 Pixeln und 500 Screen-Space-Curvature-Flow-Iterationen gewählt. So sind die Kosten eines Zeichenvorgangs im Vergleich zu einem Zeitschritt mit 274 ms gegenüber 169 ms relativ teuer. Dabei wird ein Großteil der Kosten durch den Screen-Space-Curvature-Flow und durch die Berechnung der Hauptachsen der Ellipsoide verursacht. Deshalb kann die Visualisierung durch die Verwendung von weniger Screen-Space-Curvature-Flow-Iterationen oder durch die Verwendung eines Sphere-Splattings anstelle des Ellipsoid-Splattings viel Rechenzeit auf Kosten der visuellen Qualität einsparen. Ein weiterer Vorteil für die Verwendung des Sphere-Splattings ist der potentiell geringere Speicherplatzverbrauch. Die Ursache hierfür ist, dass die Visualisierung für das Sphere-Splatting nicht die Hauptachsen der Ellipsoide abspeichern muss. Dies ist aber nur nachteilig, falls die Visualisierung die Hauptachsen nicht über Zeigeraliasing in gerade nicht genutzte Arrays der SPH-Simulation abspeichern kann.

Je seltener jedoch die Visualisierung berechnet wird, umso geringer fallen deren Kosten ins Gewicht. Weitere Tests ergaben, dass bei feiner aufgelösten Simulationen es ausreichend ist zum Beispiel nur jeden zehnten oder zwanzigsten Zeitschritt zu zeichnen, wodurch sich die Kosten der Visualisierung insgesamt auch auf ein Zehntel oder ein Zwanzigstel reduzieren. Weitere Tests zeigten zudem, dass sich diese Art der Visualisierung sehr gut für Echtzeit-Rendering eignet. Hierfür müssen aber die Zahl der Partikel der SPH-Simulation entsprechend klein gewählt und nur wenig Screen-Space-Curvature-Flow-Iterationen berechnet werden. Verwendet die SPH-Simulation beispielhaft nur 300 000 Partikel sowie 50 Screen-Space-Curvature-Flow-Iterationen und zeichnet die Visualisierung jeden zweiten Zeitschritt, so lassen sich in etwa 20 FPS erzielen. Die Beobachtung deckt sich mit der Literatur überein, welche diese Art der Visualisierung für die Echtzeit-Computergraphik empfiehlt. Weiterhin ist es problematisch, dass sich das gewählte SPH-Verfahren aus dem letzten Kapitel wegen der Steifheit und den daraus resultierenden kleinen Schrittweiten extrem schlecht für die Echtzeitcomputergraphik eignet. Aus dem Grund gibt es auch spezielle SPH-Verfahren für die Echtzeitcomputergraphik, die die

physikalische Exaktheit für wesentliche größere Zeitschrittweiten opfern. Die schwere Komprimierbarkeit einer Flüssigkeit ist aber auch für die visuelle Qualität entscheidend. Deswegen beschäftigen sich solche Verfahren der Computergraphik oft damit, wie sie die schwere Komprimierbarkeit näherungsweise bei großen Zeitschritten sicherstellen können. Ein solches Verfahren wird beispielhaft in dem Artikel [MM] vorgestellt. In einer frühen Version dieser Arbeit wurde eben dieses Verfahren testweise implementiert. Dabei zeigte sich, dass die Simulation mit diesem Verfahren tatsächlich sehr große Schrittweiten erlaubt. Jedoch war die Wahl der Konstanten der Simulation problematisch. Algorithmisch gesehen bestehen diese Verfahren der Echtzeitcomputergraphik wieder hauptsächlich aus der Berechnung von Fixed-Radius-Near-Neighbors-Problemen. Wegen der Modularität der Berechnungen dieser Arbeit ließen sich die SPH-Verfahren für die Echtzeitcomputergraphik ebenfalls leicht implementieren.

5.9 Fazit

Schließlich soll das Fazit aus dem Kapitel der Visualisierung gezogen werden. So stellte dieses Kapitel eine Visualisierung vor. Mit ihr lässt sich die Flüssigkeit zwar nicht perfekt realistisch aber dennoch wirklichkeitsnah darstellen. Dafür verwendet die Visualisierung eine Ellipsoid-Splatting-Technik, um ein Tiefenbild und ein Dickebild zu erzeugen. Anschließend glättet sie das Tiefenbild mit dem Screen-Space-Curvature-Flow-Verfahren. Zuletzt setzt sie das geglättete Tiefenbild und das Dickebild zum finalen Renderergebnis zusammen. Die Berechnungen der Visualisierung sind zwar im Vergleich zu einem SPH-Zeitschritt relativ teuer. Führt berechnet die GPU die Visualisierungs allerdings nur alle paar Zeitschritte so ist die Visualisierung sehr günstig. Allerdings besteht sowohl bei der Performance der Berechnungen als auch bei der visuellen Qualität der Visualisierung weiterhin Verbesserungspotential.

6 Erweiterung der SPH-Implementierung auf einen GPU-Cluster

6.1 Grundlegende Überlegungen

In diesem Kapitel soll vorgestellt werden, wie die Single-GPU-SPH-Implementierung mit ihrer Visualisierung auf einen GPU-Cluster erweitert worden ist. Zusätzlich soll die so entstandene Cluster-Implementierung untersucht werden. Dafür wird in diesem Punkt zunächst ein Überblick über die grundlegenden Überlegungen gegeben, welche für den Entwurf einer Implementierung für einen GPU-Cluster nötig sind.

So besitzt ein solcher GPU-Cluster die Eigenschaft, dass er aus einem oder mehreren Knoten besteht, wobei ein jeder Knoten prinzipiell eine oder mehrere GPUs besitzen kann. Es gilt hierbei zu beachten, dass weder die einzelnen GPUs innerhalb eines Knotens, noch die GPUs aus unterschiedlichen Knoten direkt auf den Speicher anderer GPUs zugreifen können. Allerdings ist ein indirekter Zugriff über den Host und das Netzwerk möglich, womit die einzelnen Knoten des Clusters miteinander verbunden sind. Deshalb muss es die allererste Überlegung sein, wie die SPH-Simulation das Gitter und damit die darinnen enthaltenen Partikel auf die GPUs im Cluster aufteilt. Dabei gilt es zu berücksichtigen, dass die Rechenlast nicht oder nur kaum vom Gittervolumen sondern hauptsächlich von der Partikelanzahl im Gitter abhängig ist. Aus diesem Grund ist es zudem wichtig, dass sich die Aufteilung während der Laufzeit dynamisch der Rechenlast der einzelnen GPUs anpassen kann. Auch benötigen die Partikel im Randbereich des Gitters einer GPU für die Fixed-Radius-Near-Neighbors-Berechnungen diejenigen Gitterdaten und Partikeldaten, welche bereits zu anderen GPUs gehören. Derjenige Bereich der Simulationsdomäne, welchen eine GPU von einer Nachbar-GPU für ihre Berechnungen im Randbereich benötigt, wird als Halo bezeichnet. Damit die GPU dennoch darauf zugreifen kann, muss das gesamte Volumen des Halos zwischen den GPUs synchronisiert werden. Da diese Synchronisation über den langsamen PCI-Express und zum Teil sogar über das noch langsamere Netzwerk erfolgen muss, ist ein kleines Verhältnis von Halovolumen im Vergleich zum Gesamtvolumen der Simulation von Vorteil. Denn daraus folgt auch ein kleines Verhältnis zwischen den langsam kopierten Daten und den schnellen Rechenoperationen. Schließlich ist es vorteilhaft, wenn die GPUs immer etwas rechnen können, während die Synchronisation erfolgt. Denn auf diese Weise verschwendet die Simulation keine Rechenleistung.

Prinzipiell kann die Simulation das Volumen der Simulationsdomäne eindimensional, zweidimensional oder dreidimensional in Blöcke aufteilen und anschließend die Blöcke den GPUs zuteilen. Beispiele für solche Aufteilungen sind in Abbildung 6.1 gezeigt. Dabei gilt im Allgemeinen je höher die Dimensionalität der Aufteilung bei konstanter GPU-Zahl ist, desto kleiner wird das Halovolumen im Verhältnis zum Gesamtvolumen der Simulation. Dadurch lohnt sich bei höheren Dimensionalitäten die Verwendung eines GPU-Clusters bereits bei einem kleineren Gesamtvolumen. Aus dem selben Grund skaliert die Simulation bei höher dimensionalen Aufteilungen besser mit der GPU-Zahl. Des Weiteren ist es bei der eindimensionalen Aufteilung problematisch, dass bei zu großen Simulationen selbst ein Block, welcher bereits nur noch eine Gitterzelle dick ist, nicht mehr in den DRAM einer GPU passen kann. Da die Simulation den Block nicht mehr kleiner machen kann, ist die maximale Größe einer solchen Simulation durch den Speicherplatz einer GPU begrenzt. Bei einer zweidimensionalen Aufteilung besteht das analoge Problem in einer weniger schlimmen Form. Erst bei einer dreidimensionalen Aufteilung ist dieses Problem komplett verschwunden. Allerdings sind die Halobehandlung im Kernel, die Kopieroperationen zwischen den GPUs und die Lastbalancierung bei niedrigeren Dimensionalitäten leichter zu implementieren und leichter zu berechnen. Des Weiteren benötigen

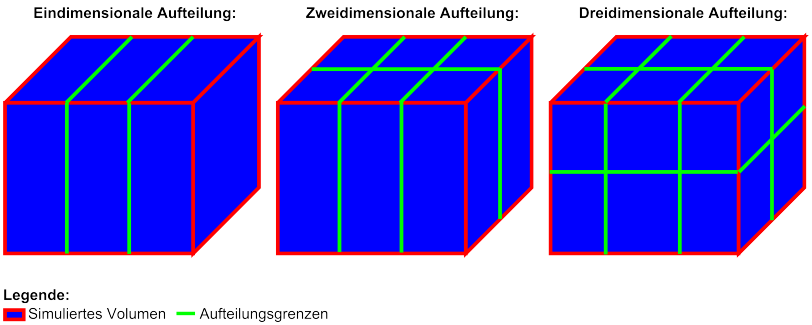


Abbildung 6.1: Beispiele für die Dimensionalität der Aufteilung des simulierten Volumens auf die GPUs des Clusters: Für diese beispielhafte Aufteilung wurden die GPUs jeweils in einem N-dimensionalen Gitter angeordnet

die unterschiedlichen Dimensionalitäten der Aufteilung auch unterschiedliche Flüssigkeitsverteilungen innerhalb der Simulationsdomäne, damit sie effizient funktionieren können. Dies soll im Folgenden kurz im Falle einer würfelförmigen Simulationsdomäne näher begründet werden. So muss bei einer eindimensionalen Aufteilung und einer würfelförmigen Simulationsdomäne die Flüssigkeit entlang der Aufteilungsachse homogen verteilt sein. Denn ansonsten befindet sich ein großer Anteil der Partikel in den Halos, wodurch wiederum viel Netzwerkbandbreite benötigt wird. Zudem kann in diesem Fall die Lastbalancierung nicht gut funktionieren. Aus analogen Gründen sollte bei einer zwei dimensionalen Aufteilung die Flüssigkeit homogen in der Aufteilungsebene verteilt sein. Bei einer dreidimensionalen Aufteilung ist es vorteilhaft, wenn die Flüssigkeit im gesamten Volumen homogen verteilt ist. Somit besitzen höherdimensionale Aufteilungen größere Anforderungen an die Flüssigkeitsverteilung als niedrigdimensionale Aufteilungen. Eine Begründung für quaderförmige Simulationsdomänen läuft analog ab. Deshalb kann eine höherdimensionale Aufteilung bei einer ungünstigen Flüssigkeitsverteilung auch eine deutlich schlechtere Performance besitzen.

6.2 Aktuelle Ansätze für die SPH-Simulation im GPU-Cluster

In diesem Punkt soll auf die aktuellen Ansätze der Literatur für die SPH-Simulation im GPU-Cluster eingegangen werden. Wie soeben erläutert besitzt jede Unterteilungsdimensionalität ihre Vor- und Nachteile. Die Literatur verwendet für kleinere Cluster, wie zum Beispiel in der Simulation [DSPHa] mit dem DualSPHysics-Simulator und 98 GPUs, jedoch meist eine eindimensionale Unterteilung. Denn hierbei wirkt sich die Größe des Halovolumens noch nicht oder nur kaum negativ auf die Performance aus. Zudem kann die SPH-Simulation bei einer eindimensionalen Aufteilung sehr leicht die Daten zwischen den GPUs synchronisieren. Da für diese Arbeit auch nur ein kleiner Cluster aus 5 GPUs zur Verfügung steht, wird ebenfalls diese Unterteilung verwendet. Deswegen werden im Folgenden nur Ansätze für die eindimensionale Unterteilung vorgestellt. Bei einer solchen eindimensionalen Aufteilung werden die Blöcke meist als Scheiben (Slices) bezeichnet.

Bei der eindimensionalen Unterteilung sind die in der Literatur erwähnten Verfahren weitestge-

hend analog. Allerdings unterscheiden sich die Verfahren der Literatur etwas darinnen, welche grundlegenden Ansätze sie für die Berechnung des Fixed-Radius-Near-Neighbors-Problems verwenden. Dementsprechend verwenden sie auch für den gewählten Ansatz spezielle Optimierungen für den GPU-Cluster. Dabei ist es meist das Ziel dieser Optimierungen ein gleichzeitiges Kopieren und Berechnen zu ermöglichen. Dies erfolgt in der Regel dadurch, dass bei der Berechnung des Fixed-Radius-Near-Neighbors-Problems die Simulation zunächst den inneren Bereich berechnet, während der Halobereich kopiert wird. Anschließend berechnet die Simulation den Randbereich. Beispielhaft wurde in [ER] die eindimensionale Unterteilung für einen einzigen Knoten mit mehreren GPUs vorgestellt. Dabei wurde das Kopieren zwischen den GPUs so optimiert, dass es in Kombination mit der Verlet-Liste, die für die Lösung des Fixed-Radius-Near-Neighbors-Problems gewählt wurde, performant funktioniert. Analog wurde in [JDc] die eindimensionale Unterteilung für einen GPU-Cluster mit mehreren GPUs pro Knoten behandelt. Bei dieser Quelle wurde das Linked-Cell-Verfahren mit einem Dynamic-Vector-Gitter, welches mit Radix-Sort gebaut wird, verwendet. Die Vorgehensweise ist dabei relativ ähnlich zu derjenigen Vorgehensweise, die diese Master-Arbeit im Folgenden verwendet.

Die Grundidee der Lastbalancierung ist ebenfalls immer identisch. Bei ihr verschiebt die Simulation die Grenzen der Scheiben der einzelnen GPUs, so dass die Rechenlast besser verteilt wird. Allerdings gibt es zwei unterschiedliche Möglichkeiten, wie die Simulation die Rechenlast der GPUs für die Lastbalancierung ermitteln kann. So kann die Simulation die Rechenlast entweder durch die Anzahl der Partikel in der jeweiligen Scheibe der GPU oder durch die Laufzeiten der SPH-Kernel bestimmen. Dabei ist die Bestimmung der Rechenlast durch die Laufzeiten meist vorteilhaft. Die Ursache hierfür ist, dass die Laufzeiten auch von der Partikelverteilung abhängig sind, welche die Lastbalancierung über die Partikelzahl nur schlecht berücksichtigen kann. Zudem sind die Laufzeiten ebenfalls von der GPU abhängig. Besitzt ein Cluster unterschiedliche GPUs, so kann dies die Lastbalancierung über die Partikelzahl auch nur schlecht berücksichtigen. Ebenfalls kann die Lastbalancierung lokal oder global erfolgen. Bei einer lokalen Lastbalancierung wird nur die Rechenlast von benachbarten GPUs berücksichtigt, während bei einer globalen Lastbalancierung die Rechenlast aller GPUs berücksichtigt wird. Dabei sind globale Verfahren wiederum vorteilhaft, da sie die Rechenlast besser und schneller aufteilen können. So wurde in [ER] ein globaler Ansatz für die Lastbalancierung vorgestellt, welcher die Rechenlast über Laufzeiten der SPH-Kernel abschätzt. Alternativ stellt der Artikel [JDc] zwei lokale Ansätze für die Lastbalancierung vor, welche die Rechenzeit beziehungsweise die Partikelanzahl berücksichtigen.

6.3 Implementierung

6.3.1 Übersicht über die Implementierung

Im Folgenden wird eine Übersicht über die Implementierung derjenigen SPH-Simulation für einen GPU-Cluster gegeben, welche für diese Arbeit erstellt wurde. So implementiert diese Arbeit die eindimensionale Aufteilung, wodurch jede GPU eine Scheibe des Gitters erhält. Diese Aufteilung findet dabei immer entlang der Z-Achse statt. Sie ist in Abbildung 6.2 und Abbildung 6.3 zu sehen. Eine solche Scheibe lässt sich, wie in Abbildung 6.4 gezeigt, noch einmal in zwei Bereiche unterteilen: Einen inneren Bereich, innerhalb welchem die GPU allein mit ihren Gitterdaten und Partikeldaten sämtliche Berechnungen ausführen kann, und einen Randbereich, für welchen sie die Daten der Halos von den beiden benachbarten GPUs benötigt.

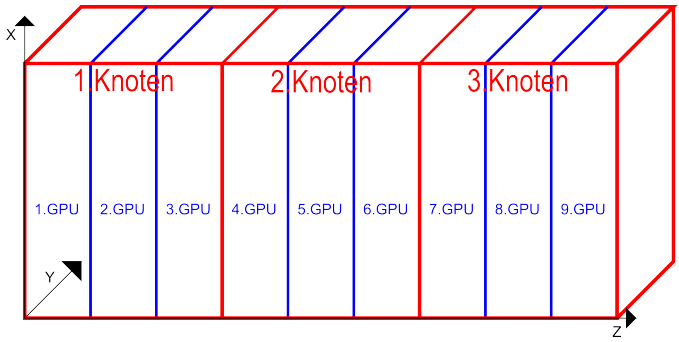


Abbildung 6.2: Aufteilung des simulierten Volumens auf die GPUs des Clusters

Da das Gitter ein per $\mathrm{Dim}_x \mathrm{Dim}_y Z + \mathrm{Dim}_x Y + X$ linearisiertes dreidimensionales Array ist, führt die Aufteilung entlang der Z-Achse dazu, dass die Gitterdaten des Halos sequentiell im Speicher der GPU liegen. Die Partikeldaten liegen innerhalb des Halos ebenfalls sequentiell im Speicher der GPU, weil die Partikel nach Gitterzellen sortiert sind. Somit benötigt die SPH-Simulation nicht extra Kernels um diejenigen Partikel innerhalb des Halos zu finden. Deshalb kann die SPH-Simulation die Synchronisation des Halos ohne die Rechenleistung der GPU zu verschwenden nur mit den Copy-Engines der GPU durchführen.

Nachdem die Aufteilung feststeht müssen für eine effiziente SPH-Simulation im GPU-Cluster folgende Features implementiert werden:

- **Punkt 6.3.2:** Das Ansteuern der GPUs über Workerthreads und der Datenaustausch über das Netzwerk
- **Punkt 6.3.3:** Die Änderungen bei der Datenhaltung
- **Punkt 6.3.4:** Die Modifikationen der Berechnungen des Fixed-Radius-Near-Neighbors-Problems
- **Punkt 6.3.5:** Die Änderungen bei der Zeitintegration und beim Bauen des Gitters
- **Punkt 6.3.6:** Die Lastbalancierung
- **Punkt 6.3.7:** Die Modifikationen des Zeichenvorgangs
- **Punkt 6.3.8:** Die Modifikation des Softwaredesigns, damit eine leichte Implementierung von alternativen Fixed-Radius-Near-Neighbors-Problemen im Cluster möglich ist

Diese Punkte sollen nun im Folgenden erläutert werden. Anschließend soll die Cluster-Version untersucht werden.

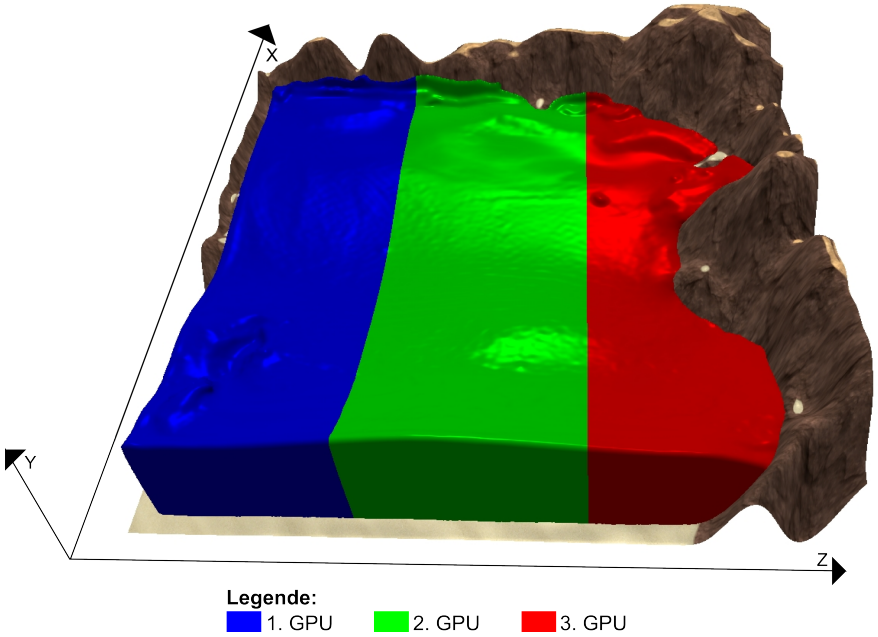


Abbildung 6.3: Aufteilung der Flüssigkeit auf die GPUs des Clusters

6.3.2 Workerthreads und Datenaustausch über das Netzwerk

Hierfür wird zuerst darauf eingegangen, wie die GPUs über Workerthreads angesteuert werden und wie der Datenaustausch über das Netzwerk von statten geht. So wird jede GPU mit ihrem eigenem Workerthread angesteuert. Damit existieren in jedem Knoten ein oder mehrere Workerthreads. Ein jeder dieser Workerthread kopiert Daten von den direkt benachbarten GPUs in dem selben Knoten durch den entsprechenden Befehl der CUDA-API zu seiner GPU hin. Bei den Daten handelt es sich entweder um die Daten des Halobereichs bei den Fixed-Radius-Near-Neighbors-Berechnungen oder um diejenige Partikel, welche sich durch die Zeitintegration in die Scheibe der GPU hineinbewegt haben. Da unterschiedliche GPUs in einem Knoten von verschiedenen Workerthreads bearbeitet werden, muss die SPH-Simulation die Workerthreads untereinander mit Barrieren synchronisieren.

Befindet sich die GPU des Threads am Rand eines Knotens, so ist ein Datenaustausch über das Netzwerk nötig. Für die Netzwerkkommunikation verwendet diese Arbeit die MPI-Bibliothek. In diesem Fall ist der Thread der GPU am Rand dafür verantwortlich diejenigen Daten, welche von dem entsprechenden Thread im benachbarten Knoten benötigt werden, von seiner GPU herunterzuladen und anschließend asynchron über MPI zu verschicken. Da der Workerthread der benachbarten GPU des anderen Knotens genau gleich handelt, kann jeder der beiden Threads auf den Empfang derjenigen Daten warten, welche er für seine GPU benötigt. Anschließend laden beide Threads die empfangenen Daten auf ihre GPUs hoch. Dadurch erfolgt ebenfalls ei-

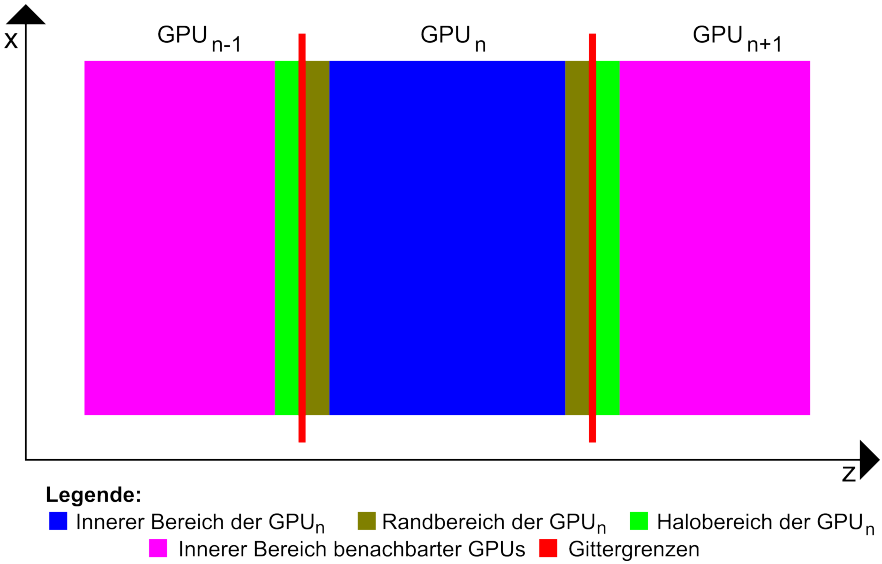


Abbildung 6.4: Bereiche in der Scheibe einer GPU

ne implizite Synchronisation der unterschiedlichen Knoten, weshalb keine teure MPI-Barriere mehr benötigt wird. Zudem erfolgt auf diese Weise auch nur eine Nachbarschaftskommunikation zwischen den GPUs. Des Weiteren rufen durch dieses Programmdesign jedoch mehrere Workerthreads pro Knoten gleichzeitig die Befehle der MPI-Bibliothek auf. Deshalb ist für die SPH-Simulation eine threadsichere Version der MPI-Bibliothek nötig. Da Open-MPIs threadsicherheit sich noch in früher Versuchsphase befindet sowie de facto nicht vorhanden ist und Intel-MPI kostenpflichtig ist, wurde in dieser Arbeit MPICH als MPI-Implementierung verwendet.

Des Weiteren wäre es auch möglich statt mehrerer Workerthreads mehrere MPI-Prozesse pro Knoten zu starten, so dass jede GPU von einem MPI-Prozess angesteuert werden könnte. In diesem Fall würde die SPH-Simulation alleine keine threadsichere MPI-Version benötigen. Da threadsichere MPI-Versionen noch selten sind, wäre dies prinzipiell vorteilhaft. Für die folgende zur SPH-Simulation asynchrone Pipeline der Visualisierung im Cluster aus Punkt 6.3.7 würde allerdings dennoch eine threadsichere MPI-Version benötigt werden. Auch würde der Quelltext nun keine Sonderbehandlung dafür benötigen, ob die Kommunikation innerhalb des selben Knotens oder mit dem benachbarten Knoten stattfinden soll. Somit ließe sich auf diese Weise der Quelltext der SPH-Simulation und der Visualisierung im Cluster stark vereinfachen. Nachteilig an mehreren MPI-Prozessen pro Knoten wäre jedoch, dass nun auch der Datenaustausch zwischen den GPUs innerhalb eines Knotens mit MPI und nicht mehr ausschließlich über die CUDA-API stattfinden müsste. Die CUDA-API führt aber einen Kopiervorgang zwischen zwei Geforce-GPUs immer dadurch aus, indem sie die Daten zuerst von der Ursprungs-GPU in den DRAM der CPU und von da aus zur Ziel-GPU kopiert. In dieser Hinsicht muss die Kommuni-

kation der GPUs innerhalb eines Knotens über MPI identisch vorgehen. Allerdings würde MPI zusätzlich einen Overhead für die Interprozesskommunikation innerhalb des Knotens verursachen. Hierbei wäre es untersuchenswert, wie groß die Performanceverluste durch den Overhead wären. Letztendlich benötigt die Visualisierung im Cluster, wie im Punkt 6.3.7 vorgestellt, eine MPI-Reduce-Operation für die Reduktion der Tiefenbilder und Dickebilder. Um Netzwerkbandbreite zu sparen ist es extrem vorteilhaft, wenn zunächst die Bilder innerhalb des Knotens reduziert werden. Hierbei wäre es wieder untersuchenswert, ob die MPI-Implementierung in dieser Hinsicht gut optimiert ist.

Schließlich soll noch kurz begründet werden, weshalb die SPH-Simulation nur eine direkte Nachbarschaftskommunikation der GPUs untereinander benötigt. So stellt die SPH-Simulation durch eine minimale Scheibendicke in Kombination mit der Schrittweitenkontrolle sicher, dass jede GPU stets nur Daten von den beiden direkten Nachbar-GPUs benötigt. Für ein Erzwingen dieser Nachbarschaftskommunikation muss eine Scheibe mindestens so dick wie die Traversierungsdistanz n_{Trav} sein. Denn in diesem Fall umfasst der Halobereich einer GPU die kompletten Scheiben der direkt benachbarten GPUs. Bei einer geringeren Dicke würde die GPU auch Daten von denjenigen GPUs benötigen, mit welchen sie nicht mehr direkt benachbart ist. Des Weiteren kann sich wegen der CFL-Bedingung für die maximale Zeitschrittweite ein Partikel pro Zeitschritt maximal nur um $0.3\Delta t \cdot 0.1c = 0.025h$ weit bewegen. Diese Strecke ist deutlich kleiner als die Kantenlänge einer Gitterzelle, die beispielhaft bei einer Traversierungsdistanz von $n_{\text{Trav}} = 3$ eine Länge von $\frac{2}{3}h$ besitzt. Dadurch kann sich auch ein Partikel durch die Zeitintegration von einer GPU aus nur zu deren direkten Nachbarn bewegen.

6.3.3 Änderungen bei der Datenhaltung

In diesem Punkt sollen die Änderungen der Datenhaltung erläutert werden. So muss die SPH-Simulation die Scheiben und damit die Gitter der GPUs für die Lastbalancierung vergrößern und verkleinern. Deswegen alloziert sie für das Gitter nicht mehr denjenigen Speicherplatz, welchen es zu Beginn benötigt, sondern soviel Speicherplatz, den das Gitter maximal durch die Lastbalancierung verbrauchen darf. Des Weiteren muss die SPH-Simulation für jede GPU zwei Gitter mit konstanter Größe für die Halobereiche der benachbarten GPUs im globalen Speicher allozieren. Durch die Lastbalancierung und durch die Zeitintegration kann sich auch die Zahl derjenigen Partikel, welche sich gerade im Volumen einer GPU befinden, erhöhen. Deshalb müssen die Partikeldatenarrays ebenfalls so groß alloziert werden, dass sie auch im schlimmsten Fall alle Partikel aufnehmen können. Zudem sind für jede GPU zwei weitere Structures aus Arrays für Partikeldaten nötig, welche dazu dienen die Partikel im Halo zwischenzuspeichern. Die SPH-Simulation verwendet die beiden Structures aus Arrays auch dafür, um bei der Zeitintegration diejenigen Partikel zwischenzuspeichern, welche sich aus dem Gitter der GPU herausbewegen.

6.3.4 Modifikationen bei den Berechnungen des Fixed-Radius-Near-Neighbors-Problems

Als Nächstes soll auf die Modifikation der Berechnungen des Fixed-Radius-Near-Neighbors-Problems eingegangen werden. Bei den Berechnungen startet der Workerthread der GPU zuerst ein Kernel um den inneren Bereich der Scheibe zu berechnen. Das Kernel geht dabei wie bereits in Punkt 4.9 vor, indem es einen GPU-Thread für jeden Partikel startet und anschließend das Fixed-Radius-Near-Neighbors-Problem für diese Partikel berechnet. Während die GPU den

inneren Bereich berechnet so kümmert sich ihr Workerthread um den Datenaustausch mit den Nachbar-GPUs über die CUDA-API und gegebenenfalls über das Netzwerk mit MPI. Dadurch aktualisiert der Thread sowohl die Partikel als auch die Gitter des Halobereichs in den extra dafür allozierten Arrays aus Punkt 6.3.3. Damit der Datenaustausch asynchron zu den inneren Berechnungen abläuft, findet er über einen separaten CUDA-Stream statt. Um den Taileffekt etwas zu reduzieren werden die Berechnungen für den Randbereich ebenfalls in diesen Stream eingereiht. Denn auf diese Weise kann die GPU, sobald die verbleibenden Thread-Blocks des Kernels des inneren Bereichs nicht mehr ausreichen um sie auszulasten, bereits mit den Berechnungen des Randbereichs anfangen. Somit wird der Tail-Effekt reduziert und die GPU kann gleichzeitig rechnen, während der Kopiervorgang läuft. Das Kernel des Randbereichs ist ebenfalls analog wie im Punkt 4.9 vorgestellt implementiert. Dabei nimmt es die selbe problemspezifische Structure mit den Berechnungsvorschriften des Fixed-Radius-Near-Neighbors-Problems als Template-Argument entgegen. Auf diese Weise entsteht wiederum eine Quelltextersparnis, da der Quelltext des Programms die Berechnungsvorschriften nicht ein zweites mal für die Berechnungen im äußeren Bereich definieren muss. Das Kernel besitzt jedoch einen kleinen Overhead. Denn bei jeder X-Gruppe muss das Kernel abfragen, ob sich die Gruppe im Gitter der GPU selbst oder im Gitter des Halobereichs befindet. Da der Randbereich jedoch in den meisten Fällen nur wenig vom gesamten Simulationsvolumen der GPU einnimmt, besitzen die Berechnungen im Randbereich insgesamt wenig Kosten. Deswegen ist der Overhead sehr wahrscheinlich zu vernachlässigen.

6.3.5 Modifikationen bei der Zeitintegration und beim des Bauens des Gitters

In diesem Punkt soll auf diejenigen Modifikationen bei der Zeitintegration und beim Bauen des Gitters eingegangen werden, welche für die Cluster Version der Arbeit implementiert worden sind. Bei der Zeitintegration ist es problematisch, dass sich durch sie Partikel aus der Scheibe einer GPU heraus und in die Scheibe einer anderen GPU hinein bewegen können. Deshalb muss die Implementierung diese Partikel möglichst effizient aus den Datenarrays ihrer alten GPU entfernen und in die Datenarrays ihrer neuen GPU einfügen. Zudem ist beim Einfügen ein Kopiervorgang zwischen den GPUs nötig. Auch hier wäre es vorteilhaft, wenn die GPU in der Zwischenzeit etwas rechnen könnte. Beim Entfernen macht sich die Implementierung zu Nutze, dass das Entfernen ohne viel Rechenaufwand beim Bauen des Gitters möglich ist, während die GPU die Partikel nach Gitterzellen sortiert. Das Finden derjenigen Partikel, welche sich aus der Scheibe herausbewegt haben, findet während der Zeitintegration statt. Für das Zwischenspeichern der gefundenen Partikel im globalen Speicher der GPU werden die Structures aus Arrays für Partikeldaten, in denen auch die Partikeldaten der Halos zwischengespeichert werden (siehe Punkt 6.3.3), verwendet. In eine Structure schreibt die SPH-Simulation diejenigen Partikel hinein, welche sich nach +Z hinausbewegen, und in die andere Structure diejenigen Partikel, welche sich nach -Z hinausbewegen. Die SPH-Simulation kopiert dann die beiden Structures von GPU zu GPU. Zudem wird das Bauen des Gitters so modifiziert, dass zumindest ein kleinerer Teil des Bauens während des Kopiervorgangs der Partikel stattfinden kann. Dies soll im Folgenden im Detail erläutert werden.

Für die Zeitintegration muss die Implementierung zunächst die globale maximale Zeitschrittweite bestimmen, welche jede GPU für sich lokal berechnet hat. Hierbei reduziert ein Thread pro Knoten die Zeitschrittgrößen der GPUs des Knotens auf ihren minimalen Wert. Anschließend wird ein MPI-Allreduce mit der Min-Operation ausgeführt um die globale maximale Zeitschrittweite zu ermitteln. Das Ergebnis teilt der Thread nun wiederum mit allen anderen Threads

des Knotens.

Als Nächstes werden die Änderungen der Zeitintegration selbst vorgestellt. Da die Zeitintegration in der Cluster-Version komplexer ist, abstrahiert beziehungsweise modularisiert sie die Implementierung durch eine Template-Funktion. Diese ist in Abbildung 6.5 zu sehen. Dabei definiert die Implementierungen die Berechnungen der Zeitintegration selbst wieder in einer problemspezifischen Structure. Die problemspezifische Structure für den ersten Teilschritt des Prädiktor-Korrektor-Verfahrens ist in Abbildung 6.6 zu sehen.

So wird bei der Zeitintegration wieder analog zu Punkt 4.10 für jedes Partikel in der Scheibe der GPU ein Thread gestartet. Dabei berechnet der Thread mit Hilfe einer Funktion der problemspezifischen Structure die neuen Strömungsgrößen des Partikels. Anschließend wird anhand der neuen Position eines jeden Partikels zusätzlich abgefragt, ob sich das Partikel noch nach der Bewegung in der Scheibe der GPU befindet. Dementsprechend passiert folgendes:

- **Das Partikel befindet sich noch in der Scheibe der GPU:** In diesem Fall schreibt der Thread die neuen aktualisierten Größen des Partikels an dessen Stelle zurück in diejenige Structure aus Arrays, die die Partikel der Scheibe der GPU enthält.
- **Das Partikel befindet sich nicht mehr in der Scheibe der GPU:** Nun schreibt der Thread um etwas Speicherbandbreite zu sparen nur noch die neue Position an die Stelle seines Partikels zurück in die Structure aus Arrays, die die Partikel der Scheibe der GPU enthält. Dadurch kann die GPU das Partikel anhand seiner neuen Position später beim Bauen des Gitters aussortieren. Zudem muss der Thread das Partikel mit nicht nur den geänderten Partikeldaten sondern mit allen Partikeldaten in diejenige Structure aus Arrays für Partikel, welche später zur entsprechenden Nachbar-GPU kopiert wird, zurückschreiben. Dafür wird zunächst per atomarer Operation Speicherplatz für das Partikel in der entsprechenden Structure alloziert und es anschließend an die entsprechende Stelle zurückgeschrieben.

Dieses Herausbewegen kann jedoch nicht nur durch die Eigenbewegung des Partikels passieren, sondern auch dadurch, dass die Lastbalancierung die Scheibengrenzen der GPUs verschiebt. Deshalb kann es vor allem durch die Lastbalancierung vorkommen, dass sich viele aufeinanderfolgende Partikel aus der Scheibe der GPU herausbewegen. Um dabei die Last etwas von den Atomic-Units zu nehmen findet diese Allozierung innerhalb der Structures aus Arrays auf Warpbasis statt. Auf diese Weise benötigt das Kernel maximal eine atomare Operation pro Warp. Dafür ist eine Prefix-Sum innerhalb eines Warps notwendig. Das Kernel implementiert die Prefix-Sum durch die performanten Warp-Shuffle-Instruktionen, welche allerdings immer noch teuer sind. Da sich in den allermeisten Fällen jedoch kein einziges Partikel des Warps aus der Scheibe heraus bewegen wird, ist sie zudem meist überflüssig. Deshalb wird vor der Berechnung der Prefix-Sum mit der All- und Any-Warp-Vote-Instruktion überprüft, ob sich überhaupt ein Partikel des Warps aus der Scheibe herausbewegt hat. Wenn nicht, so kann die Prefix-Sum übersprungen werden. Durch diese Optimierungen verhält sich diese Version der Zeitintegration weitestgehend wie die Version ohne GPU-Cluster, weshalb sie ebenfalls stark durch die Speicherbandbreite limitiert ist.

Schließlich wird auf die Modifikation beim Bauen des Gitters eingegangen. Hierfür wurde nur die Version der Konstruktion mit Counting-Sort spezifisch auf den GPU-Cluster optimiert. Deshalb wird auch nur sie hier erläutert. So startet beim Bauen des Gitters zunächst jeder Workerthread den Kopiervorgang der Structures aus Arrays mit denjenigen Partikeln, welche sich

```
template<class TimeIntegrationStruct>
__global__ void MoveParticlesKernel(ParticleData PartCenter, ParticleData PartUp, ParticleData PartDown,
                                    ParticleGrid Grid)
{
  uint ThreadID = blockIdx.x * blockDim.x + threadIdx.x;

  TimeIntegrationStruct TI;
  TI.Integrate(PartCenter, ThreadID);
  bool WriteBackDown = Grid.ZMin > TI.GetPositionNew().z;
  bool WriteBackUp = Grid.ZMax <= TI.GetPositionNew().z;
  bool WriteBackCenter = !(WriteBackUp || WriteBackDown);

  if(WriteBackCenter)
    TI.WriteBack<ChangesOnly>(PartCenter, ThreadID);
  else
    TI.WriteBack<PositionOnly>(PartCenter, ThreadID);

  if(__all(WriteBackCenter))
    return;

  if(__any(WriteBackDown))
  {
    int WriteBackDownID = AllocateParticlesPerWarp(PartDown,WriteBackDown);
    if(WriteBackDown)
      TI.WriteBack<AllData>(PartDown, WriteBackDownID);
  }

  if(__any(WriteBackUp))
  {
    int WriteBackUpID = AllocateParticlesPerWarp(PartUp,WriteBackUp);
    if(WriteBackUp)
      TI.WriteBack<AllData>(PartUp, WriteBackUpID);
  }
}
```

Abbildung 6.5: Template für das Kernel der Zeitintegration im Cluster

in die Scheibe beziehungsweise in das Gitter seiner GPU hineinbewegt haben. Dabei werden die Partikel, welche sich in die Scheibe der GPU hineinbewegen, hinten in diejenige Structure aus Arrays, die die Partikel der Scheibe der GPU enthält, eingefügt.

Während des Kopiervorgangs kann die GPU bereits den Initialisierungsschritt des Counting-Sorts komplett und den Einfügen-Schritt des Counting-Sorts teilweise ausführen. Denn erst für die Prefix-Sum benötigt die GPU die Daten von allen Partikeln, die sich in ihrem Gitter befinden. Dabei wird der Einfügen-Schritt zunächst nur mit denjenigen Partikeln ausgeführt, welche sich zu Beginn des Bauens bereits im Gitter der GPU befanden. Das Kernel dieses Schritts wird ebenfalls so modifiziert, dass das Inkrementieren nur statt findet, wenn sich der entsprechende Partikel noch im Gitter der GPU befindet. Ist der Kopiervorgang abgeschlossen, so wird der Einfügen-Schritt noch einmal mit den neu hinzugekommenen Partikeln ausgeführt. Anschließend findet die Prefix-Sum analog zu der Version ohne Cluster statt. Zuletzt muss die Permutation modifiziert werden, wobei in der Implementierung nur die Rückwärtspermutation

```
struct PredictorStructFirstSubStep
{
  float3 PositionT; float3 VelocityT; float DensityT;
  float3 PositionTHalf; float3 VelocityTHalf; float DensityTHalf;

  float3 GetPositionNew()
  {
    return PositionTHalf;
  }

  void Integrate(ParticleData &Particles, uint PartID)
  {
    float4 Changes = Particles.Changes[PartID];
    float3 Acceleration = make_float3(Changes);
    float DensityChange = Changes.w;

    float4 PositionAndDensity = Particles.PositionsAndDensitiesT[PartID];
    PositionT = make_float3(PositionAndDensity);
    VelocityT = make_float3(Particles.VelocitiesT[PartID]);
    DensityT = PositionAndDensity.w;

    PositionTHalf = PositionT + 0.5f*DT*VelocityT;
    VelocityTHalf = VelocityT + 0.5f*DT*Acceleration;
    DensityTHalf = DensityT + 0.5f*DT*DensityChange;
  }

  template<uint WriteBackAmount>
  void WriteBack(ParticleData &Particles, uint PartID)
  {
    if(WriteBackAmount == PositionOnly ||
       WriteBackAmount == ChangesOnly ||
       WriteBackAmount == AllData)
       Particles.PositionsTHalf[PartID] = make_float4(PositionTHalf, DensityTHalf);

    if(WriteBackAmount == ChangesOnly ||
       WriteBackAmount == AllData)
       Particles.VelocitiesTHalf[PartID] = make_float4(VelocityTHalf);

    if(WriteBackAmount == AllData)
    {
       Particles.PositionsAndDensitiesT[PartID] = make_float4(PositionT, DensityT);
       Particles.VelocitiesT[PartID] = make_float4(VelocityT);
    }
  }
};
```

Abbildung 6.6: Problemspezifische Structure für den ersten Schritt des Prädiktor-Korrektor-Verfahrens

angepasst wurde. So berechnet das Kernel nur noch die Permutationsindexe von denjenigen Partikeln, welche sich noch in der Scheibe der entsprechenden GPU befinden. Die Permutation selbst läuft wieder analog zur Version ohne GPU-Cluster ab.

Auf diese Weise sortiert die GPU diejenigen Partikel, welche die Scheibe der GPU verlassen haben, nun beim Bauen des Gitters aus. Zusätzlich führt sie zumindest die Berechnungen der Gitterkonstruktion teilweise während des Kopiervorgangs durch. Allerdings sind die Schritte der Initialisierung und des Einfügens günstig und kosten nur 1% der Gesamtlaufzeit. Deshalb muss die GPU beim Bauen des Gitters, falls währenddessen viele Partikel kopiert werden, vermutlich dennoch auf das Kopieren warten.

6.3.6 Lastbalancierung

Als Nächstes soll auf die Lastbalancierung eingegangen werden. Bei einer eindimensionalen Aufteilung verwirklichen SPH-Simulationen die Lastbalancierung dadurch, dass sie die Scheibengrenzen entlang der Z-Achse verschieben, damit die GPUs gleichmäßig ausgelastet sind.

In dieser Arbeit wird nur ein einfacher lokaler Ansatz für die Lastbalancierung verwendet. Zusätzlich wird die Rechenlast über die Laufzeit ermittelt, da eine solche Lastbalancierung vorteilhafter ist. Dabei wird die Rechenlast in dieser Arbeit durch diejenige Zeit bestimmt, die eine GPU benötigt um die Berechnungen des Fixed-Radius-Near-Neighbors-Problems im inneren Bereich ihrer Scheibe durchzuführen. Da die Lastbalancierung nur alle paar Zeitschritte durchgeführt wird, werden diese einzelnen Laufzeiten für das Abschätzen der Rechenlast aufsummiert. Somit wird die Rechenlast für die Berechnungen des Fixed-Radius-Near-Neighbors-Problems im Halobereich, der Gitterkonstruktion und der Zeitintegration wegen ihrer Geringfügigkeit ignoriert.

Das Verschieben der Scheibengrenzen entlang der Z-Achse für die Lastbalancierung findet alle paar Zeitschritte zu Beginn der Zeitintegration statt. Die unmittelbar folgende Zeitintegration sorgt dann automatisch dafür, dass die Partikel passend zwischen den GPUs kopiert werden. Je nachdem ob eine benachbarte GPU für die Berechnungen der Fixed-Radius-Near-Neighbors-Probleme länger oder kürzer gebraucht hat, wird die Scheibe der GPU an dieser Stelle entlang der Z-Achse entsprechend um eine Gitterzelle vergrößert oder verkleinert. Dabei darf die minimale Länge entlang der Z-Achse nicht unterschritten werden, da ansonsten bei der Implementierung jede GPU nicht nur mit den direkten Nachbar-GPUs kommunizieren müsste. Ebenfalls ist die maximale Länge beschränkt, da ein größeres Gitter nicht mehr in den dafür allozierten Speicher der GPU passen würde. Des Weiteren findet die Verschiebung der Scheibengrenzen nur alle paar Zeitschritte statt, weil sie teuer ist. Denn wegen ihr müssen viele Partikeldaten zwischen den GPUs übertragen werden und zudem können die GPUs in dieser Zeit beim Bauen des Gitters nur wenig berechnen. Da sich die Partikel wegen der Schrittweitenkontrolle pro Zeitschritt jedoch deutlich weniger weit als eine Gitterzelle bewegen, kann die Rechenlast trotz der seltenen Lastbalancierung gut auf die GPUs verteilt werden.

Diese Art der lokalen Lastbalancierung ist jedoch im Gegensatz zu fortgeschrittenen Ansätzen aus der Literatur nur vergleichsweise einfach. Weiterführend könnte deshalb die Lastbalancierung zum Beispiel mit dem vielversprechenden globalen Ansatz aus [ER] verbessert werden.

6.3.7 Modifikation des Zeichenvorgangs

In diesem Punkt soll vorgestellt werden, wie der Zeichenvorgang für den GPU-Cluster erweitert worden ist. Dabei teilen sich die nun folgenden Erläuterung in zwei Teile auf. So stellt dieser Punkt zunächst die Grundidee der Compositing-Technik vor, wodurch die Implementierung die Rechenlast der Visualisierung effizient auf die einzelnen GPUs verteilen kann. Anschließend erklärt er, wie die Implementierung die Compositing-Technik effizient in einer Pipeline umsetzt.

Hierfür wird zuerst auf die Grundidee des Compositings eingegangen. Da der Screen-Space-Curvature-Flow und das finale Shading lediglich von der Partikelanzahl unabhängige Kosten besitzen, welche auch eine einzige GPU problemlos bewältigen kann, muss nur die Rechenlast des Splattings, die mit der Partikelanzahl zunimmt, auf die GPUs des Clusters verteilt werden. Dies kann effizient mit einer Compositing-Technik geschehen, deren Grundidee beispielhaft noch einmal allgemein in dem Artikel [TR] erläutert ist. Eine solche Compositing-Technik basiert immer darauf, dass jede GPU ein Teilbild mit den in ihrem DRAM vorhandenen Daten rendert. Anschließend werden alle Teilbilder per Reduktionsoperation zu einem einzigen Bild zusammengesetzt. Im Speziellen kann das Compositing beim Splatting ausnutzen, dass es mehrere Tiefenbilder korrekt zu einem einzigen Tiefenbild kombinieren beziehungsweise reduzieren kann, indem es für jeden Pixel das Minimum aus allen Tiefenbildern verwendet. Ebenso kann es mehrere Dickebilder korrekt zu einem einzigen Dickebild kombinieren, indem es die Dickewerte aus allen Bildern für jeden Pixel aufsummiert. Dies ist in Abbildung 6.7 zu sehen. Dadurch kann jede GPU das Splatting für sich mit denjenigen Partikeln, welche sich gerade in ihrer Scheibe und damit in ihrem Speicher befinden, ausführen, um ein Tiefenbild und ein Dickebild zu erzeugen. Somit kann die Rechenleistung aller GPUs für das Splatting verwendet werden. Diese Bilder lassen sich danach zu einem einzigen Tiefenbild beziehungsweise Dickebild reduzieren. Auf diese Weise vermeidet die Implementierung ebenfalls, dass für das Splatting selbst Partikel zwischen den einzelnen GPUs kopiert werden müssen. Dadurch skaliert die für die Reduktion benötigte PCI-Express-Bandbreite und Netzwerkbandbreite nur mit der Auflösung, der GPU-Zahl und der Knotenzahl, nicht jedoch mit der Partikelanzahl.

Diese Grundidee muss allerdings nun effizient in eine Pipeline implementiert werden. Diese Pipeline ist in Abbildung 6.8 gezeigt. So berechnet für die Visualisierung jede GPU zuerst die Hauptachsen der Ellipsoide gemäß der generischen Version des Fixed-Radius-Near-Neighbors-Problems aus Punkt 4.9. Anschließend findet der Zeichenvorgang mit OpenGL selbst statt, bei welchem wieder jede GPU durch ihren eigenen Workerthread angesteuert wird. Dabei erstellt jede GPU zuerst ein Tiefenbild des Terrains. Obwohl das Tiefenbild des Terrains auf allen GPUs das gleiche ist, wird hierbei von eine Lastverteilung abgesehen, da das Erstellen sehr günstig ist. Dann erfolgt das Splatting der Ellipsoide, durch welches jede GPU ein Tiefenbild und ein Dickebild aus ihren Partikeln erzeugt. Schließlich werden die Bilder des Splattings von den GPUs in den Speicher des Hosts vom jeweiligen Knoten kopiert. Dort erfolgt zuerst eine Reduktion dieser Bilder per Additions-Operation beziehungsweise per Minimums-Operation innerhalb des Knotens. Dadurch besitzt jeder Knoten ein Dickebild und ein Tiefenbild, welche sich aus den Dickebildern beziehungsweise Tiefenbildern all seiner GPUs zusammensetzen. Diese Bilder werden anschließend per MPI-Reduktion per Additions-Operation beziehungsweise per Minimums-Operation zu einem einzigen Dickebild und Tiefenbild reduziert. Als Letztes kann darauf der Screen-Space-Curvature-Flow und das finale Shading angewendet werden.

Da die GPUs gleichzeitig kopieren und rechnen können, wird in der Pipeline mit den SPH-Berechnungen fortgefahren, sobald die GPUs mit dem Splatting fertig sind. Dadurch findet das Herunterladen von den GPUs auf den Host und das Reduzieren über MPI asynchron zu den

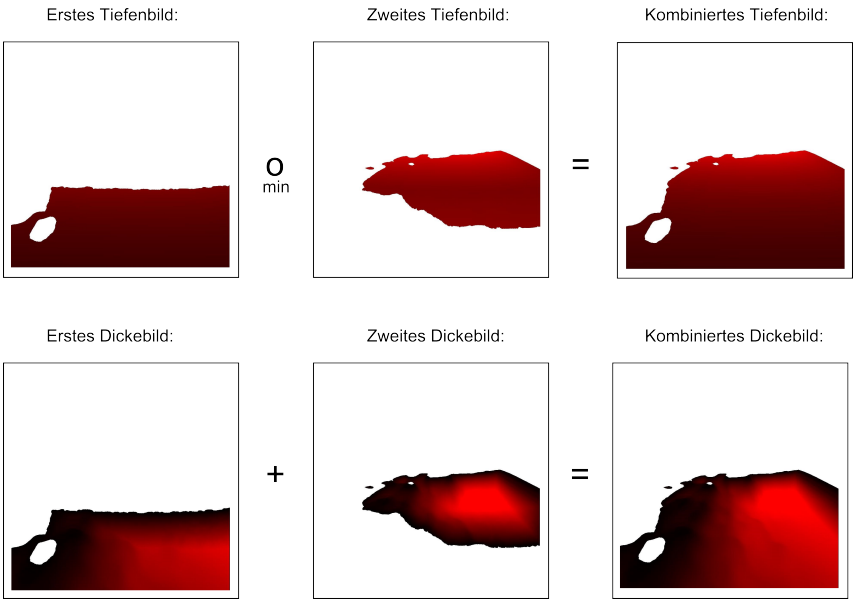


Abbildung 6.7: Kombination mehrerer Dickebilder und Tiefenbilder zu einem einzigen Dickebild beziehungsweise Tiefenbild

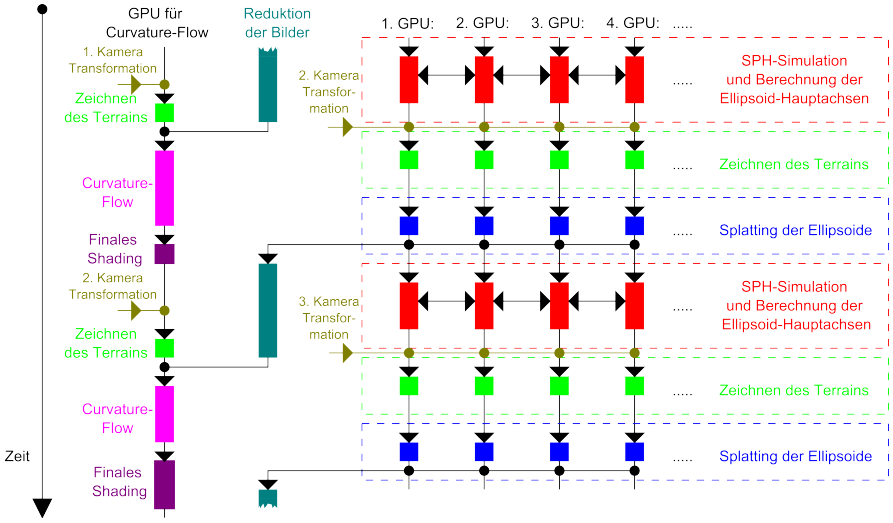


Abbildung 6.8: Pipeline der Visualisierung im Cluster

Berechnungen der SPH-Simulation statt. Da die SPH-Berechnungen ebenfalls MPI verwenden benötigt die Reduktion für die Asynchronität eine eigene MPI-Kommunikationsgruppe. Diese Asynchronität ist sinnvoll, da die Reduktion über das Netzwerk lange dauert und die GPUs dadurch währenddessen weiter an der SPH-Simulation rechnen können. Allerdings ist die Netzwerkbandbreite für die SPH-Simulation währenddessen reduziert.

Das Erstellen des Farbbild des Terrains, der Screen-Space-Curvature-Flow und das finale Shading finden dabei mit einer separaten GPU statt, welche ebenfalls über einen separaten Prozess angesteuert wird. Diese GPU ist mit ihrem Prozess nur dafür zuständig und nimmt nicht an der physikalischen SPH-Simulation teil. Dadurch wird vermieden, dass andere GPUs für die Simulation auf eine einzige GPU, während diese den Screen-Space-Curvature-Flow berechnet, warten müssen. Dieser Prozess übernimmt ebenfalls die Verarbeitung und das Versenden der Kamerabewegung durch den Benutzer über MPI-Scatter-Operationen. Dabei ist es eine weitere Herausforderung, dass zu Beginn jedes Zeichenvorgangs für alle Knoten ein und die selbe aktuelle Kameratransformationsmatrix zur Verfügung stehen muss, welche der Benutzer zur Laufzeit verändern können soll. Hier gilt es zudem zu berücksichtigen, dass der Zeichenvorgang für ein gegebenes Bild auf dem Knoten für den Screen-Space-Curvature-Flow immer wesentlich später startet, als die Zeichenvorgänge auf den SPH-Simulationsknoten. Deshalb findet das Verteilen der Kameratransformationsmatrix ebenfalls asynchron zu den SPH-Berechnungen und der Reduktion in einer eigenen MPI-Kommunikationsgruppe durch einen separaten Thread pro Knoten statt. Zusätzlich wird die Matrix über First-In-First-Out-Buffer auf den einzelnen Knoten zwischengespeichert. Denn dadurch wird ihr Vorhandensein zu Beginn eines Zeichenvorgangs auf allen Knoten garantiert.

6.3.8 Software-Design

In diesem Punkt soll kurz das abstrahierende Software-Design vorgestellt werden, wie es in dieser Arbeit für die physikalischen SPH-Berechnungen implementiert worden ist.

So zeigte es sich in den vorangegangenen Kapiteln, dass sich die einzelnen Pässe des Fixed-Radius-Near-Problems nur an wenigen Stellen des gesamten Algorithmus durch ihre problemspezifischen Berechnungen unterscheiden. Die generelle Vorgehensweise ist aber immer identisch. Hierunter fällt zum Beispiel das Traversieren des Gitters oder die Synchronisation des Halos im GPU-Cluster. Analog verhält es sich mit dem Bewegen der Partikel durch die Zeitintegration und dem Bauen des Gitters. Deswegen ist es sinnvoll die SPH-Berechnungen im Software-Design etwas zu abstrahieren. Aus diesem Grund implementierte die Arbeit eine abstrakte SPH-Löser-Klasse. Dabei müssen die davon abgeleiteten Klassen die virtuelle Funktion des Workerthreads überschrieben. Die Funktion muss dann die Berechnungen der Fixed-Radius-Near-Neighbors-Probleme und der Zeitintegrationen vorgeben. Eine solche Funktion für dasjenige SPH-Verfahren, bei welchem die Dichte zeitlich integriert wird, ist in Abbildung 6.9 zu sehen.

Für die Berechnung eines Fixed-Radius-Near-Neighbors-Problems muss ein Workerthread eine Template-Funktion der abstrakten SPH-Löser-Klasse aufrufen. Als Template-Parameter dienen wieder die problemspezifischen Structures aus 4.9, welche nur die Berechnungen des Fixed-Radius-Near-Neighbors-Problems beinhalten. Zusätzlich muss übergeben werden, welche Daten zu synchronisieren sind. Diese Funktion ist dann nun dafür zuständig all diejenigen Vorgänge, welche bei der Berechnung jedes Fixed-Radius-Near-Neighbors-Problems identisch sind, auszuführen. So synchronisiert sie wie vorgegeben das Halo und startet die Fixed-Radius-Near-

Neighbors-Kernel mit den problemspezifischen Berechnungen im inneren Bereich sowie im äußeren Bereich.

Die Zeitintegration läuft analog ab. Der Workerthread ruft eine Template-Funktion der abstrakten SPH-Löser-Klasse auf, welcher er als Template-Parameter die problemspezifische Structure für die Zeitintegration übergibt. Zusätzlich gibt er beim Funktionsaufruf an, welche Daten der sich bewegenden Partikel zwischen den GPUs kopiert werden müssen. Anschließend kümmert sich die Template-Funktion um das Starten des Kernels der Zeitintegration, das Kopieren der Partikeldaten, und das Bauen des Gitters. Für das Bauen des Gitters wird jedoch zusätzlich dasjenige Array mit Partikeldaten benötigt, in welchem die nach der Zeitintegration aktuellen Positionen stehen. Der Index dieses Arrays innerhalb der Structure aus Arrays muss deshalb ebenfalls übergeben werden.

Auf diese Art kann ein Programmierer einfach neue SPH-Verfahren für GPU-Cluster implementieren, da er hierfür nur die problemspezifischen Berechnungen selbst definieren muss. Dies ist insbesondere vorteilhaft, da es kein einziges festes sondern viele unterschiedliche SPH-Verfahren gibt. Ebenso kann er mit dieser Klasse problemlos andere Fixed-Radius-Near-Neighbors-Probleme wie Moleküldynamik-Simulationen auf einem GPU-Cluster berechnen. Allerdings gibt es bei der abstrakten Implementierung ebenfalls Einschränkungen. So gibt es gegenwärtig nur eine Structure aus Arrays für Partikeldaten der Flüssigkeit. Ebenfalls ist nur ein Gitter vorgesehen. Muss eine Simulation mehrere Partikeltypen, wie zum Beispiel einen Partikeltyp für partikelbasierte Festkörperrandbedingung, in unterschiedlichen Structures abspeichern, so ist dies momentan nicht möglich. Zudem ist die Visualisierung selbst zwar logisch stark von der SPH-Löser-Klasse getrennt. Jedoch ist sie weitestgehend unmodular, wodurch sie die Flüssigkeit nur so visualisieren kann, wie es in Kapitel 5 beschrieben worden ist.

6.4 Untersuchungen

6.4.1 Test-Cluster, Messmethoden, Testszene

Im Folgenden sollen die Untersuchungen der Cluster-Version durchgeführt werden. Dafür müssen zunächst der Test-Cluster, die Messmethoden und die Testszene vorgestellt werden. Als erstes soll kurz der Test-Cluster vorgestellt werden. Dieser besteht aus folgenden Knoten mit folgenden GPUs:

1. Knoten:

- **1. GPU:** Geforce Titan Black
- **2. GPU:** Geforce Titan

2. Knoten:

- **1. GPU:** Geforce Titan Black
- **2. GPU:** Geforce Titan Black

3. Knoten:

- **1. GPU:** Geforce Titan

Visualisierungsknoten: 1 Geforce 750 GTX Ti

```
void IntegrateDensitySPHSolver::WorkerThreadFunction()
{
  if(StepCount % StepsPerShepard == 0)
  {
    CalcFRNNProblem<ShepardProblemStruct>(PositionsAndDensitiesT);
    SwapParticlePointer(PositionsAndDensitiesT, PositionsAndDensitiesAfterShepard);
  }

  CalcFRNNProblem<ChangesProblemStructFirstSubStep>(PositionsAndDensitiesT | VelocitiesT);

  MoveParticlesAndBuildGrid<PredictorStructFirstSubStep>(PositionsAndDensitiesTHalf, // New Positions
                                                         PositionsAndDensitiesT| VelocitiesT |
                                                         PositionsAndDensitiesTHalf | VelocitiesTHalf);

  CalcFRNNProblem<ChangesProblemStructSecondSubStep>(PositionsAndDensitiesTHalf | VelocitiesTHalf);

  MoveParticlesAndBuildGrid<PredictorStructSecondSubStep>(PositionsAndDensitiesT, // New Positions
                                                          PositionsAndDensitiesT| VelocitiesT);

  if( (StepCount +1) % StepsPerVisualization == 0)
  {
    CalcFRNNProblem<WeightedMeansProblemStruct>(PositionsAndDensitiesT);
    CalcFRNNProblem<EllipsoidAxisProblemStruct>(WeightedMeans);
  }
}
```

Abbildung 6.9: Workerthreadfunktion für das SPH-Verfahren mit zeitlicher Integration der Dichte

Die Knoten verwenden allesamt das CUDA-Toolkit 6.5, die GPU-Treiberversion 340.58 sowie ein Open-Suse-Linux als Betriebssystem. Alle Knoten sind zudem sternförmig über ein 1 Gigabit/s Ethernet durch einen einzigen Ethernet-Switch miteinander verbunden. Bei ihm handelt es sich um einen HP 1800-24G. Die Geforce Titan Blacks sind ebenfalls GPUs der Compute-Capability 3.5, besitzen im Vergleich zu den herkömmlichen Titans 15 statt 14 Multiprozessoren und sind etwas schneller getaktet. Dadurch erhöht sich ihre Rechenleistung ohne Boost von 4500 SP-GFLOPS auf 5121 SP-GFLOPS und ihre Speicherbandbreite von 288 GB/s auf 337 GB/s. Um Messfehler durch den Boost zu vermeiden wurde wieder im gesamten Cluster die Double-Precision-Funktion aktiviert, damit der Boost Takt bei allen verwendeten GPUs deutlich geringer wird.

Für die SPH-Simulation wird wiederum dasjenige Verfahren untersucht, bei welchem die Dichte zeitlich integriert wird. Für die Tests werden insgesamt 500 Zeitschritte berechnet. Die Lastbalancierung wird nur alle 10 Zeitschritte ausgeführt. Denn dadurch kann die SPH-Simulation selbst die allermeiste Zeit die Rechenleistung der GPUs ausnutzen. Zusätzlich wird dadurch, wenn die Reduktion der Tiefenbilder und Dickebilder nicht berücksichtigt wird, ein Großteil der Netzwerkbandbreite für die Synchronisation des Halos und nicht für die Bewegung der Partikel benötigt. Für die Visualisierung wird wiederum eine Auflösung von 1024 auf 1024 gewählt. Zudem wird die Visualisierung nur alle 20 Zeitschritte aufgerufen. Dadurch spielen die Kosten des Splattings und der Berechnung der Hauptachsen der Ellipsoide nur eine gerin-

ge Bedeutung für die Performance. Theoretisch wäre es sinnvoll die Visualisierung nicht alle paar Zeitschritte sondern in Abhängigkeit von der Simulationszeit aufzurufen. Da jedoch im Folgenden die Partikelzahl über den Startabstand variiert wird, könnte dies die angestrebten Erkenntnisse aus den Messungen erschweren. Deshalb wird hierauf verzichtet. Zusätzlich werden 500 Screen-Space-Curvature-Flow-Iterationen verwendet. Hier wäre ebenfalls eine Anpassung an den Startabstand sinnvoll, da bei größeren Startabständen mehr Glättung benötigt wird. Da es wiederum die angestrebten Erkenntnisse der Messergebnisse erschweren würde, wird darauf verzichtet.

Im Folgenden werden die GPU-Auslastungen und die Netzwerkbandbreiten gemessen. Die GPU-Auslastung wird hierbei definiert als das Verhältnis zwischen derjenigen Zeit, in der die GPU etwas berechnet, und der Gesamtlaufzeit. Die Rechenzeit lässt sich dabei leicht über CUDA-Events bestimmen. Interessanterweise gibt die GPU-Auslastung dadurch nicht an, wie gut die GPU während ihrer Rechenzeit ausgenutzt wird. So lässt sich beispielhaft bereits mit einem einzigen GPU-Thread, welcher die Rechenleistung der GPU nur extrem schlecht ausnutzen kann, eine GPU-Auslastung von 100% erzielen. Allerdings lässt sich durch die GPU-Auslastung leicht erkennen, ob die gesamte Simulation durch etwas anderes, wie die Netzwerkbandbreite oder durch das Warten auf andere GPUs, limitiert wird.

Die Netzwerkbandbreite wird im Falle, dass keine Visualisierung verwendet wird, direkt berechnet. Dies hat den Vorteil, dass die Berechnung sehr genau ist. Im Falle der Simulation mit Visualisierung lässt sich die Netzwerkbandbreite jedoch nicht direkt berechnen, da es unklar ist, wie das MPI-Reduce für die Tiefenbilder und die Dickebilder intern funktioniert. Deshalb wird in diesem Fall der Systemmonitor für das Ablesen der Netzwerkbandbreite verwendet. Hier ist allerdings unklar, wie der Systemmonitor den Wert genau berechnet. Zudem ist der Systemmonitor nur schwer abzulesen, wodurch sich ein großer Fehler ergibt. Allerdings stimmte der durch das Ablesen ermittelte Wert bei diversen einfachen Tests in etwa mit dem direkt berechneten Wert überein. Es ist im Folgenden zudem interessant zu sehen, wie gut die Simulation die Netzwerkbandbreite ausnutzt. Um zu sehen wie viel Bandbreite überhaupt maximal über das Netzwerk erreicht werden kann wurde ein einfacher Benchmark geschrieben. Da das Ethernet-Netzwerk full-duplex ist, scheint es sinnvoll sowohl den unidirektionalen Transfer als auch den bidirektionalen Transfer zu untersuchen. Dabei ergab das Benchmark einen Wert von 115 MB/s mit unidirektionalem Transfer. Bei bidirektionalem Transfer fiel der Wert auf jeweils 102 MB/s in jede der beiden Richtungen ab. Bei einer gegebenen Transferart kann nun aus dem Verhältnis zwischen maximaler Bandbreite und erreichter Bandbreite die Ausnutzung der Netzwerkbandbreite berechnet werden.

Als Testszene wird eine schiefe Ebene gewählt, deren eine Hälfte in etwa mit Flüssigkeit überschwemmt ist. Zu Beginn ist das Volumen der Simulation gleichmäßig auf die GPUs der SPH-Simulation aufgeteilt. Zusätzlich verläuft die Neigung der Ebene entlang der Z-Achse der Simulation. Dadurch besitzt jede GPU in etwa zu Beginn gleich viele Partikel und dementsprechend in etwa die gleiche Rechenlast. Dies ist vorteilhaft, da die Lastbalancierung bei ungünstigen Anfangsverteilungen viele Zeitschritte benötigt um die Rechenlast auf die GPUs aufzuteilen. Auf diese Weise könnten die Messergebnisse in Kombination mit dem kurzen Messintervall verfälscht werden. Da die Geforce Titan Blacks jedoch in etwa 14 % schneller als die Geforce Titans und die Partikel zusätzlich zu Beginn gleichmäßig verteilt sind, wird die Lastbalancierung dennoch etwas benötigt. Die Wahl der Szene hat auch den weiteren Vorteil, dass die Flüssigkeitsverteilung für die Aufteilung der Simulation in Scheiben entlang der Z-Achse ebenfalls optimal ist. Auf diese Weise muss die Simulation weniger Partikeldaten im Halobereich syn-

chronisieren.

6.4.2 Skalierung mit der Partikelzahl

Anschließend kann mit den Untersuchungen selbst begonnen werden. Als erstes soll die Skalierung der Simulation inklusive Visualisierung mit der Partikelzahl bei der maximalen Knotenzahl von drei Knoten und einem Visualisierungsknoten untersucht werden. Dabei wird die Partikelzahl über die Änderung des Startabstandes der Partikel Δx variiert. Die so gemessenen Laufzeiten, Netzwerkbandbreiten und GPU-Auslastungen wurden in Abbildung 6.10 eingetragen. Die Netzwerkbandbreite wurde bei der Messung aus dem Systemmonitor abgelesen, weil die Messung mit Visualisierung durchgeführt wurde.

Die durchgeführten Messungen sollen als Nächstes diskutiert werden. Zuerst soll kurz auf die Qualität der Messung der Netzwerkbandbreite eingegangen werden. Denn diese ist durch das Ablesen durch den Systemmonitor potentiell von einem großen Fehler behaftet. Hierfür werden zunächst die Messungen mit den großen Partikelanzahlen herangezogen. Denn hier trägt wahrscheinlich das MPI-Reduce der Dicke- und Tiefenbilder nur einen Bruchteil zur Gesamtbandbreite bei, da dessen benötigtes Netzwerkvolumen nicht mit der Partikelzahl skaliert. Die Vermutung wird dadurch, dass die Bandbreite des Visualisierungsknotens gering ist gestützt. In diesem Fall verursacht die SPH-Simulation selbst nun den Großteil der benötigten Netzwerkbandbreite. Des Weiteren muss der zweite Knoten der SPH-Simulation mit dem ersten und dritten Knoten, welche beide am Rand der SPH-Simulation liegen, kommunizieren, während der erste und dritte Knoten nur mit dem mittleren Knoten kommunizieren müssen. Deshalb muss die eingehende Netzwerkbandbreite in den inneren Knoten die Summe der ausgehenden Netzwerkbandbreiten der beiden Randknoten sein. Das gleiche gilt für die ausgehende Netzwerkbandbreite des zweiten Simulationsknotens und die eingehenden Netzwerkbandbreiten in den beiden Randknoten. Daraus lässt sich ein relativer Fehler für die Summe der eingehenden und ausgehenden Bandbreiten der beiden Randknoten in Relation zu den Bandbreiten des mittleren Knotens berechnen. Die Fehler wurden bei den großen Partikelzahlen von 11 Millionen bis 126 Millionen berechnet und anschließend wurde ihr Betrag gemittelt. Dadurch ergibt sich ein mittlerer Fehler von in etwa 5%. Bei den geringen Partikelzahlen von 0.52 bis 2.1 Millionen wird der Fehler deutlich größer, da hier die Reduktion der Bilder mehr zur Bandbreite beiträgt. So beträgt der Fehler bei der Partikelzahl von 0.52 Millionen bereits 19% und bei einer Partikelzahl von 0.98 Millionen noch 12%. Dies verdeutlicht, dass die Messung trotz ihrer Ungenauigkeit ausreichend gut ist, und für die weitere Diskussion verwendet werden kann.

Als Nächstes soll auf die Skalierung selbst eingegangen werden. Hierfür ist es interessant zu sehen, wie sich der Partikeldurchsatz, also das Verhältnis zwischen Partikelzahl und Laufzeit, des Clusters mit zunehmender Partikelzahl ändert. Der so berechnete Partikeldurchsatz wurde noch auf den maximal gemessenen Partikeldurchsatz normiert und in die Tabelle in Abbildung 6.11 eingetragen. Zusätzlich ist die Netzwerkbandbreite des zweiten Knotens von Interesse, da diese in den Messungen am höchsten ist, und deshalb potentiell als erstes limitieren wird. Da der zweite Knoten in etwa genauso viel Daten verschickt wie empfängt wurde die Ausnutzung der Netzwerkbandbreite im Falle eines bidirektionalen Transfers berechnet und in Abbildung 6.11 eingetragen. Ebenso wurde die mittlere GPU-Auslastung aller Simulationsknoten berechnet und in die selbe Tabelle eingetragen. Anschließend wurden alle drei in das Diagramm in Abbildung 6.12 eingetragen. Das Diagramm demonstriert, dass der Partikeldurchsatz für 69 Millionen Partikel am höchsten ist, und sowohl für mehr Partikel als auch für weniger Partikel

Δx **in m**	1.10	0.90	0.70	0.50	0.40	0.35
Partikelzahl in Mio.	0.52	0.98	2.1	5.8	11	17
Laufzeit in s	17	23	38	72	139	216
1. Simulationsknoten						
Auslastung der 1. GPU	40%	49%	65%	85%	90%	90%
Auslastung der 2. GPU	49%	58%	74%	90%	94%	95%
Netzwerk I/O in MB/s	46 /47	45 /46	45 / 41	39 /40	34 / 29	27 /24
2. Simulationsknoten						
Auslastung der 1. GPU	47%	56%	74%	89%	92%	94%
Auslastung der 2. GPU	46%	55%	72%	88%	91%	93%
Netzwerk I/O in MB/s	72 / 74	76/ 77	75 / 76	75 / 75	60 / 61	49 / 50
3. Simulationsknoten						
Auslastung der 1. GPU	46%	52%	69%	86%	89%	91%
Netzwerk I/O in MB/s	42 / 39	42 /39	40 /41	40 /41	29 /33	24 / 25
Visualisierungsknoten						
Auslastung der GPU	53%	39%	24%	13%	7%	5%
Netzwerk I/O in MB/s	19 /10	13 / 7	7 / 4	4.5 /2	2.6 / 1.3	1.6 / 0.8

Δx **in m**	0.30	0.25	0.22	0.20	0.19	0.18
Partikelzahl in Mio.	27	47	69	92	106	126
Laufzeit in s	331	477	688	963	1249	1349
1. Simulationsknoten						
Auslastung der 1. GPU	92%	94%	95%	94%	92%	97%
Auslastung der 2. GPU	95%	95%	97%	96%	94%	98%
Netzwerk I/O in MB/s	23 / 26	25 /25	23 / 24	19 / 17	18 /17	14 / 13
2. Simulationsknoten						
Auslastung der 1. GPU	94%	96%	96%	96%	94%	97%
Auslastung der 2. GPU	94%	94%	95%	93%	93%	97%
Netzwerk I/O in MB/s	49 / 47	44 / 44	40 / 40	36 / 36	33 / 32	28 / 28
3. Simulationsknoten						
Auslastung der 1. GPU	93%	94%	95%	93%	92%	97%
Netzwerk I/O in MB/s	24 /27	22 /22	20 / 20	16 / 18	15 /17	14 / 15
Visualisierungsknoten						
Auslastung der GPU	3%	2%	1.4%	1.0%	0.8%	0.7%
Netzwerk I/O in MB/s	0.9 / 0.5	0.5 / 0.3	0.4 / 0.2	0.3 / 0.2	0.3 / 0.2	0.2 / 0.1

Abbildung 6.10: Gemessene Laufzeiten, GPU-Auslastungen und Netzwerkbandbreiten bei der Skalierung mit der Partikelzahl

Partikelzahl in Mio.	0.52	0.98	2.1	5.8	11	17
Normierter Partikeldurchsatz	30%	39%	55%	80%	79%	78%
Mittlere GPU-Auslastung	46%	54%	71%	88%	91%	93%
Netzwerkausnutzung des zweiten Simulationsknotens	72%	75%	74%	74%	64%	49%

Partikelzahl in Mio.	27	47	69	92	106	126
Normierter Partikeldurchsatz	81%	98%	100%	95%	86%	93%
Mittlere GPU-Auslastung	94%	95%	96%	94%	93%	97%
Netzwerkausnutzung des zweiten Simulationsknotens	47%	43%	39%	35%	32%	27%

Abbildung 6.11: Berechneter normierter Partikeldurchsatz, mittlere GPU-Auslastung und Netzwerkausnutzung des zweiten Simulationsknotens bei der Skalierung mit der Partikelzahl

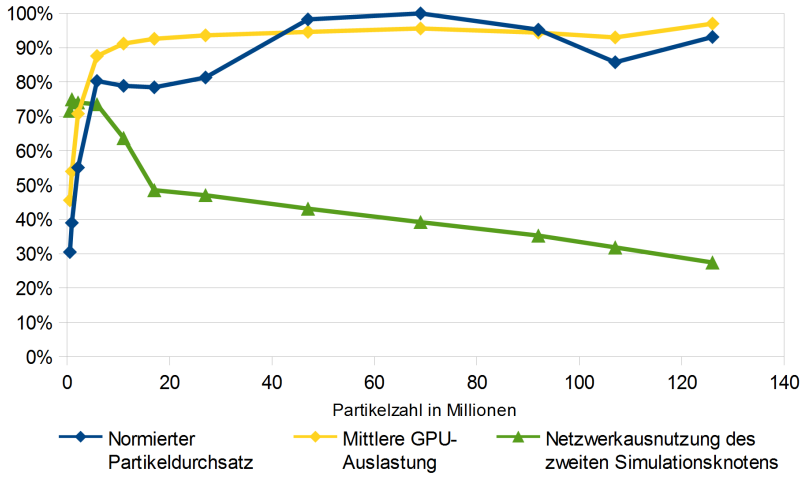


Abbildung 6.12: Diagramm für die Skalierungseffizienz, die mittlere GPU-Auslastung, und die Auslastung der Netzwerkbandbreite des zweiten Simulationsknotens bei der Skalierung mit der Partikelzahl

abnimmt. Insbesondere zeigt das Diagramm eine steile Abnahme bei weniger als 2.1 Millionen Partikeln.

Die leichte Abnahme bei mehr als 69 Millionen Partikeln ist wahrscheinlich auf den damit verbundenen größeren Working-Set der SPH-Simulation und die schlechteren L2-Cache-Hitrates

zurückzuführen. Die genauen Gründe und Auswirkungen der schlechter werdenden L2-Cache-Hitrates bei der Berechnung der Fixed-Radius-Near-Neighbors-Probleme wurden bereits in Punkt 4.9.6 erläutert. Die Vermutung der Abnahme der Performance wegen schlechter werdenden L2-Cache-Hitrates wird von der Tatsache untermauert, dass die mittlere gemessene GPU-Auslastung bei der Partikelzahl von 69 Millionen Partikeln 96% beträgt, und bei den höheren Partikelzahlen nur minimal zunimmt. Zudem sind die Messergebnisse für die Netzwerkbandbreiten aller Knoten bei den großen Partikelzahlen im Vergleich zu denjenigen Messergebnissen bei kleineren Partikelzahlen vergleichsweise niedrig. Deshalb haben die Bandbreiten sehr wahrscheinlich nur einen kleinen Einfluss auf die Performance. Dadurch wird die Vermutung, dass die niedrigeren L2-Cache-Hitrates den Performance-Verlust verursachen, gestützt. Allerdings beträgt der normierte Partikeldurchsatz bei der höchsten vermessenen Partikelzahl von 126 Millionen Partikeln immer noch 93%. Aus diesem Grund gehen bei der höchsten vermessenen Partikelzahl von 126 Millionen nur bis zu 7% Performance gegenüber des maximal gemessenen Partikeldurchsatzes bei 69 Millionen Partikeln wegen schlechterer L2-Cache-Hitrates verloren. Interessanterweise ergaben die Messungen der hier nicht verwendeten Modifikation des L2-Cache-Blockings für die Fixed-Radius-Near-Neighbors-Berechnungen in Punkt 4.9.6 einen Performancegewinn von in etwa 3% bei einer ähnlich großen Partikelzahl. So wurde die Messung des Blockings mit einer Partikelzahl von 20.5 Millionen Partikeln für eine einzige GPU durchgeführt, während die hiesigen Skalierungsuntersuchung einen maximalen Wert von 25 Millionen Partikel pro GPU vermessen haben. Somit kann der Performancegewinn bei den Fixed-Radius-Near-Neighbors-Berechnungen durch das L2-Cache-Blocking den Abfall nicht komplett kompensieren. Eine mögliche Ursache hierfür ist, dass ebenfalls die Kosten der Gitterkonstruktion, bei welcher das Blocking nicht vorhanden ist, wegen den schlechter werdenden L2-Cache-Hitrates zunehmen. Der Effekt der schlechter werdenden L2-Hitrates wird jedoch in den Messungen wahrscheinlich ein wenig durch die etwas höher werdenden GPU-Auslastungen kompensiert. Diese größer werdenden Auslastungen sind wahrscheinlich darauf zurückzuführen, dass die Rechenlast bei mehr Partikeln besser auf die einzelnen GPUs verteilt werden kann. Dadurch müssen die GPUs nicht mehr so lange auf die am längsten rechnende GPU warten. Des Weiteren ist der normierte Partikeldurchsatz bei 106 Millionen Partikeln mit 89% etwas schlechter, als mit 93% bei der maximalen Partikelzahl von 116 Millionen Partikeln. Der unerwartete geringere Wert wurde durch ein Überprüfen der Messung bestätigt. Deshalb sind die Ursachen für das Abweichen unklar und würden weitere Untersuchungen benötigen.

Die starke Abnahme des Partikeldurchsatzes bei weniger als 5.8 Millionen Partikeln ist sehr wahrscheinlich größtenteils auf die Limitierung durch die Netzwerkbandbreite zurückzuführen. Ein Teil dieser Abnahme kann auch durch den Tail-Effekt oder die länger werdenden Hauptachsen der Ellipsoide verursacht werden. Auch befinden sich anteilig mehr Partikel in den Randbereichen der Scheiben der GPUs, wodurch sie durch das etwas langsamere Rand-Kernel berechnet werden müssen.

Zunächst soll auf die Limitierung durch die Netzwerkbandbreite eingegangen werden. Die Limitierung durch die Netzwerkbandbreite für kleine Partikelzahlen rührt daher, dass die Partikelzahl durch die Größe des Startabstandes festgelegt wird. Ein größerer Startabstand erhöht jedoch auch das Verhältnis zwischen denjenigen Partikeln, welche sich im inneren Teil der Scheibe einer GPU und dem Halobereich befinden. Da die SPH-Simulation die Halobereiche bei der Berechnung der Fixed-Radius-Near-Neighbors-Probleme über das Netzwerk übertragen muss, wird dadurch auch das Verhältnis zwischen Rechenoperationen und Kopieroperationen über das Netzwerk geändert. Zusätzlich werden wegen der kleineren Laufzeit pro Zeitschritt die Dicke- und Tiefenbilder häufiger berechnet. Somit müssen sie auch häufiger über das

Netzwerk reduziert werden. Die Vermutung für die Limitierung durch die Netzwerkbandbreite wird dadurch gestützt, dass die mittlere GPU-Auslastung bei weniger als 2.1 Millionen Partikeln ebenfalls stark abnimmt. Bei der Betrachtung der Netzwerkbandbreite des zweiten SPH-Simulationsknotens zeigt sich, dass er am meisten Daten über das Netzwerk verschicken muss. Des Weiteren nimmt dessen Netzwerkbandbreite von 126 Millionen Partikeln bis 5.8 Millionen Partikel zu. Bei weniger Partikeln bleibt sie mit in etwa gesendeten 75 MB/s und empfangenen 75 MB/s konstant. Aus diesem Grund limitiert mit Wahrscheinlichkeit die Netzwerkbandbreite des inneren Knotens die Performance des gesamten Clusters. Die Ursache hierfür ist, dass er als einziger Knoten mit zwei anderen Knoten für die SPH-Simulation kommunizieren muss, während die beiden Randknoten jeweils nur mit dem zweiten Knoten kommunizieren müssen. Seine Netzwerkausnutzung beträgt allerdings nur 72%. Daraus folgt, dass 28% der Netzwerkbandbreite im Vergleich zur optimalen Ausnutzung verloren gehen. Eine mögliche Ursache hierfür ist, dass die Pipeline der SPH-Simulation, während die GPUs das Fixed-Radius-Near-Neighbors-Problem der Partikel im Randbereich ihrer Scheibe berechnen, die Zeitintegration ausführen bauen, keinen Netzwerktransfer durchführt. Auch kann bei der Gitterkonstruktion nicht durchgehend ein Netzwerktransfer stattfinden. Dadurch kann die Netzwerkbandbreite durch die SPH-Simulation niemals vollkommen ausgelastet werden. Ebenso kann die SPH-Simulation selbst keinen Netzwerktransfer während der Berechnung des Splattings ausführen. Allerdings kann die Visualisierung im Hintergrund während der SPH-Simulation ein Tiefenbild und ein Dickebild reduzieren, wodurch diese Effekte etwas und zeitweilig kompensiert werden können.

Ein weiterer wahrscheinlicher Grund für den Abfall der Performance bei niedrigen Partikelzahlen ist der Tail-Effekt, Der Tail-Effekt selbst lässt sich weniger direkt mit den Messungen untermauern, als theoretisch überlegen. So besitzt jede der 5 GPUs bei 0.98 Millionen Partikeln nur noch in etwa 200 000 Partikel; bei 0.52 Millionen sind es nur noch 100 000 Partikel. Bei maximaler Occupancy kann eine Geforce Titan 28 672 Partikel und eine Geforce Titan Black 30 720 Partikel gleichzeitig berechnen. Dadurch kann der Tail bei 0.98 Millionen Partikel im schlimmsten Fall 15% zusätzliche Laufzeit verursachen. Bei 0.52 Millionen Partikel sind es bereits 30 Prozent. Ein mögliches Indiz für den Tail-Effekt ist, dass die GPU-Auslastung, welche nur aussagt ob die GPU überhaupt etwas berechnen muss, bei kleinen Partikelzahlen weniger stark abnimmt als der Partikeldurchsatz. Interessanterweise tritt der Tail-Effekt im Falle einer Limitierung durch die Netzwerkbandbreite zwei mal bei jeder Berechnung eines Fixed-Radius-Near-Neighbors-Problems auf: Einmal bei der Berechnung des inneren Teils der Scheibe, weil wegen der Limitierung durch die Netzwerkbandbreite noch nicht sofort mit dem Randbereich begonnen werden kann. Anschließend tritt er noch einmal bei der Berechnung des Randbereichs auf. Da der Tail-Effekt des inneren Bereichs nur durch die Limitierung durch die Netzwerkbandbreite verursacht wird, ist nur der Tail-Effekt bei der Berechnung des Randbereichs für die Gesamtlaufzeit von Bedeutung.

Es gibt jedoch noch alternative Erklärungen dafür, dass die GPU-Auslastung weniger stark als der Partikeldurchsatz abnimmt. So werden die Hauptachsen der Ellipsoide mit zunehmenden Startabstand länger. Dadurch müssen beim Splatting für jedes Ellipsoid viel mehr Fragmente erstellt werden, wodurch sich die Kosten pro Partikel erhöhen und der Partikeldurchsatz wiederum abnimmt. Auch kann das Kernel für den Randbereich dafür verantwortlich sein. Denn bei größeren Startabständen befinden sich anteilig gesehen mehr Partikel in den Randbereichen der Gitter der GPUs. Das Kernel für den Randbereich besitzt jedoch zusätzlich eine teure Randbehandlungen, wodurch ebenfalls die Rechenzeit pro Partikel zunimmt.

Auf Grund des gemessenen normierten Partikeldurchsatzes lässt sich insgesamt sagen, dass die Simulation in dem Bereich von weniger als 5.8 Millionen Partikeln primär durch die Limitierung der Netzwerkbandbreite viel Performance verliert. So gehen beispielhaft in etwa 70% Performance bei 0.52 Millionen Partikeln und immer noch 45% Performance bei 2.1 Millionen Partikeln gegenüber des maximalen Partikeldurchsatzes verloren.

Nun soll auf die leichte Abnahme des Partikeldurchsatzes zwischen den Partikelzahlen von 5.8 Millionen bis 47 Millionen eingegangen werden. In diesem Bereich ist die gemessene Netzwerkbandbreite vergleichsweise niedrig, wodurch sie sich nur kaum negativ auf die Performance auswirken kann. Dies wird durch die mittlere GPU-Auslastung mit mindestens 88% gestützt, die zusätzlich in diesem Bereich in etwa konstant ist. Auch kann der Taileffekt bei einer Partikelzahl von 5.8 Millionen Partikel pro GPU nur noch maximal 3% der Performance bei der Berechnung eines Fixed-Radius-Near-Neighbors-Problems kosten. Deshalb bleiben als wahrscheinliche Ursache noch die länger werdenden Hauptachsen oder die zunehmenden Kosten durch die Berechnungen des Rand-Kernels übrig. Für sicherere Erkenntnisse müssten jedoch weitere Untersuchungen durchgeführt werden.

Als Nächstes soll kurz auf die GPU-Auslastung der SPH-Simulationsknoten eingegangen werden. Bei der Betrachtung der Auslastungen der unterschiedlichen GPUs bei hohen Partikelzahlen zeigt sich, dass die Auslastungen in etwa bei allen GPUs gleich groß sind. Da sich die Leistungen von den Titans und den Titan Blacks in etwa um 14% unterscheiden und die Partikel zu Beginn gleichmäßig auf die GPUs verteilt sind, lässt sich hieraus folgern, dass die Lastbalancierung dort soweit gut funktioniert. Bei den GPU-Auslastungen der Messungen mit niedrigen Partikelzahlen werden die Schwankungen größer. Die größeren Schwankungen sind wahrscheinlich darauf zurückzuführen, dass die Scheibendicke bei weniger Partikeln auch kleiner wird, wodurch die Granularität der Lastbalancierung abnimmt. Insbesondere ist die erste GPU des ersten Knotens immer etwas schlechter ausgelastet. Die Ursachen hierfür sind jedoch unklar. Zudem wurde befürchtet, dass beim Bauen des Gitters die GPUs des Cluster auf den Transfer derjenigen Partikel, die sich zwischen den Knoten bewegt haben, warten müssen. Wegen der hohen GPU-Auslastungen bei mehr als 5.8 Millionen kann das Warten nur wenig zur Gesamtlaufzeit beitragen. Schließlich fallen Auslastungen der GPUs erst bei weniger als 5.8 Millionen Partikeln deutlich ab. Dies ist eben der Bereich indem die Netzwerkbandbreite limitiert.

Abschließend sollen kurz die Messungen des Visualisierungsknotens diskutiert werden. So ist die Bandbreite des Visualisierungsknotens von 126 Millionen bis 17 Millionen Partikeln mit weniger als zwei MB/s gering und steigt bei weniger Partikeln auf bis zu 29 MB/s stark an. Dies ist auf die geringeren Kosten für einen SPH-Simulationsschritt zurückzuführen, wodurch die Visualisierung wiederum häufiger stattfindet. Daraus folgt zudem, dass die Reduktion nur bei niedrigen Partikelzahlen viel Netzwerkbandbreite benötigt. Interessanterweise sendet der Visualisierungsknoten auch vergleichsweise immer halb so viele Daten, wie er empfängt. Dies scheint etwas unproduktiv zu sein, da der primäre Knoten bei einem Reduce-Befehl am Schluss alle reduzierten Daten erhalten und dafür keinerlei Daten verschicken müsste. Die Ursache hierfür ist, dass MPICH wahrscheinlich einen fortgeschrittenen Reduktionsalgorithmus verwendet, der unter anderem die Full-Duplex-Funktionalität des Ethernets ausnutzt. Damit nun der erste Knoten bei der Reduktion nicht auch die Full-Duplex-Funktionalität ausnutzt und dadurch teure Netzwerkbandbreite verschwendet, muss er im Algorithmus besonders behandelt werden. Eine solche Sonderbehandlung wurde in MPICH vermutlich nicht implementiert. Die GPU-Auslastung des Visualisierungsknotens nimmt aus analogen Gründen wie die Netz-

werkbandbreite für kleine Partikelzahlen stark auf bis zu 53% zu. In diesem Fall werden die GPUs der SPH-Simulationsknoten wahrscheinlich durch die extra Visualisierungs-GPU deutlich besser ausgelastet. Denn so vermiedet die Visualisierung, dass eventuell eine GPU alle anderen GPUs aufhält. Allerdings benötigt der Visualisierungsknoten für die Reduktion zusätzliche Netzwerkbandbreite und das, obwohl bei kleinen Partikelzahlen bereits eine Limitierung durch die Netzwerkbandbreite vorliegt. Somit ist es insgesamt fraglich, ob der zusätzliche Visualisierungsknoten gerade bei kleinen Partikelzahlen performancesteigernd wirkt. Zusätzlich ist bei höheren Partikelzahlen die Auslastung des Visualisierungsknotens relativ niedrig. So beträgt sie bei 126 Millionen Partikeln nur noch 0.7%. In diesem Fall der Performance-Gewinn durch die Visualisierungs-GPU mit wahrscheinlich deutlich weniger als 0.7% ebenfalls gering. Deswegen ist es fraglich, ob die Idee für einen extra Visualisierungsknoten überhaupt oder deutlich laufzeitverkürzend ist.

So lässt sich als Fazit dieser Untersuchung der Skalierung mit der Partikelzahl sagen, dass sich bei hohen Partikelzahlen ein hoher Partikeldurchsatz und eine hohe GPU-Auslastung erzielen lassen. Zudem sind der Partikeldurchsatz und die GPU-Auslastungen bei kleinen Partikelzahlen wegen einer Limitierung primär durch die Netzwerkbandbreite niedrig. Zusätzlich nimmt der Partikeldurchsatz bei großen Simulationsgrößen wegen einer kleiner werdenden L2-Cache-Hitrate ab. Diese Beobachtungen erfüllen soweit auch die Erwartungen an eine Cluster-Simulation. Denn bei solchen Simulationen ist es üblich, dass bei kleineren Problemgrößen die Netzwerkbandbreiten die Performance limitieren, während bei größeren Problemgrößen der Durchsatz durch eine niedrigere Cache-Hitrate reduziert wird.

6.4.3 Skalierung mit der GPU-Zahl ohne Limitierung durch Netzwerkbandbreite

Als Nächstes soll die Skalierung der Performance mit der GPU-Zahl bestimmt werden. So gibt es generell zwei Probleme, welche die Skalierung mit der GPU-Zahl beeinflussen können: Die Limitierung durch die Netzwerkbandbreite oder das Warten des gesamten Clusters auf die am längsten rechnende GPU. In diesem Punkt soll zunächst das Warten auf die am längsten rechnende GPU untersucht werden. Dabei wird wiederum die SPH-Simulation mit Visualisierung herangezogen. Zusätzlich wurde ein Startabstand von 0.33 m gewählt. Dadurch ergeben sich insgesamt 20.5 Millionen Partikel. Bei dieser Partikelzahl spielt gemäß den Messungen aus dem letzten Punkt wahrscheinlich weder die Auslastung der Netzwerkbandbreite noch die Auslastung des Visualisierungsknotens einen merklichen Einfluss auf die Performance. Deshalb werden sie im Folgenden nicht weiter untersucht. Die für diesen Punkt durchgeführten Skalierungsmessungen reduzieren die Knotenzahl und die GPU-Zahl des Clusters bei konstantem Simulationsvolumen beziehungsweise bei konstanter Partikelzahl. Für jede GPU-Zahl werden die Auslastungen der GPUs und die Laufzeiten gemessen. Die Messergebnisse wurden in Abbildung 6.13 eingetragen.

Als Nächstes sollen die Messungen diskutiert werden. So ist es bei dem verwendeten Cluster jedoch problematisch, dass er für die SPH-Simulation unterschiedliche GPUs verwendet. Denn dadurch könnten die Ergebnisse der Skalierungsuntersuchung verfälscht werden. Deswegen werden die Untersuchungsergebnisse im Folgenden auf die FLOPS-Zahl der ersten GPU des Clusters, einer Titan Black normiert. Dies ist jedoch nur eine grobe Näherung. Eine Testmessung, bei welcher analog zu obigen Messbedingungen eine einzige Titan statt einer Titan Black gebenchmarkt wurde, untermauerte die Vermutung jedoch. So ergab die Messung eine Laufzeit von 1188 s für die Titan und eine Laufzeit von 1069 s für die Titan Black. Dadurch

GPU-Zahl	1	2	3	4	5
Laufzeit in s	1069	607	413	302	248
1. Simulationsknoten					
Auslastung der 1. GPU	100%	97%	95%	93%	91%
Auslastung der 2. GPU	-	97%	95%	96%	94%
2. Simulationsknoten					
Auslastung der 1. GPU	-	-	97%	95%	94%
Auslastung der 2. GPU	-	-	-	92%	93%
3. Simulationsknoten					
Auslastung der 1. GPU	-	-	-	-	92%

Abbildung 6.13: Gemessene Laufzeit und GPU-Auslastungen in Abhängigkeit der GPU-Zahl ohne Limitierung durch die Netzwerkbandbreite

GPU-Zahl	1	2	3	4	5
Normierte FLOPS	100%	188%	288%	388%	476%
Speedupfaktor	100%	176%	258%	360%	431%
Skalierungseffizienz	100%	94%	90%	91%	91%
Mittlere GPU-Auslastung	100%	97%	96%	94%	93%

Abbildung 6.14: Berechnete normierte FLOPS, Speedupfaktor, Skalierungseffizienz und mittlere GPU-Auslastung in Abhängigkeit der GPU-Zahl ohne Limitierung durch die Netzwerkbandbreite

ergibt sich ein Laufzeitenverhältnis von 1.11 zu 1 zwischen den beiden GPUs. Das stimmt gut mit dem FLOPS Verhältnis von 1 zu 1.14 überein. So lässt sich aus den Messergebnissen in Abbildung 6.13 ein Speedupfaktor bezüglich derjenigen Messung mit einer GPU berechnen. Dieser Speedupfaktor wurde mitsamt der normierten FLOPS-Zahl für die entsprechende GPU-Konfiguration in Abbildung 6.14 eingetragen. Auch wurde der Speedupfaktor in Abhängigkeit von der normierten FLOPS-Zahl in das Diagramm in Abbildung 6.15 eingezeichnet. Des Weiteren lässt sich aus dem Verhältnis zwischen dem Speedupfaktor und der normierten FLOPS-Zahl eine Skalierungseffizienz bestimmen. Die so berechnete Skalierungseffizienz wurde ebenfalls in die Tabelle in Abbildung 6.14 und in Abhängigkeit von der GPU-Zahl in das Diagramm in Abbildung 6.16 eingetragen.

Beim Betrachten von dem Diagramm mit dem Speedupfaktor in Abbildung 6.15 fällt auf, dass die Simulation im Messbereich in etwa optimal skaliert. So besitzen 5 GPUs eine normierte FLOPS-Zahl von 476% weshalb auch ein Speedup von 476% zu erwarten wäre. Tatsächlich beträgt er jedoch 431%. Aus dem Diagramm in Abbildung 6.16 mit der Skalierungseffizienz folgt zusätzlich, dass die Skalierungseffizienz bei 5 GPUs 91% beträgt, weshalb nur 9% Performance gegenüber optimaler Skalierung verloren gehen. Interessanterweise fällt die Skalierungseffizienz umso weniger ab, desto mehr GPUs beteiligt sind. Dies ist wahrscheinlich auf statistische Effekte zurückzuführen, da bei mehr GPUs die Schwankung der maximalen Rechenzeit unter allen GPUs abnimmt, wodurch die Wartezeit wiederum einen konstanteren Wert

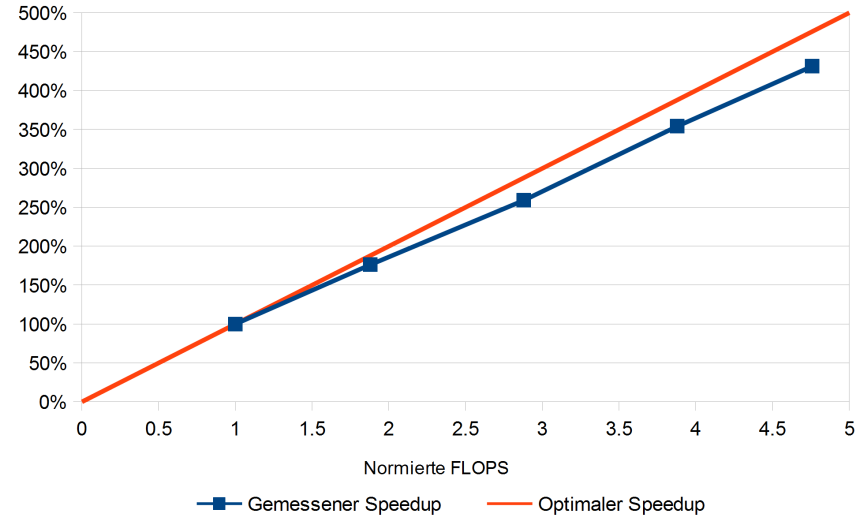


Abbildung 6.15: Diagramm des Speedups für die Skalierung mit der GPU-Zahl ohne Limitierung durch die Netzwerkbandbreite

einnimmt. Des Weiteren ist die Skalierungseffizienz bei einer Simulation mit mehr als einer GPU immer deutlich niedriger als die mittlere GPU-Auslastung. Teilweise lässt sich das wieder mit dem Tail-Effekt erklären. Denn bei 20.5 Millionen Partikeln insgesamt und 5 GPUs ergeben sich nur noch 4.3 Millionen Partikel pro GPU. Dadurch kann der Taileffekt wiederum bis zu 0.7% Performance kosten.

6.4.4 Skalierung mit der GPU-Zahl bei einer Limitierung durch Netzwerkbandbreite

Im letzten Punkt wurde behandelt, wie die Simulation im denjenigen Fall mit der GPU-Zahl skaliert, wenn die Netzwerkbandbreite die Performance nicht limitiert. In diesem Punkt soll deshalb gegensätzlich untersucht werden, wie die Simulation mit der GPU-Zahl skaliert, sofern die Netzwerkbandbreite limitiert. Da der Testcluster mit 5 GPUs verglichen mit Supercomputern, die mehrere hundert oder tausend GPUs besitzen, nur klein ist, scheint es weniger sinnvoll diese Skalierung durch das Hinzufügen oder Hinwegnehmen von GPUs beziehungsweise von Knoten zu testen. Denn so würden lediglich die drei Spezialfälle untersucht werden: Bei drei Knoten müsste für die SPH-Simulation ohne Visualisierung der zweite innere Knoten mit seinen zwei Nachbarn kommunizieren. Die anderen beiden Randknoten müssten jeweils nur noch mit einem nämlich dem inneren Nachbarn kommunizieren. Bei zwei Knoten für die SPH-Simulation müssten die beiden Randknoten nur noch untereinander kommunizieren. Bei einem Knoten würde gar keine Netzwerkkommunikation für die SPH-Simulation mehr stattfin-

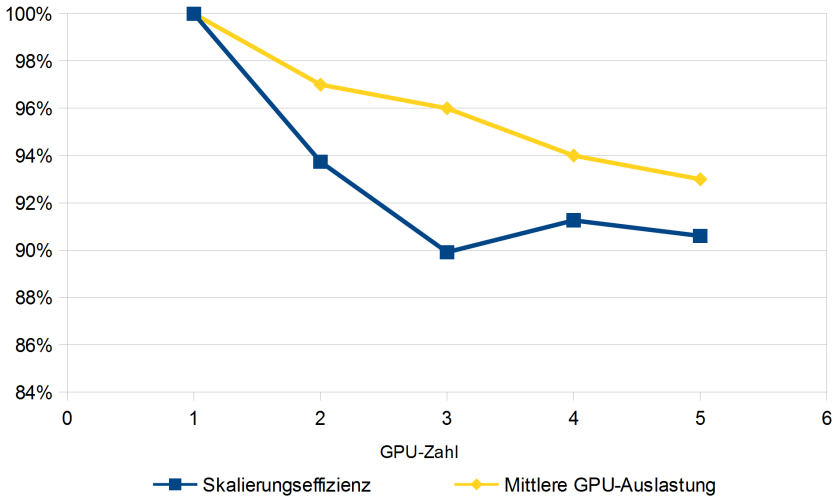


Abbildung 6.16: Diagramm der mittleren GPU-Auslastung und der Skalierungseffizienz für die Skalierung mit der GPU-Zahl ohne Limitierung durch die Netzwerkbandbreite

den. Somit würde bei der SPH-Simulation mit drei Knoten wesentlich eher eine Limitierung durch die Netzwerkbandbreite eintreten, als bei zwei Knoten. Bei einem Knoten würde bei einer SPH-Simulation ohne Visualisierung gar keine Limitierung durch die Netzwerkbandbreite mehr eintreten können.

Deswegen wird für die Untersuchung der Skalierung bei einer Limitierung durch die Netzwerkbandbreite ein anderer Ansatz gewählt. So werden bei der folgenden Messung zusätzliche nicht existente Knoten und GPUs, die in etwa einer Titan oder einer Titan Black entsprechen, simuliert, indem das tatsächlich simulierte Volumen der Simulation dementsprechend verkleinert wird. Denn bei der implementierten SPH-Simulation mit vielen Knoten und ohne Visualisierung sollten alle Knoten außer den beiden Randknoten immer in etwa die gleiche Netzwerkbandbreite benötigen. Diesbezüglich ist es für das Verhalten eines solchen Clusters unerheblich ob diese Knoten tatsächlich vorhanden sind, sofern der Cluster mindestens einen inneren Knoten besitzt, welcher mit zwei Nachbarn kommunizieren muss. Somit lässt sich das Skalierungsverhalten eines größeren Clusters durch einen wesentlich kleineren Cluster näherungsweise simulieren, indem das Volumen der Simulation dementsprechend verkleinert wird. Allerdings kann dadurch nicht die Skalierung der MPI-Reduktion der Tiefenbilder und Dickebilder simuliert werden. Deshalb finden folgende Messungen ohne Visualisierung statt. Ebenfalls können auf diese Weise nicht die Performanceverluste simuliert werden, welche dadurch entstehen, dass der gesamte Cluster immer auf die am längsten rechnende GPU warten muss. Analog kann nicht simuliert werden, dass der gesamte Cluster eventuell auf denjenigen Knoten, welcher am längsten für den Netzwerktransfer benötigt, warten muss.

So wird für die Messung das gesamte tatsächlich simulierte Volumen verkleinert, um zusätzliche nicht vorhandene Knoten und GPUs zu simulieren. Das verkleinerte Volumen wird wiederum am Anfang der Simulation gleichmäßig in Scheiben aufgeteilt, wobei jede vorhandene GPU eine solche Scheibe erhält. Auf diese Weise wird indirekt die anfängliche Scheibendicke variiert. Als Startabstand für die Partikel wurde ein Wert von 0.19 m gewählt. Dadurch ergibt sich insgesamt eine Problemgröße von 108 Millionen Partikeln. Zusätzlich ist das Gitter entlang der Z-Achse 1055 Gitterzellen lang. Auf diese Weise beträgt die anfängliche Scheibendicke von einer GPU 211 Gitterzellen, wenn die fünf tatsächlich vorhandenen GPUs die gesamte Simulation, also alle 108 Millionen Partikel, berechnen. Für jede simulierte GPU-Zahl wurden die Laufzeit und die GPU-Auslastungen gemessen. Zusätzlich wurde die Netzwerkbandbreite diesmal aus der Menge der übertragenen Daten berechnet. All das wurde ins Diagramm in Abbildung 6.17 eingetragen.

Aus den Messungen lässt sich nun ein Speedupfaktor bezüglich zur tatsächlich vorhandenen GPU-Zahl von fünf GPUs berechnen. Die Rechenergebnisse sind in Abbildung 6.18 zu sehen. Anschließend wurde er in das Diagramm in Abbildung 6.19 eingetragen. Zusätzlich lässt sich aus der Laufzeit eine Skalierungseffizienz bezüglich der fünf tatsächlich vorhandenen GPUs berechnen. Diese ist ebenfalls in Abbildung 6.18 zu sehen und wurde in das Diagramm in Abbildung 6.20 eingetragen. Ebenso wurde wieder aus der potentiell limitierenden Netzwerkbandbreite des zweiten Simulationsknotens die Ausnutzung der Netzwerkbandbreite im Falle des bidirektionalen Transfers ermittelt, da der zweite Knoten in etwa so viele Daten versendet wie empfängt. Auch wurde die mittlere GPU-Auslastung aller beteiligten GPUs berechnet. Die Netzwerkausnutzung des zweiten Simulationsknotens und die mittlere GPU-Auslastung wurden ebenfalls in Abbildung 6.18 und in das Diagramm der Skalierungseffizienz in Abbildung 6.20 eingetragen.

Das Diagramm für den Speedupfaktor in Abbildung 6.19 zeigt deutlich, dass die Simulation bei der verwendeten Partikelzahl und Testszene in etwa gut linear bis 15 GPUs skaliert. So müsste im Optimalfall bei 15 GPUs der Speedup-Faktor 300% betragen. Tatsächlich beträgt der berechnete Speedupfaktor 274%. Das Diagramm mit der Skalierungseffizienz in Abbildung 6.20 zeigt, dass die Skalierungseffizienz bei dieser GPU-Zahl von 15 GPUs immer noch 91% misst, wodurch nur 9% Performance gegenüber optimaler Skalierung verloren gehen. Bei mehr als 15 GPUs kommt die Skalierung fast vollkommen zu erliegen. So ändert sich die Laufzeit durch das simulierte Hinzufügen von GPUs kaum. Das Erliegen ist im diesem Fall der Limitierung durch die Netzwerkbandbreite des zweiten Knotens geschuldet. Diese steigt bis 15 GPUs auf einen Wert von circa 85 MB/s für die Lesebandbreite und für die Schreibbandbreite an. Anschließend bleibt die Netzwerkbandbreite beim simulierten Hinzufügen von weiteren GPUs nahezu konstant. Ein weiteres Indiz für die Limitierung durch die Netzwerkbandbreite ist im Diagramm in Abbildung 6.20 die mit der simulierten GPU-Zahl stark abfallende mittlere GPU-Auslastung. Interessanterweise ist die maximal erreichte Netzwerkbandbreite der hiesigen Messreihe mit 86 MB/s eingehend und 92 MB/s ausgehend höher als die maximal erreichte Netzwerkbandbreite, die bei der Skalierung durch die Partikelzahl gemessen wurde und nur 76 MB/s eingehend und 77 MB/s ausgehend betrug. Dieser höhere Wert ist nicht auf die unterschiedliche Testmethodik zurückzuführen, da der Systemmonitor bei der hiesigen Messreihe in etwa den gleichen höheren Wert anzeigt. Daraus folgt, dass die SPH-Simulation ohne Visualisierung die Netzwerkbandbreite besser ausnutzen kann als mit Visualisierung. Insgesamt kann das Verfahren jedoch hier auch nur bis zu 87% der Netzwerkbandbreite ausnutzen. Die Gründe dafür, dass dieser Wert weit von den 100% entfernt ist, sind wahrscheinlich analog wie bei der Diskussion der Skalierung mit der Partikelzahl.

Simulierte GPU-Zahl	5	7	10	13	15	17
Scheibendicke in Gitterzellen	211	151	106	81	70	62
Laufzeit in s	1213	940	640	497	443	436
1. Simulationsknoten						
Auslastung der 1. GPU	90%	91%	85%	88%	83%	74%
Auslastung der 2. GPU	95%	93%	90%	93%	89%	78%
Netzwerk I/O in MB/s	13/16	18/23	25/30	33/41	37/46	37/47
2. Simulationsknoten						
Auslastung der 1. GPU	94%	93%	94%	92%	88%	79%
Auslastung der 2. GPU	93 %	94%	94%	93%	88%	80%
Netzwerk I/O in MB/s	31/29	43/42	57/57	73/73	84/82	85/85
3. Simulationsknoten						
Auslastung der 1. GPU	93%	94%	94%	90%	87%	79%
Netzwerk I/O in MB/s	16/15	23/20	33/27	40/33	45/38	48/38

Simulierte GPU-Zahl	20	25	30	35	40	45
Scheibendicke in Gitterzellen	53	42	35	30	26	23
Laufzeit in s	427	414	430	418	411	421
1. Simulationsknoten						
Auslastung der 1. GPU	67%	52%	43%	35%	30%	25%
Auslastung der 2. GPU	68%	57%	47%	40%	33%	30%
Netzwerk I/O in MB/s	37/45	40/44	39/47	40/46	43/47	41/46
2. Simulationsknoten						
Auslastung der 1. GPU	69%	56%	48%	40%	34%	29%
Auslastung der 2. GPU	69%	55%	46%	37%	33%	29%
Netzwerk I/O in MB/s	81/86	81/90	86/90	84/86	86/90	86/92
3. Simulationsknoten						
Auslastung der 1. GPU	67%	54%	43%	35%	32%	28%
Netzwerk I/O in MB/s	49/36	51/38	51/39	46/38	47/38	50/38

Abbildung 6.17: Gemessene GPU-Auslastungen und Netzwerkbandbreiten für die Skalierung mit der GPU-Zahl bei einer Limitierung durch die Netzwerkbandbreite

Simulierte GPU-Zahl	5	7	10	13	15	17
Speedup-Faktor	100%	129%	190%	244%	274%	278%
Skalierungseffizienz	100%	92%	95%	94%	91%	81%
Mittlere GPU-Auslastung	93 %	93%	91%	91%	87%	78%
Netzwerkausnutzung des zweiten Simulationsknotens	29%	42%	56%	72%	81%	83%

Simulierte GPU-Zahl	20	25	30	35	40	45
Speedup-Faktor	284%	293%	282%	290%	295%	288%
Skalierungseffizienz	71%	58%	47%	41%	37%	32%
Mittlere GPU-Auslastung	68%	55%	45%	37%	32%	28 %
Netzwerkausnutzung des zweiten Simulationsknotens	82%	84%	86%	83%	86%	87%

Abbildung 6.18: Berechneter Speedupfaktor, Skalierungseffizienz, mittlere GPU-Auslastung und Ausnutzung der Netzwerkbandbreite des zweiten Simulationsknotens für die Skalierung mit der GPU-Zahl bei einer Limitierung durch die Netzwerkbandbreite

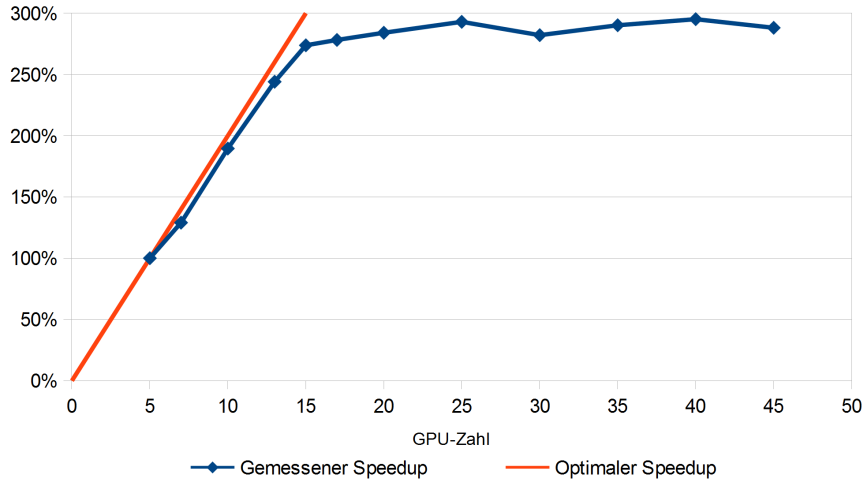


Abbildung 6.19: Diagramm des Speedupfaktors für die Skalierung mit der GPU-Zahl bei einer Limitierung durch die Netzwerkbandbreite

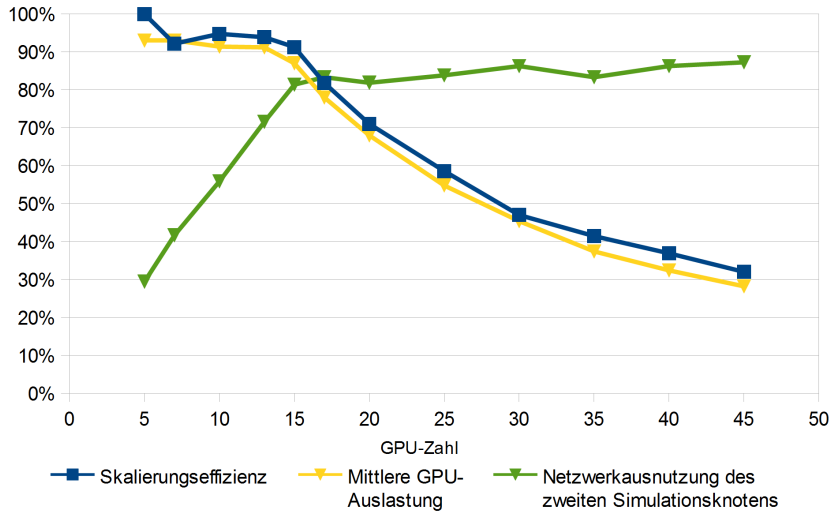


Abbildung 6.20: Diagramm der Skalierungseffizienz, mittleren GPU-Auslastung und Ausnutzung der Netzwerkbandbreite des zweiten Simulationsknotens für die Skalierung mit der GPU-Zahl bei einer Limitierung durch Netzwerkbandbreite

Des Weiteren ist das Halo bei der verwendeten Gitterzellengröße von 1/3 der Cutoff-Distance 3 Gitterzellen dick. Dadurch ergibt sich bei derjenigen GPU-Zahl von 15 GPUs, bis zu welcher eine gute Skalierung vorliegt, ein Volumenverhältnis zwischen dem Halobereich und dem inneren Bereich der Scheibe einer GPU von 0.09 zu 1. Dieser Wert lässt sich prinzipiell gut dafür verwenden, um vorherzusagen, bis zu welcher GPU-Zahl die SPH-Simulation in einem Cluster auch bei anderen Testszenen skalieren wird. Dabei müssen allerdings die Voraussetzung der ähnlichen GPUs, der homogenen Flüssigkeitsverteilung und der ähnlichen Netzwerkhardware gegeben sein. Bei besserer Netzwerkhardware verkleinert sich der Wert dementsprechend, während er sich bei besseren GPUs oder einer inhomogenen Aufteilung innerhalb der Szene dementsprechend vergrößert.

Als Fazit dieser Untersuchung lässt sich sagen, dass die Simulation bei der gewählten Testszene bis zu 15 GPUs gut skaliert. Bei mehr GPUs tritt eine Limitierung durch die Netzwerkbandbreite ein.

6.5 Fazit

Schließlich soll das Fazit aus diesem Kapitel gezogen werden. So wurde die SPH-Simulation aus dem Kapitel 4 mit ihrer Visualisierung aus dem Kapitel 5 so erweitert, dass sie durch einen GPU-Cluster berechnet werden kann. Dafür wurden sowohl die Berechnungen der SPH-Simulation selbst als auch die Berechnungen der Visualisierung auf die Knoten des Clusters

verteilt. Für die Verteilung der SPH-Simulation wurde das Simulationsvolumen entlang der Z-Achse eindimensional in Scheiben aufgeteilt, welche wiederum den einzelnen GPUs zugewiesen worden sind. Da sowohl für die Konstruktion des Gitters als auch für die Berechnungen der Fixed-Radius-Near-Neighbors-Probleme eine Kommunikation innerhalb des Clusters notwendig ist, wurde beides so erweitert, dass es gleichzeitig einen Netzwerktransfer und ein Berechnen auf den GPUs erlaubt.

Die Berechnungen der Visualisierung wurden per Compositing-Technik im Cluster verteilt. Dafür berechnet als erstes jede GPU ein Tiefenbild und ein Dickebild durch das Ellipsoid-Splatting, welche dann zu einem einzigen Tiefenbild beziehungsweise Dickebild zusammengesetzt werden. Darauf wendet die Visualisierung dann den Screen-Space-Curvature-Flow und das finale Shading an. Damit kein Warten auf den Netzwerktransfer während der Visualisierung stattfinden muss, wurde die Visualisierung ebenfalls effizient in einer asynchronen Pipeline implementiert.

Bei den Untersuchungen wurde gezeigt, dass die Simulation sowohl sehr gut mit der Partikelzahl als auch mit der GPU-Zahl beziehungsweise Knotenzahl skaliert. Allerdings treten bei zu großen Partikelzahlen und konstanter GPU-Zahl Performanceeinbrüche wegen schlechter werdenden L2-Cache-Hitrates auf. Bei zu kleinen Partikelzahlen beziehungsweise zu vielen GPUs treten ebenfalls Performanceeinbrüche auf, da dann die Netzwerkbandbreite limitiert. Zusätzlich wird die Performance leicht dadurch reduziert, dass die GPUs untereinander auf ihre Berechnungen warten müssen.

6.6 Ausblick

In diesem Kapitel wurde die SPH-Simulation mit Visualisierung auf einen GPU-Cluster erweitert und auf die Skalierbarkeit untersucht. Zusätzlich wurden bereits einige kleinere weiterführende Verbesserungsansätze für die Cluster-Version vorgestellt. Allerdings konnten wegen des beschränkten Umfangs und beschränkter finanzieller Mittel dieser Arbeit nicht alle Untersuchungen durchgeführt werden. Ebenso konnten nicht alle sinnvoll erscheinenden Optimierungen implementiert und weiter untersucht werden. Dabei scheinen neben den bereits genannten Verbesserungen folgende Optimierungen und Untersuchungen am sinnvollsten:

- **Weitere Skalierungsuntersuchungen:** In dieser Arbeit wurden nur drei Versuchsreihen durchgeführt: Eine Messreihe für die Skalierung mit der Partikelzahl bei konstanter Knotenzahl, eine Messung der Skalierung mit der GPU-Zahl ohne Limitierung durch die Netzwerkbandbreite und eine synthetische Messreihe für die Skalierung mit der GPU-Zahl mit Limitierung durch die Netzwerkbandbreite. Dabei wären jedoch zusätzliche Messreihen von Interesse. Beispielhaft könnte eine Untersuchung weitere Messreihen bei Testszenen durchführen, welche für die eindimensionale Unterteilung weniger vorteilhaft sind. Alternativ könnte sie direkt die Lastbalancierung untersuchen. Des Weiteren wäre es interessant, für die Untersuchungen einen größeren Cluster verwenden zu können.
- **Verwendung von CPUs für die SPH-Simulation:** Bislang wurden nur die GPUs des Clusters für die SPH-Simulation verwendet. Jedoch besitzen GPU-Cluster in der Regel auch leistungsfähige CPUs. Diese werden momentan fast ausschließlich für das Ansteuern der GPUs und für die MPI-Kommunikation verwendet. Dadurch ist ihre Auslastung während der Simulation gering. Deshalb wäre es interessant diese auch für die Simulation zu verwenden. In einem Performancevergleich in [ACb] zeigte es sich jedoch, dass bei

einer SPH-Simulation eine Graphikkarte vom Typ Geforce 480 GTX um den Faktor 13 schneller als eine 4-Kern Intel Core-i7 940 CPU ist. Erschwerend kommt hinzu, dass bei einer Verwendung von CPUs durch die feinere Aufteilung die GPUs potentiell schlechter ausgelastet werden könnten. Zusätzlich verursacht die höhere CPU-Auslastung, dass die Ansteuerung der GPUs mit höheren Latenzen verbunden ist. Deshalb wäre der maximale Nutzen dieser Optimierung wahrscheinlich sehr beschränkt.

- **Berechnung des Screen-Space-Curvature-Flows auf der CPU:** Momentan wird eine eigene GPU für den Screen-Space-Curvature-Flow verwendet. Diese ist jedoch mit meist weniger als 1% schlecht ausgelastet. Deshalb scheint es weniger sinnvoll einen Knoten extra für diese Berechnung bereit zu stellen. Dennoch sollte die SPH-Simulation ein eventuelles Warten der übrigen GPUs vermieden, während der Screen-Space-Curvature-Flow berechnet wird. Ansonsten könnte sie so vielleicht wenige Prozent an Performance des gesamten Clusters verlieren. Deshalb scheint es sinnvoll einen untätigen CPU-Kern für diese Berechnung zu verwenden.

- **Optimierung des Screen-Space-Curvature-Flows für eine Multi-GPU-Version mit nur einem Knoten:** Des Weiteren wäre eine Optimierung des Screen-Space-Curvature-Flows für eine Multi-GPU-Version mit nur einen Knoten vorteilhaft. Denn eine Version, welche nur auf einen einzigen Knoten läuft, würde sich prinzipiell bereits bei kleinen Partikelzahlen lohnen. In diesem Fall sind die Kosten für den Screen-Space-Curvature-Flow im Vergleich zur übrigen Rechenlast jedoch relativ hoch. So ist es in der aktuellen Implementierung möglich zwei MPI-Prozesse auf einen einzigen Knoten zu starten, wobei einer von beiden dann nur die Berechnung des Screen-Space-Curvature-Flows und des finalen Shadings ausführt. Dies führt nun wiederum dazu, dass diejenige GPU, welche den Screen-Space-Curvature-Flow und die SPH-Simulation berechnet, besser und diejenigen GPUs, welche nur die SPH-Simulation berechnen, schlechter ausgelastet sind. Aus diesem Grund wären bei einer Multi-GPU-Version mit nur einen Knoten weitere Optimierungen bezüglich des Screen-Space-Curvature-Flows nötig.

- **Wahl einer anderen Aufteilung des simulierten Volumens auf die GPUs des Clusters:** Da höhere Dimensionalitäten für die Aufteilung des simulierten Volumens auf die GPUs des Clusters prinzipiell besser skalieren, wäre eine Implementierung und Untersuchung von diesen ebenfalls lohnenswert. Um eine solche Aufteilung sinnvoll testen zu können wird allerdings zusätzlich ein größerer GPU-Cluster benötigt.

- **Durchführen des Ellipsoid-Splattings während des Netzwerktransfers:** Aktuell wird die Netzwerkbandbreite nicht für die SPH-Simulation verwendet, während das Splatting stattfindet. Dies könnte die Visualisierung vermeiden indem sie das komplette Splatting oder Teile davon berechnet, während die SPH-Simulation auf das Netzwerk wartet. Dadurch ließe sich wiederum die Netzwerkbandbreite und die GPUs besser auslasten.

- **Einbauen von „Randgittern“ am Rand der Knoten:** Bei der bisherigen Implementierung kann die GPU bei der Konstruktion des Gitters nur wenige Berechnungen durchführen, während die Partikel, welche sich in das Gitter der GPU hineinbewegt haben, über das Netzwerk übertragen werden. Dadurch muss sie dort potentiell auf den Netzwerktransfer warten. Dies ließe sich fast komplett vermeiden, indem jede GPU, welche sich am Rand des Knotens befindet, zusätzlich zu ihrem regulären Gitter ein separates Randgitter besitzt. Dadurch würden sich die Partikel, die sich von einem anderen Knoten aus in die Scheibe der GPU hineinbewegen, zunächst im Randgitter sein. Auf diese

Weise kann das reguläre Gitter bereits komplett gebaut werden und dort schon mit der Berechnung des ersten Fixed-Radius-Near-Neighbors-Problems begonnen werden, ohne dass der Transfer der Partikel in das Randgitter über das Netzwerk komplett sein muss. Dadurch ließe sich wiederum die GPU etwas besser auslasten.

- **Reduktion der Netzwerkbandbreite durch Kompression:** Da die Netzwerkbandbreite eine sehr knappe Ressource ist, wäre es sinnvoll die übertragenen Daten zu komprimieren. Hierfür könnte die SPH-Simulation zunächst eine sehr einfache Kompression verwenden, welche einen Float4 zu einem Float3 komprimiert, sofern dessen vierte Komponente nicht benötigt werden sollte. Im Speziellen könnte hier die Netzwerkbandbreite der vierten Float4-Komponente der Geschwindigkeit beim Synchronisieren des Halos und beim Transfer der Partikel eingespart werden. Alternativ könnte die SPH-Simulation die brachliegende Rechenleistung der CPU für komplexere Kompressionsalgorithmen verwenden.

- **Rekonstruktion des Gitters aus den Positionen der Partikel im Halo:** Momentan wird das Gitter im Halo ebenfalls über das Netzwerk synchronisiert. Da das Gitter aber aus den Positionen der ebenfalls synchronisierten Partikel im Halo folgt, könnte die Simulation das Gitter direkt aus diesen Positionen rekonstruieren. Deshalb ist eine Synchronisation über das Netzwerk nicht nötig.

- **Implementierung von Speichermanagement in Kombination mit der Lastbalancierung:** Momentan werden die Arrays für Partikeldaten zu Beginn mit einer festen Größe alloziert. Ebenso wird das Array für das Gitter mit fester Größe alloziert. In einem Cluster können aber prinzipiell unterschiedliche GPUs mit unterschiedlich viel DRAM vorhanden sein. Aus dem Grund wäre es sinnvoll die Allozierung gemäß der DRAM-Größe der GPUs zu wählen. Zusätzlich kann es durch die Lastbalancierung vorkommen, dass die GPU ihr Gitter weiter vergrößern müsste, so dass der allozierte Speicherplatz für das Gitter nicht mehr ausreichen würde. In diesem Fall könnte die Größe der Gitterzellen erhöht werden, so dass das Gitter weniger Speicherplatz benötigt. Alternativ könnte mehr Speicherplatz für das Gitter und weniger Speicherplatz für Partikel alloziert werden. Ebenfalls wären in diesem Fall Optimierungen sinnvoll, damit Gitterbereiche ohne Partikel nicht im DRAM der GPU abgespeichert werden müssen. Analog kann es passieren, dass die Lastbalancierung das Gitter weiter vergrößert, wonach mehr Partikel in der Scheibe der GPU sind als in dem DRAM der GPU Platz finden. Das selbe Problem kann durch die Eigenbewegung der Partikel auftreten. In diesem Fall müsste das Gitter der GPU wieder verkleinert werden. Alternativ könnte der DRAM der CPU zum Auslagern verwendet werden.

7 Fazit

In diesem allerletzten Kapitel soll das Fazit der gesamten Arbeit gezogen werden. Für das letzte Fazit werden noch einmal die Fazite der einzelnen Kapitel genannt. Somit werden hier das Fazit des Kapitels der GPU-Grundlagen, der SPH-Grundlagen, der Single-GPU-SPH-Simulation, der SPH-Visualisierung und der SPH-Simulation auf einem GPU-Cluster wiederholt.

Das Kapitel 2 führte den Leser in die CUDA- und OpenGL-Grundlagen ein. Dafür wurde zunächst der Hardware-Aufbau einer GPU anhand eines GK-110 erklärt. Auch wurde die Kernelausführung auf der GPU erläutert. Hierfür wurden insbesondere die SIMT- beziehungsweise SIMD-Eigenschaften der GPU und die damit verbundene Warp-Ausführungseffizienz hervorgehoben. Ebenso wurde erläutert, wie die GPU Latenzen überbrückt. Auch stellte das Kapitel die stark mit der Latenzüberbrückung verbundene Occupancy vor. Zudem wurde auf die einzelnen Speicherbereiche einer GPU in CUDA eingegangen. Zusätzlich wurde kurz die Programmierung von CUDA mit CUDA-C gezeigt. Schließlich demonstrierte das Kapitel die grundlegende OpenGL-Funktionalität an einer einfachen OpenGL-Pipeline.

Als Nächstes wurden im Kapitel 3 die physikalischen Grundlagen einer SPH-Simulation diskutiert. So diskretisiert ein SPH-Verfahren die Flüssigkeit zunächst durch einzelne Partikel. Dabei übernehmen die einzelnen Partikel die Strömungsgrößen der Flüssigkeit, also die Dichte und die Geschwindigkeit. Die Partikel dienen zudem zur kurzreichweitigen Interpolation der Felder derjenigen Differentialgleichungen, welche die Flüssigkeit beschreiben. Durch die Interpolation wird die Ortsdiskretisierung ausgeführt, womit die SPH-Simulation die zeitlichen Änderungen der Strömungsgrößen der Partikel bestimmt. Da die Interpolation kurzreichweitig ist, handelt es sich bei ihr algorithmisch gesehen um ein Fixed-Radius-Near-Neighbors-Problem. Dabei gilt zu beachten, dass die Lösung der Fixed-Radius-Near-Neighbors-Probleme bei einer SPH-Simulation immer am meisten Rechenzeit kostet. Anschließend kann die Zeitdiskretisierung erfolgen, wofür die Arbeit ein Prädiktor-Korrektor-Verfahren zweiter Ordnung verwendet.

Im Kapitel 4 beschrieb die Arbeit, wie die physikalischen Grundlagen aus dem vorherigen Kapitel in eine SPH-Simulation für eine einzige GPU umgesetzt wurden. Für die Lösung der Fixed-Radius-Near-Neighbors-Probleme verwendete die Arbeit als Space-Partitioning-Datenstruktur ein Dynamic-Vector-Gitter, in welchem die Partikel einsortiert wurden. Für die Konstruktion des Gitters wurde ein atomarer Counting-Sort-Ansatz verwendet. Für die Berechnungen selbst nutzte die Arbeit den Linked-Cell-Ansatz. Bei ihm wird über die Partikel selbst parallelisiert. Für jedes Partikel werden die benachbarten Gitterzellen nach Nachbarpartikeln innerhalb der Cutoff-Distance untersucht, um mit ihnen die Interpolation ausführen zu können. Da die Fixed-Radius-Near-Neighbors-Berechnungen einen Großteil der Laufzeit des Programms ausmachen, wurde versucht diese in Kombination mit dem verwendeten Linked-Cell-Ansatz weiter zu optimieren. Dabei wurde besonderen Wert darauf gelegt, die Auswirkungen der limitierenden Latenzen zu reduzieren und die niedrige Warp-Ausführungseffizienz zu optimieren. Die Untersuchungen fanden beispielhaft anhand zwei Fixed-Radius-Near-Neighbors-Problemen aus der SPH-Simulation, nämlich dem Änderungen-Kernel und dem Shepard-Kernel, statt. Durch die Optimierungen ließen sich zwar die Laufzeiten des Änderungen-Kernels und des Shepard-Kernels deutlich reduzieren. Dennoch verursachen die Latenzen und die Warp-Ausführungseffizienz weiterhin große Performanceverluste.

Als Nächstes wurde die Simulation im Kapitel 5 um eine Visualisierung erweitert. Bei der Visualisierung verwendete die Arbeit eine Ellipsoid-Splatting-Technik. Dabei zeichnet das Splatting die Partikel der SPH-Simulation als Ellipsoide. Für dieses Splatting muss zunächst für

jedes Partikel die Anisotropie der Partikelverteilung an dessen Stelle per Fixed-Radius-Near-Neighbors-Berechnung bestimmt werden. Aus dieser Anisotropie lassen sich die Hauptachsen der Ellipsoide bestimmen. Anschließend werden die Ellipsoide mit OpenGL gezeichnet, wodurch die GPU ein Tiefenbild und ein Dickebild erstellt. Nun wird das Tiefenbild per Screen-Space-Curvature-Flow geglättet, damit der Betrachter die Wölbungen der Ellipsoide im finalen Renderergebnis nicht mehr erkennen kann. Zuletzt wird das geglättete Tiefenbild und Dickebild durch das finale Shading zum fertigen Renderergebnis zusammengesetzt. Dabei dient das Dickebild dafür die Durchsichtigkeit der Flüssigkeit zu bestimmen und das Tiefenbild dafür die Reflexion der Flüssigkeitsoberfläche zu ermitteln.

Zu Letzt wurde die Simulation im Kapitel 6 mitsamt Visualisierung auf einen GPU-Cluster erweitert. Anschließend wurde die so entstandene GPU-Cluster-Version untersucht. Für die Erweiterungen wurde das Volumen der Simulation und damit die in dem Volumen enthaltenen Partikel eindimensional in Scheiben auf die einzelnen GPUs aufgeteilt. Sowohl für die Bewegung der Partikel als auch für die Berechnungen der Fixed-Radius-Near-Neighbors-Probleme ist jedoch eine Kommunikation innerhalb des Clusters notwendig. Deshalb wurde die Simulation so modifiziert, dass sie gleichzeitig einen Netzwerktransfer und ein Berechnen auf den GPUs erlaubt. Zusätzlich verwendete die Arbeit eine Lastbalancierung, die die Scheibendicke anhand der Kernel-Laufzeiten variiert, so dass die GPUs des Clusters gleichmäßig ausgelastet werden. Ebenso wurden die Berechnungen der Visualisierung auf die Knoten des Cluster per Compositing-Technik aufgeteilt. Dafür berechnet jede GPU zunächst mit ihren eigenen Partikeln ein Tiefenbild und ein Dickebild, die dann über das Netzwerk per Min-Operation beziehungsweise per Additionsoperation zu einem einzigen Tiefenbild und Dickebild reduziert werden. Die Reduktion der Tiefenbilder und Dickebilder findet asynchron in einer Pipeline statt, damit die GPUs währenddessen weiter die SPH-Simulation berechnen können. Anschließend finden der Screen-Space-Curvature-Flow und das finale Shading auf einem separaten Knoten des Clusters statt, so dass die GPUs der anderen Knoten nicht ungleichmäßig ausgelastet werden. Bei den Untersuchungen zeigte sich, dass die Simulation sehr gut mit der Partikelzahl als auch mit der GPU-Zahl beziehungsweise Knotenzahl skaliert. Wird die Partikelzahl der Simulation bei konstanter GPU-Zahl zu groß, so treten Performanceeinbrüche wegen schlechter werdenden Cache-Hitrates auf. Werden zu wenige Partikel bei zu vielen GPUs simuliert, so geht ebenfalls Performance verloren, da in diesem Fall die Netzwerkbandbreite limitiert. Zusätzlich wird die Laufzeit etwas dadurch erhöht, dass die GPUs des Clusters trotz Lastbalancierung ein wenig auf ihre Berechnungen untereinander warten müssen. Abgesehen von diesen Limitierungen skaliert die SPH-Simulation jedoch sehr gut.

8 Anhang

8.1 Praktische Arbeit

Die auf einer DVD beiliegende praktische Arbeit wurde mit CUDA 6.5 und Nsight Eclipse programmiert. Zusätzlich wird Linux, Glew und ein threadsicheres MPI benötigt. Außerdem funktioniert das Programm nur mit NVIDA-GPUs der Compute-Capability 3.5 oder höher. Das Programm wurde nur mit GPUs der Compute-Capability 3.5 und 5.0 getestet. Da CUDA prinzipiell aufwärtskompatibel ist sollte das Programm auch auf moderneren Compute-Capabilities lauffähig sein.

8.2 Quellenangaben

[ACa] *"GPUs, a New Tool of Acceleration in CFD: Efficiency and Reliability on Smoothed Particle Hydrodynamics Methods "*
Alejandro C. Crespo, Jose M. Dominguez, Anxo Barreiro, Moncho Gomez-Gesteira, Benedict D. Rogers
Artikel - Plos One - 2011

[ACb] *"DualSPHysics, new GPU computing on SPH models"*
Alejandro C. Crespo, Jose M. Dominguez, Anxo Barreiro, Moncho Gomez-Gesteira, Benedict D. Rogers
Präsentation - Schriftenreihe Schiffbau 6th SPHERIC- 2011

[CB] *"Eigenvectors of 3x3 symmetric matrix "*
Connelly Barnes
Blog - barnesc.blogspot.com/2007/02/eigenvectors-of-3x3-symmetric-matrix.html - 2007

[CH] *"Entwicklung eines Smoothed Particle Hydrodynamics (SPH) Codes zur numerischen Vorhersage des Primärzerfalls an Brennstoffeinspritzdüsen"*
Corina Hoefler
Dissertation - Karlsruher Institut für Technologie - 2013

[DSPHa] *"Dual SPHysics"*
The University of Manchester and Universida de Vigo
Demonstrationsvideo - http://youtu.be/rHq6rcdqHzM?list=UU7I8ftAldTXKAWD6_VxE5Iw - 2014

[DSPHb] *"Dual SPHysics"*
The University of Manchester and Universida de Vigo
Quelltext - 2014

[DV] *"Fluid Mechanics and the SPH Method"*
Damien Violeau
Buch - ISBN: 978-0-19-965552-6 - 2012

[DW] *"OpenGL 4.0 Shading Language Cookbook"*
David Wolff
Buch - ISBN: 978-1849514767 - Juli 2011

[GGa] *"SPHysics - developtment of a free-surface fluid solver - Part 1: Theory and formulations "*
M. Gomez-Gesteira , B.D. Rogers, A.J.C. Crespo, R.A. Dalrymple, M. Narayanaswamy,J.M. Dominguez
Artikel - Computer and Geosciences 48 - 2012

[GGb] *"SPHysics – development of a free-surface fluid solver – Part 2: Efficiency and test cases"*
M. Gomez-Gesteira, A.J.C. Crespo, B.D. Rogers, R.A. Dalrymple, J.M. Dominguez, A. Barreiro
Artikel - Computer and Geosciences 48 - 2012

[GM] *"Smoothed particle hydrodynamics: theory and application to non-spherical stars"*
R. A. Gingold, J. J. Monaghan
Artikel - Monthly Notices of the Royal Astronomical Society - Volume 181 Issue 3 - 1977

[JA] *"General purpose molecular dynamics simulations fully implemented on graphics processing units"*
Joshua A. Anderson , Chris D. Lorenz, A. Travesset
Artikel - Journal of Computational Physics 227 - 2008

[ER] *"Advances in Multi-GPU Smoothed Particle Hydrodynamics Simulations"*
Eugenio Rustico, Giuseppe Bilotta, Alexis Herault, Ciro Del Negro, Giovanni Gallo
Artikel - TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS - 2014

[JDa] *"New OpenMP-MPI-CUDA implementation for parallel SPH simulations on heterogeneous CPU-GPU clusters "*
J.M. Domínguez, A.J.C. Crespo, D. Valdez-Balderas, B.D. Rogers, M. Gómez-Gesteira
Präsentation - 7th International SPHERIC Workshop - 2012

[JDb] *"Neighbour lists in smoothed particle hydrodynamics"*
J. M. Domínguez, A. J. C. Crespo, M. Gómez-Gesteira, J. C. Marongiu
Artikel - INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN FLUIDS - 2010

[JDc] *"New multi-GPU implementation for smoothed particle hydrodynamics on heterogeneous clusters"*
J.M. Domínguez, A.J.C. Crespo, D. Valdez-Balderas, B.D. Rogers, M. Gómez-Gesteira
Artikel - Computer Physics Communications 184 - 2013

[JY] *"Reconstructing Surfaces of Particle-Based Fluids Using Anisotropic Kernels"*
Jihun Yu, Greg Turk
Artikel - ACM Transactions on Graphics, Vol. 32, No. 1 - 2013

[MM] *"Position Based Fluids "*
Miles Macklin, Matthias Müller
Artikel - ACM Transactions on Graphics - 2013

[MH] *"Radix Tricks"*
Michael Herf
Artikel - http://stereopsis.com/radix.html - Dezember 2001

[MaH] *"GPU Gems 3 - Chapter 39. Parallel Prefix Sum (Scan) with CUDA"*
Mark Harris,Shubhabrata Sengupta, John D. Owens
Buch - ISBN-13: 978-0-321-51526-1 - 2008

[NVa] *"CUDA-Programming-Guide"*
NVIDIA
Guide - http://docs.nvidia.com/cuda/cuda-c-programming-guide/

[NVb] *"NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK-110 "*
NVIDIA
Whitepaper - http://www.nvidia.de/content/PDF/kepler/
NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[PG] *"Interactive SPH Simulation and Rendering on the GPU"*
Prashant Goswami, Philipp Schlegel, Barbara
Artikel - Symposium on Computer Animation - 2010

[RF] *"Efficient High-Quality Volume Rendering of SPH Data"*
Roland Fraedrich, Stefan Auer, Rüdiger Westermann
Artikel -IEEE Transactions on Visualization and Computer Graphics 16 - 2010

[RHa] *"Fluid v3 Development"*
Rama Hoetzlein
Dokumentation - http://www.rchoetzlein.com/fluids3/?page_id=32 - 2014

[RHb] *"FAST FIXED-RADIUS NEAREST NEIGHBORS: INTERACTIVE MILLION-PARTICLE FLUIDS"*
Rama Hoetzlein
Präsentation - GPU Technology Conference - 2014

[SG] *"Particle Simulation using CUDA "*
Simon Green
Dokumentation - CUDA SDK 6.5 - 2014

[SGG] *"Optimizing and Auto-tuning Belief Propagation on the GPU"*
Scott Grauer-Gray and John Cavazos
Artikel - Lecture Notes in Computer Science Volume 6548 - 2011

[TexTer] "-"
Autoren unbekannt
Sand: http://www.textures123.com/free-texture/sand/sand-texture4.jpg
Gras: http://xmoto.tuxfamily.org/sprites/Textures/Textures/T_Grass3_944.jpg
Felsen: http://kcgd.info/wp-content/uploads/2014/04/rock-tile-1.jpg
Skybox: http://www.redsorceress.com/skyboxs/hourglass.zip

[TexSky] "-"
Autor unbekannt
http://www.redsorceress.com/skyboxs/hourglass.zip

[TR] *"Does Your Software Scale ? Multi-GPU Scaling for Large Data Visualization "*
Thomas Ruge
Präsentation - NVision 08 - 2008

[TH] *"Sliced Data Structure for Particle-Based Simulations on GPUs"*
Takahiro Harada, Seiichi Koshizuka, Yoichiro Kawaguchi
Artikel - Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia - 2007

[VV] *"Better Performance at Lower Occupancy"*
Vasily Volkov

Präsentation - UC Berkeley - www.cs.berkeley.edu/ volkov/volkov10-GTC.pdf - September 2010

[WL] *"Screen Space Fluid Rendering with Curvature Flow"*
Wladimir J. van der Laan, Simon Green, Miguel Sainz
Artikel - Proceedings of the 2009 symposium on Interactive 3D graphics and games -2009

[WP] *"Texture Mapping"*
Wikipedia
Artikel -http://en.wikipedia.org/wiki/Texture_mapping - 2014

8.3 Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig angefertigt und ohne fremde Hilfe verfasst, keine neben den von mir angegebenen Hilfsmitteln und Quellen dazu verwendet und die den benutzten Werken inhaltlich oder wörtlich entnommenen Stellen als solche kenntlich gemacht habe. Des Weiteren versichere ich, dass ich diese Arbeit nicht bereits zur Erlangung eines akademischen Grades eingereicht habe. Ich versichere auch, dass die elektronische Form der Masterarbeit auf der CD-ROM der gebundenen Fassung der Masterarbeit entspricht.

Bayreuth, Januar 2015

_____

Tim Werner