Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZFI-BM-2011-01

Bachelorarbeit

im Studiengang "Angewandte Informatik"

GPU-basiertes Volumen Ray Casting am Beispiel einer Strömungssimulation

Sebastian Wolter

am Institut für

Numerische und Angewandte Mathematik

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen

18. September 2011

Georg-August-Universität Göttingen
Zentrum für Informatik

Goldschmidtstraße 7
37077 Göttingen
Germany

Tel. +49 (5 51) 39-1 72 000
Fax +49 (5 51) 39-1 44 03
Email info@informatik.uni-goettingen.de
WWW www.informatik.uni-goettingen.de

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den 18. September 2011

Bachelorarbeit

GPU-basiertes Volumen Ray Casting am Beispiel einer Strömungssimulation

Sebastian Wolter

18. September 2011

Betreut durch Prof. Dr. Gert Lube und Dr. Jochen Schulz
Institut für Numerische und Angewandte Mathematik
Georg-August-Universität Göttingen

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Durch die fortlaufende Entwicklung der Computertechnik können immer bessere Computersimulationen von naturwissenschaftlichen Phänomenen durchgeführt werden. Hierbei fallen enorme Datenmengen an, die verarbeitet und veranschaulicht werden müssen, um das Ergebnis besser zu verstehen und analysieren zu können. Die Visualisierung von zweidimensionalen Datensätzen ist vergleichsweise einfach. Der Datensatz besitzt für die zwei Raumdimensionen einen Wert, diese können durch Färbung eines Bildschirmpixels ihren Wert repräsentieren. Bei drei-dimensionalen Daten muss anders verfahren werden. Zwar könnte man durch Fixierung der dritten Dimension einen Teil des Datensatzes abbilden, vergleichbar mit einer Computertomographie, würde jedoch das Gesamtbild nicht verstehen. Geeignete Visualisierungsverfahren sollten also direkt erkennen lassen „wo sich etwas tut“.
Eine wichtige Datenquelle ist der Bereich der numerischen Strömungsmechanik (Computational Fluid Dynamics), die durch Approximation mit numerischen Verfahren das Strömungsverhalten von newtonschen Flüssigkeiten und Gasen (Fluide), welches durch die Navier-Stokes-Gleichungen beschrieben sind, berechnen. Als Anwendung sei hier die Entwicklung von Energie sparenden Autos (durch Reduzierung des Luftwiderstands), die Wettervorhersage oder die Optimierung von Klimaanlagen genannt. Bei hoher Präzision der Ergebnisse ist die Rechenzeit entsprechend hoch, so dass eine Simulation mehrere Tage, Wochen oder Monate dauert unter Einsatz von hunderten von Rechnern.
In der Computergrafik ist eine so hohe Präzision der Ergebnisse nicht nötig. In der Film- und Spieleindustrie als auch in den Medien werden (pseudo-)realistische Fluidsimulationen gebraucht, die zwar ungenauer sind, allerdings auch sehr schnell zu berechnen sind. Durch das Aufkommen von Mehrkernprozessoren und programmierbaren Grafikkarten können durch Parallelisierung selbst auf einem Standard-PC mit entsprechender Hardware (pseudo-)realistische Fluidsimulationen berechnet werden.

1.2 Ziel der Arbeit

Diese Arbeit besteht aus einer Strömungssimulation basiert auf der Arbeit „Stable Fluids“ von Jos Stam [1] und der Visualisierung der entstehenden 3D-Daten.

Jos Stam ist Mitentwickler der größten 3D-Visualisierungs- und -Animationssoftware (Maya), die momentan auf dem Markt erhältlich ist, welche hauptsächlich in der Film- und Fernsehindustrie eingesetzt wird. In seiner Arbeit stellte er ein stabiles Verfahren vor, mit dem die Navier-Stokes-Gleichungen für inkompressible Fluide schnell berechnet werden können. Auf Basis dessen wird in der Bachelorarbeit eine 3D-Variante implementiert werden. Hierzu wird die Programmierschnittstelle CUDA zur Parallelisierung auf NVIDIA-Grafikkarten verwendet werden. Die Visualisierung soll durch Volumen Ray[1] Casting erfolgen.

1.3 Gliederung der Arbeit

Der erste Teil der Arbeit beschreibt verschiedene Techniken zur Visualisierung von 3D-Volumendaten. Hierbei wird das verwendete Verfahren Volumen Ray-Casting aufgrund seiner guten Parallelisierbarkeit detailliert betrachtet. Dann folgen die mathematischen Grundlagen zu den Navier-Stokes-Gleichungen und der Beschreibung nach Jos Stam. In Abschnitt 3 folgt eine Beschreibung der verwendeten Architektur und ein kurzer Abriss über die Besonderheiten der Programmiersprache CUDA, neben Details zur Implementierung und Erklärungen zum parallelisierten Quellcode. In Abschnitt 4 werden dann die Ergebnisse, Benchmarks und Laufzeitanalysen präsentiert, gefolgt von dem Fazit und dem Ausblick auf mögliche Erweiterungen und Verbesserungen in Abschnitt 6.

[1] Strahl

2 Grundlagen

In diesem Abschnitt werden die theoretischen Grundlagen gelegt, auf denen der später implementierte Algorithmus basiert. Hierbei werden zunächst die Grundlagen für die Visualisierung und später die Grundlagen der Strömungssímulation gelegt.

2.1 Volumenrendering

Um Volumendaten zu visualisieren wird ein entsprechendes Rendering-Verfahren benötigt. Die Art der Datenrepräsentation ist hierbei entscheidend für die Auswahl eines geeigneten Verfahrens, daher werden im Folgenden verschiedene Arten definiert.

2.1.1 Datenrepräsentation

Volumendaten sind Daten, die sich in folgender Form schreiben lassen:

$$data = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \tag{2.1}$$

x, y, z geben die Raumkoordinaten an, w bezeichne einen skalaren Wert. Dieser räumliche Datenwert wird auch als Voxel bezeichnet (volumetric pixel). Die interne Struktur kann durchaus unterschiedlich sein:

- kartesisch: angeordnet in einem Würfelgitter
- regulär: Kartesisch, mit rechteckigem Gitter
- unstrukturiert: Datenpunkte sind unstrukturiert im Volumen definiert
- hybrid: Kombination aus strukturierten und unstrukturierten Daten.

Da reguläre Gitter sowohl für das Volumenrendering, als auch für die Simulation einfacher zu rechnen sind, wird nun angenommen, das die Daten in einem regulären 3D-Gitter vorliegen.

2.1.2 Volumenvisualisierung

Das Ziel des Volumenrendering[1] ist es, aus 3D-Datensätzen eine 2D-Projektion zu erhalten. Man schafft sich die Möglichkeit, sowohl Objekte zu betrachten, die sich schwer durch Polygone approximieren lassen, als auch in Objekte hineinzuschauen (z. B. Temperaturverläufe, Gaswolken, Knochen bei medizinischen CT-Daten).
Die Verfahren können in zwei Arten eingeteilt werden: indirekte und direkte Verfahren.
Indirekte Verfahren extrahieren Oberflächen gleichen Werts (Isolinien/Isoflächen) und stellen diese durch traditionelle Polygonbasierte Verfahren dar. Als Beispiel sei hier der Marching Cubes Algorithmus genannt, welches versucht ein Voxelmodell durch Oberflächenmodelle zu approximieren. indem es zunächst das Voxelmodell in kleine Würfel zerlegt, von einem Würfel zum nächsten „marschiert“ und bestimmt, wie das Objekt den Würfel schneidet.
Direkte Verfahren betrachten den Datensatz als transparentes Medium, bei dem jeder Wert einer Farbe und einer Opazität[2] zugeordnet wird. Je nach Realitätsanspruch kann zusätzlich die Emission und Absorption modelliert werden. Beispiele hierfür sind der Splatting-Algorithmus, der Shear-Warp-Algorihmus und das Ray Casting[14].

2.1.3 Ray Casting

Das Ray Casting ist eine Abwandlung des Ray Tracing, und wurde zuerst 1988 von Marc Levoy [6] beschrieben. Da das Ray Casting allerdings sehr aufwändig ist, war Echtzeit-Raycasting lange Zeit nicht möglich, da je nach Auflösung Millionen Strahlberechnungen durchgeführt werden müssen. Durch die Entwicklung schneller Grafikkarten ist es nun möglich durch Parallelisierung der Strahlberechnungen eine Visualisierung eines Datensatzes durch die Ray Casting-Technik in Echtzeit zu ermöglich. Das grundlegende Verfahren zur Volumenvisualisierung mittels Ray Casting ist in Abbildung 2.1 angedeutet und soll im Folgenden kurz beschrieben werden.

Ablauf

Die virtuelle Darstellungsumgebung besteht aus mindestens vier Elementen:

- Volumendatensatz V, umgeben von einer achsenparallelen Box B
- Betrachterposition (Kamera) $C = \begin{pmatrix} C_x \\ C_y \\ C_z \end{pmatrix}$

[1] Rendering, zu deutsch: berechnen
[2] Lichtundurchlässigkeit

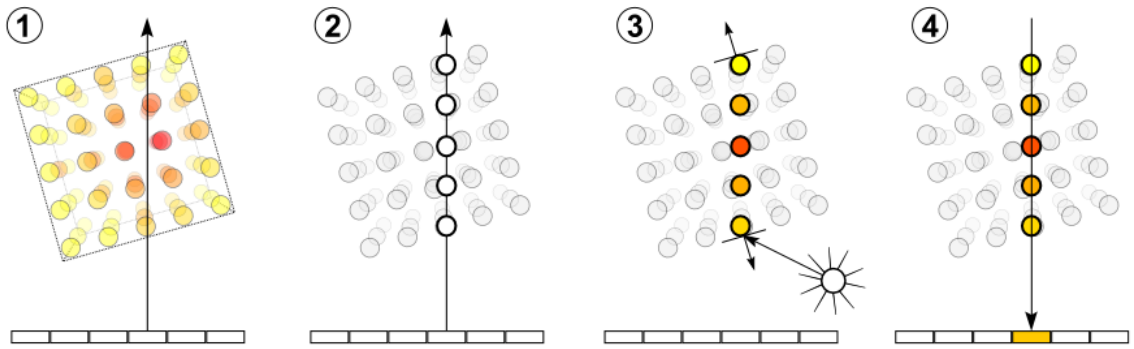


Abbildung 2.1: Ablauf des Ray Casting (ohne Transferfunktion)[16]

- Bildfläche F, dessen Größe durch die Auflösung (z.B. 512x512), einem Aufhängepunkt E_s und zwei normierten Richtungsvektoren E_r, E_t definiert ist:
$$\begin{pmatrix} P_{s_x} \\ P_{s_y} \\ P_{s_z} \end{pmatrix} + x * \begin{pmatrix} E_{r_x} \\ E_{r_y} \\ E_{r_z} \end{pmatrix} + y * \begin{pmatrix} E_{t_x} \\ E_{t_y} \\ E_{t_z} \end{pmatrix}$$
- Lichtquelle $L = \begin{pmatrix} L_x \\ L_y \\ L_z \end{pmatrix}$.

Der Betrachter befindet sich an einer Kamera-Position $C \in \mathbb{R}^3$. Vor ihm befindet sich die durch Länge und Breite begrenzte Bildfläche F, welche in $k = w \cdot h$ Pixel ($w = Breite$, $h = Höhe$) aufgeteilt ist. Das zu betrachtende Volumen V ist als Inhalt einer achsenparallelen Box B definiert. Der Ablauf des Ray Casting besteht nun aus den folgenden Schritten:

1. Konstruiere für jeden Punkt (x, y) der Ebene F einen Strahl R_{xy} mit
$$R_{xy} = \underbrace{\begin{pmatrix} C_x \\ C_y \\ C_z \end{pmatrix}}_{R_s} + t \cdot \underbrace{\begin{pmatrix} C_x - (E_{s_x} + x * E_{r_x} + y * E_{t_x}) \\ C_y - (E_{s_y} + x * E_{r_y} + y * E_{t_y}) \\ C_z - (E_{s_z} + x * E_{r_y} + y * E_{t_z}) \end{pmatrix}}_{R_d}.$$
2. Prüfe auf Schnitt mit der Volumenumgebungsbox B.
3. Bei Schnitt: Tauche in das Volumen V beginnend mit dem ersten (oder zweiten) Schnitt und sample in festen Intervallen einen Wert S aus dem Volumen durch trilineare Interpolation.

4. Anhand einer Transferfunktion $f : S \rightarrow RBGA$ wird für den skalaren Wert eine Farbe und Opazität (Alpha-Wert) bestimmt.

5. Ein lokales Beleuchtungsmodell wird auf den gesampelten Punkt angewendet.

6. Ein Kompositionsschema kombiniert die gesampelten Farbwerte zu einer Endfarbe G.

7. Der Bildschirm an der Stelle (x, y) auf die Farbe G gesetzt.

Die im Ablauf benötigten Operationen und Modelle werden nun eingeführt. Als Farbmodell wird im Allgemeinen das RGBA-Modell verwendet, daher wird dieses nun kurz beschrieben.

2.1.4 RGB(A)-Farbmodell

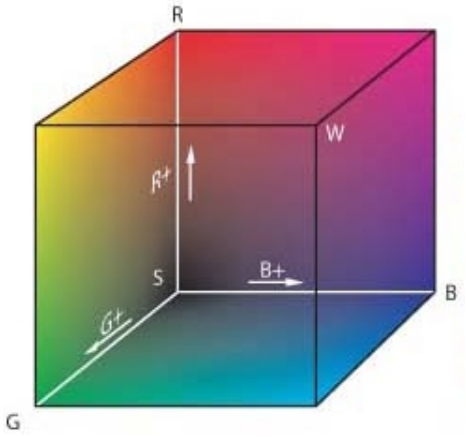


Abbildung 2.2: Das RGB-Modell

Das RGB Modell beschreibt ein additives Farbmodell durch die Primärfarben Rot, Grün und Blau. Konkrete Farbwerte können als Vektoren $\in [0, 1]^3$ oder $[0, 255]^3$ beschrieben werden (siehe Abbildung 2.2). Eine Erweiterung ist das RBGA Modell. Hier wird zusätzlich die Transparenzinformation α mit angegeben. $\alpha = 0$ bedeutet vollständige Transparenz, $\alpha = 1$ vollständige Opazität. Ein Farbwert $f \in RBGA$ kann durch die Transformation $f \cdot \alpha$ in das RGB Modell überführt werden.
Um zu berechnen, ob ein Strahl die Volumenumgebungsbox schneidet oder nicht, wird die entsprechende Schnittpunktberechnung benötigt. Da in der Implementierung keine weiteren Objekte verwendet werden ist die folgende Ray-Box-Schnittberechnung die einzig verwendete.

2.1.5 Ray-Box-Schnittberechnung

Die Box-Strahl-Schnittpunktberechnung soll mit Hilfe der „Slab-Methode“ von Kay und Kayja [8] erfolgen. „Slab“ bezeichnet den Bereich zwischen zwei parallelen Ebenen. Der Schnitt zwischen einem Strahl und einer Achsenparallelen Box definiert durch die Punkte o_{min} und o_{max} kann auf folgende Weise berechnet werden:

```
tnear = -infinity
tfar = infinity
Für jede Dimension i=1:3
    if(Ray.d[i]==0) {
        if(Ray.s[i]<omin[i] Or Ray.s[i]>omax[i])
            return false // kein Hit
    } else {
        t1=(omin[i]-Ray.s[i])/Ray.d[i]
        t2=(omax[i]-Ray.s[i])/Ray.d[i]
        if(t1>t2) swap(t1,t2)
        if(t1>tnear) tnear=t1
        if(t2<tfar) tfar=t2
        if(tnear>tfar) return false // Box wird nicht getroffen
        if(tfar<0) return false // Box befindet sich hinter dem Strahl
    }
return true
// tnear führt mit ray.s+tnear*ray.d zu 1. Schnittpunkt
// tfar entsprechend zum 2. Schnittpunkt
```

Ein Beleuchtungsmodell wird verwendet, um den Kontrast zu erhöhen und Helligkeitsunterschiede zu erzeugen.

2.1.6 Beleuchtungsmodell

Führt man eine Beleuchtung der einzelnen Sample-Punkte durch, so benötigt man ein nicht zu aufwendiges lokales Beleuchtungsmodell, da dies für jeden Sample-Punkt angewendet wird. In der Computergrafik wird häufig das Phong-Modell [7] verwendet.

$$I_{out} = I_a k_{amb} + I_{in}(k_{diff} cos(L \cdot N) + k_s (R \cdot V)^n) \tag{2.2}$$

I_{in} : Intensität des einfallenden Lichts
I_a : Intensität des Umgebungslichts
k_{amb} : empirische Materialkonstante
k_{diff} : empirischer Faktor für Reflexion des diffusen Lichts (diffuse Materialfarbe)
k_s : empirischer Faktor für Reflexion des spiegelnden Lichts (spiegelnde Materialfarbe)

n : konstanter Faktor zur Beschreibung der Rauheit
L : Lichtrichtung
N : Normalenvektor
V : Richtung zum Betrachter
$R = (2 * (N \cdot L)N) - L$: Reflektionsvektor (normalisiert)

Dieses Modell ist allerdings kein physikalisch.basiertes Modell, sondern rein empirisch. Das Phong-Modell benötigt den Normalenvektor am Sample-Punkt. Da allerdings in einem Volumendatensatz keine Normalen vorhanden sind, wird stattdessen der Gradient $\nabla data_{i,j,k}$ verwendet. Dieser wird durch das Zentraldifferenzenschema gebildet. Siehe hierzu Abschnitt 2.2.2.

2.1.7 Transferfunktion

Die Aufgabe der Transferfunktion ist, anhand des Datenwerts eine Farbe und die Opazität festzulegen. Somit können Regionen priorisiert werden. Es können sowohl ein- als auch mehrdimensionale Transferfunktionen verwendet werden. Eindimensionale Transferfunktionen sind besonders einfach und schnell zu implementieren. Es können sowohl Funktionen von Hand programmiert werden, als auch mathematische Funktionen verwendet werden. Ein Beispiel für eine mehrdimensionale Funktion:

$$\alpha(x_i) = \begin{cases} 1, & falls\, |\nabla f(x_i)| = 0\, und\, f(x_i) = f_v \\ 1 - \frac{1}{r}\left|\frac{f_v - f(x_i)}{\nabla f(x_i)}\right|, & falls\, |\nabla f(x_i)| > 0\, und\, f(x_i) - r|\nabla f(x_i)| \leq f_v \leq f(x_i) + r|\nabla f(x_i)| \\ 0, & sonst \end{cases} \tag{2.3}$$

Für jeden Wert f_v wird also eine Opazität α_v definiert, die in Abhängigkeit des Gradienten abnimmt. r ist ein Faktor zur Steuerung der Opazitätsabnahme (Fall-Off), $f(x_i)$ Volumen-Datenwert an der Stelle x_i.

2.1.8 Komposition

Im Kompositionsschritt werden nun die gesampelten Farbdaten nach einem bestimmten Schema zusammengeführt.

Beispiele für Schemata

- Maximum-Value
- Average-Value
- First-Value

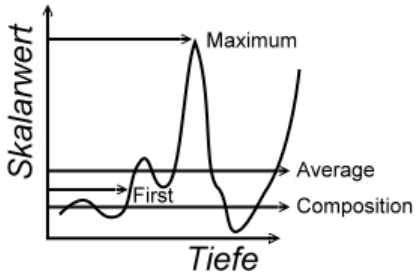


Abbildung 2.3: Kompositionsschritt

- Composition

Die Schemata sind in Abbildung 2.3 dargestellt.

Maximum Value

Bei Maximum Value wird die Farbe des Punktes übernommen, der das Maximum angenommen hat entlang des Strahls durch das Volumen.

Average Value

Bei Average Value wird über alle Farbwerte entlang des Strahls gemittelt.

First Value

Bei First Value wird der Farbwert des ersten Datenpunkts genommen, dessen Wert über einem Anfangsschwellenwert liegt.

Komposition

Bei der eigentlichen Komposition werden die Farbwerte und Alphawerte nach einem front-to-back oder back-to-front Schema aufaddiert.

$$back - to - front: \;\; C_{i+1} = c_i * (1 - \alpha_i) + C_i \cdot \alpha_i \tag{2.4}$$

$$\begin{aligned} front - to - back: \;\; C_{i+1} &= \alpha_i \cdot c_i + (1 - \alpha_i) \cdot A_i \cdot C_i \\ A_{i+1} &= \alpha_i + (1 - \alpha_i) \cdot A_i \end{aligned} \tag{2.5}$$

C_i : aktuell aufsummierte Zwischenfarbe
A_i : aktuell aufsummierte Zwischenopazität

α_i: Opazität des aktuellen Samples
c_i : Farbwert des aktuellen Samples
Das Kompositionsverfahren lässt sich aus dem Volumen Rendering Integral herleiten [10].

Volumen Rendering Integral

Tritt Licht durch ein Feld (Volumendatensatz), so kommt es zu einer Dämpfung der anfänglichen in das Feld eintretenden Lichtintensität. Sei $r(t)$ ein Sichtstrahl mit Abstandsvariable t zum Beobachter. Sei $u : R^3 \rightarrow R^+$ die Absorptionsfunktion. Die Intensitätsabschwächung durch das Volumen lässt sich beschreiben durch

$$e^{-\int_0^t u(r(t'))dt'}. \tag{2.6}$$

Sei $c(r(t))$ die Farbe im Punkt $r(t)$. Somit ergibt sich das Volumen Rendering Integral:

$$C = \int_o^T c(r(t)) \cdot e^{-\int_0^t u(r(t'))dt'} dt. \tag{2.7}$$

Analytisch ist dieses Integral nicht berechenbar, daher muss numerisch approximiert werden. Sei $T = max(t)$ die maximale Distanz zum Beobachter. Man unterteilt nun T in k äquidistante Segmente. Nun ersetzt man das Integral durch endliche Riemannsummen.

$$C = \sum_{i=0}^{k} (c_i \cdot e^{-\sum_{j=0}^{i-1} u_j}) = \sum_{i=0}^{k} (c_i \cdot \prod_{j=0}^{i-1} e^{-u_j}). \tag{2.8}$$

Als Vereinfachung wird nun $e^{-u_j} = (1 - \alpha_j)$ gesetzt. Dies führt zu:

$$C = \sum_{i=0}^{k} (c_i \cdot \prod_{j=0}^{i-1} (1 - \alpha_j)). \tag{2.9}$$

2.2 Fluidsimulation

Im folgenden Abschnitt werden die Navier-Stokes-Gleichungen und das Modell der Stable Fluids beschrieben.

2.2.1 Navier-Stokes-Gleichungen

Basierend auf den Euler-Gleichungen erweiterten Claude Louis Marie Henri Navier und George Gabriel Stokes Mitte des 19. Jahrhunderts auf reibungsbehaftete newtonsche Fluide wie Wasser, Öl oder Luft. Im Falle eines inkompressiblen und homogenen (konstante

Dichte in Raum und Zeit) Fluids kann die Änderung der Geschwindigkeit zum Zeitpunkt t folgendermaßen beschrieben werden:

$$\frac{\partial u}{\partial t} = -(u \cdot \nabla)u - \frac{1}{d}\nabla p + \nu\nabla^2 u + f \tag{2.10}$$

$$\nabla \cdot u = 0 \tag{2.11}$$

Hierbei bezeichnet u das Geschwindigkeitsfeld, t die Zeit, d die Dichte, p das Druckfeld, ν die Viskosität und f den Einfluss externer Kräfte. Die erste Gleichung (Impulserhaltungsgleichung) beschreibt die Änderung der Geschwindigkeit in Abhängigkeit der genannten Größen, die zweite Gleichung (Masseerhaltungsgleichung) sorgt durch die Divergenzfreiheit des Geschwindigkeitsfeldes für die Masseerhaltung. Die Impulserhaltungsgleichung besteht aus drei Teilen: Advektion[3], Druck, Diffusion und externe Beschleunigung.

Advektion

Der Advektionsterm $-(u \cdot \nabla)u$ beschreibt, wie Änderungen des Geschwindigkeitsfeldes sich entlang desselbigen ausbreiten, wodurch die Navier-Stokes-Gleichungen ihre Nichtlinearität erhalten.

$$-(u \cdot \nabla u) = -\begin{pmatrix} u_x \frac{\partial u_x}{\partial x} + u_y \frac{\partial u_x}{\partial y} + u_z \frac{\partial u_x}{\partial z} \\ u_x \frac{\partial u_y}{\partial x} + u_y \frac{\partial u_y}{\partial y} + u_z \frac{\partial u_y}{\partial z} \\ u_x \frac{\partial u_z}{\partial x} + u_y \frac{\partial u_z}{\partial y} + u_z \frac{\partial u_z}{\partial z} \end{pmatrix} \tag{2.12}$$

Um den Advektionsterm zu approximieren, verwendet Jos Stam die sogenannte Charakteristikenmethode (Semi-Lagrange-Methode), da sie im Gegensatz zur Finiten-Differenzen-Approximation für beliebige Zeitschritte stabil bleibt. Bei jedem Zeitschritt werden die Fluidpartikel durch ihr eigenes Geschwindigkeitsfeld bewegt. Die Geschwindigkeit eines Partikels x zum Zeitpunkt $t + \Delta t$ ergibt sich, indem man das Fluidpartikel entlang seines Pfades (der Charakteristik) zurückverfolgt, um seine Position P zum Zeitpunkt $t - \Delta t$ zu bestimmen. Die Geschwindigkeit bewegt sich also entlang dieses Pfades und wurde von der Position P advektiert (siehe hierzu auch Abbildung 2.4 und 2.5). Daher bestimmt man die Geschwindigkeitswerte am Punkt P im Gitter (z. B. durch trilineare Interpolation) und übernimmt diese als neue Werte zum Zeitpunkt $t + \Delta t$. Da dies eine Mischung aus einem partikelbasierten und einem gitterbasierten Ansatz darstellt, bezeichnet man dies wie oben schon erwähnt als Semi-Lagrange-Methode. Vorteile dieser Methode sind die unkonditionelle Stabilität und die einfache Implementierung. Die Stabilität folgt daher, dass der Maximalwert zum neuen Zeitpunkt $t + \Delta t$ niemals größer ist, als der Maximalwert im vorherigen Zeitpunkt. Somit kommt es zu keiner Explosion der Geschwindigkeit. Stam führt aus, dass die Methode der Charakteristiken an Präzision gewinnt, wenn das Feld vor

[3] von lat. advehi - heranbewegen

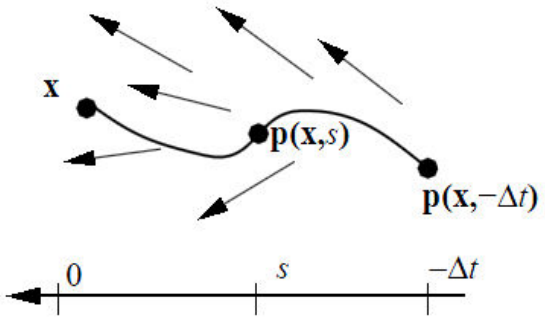


Abbildung 2.4: Charakteristik [1]

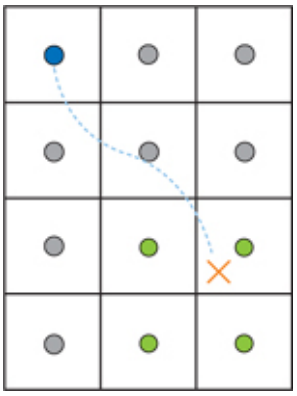


Abbildung 2.5: Backtracing [9]

Anwendung divergenzfrei ist. Dies erklärt die zweimalige Anwendung des Projektionsschrittes in der Implementierung (siehe Abschnitt 3.3).

Druck

Der Druckterm beschreibt die Beschleunigung, die durch den Druckgradienten verursacht wird.

$$-\frac{1}{d}\nabla p = -\frac{1}{d}\begin{pmatrix} \frac{\partial p}{\partial x} \\ \frac{\partial p}{\partial y} \\ \frac{\partial p}{\partial z} \end{pmatrix}. \tag{2.13}$$

Diffusion

Der Diffusionsterm beschreibt die Diffusion des Fluids in Abhängigkeit der Viskosität. Je höher die Viskosität, desto dickflüssiger ist das Fluid.

$$\nu\nabla^2 u = \nu\begin{pmatrix} \frac{\partial^2 u_x}{\partial x^2} + \frac{\partial^2 u_x}{\partial y^2} + \frac{\partial^2 u_x}{\partial z^2} \\ \frac{\partial^2 u_y}{\partial x^2} + \frac{\partial^2 u_y}{\partial y^2} + \frac{\partial^2 u_y}{\partial z^2} \\ \frac{\partial^2 u_z}{\partial x^2} + \frac{\partial^2 u_z}{\partial y^2} + \frac{\partial^2 u_z}{\partial z^2} \end{pmatrix}. \tag{2.14}$$

Externe Beschleunigung

Der letzte Term beschreibt den Einfluss von externen Beschleunigungsfaktoren. Beispiele für solche Faktoren sind

- Gravitation: $f_{grav} = -\alpha \cdot d$

- thermaler Auftrieb: $f_{bouyancy}$ (siehe Abschnitt 2.2.7)
- Vorticity Confinement: $f_{vorticity}$ (siehe Abschnitt 2.2.8).

2.2.2 Diskretisierung

Um die Navier-Stokes-Gleichungen numerisch zu lösen, müssen diese diskretisiert werden. Daher teilt man den Raum durch ein regelmäßiges kartesiches Gitter in Würfel äquidistanter Größe ein. In jedem Würfel (Zelle) bzw. den Zellzentren werden die Variablen (Geschwindigkeit, Druck, etc.) gespeichert. Man nennt diese Art Gitter auch „collocated grid“. Da die Variablen nur in den Zellzentren definiert sind, müssen die Werte dazwischen interpoliert werden.

Nabla-Operator

Da in den Naver-Stokes-Gleichungen der ∇- Operator in verschiedener Ausführung gebraucht wird, wird dieser nun kurz definiert mit dazugehöriger diskretisierten Form.

Gradient

Im $\mathbb{R}^3$ ist der Gradient eines skalaren Feldes F folgendermaßen definiert:

$$\nabla F = \left(\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}, \frac{\partial F}{\partial z} \right) \tag{2.15}$$

Diskretisierung durch die Zentral-Differenzen-Form ergibt:

$$\nabla F_{i,j,k} = \left(\frac{F_{i+1,j,k} - F_{i-1,j,k}}{2h}, \frac{F_{i,j+1,k} - F_{i,j-1,k}}{2h}, \frac{F_{i,j,k+1} - F_{i,j,k-1}}{2h} \right) \tag{2.16}$$

Divergenz

Im $\mathbb{R}^3$ ist die Divergenz, angewendet auf ein Vektorfeld $F = (u, v, w)_{i,j,k}$, folgendermaßen definiert:

$$\nabla \cdot F = \left(\frac{\partial F}{\partial x} + \frac{\partial F}{\partial y} + \frac{\partial F}{\partial z} \right) \tag{2.17}$$

Diskretisierung durch die Zentral-Differenzen-Form ergibt:

$$\nabla \cdot F_{i,j,k} = \frac{u_{i+1,j,k} - u_{i-1,j,k}}{2h} + \frac{v_{i,j+1,k} - v_{i,j-1,k}}{2h} + \frac{w_{i,j,k+1} - w_{i,j,k-1}}{2h} \tag{2.18}$$

Laplace-Operator

Im $\mathbb{R}^3$ ist der Laplace-Operator angewendet auf ein skalares Feld F folgendermaßen definiert:

$$\nabla^2 F = \left(\frac{\partial^2 F}{\partial x^2} + \frac{\partial^2 F}{\partial y^2} + \frac{\partial^2 F}{\partial z^2} \right) \tag{2.19}$$

Diskretisierung durch die Zentral-Differenzen-Form ergibt:

$$\nabla^2 F_{i,j,k} = \frac{F_{i+1,j,k} - 2F_{i,j,k} + F_{i-1,j,k}}{(\Delta x)^2} + \frac{F_{i,j+1,k} - 2F_{i,j,k} + F_{i,j-1,k}}{(\Delta y)^2} + \frac{F_{i,j,k+1} - 2F_{i,j,k} + F_{i,j,k-1}}{(\Delta z)^2} \tag{2.20}$$

Operator Splitting

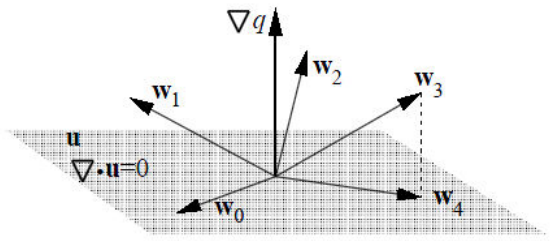


Abbildung 2.6: Operator Splitting [1]

Zur Berechnung der Navier-Stokes-Gleichungen wird das Problem in einfachere partielle Differentialgleichungen aufgeteilt. Für gegebenen Startzustands $u_0 = u(x, 0)$ wird die Lösung $u_{t+\Delta t}$ in vier Schritten berechnet (siehe hierzu auch Abbildung 2.6):

$$w_0(x) \overset{ext.Kraefte}{\rightarrow} w_1(x) \overset{Advektion}{\rightarrow} w_2(x) \overset{Diffusion}{\rightarrow} w_3(x) \overset{Projektion}{\rightarrow} w_4(x)$$

mit $w_0(x) = u(x, t)$, $u(x, t + \Delta t) = w_4(x)$.

Externe Kräfte

$$\frac{\partial w_1}{\partial t} = f \tag{2.21}$$

wird approximiert durch

$$\frac{w_1(x) - w_0(x)}{\Delta t} = f(x, t) \Leftrightarrow w_1(x) = w_0(x) + \Delta t f(x, t) \tag{2.22}$$

unter der Annahme, dass sich während eines Zeitschritts die Kräfte nicht ändern.

Advektion

$$\frac{\partial w_2}{\partial t} = -(w_2 \cdot \nabla w_2) \tag{2.23}$$

Wie bereits in Abschnitt 2.2.1 ausgeführt verwendet Stam eine Semi-Lagrange Methode, um den Advektionsschritt zu berechnen. Die grundlegende Idee der Semi-Lagrange Methode ist, einen Punkt x rückwärts in der Zeit ($-\Delta t$) entlang seines Pfades P zu verfolgen und den gefundenen Vektor als neuen Wert zu übernehmen. Sei $p(x,t)$ eine Funktion, die die erwähnte Punktverfolgung anwendet, so lässt sich der Advektionsterm folgendermaßen approximieren:

$$w_2(x) = w_1(p(x,-\Delta t)). \tag{2.24}$$

Diffusion

$$\frac{\partial w_3}{\partial t} = \nu \nabla^2 w_3 \tag{2.25}$$

Dies wird approximiert durch:

$$\frac{w_3 - w_2}{\Delta t} = \nu \nabla^2 w_3 \tag{2.26}$$

$$w_3 - \nu \Delta t \nabla^2 w_3 = w_2 \tag{2.27}$$

$$(I - \nu \Delta t \nabla^2) w_3 = w_2. \tag{2.28}$$

Dies führt durch Diskretisierung des ∇^2-Operators zu einem dünnbesetzten linearen Gleichungssystem.

$$w^3_{i,j,k} - \frac{\nu \Delta t}{h^2} \left(w^3_{i-1,j,k} + w^3_{i,j-1,k} + w^3_{i,j,k-1} + w^3_{i+1,j,k} + w^3_{i,j+1,k} + w^3_{i-1,j,k+1} - 6w^3_{i,j,k} \right) = w^2_{i,j,k} \tag{2.29}$$

Projektion

Um den Druckterm zu approximieren, bedient man sich der Helmholtz-Hodge-Zerlegung, welche besagt, dass jedes Vektorfeld w in ein divergenzfreies Vektorfeld u und ein Gradientenfeld zerlegt werden kann:

$$w = u + \nabla q \tag{2.30}$$

Somit kann ein Projektionsoperator P definiert werden, der das Vektorfeld w auf das divergenzfreie Vektorfeld u abbildet. Dies geschieht implizit, durch Multiplikation der beide Seiten der Gleichung mit ∇ unter Beachtung dass $\nabla \cdot u = 0$:

$$\nabla \cdot w = \nabla^2 q \tag{2.31}$$

Dies entspricht einer Poisson-Gleichung für das Skalarfeld q mit Neumann Randbedingung $\frac{\partial q}{\partial n} = 0$ auf dem Rand ∂D Hat man q errechnet, so ergibt sich u durch:

$$u = w - \nabla q \tag{2.32}$$

Wendet man den Projektionsoperator auf die Navier-Stokes-Gleichungen an, so ergibt sich eine Gleichung für die Geschwindigkeit:

$$\frac{\partial u}{\partial t} = P(-(u \cdot \nabla)u + \nu \nabla^2 u + f). \tag{2.33}$$

Eigenschaften des Projektionsoperators sind: $Pu = u, P\nabla p = 0$.
Somit berechnet sich der letzte Schritt durch:

$$\nabla^2 q = \nabla \cdot w_3 \quad w_4 = w_3 - \nabla q \tag{2.34}$$

Das Lösen der Poissongleichung ergibt auch wieder ein dünnbesetztes lineares Gleichungssystem:

$$\begin{aligned} \frac{1}{h^2}(q_{i-1,j,k} + q_{i,j-1,k} + q_{i,j,k-1} + q_{i+1,j,k} + q_{i,j+1,k} + q_{i,j,k+1} - 6q_{i,j,k}) = \\ \frac{1}{2h}(w^3_{i+1,j,k} - w^3_{i-1,j,k} + w^3_{i,j+1,k} - w^3_{i,j-1,k} + w^3_{i,j,k+1} - w^3_{i,j,k-1}) \end{aligned}. \tag{2.35}$$

2.2.3 Randbedingungen

Die Randbedingungen ermöglichen es, Bedingungen für das Randverhalten zu setzen.

No Penetration Randbedingung

Die No Penetration Randbedingung sorgt dafür, dass die Geschwindigkeit an den Randzellen senkrecht auf den Normalen steht und somit das Fluid am Rand entlang fließt:

$$u \cdot n = 0. \tag{2.36}$$

Dies impliziert, dass:

- $u_{0,j,k} = u_{N_{max},j,k} = 0$ an Wänden mit fester x-Koordinate
- $v_{i,0,k} = v_{i,N_{max},k} = 0$ an Wänden mit fester y-Koordinate
- $w_{i,j,0} = w_{i,j,N_{max}} = 0$ an Wänden mit fester z-Koordinate

Neumann Randbedingung

$$\frac{\partial u}{\partial n} = \nabla u \cdot n = 0 \tag{2.37}$$

führt dies zu folgenden Bedingungen:

- $u_{0,j,k} - u_{1,j,k} = 0$, $u_{N_{max},j,k} - u_{N_{max}-1,j,k} = 0$ bei horizontalen Wänden
- $v_{i,j,0} - v_{i,j,1} = 0$, $v_{i,j,N_{max}} - v_{i,j,N_{max}-1} = 0$ bei horizontalen Wänden
- $w_{i,0,k} - w_{i,1,k} = 0$, $w_{i,N_{max},k} - w_{i,N_{max}-1,k} = 0$ bei vertikalen Wänden.

Diese Randbedingung setzt die Richtungsableitung entlang der Rand-Normalen gleich Null. In [1] und [2] verwendet Stam die Neumann Randbedingung für die Skalarfelder und sowohl die Neumann, als auch die No-Penetration Randbedingung für das Geschwindigkeitsfeld.

2.2.4 Transport von Skalaren

Führt man eine Substanz (Skalar) a in das Fluid ein, so wird diese durch das Geschwindigkeitsfeld beschwegt, während es sich durch Diffusion mit den Nachbarzellen austauscht. Beispiele für Skalare sind Dichte von Nebel, Rauch oder die Temperatur. Die zeitliche Veränderung kann durch folgende Gleichung definiert werden:

$$\frac{\partial a}{\partial t} = -(u \cdot \nabla)a + \kappa_a \nabla^2 + S_a \tag{2.38}$$

Hierbei ist κ_a die entsprechende Diffusionskonstante, S_a beschreibt die externe Zu/Abnahme des Skalars (bei der Temperatur beispielsweise eine Heizplatte oder eine Kühlung). Die Transportgleichung entspricht der Impulserhaltungsgleichung und kann somit auf die gleiche Art gelöst werden. Ein Projektionsschritt ist hier nicht nötig, da kein Druckterm vorhanden ist.

2.2.5 Thermaler Auftrieb

Ein Beispiel für externe Kräfte ist die thermale Auftriebskraft bei der Simulation von Gasen unter Berücksichtigung der Temperatur. Heisse Luft soll aufsteigen, kalte Luft fallen. Bezugnehmend auf [3] wird die thermale Auftriebkraft folgendermaßen definiert:

$$f_{buoyancy} = \beta(T - T_{amb}) * \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \tag{2.39}$$

β ist die Temperaturkonstante, T ist die Temperatur, T_{amb} ist die Umgebungstemperatur.

2.2.6 Vorticity Confinement

Aufgrund des vergleichsweise groben Gitters tritt zwangsläufig numerische Dissipation auf. Turbulente Strukturen werden hierdurch gedämpft. In [3] wird die Methode des Vorticity Confinement vorgestellt, welche versucht die gedämpften Turbulenzen wieder in das Fluid zurückzuführen.
Zunächst wird die Wirbelstärke berechnet. Diese ist definiert als die Rotation des Geschwindigkeitsfelds

$$\omega = \nabla \times u \tag{2.40}$$

Diskretisiert in der Zentraldifferenzenform:

$$\omega_{i,j,k} = \frac{1}{2h} \begin{pmatrix} w_{i,j+1,k} - w_{i,j-1,k} - v_{i,j,k+1} + v_{i,j,k-1} \\ u_{i,j,k+1} - u_{i,j,k-1} - w_{i+1,j,k} + w_{i-1,j,k} \\ v_{i+1,j,k} - v_{i-1,j,k} - u_{i,j+1,k} + u_{i,j-1,k} \end{pmatrix}. \tag{2.41}$$

Dann berechnet man den normalisierten Wirbelstärke-Ortsvektor, der von Bereichen mit kleiner Wirbelstärke zu Bereichen mit hohen Wirbelstärke zeigt.

$$N = \frac{\nabla|\omega|}{|\nabla|\omega||} \tag{2.42}$$

Diskretisiert in der Zentraldifferenzenform:

$$\nabla|\omega|_{i,j,k} = \frac{1}{2h} \begin{pmatrix} |\omega_{i+1,j,k}| - |\omega_{i-1,j,k}| \\ |\omega_{i,j+1,k}| - |\omega_{i,j-1,k}| \\ |\omega_{i,j,k+1}| - |\omega_{i,j,k-1}| \end{pmatrix} \tag{2.43}$$

Die Vorticity Confinement-Kraft berechnet sich nun als:

$$f_{vorticity} = \epsilon h(N \times \omega) \tag{2.44}$$

$\epsilon > 0$ steuert hierbei die Größe des Wirbelstärke-Erhaltungseffekts. Somit wird versucht die "verlorengegangene" Energie über die Rechte Seite der Impulserhaltungsgleichung wieder in das System zurückzuführen.

3 Entwurf und Implementierung

In diesem Kapitel wird die verwendete Programmiersprache und die Eigenschaften der Grafikkarten vorgestellt. Zudem wird die grundlegende Implementierung des Codes beschrieben, sowie die parallelisierten Erweiterungen erklärt.

3.1 Programmierumgebung

Aufgrund der rasanten Entwicklung in der 3D Grafik haben sich Grafikkarten (GPU[1]) zu hochparallelen Mehrkernprozessoren entwickelt, die man insbesondere für wissenschaftliche Simulationen gut nutzen kann. Während man früher die sogenannten Shader[2] nur für bestimmte Rendering-Effekte nutzte, können diese (Unified Shader) mittlerweile effizient programmiert werden. Jeder Unified Shader besitzt (siehe in Abbildung 3.1) eine eigene ALU (Arithmetic Logical Unit) und kann damit Rechenoperationen parallel ausführen.

Abbildung 3.1: Architekturvergleich CPU, GPU

Die Programmierumgebung CUDA (Compute Unified Device Architecture) wurde 2006

[1]Graphics Processing Unit
[2]Shader waren früher nur für feste Renderingeffekte verantwortlich - z.b Schattierung

von der Firma NVIDIA [3] entwickelt und erlaubt eine GPU-Programmierung durch die Programmiersprache CUDA C, eine Erweiterung der Sprache C. CUDA ist nur auf Grafikkarten der Firma NVIDIA lauffähig[4].

3.1.1 Hardware-Architektur

Die folgenden Ausführungen bezüglich der CUDA-Architektur stammen aus dem CUDA Programming Guide [11], der CUDA Architektur Overview und weiteren Dokumenten, die sich auf der NVIDIA-Homepage [5] und dem CUDA v4.0 Toolkit beiliegend zu finden sind. Die Modellbeschreibungen können anhand aktueller Datenblättern nachvollzogen werden [12].
Je nach Modell verfügt eine NVIDIA-Grafikkarte über mehrere Streaming Multiprozessoren (SM). Jede dieser Multiprozessoren hat 8-32 (je nach Modell) Kerne. Wird eine Funktion auf der Grafikkarte ausgeführt (auch Kernel[6] genannt), so führt jeder SM und jeder Kern das Programm aus. NVIDIA bezeichnet diese Architektur auch als "Single Instruction Multiple Threads"-Architektur. Bei der Programmausführung wird zudem eine Topologiehierarchie angegeben. Es wird zwischen Threads und Blöcken unterschieden. Ein Thread ist eine Instanz des Kernels, die auf einem Kern gerechnet werden kann. Ein Block ist definiert als Gruppierung von bis zu 1024 Threads, mehrere Blöcke können in einer ein- bis dreidimensionalen Topologie geordnet sein (Grid). Ein SM arbeitet eine Reihe von Blöcken nacheinander ab. Die Zuordnung von Blöcken auf die SM erfolgt jedoch automatisch. Je 32 Threads bilden einen Warp und werden als Gruppe auf einem SM ausgeführt. Bis zu 8 Blöcke bekommen "gleichzeitig" Rechenzeit von einem SM (sie bekommen wie bei einer CPU[7] Zeitscheiben zugewiesen).

Die folgenden Speicherarten stehen einem Kernel zur Verfügung: Das Zusammenspiel aller Speicherarten ist in Abbildung 3.2 dargestellt.

Register

Jedem Block stehen bis zu 32768 Register zur Verfügung, die auf die Threads aufgeteilt werden. Register sind die performantesten Speichereinheiten und werden implizit benutzt, sofern ihr Inhalt zur Compile-Zeit[8] bekannt ist[9].

[3] einer der größten Hersteller von Grafikkarten
[4] ein Emulationsmodus ist seit Version 4 nicht mehr vorhanden
[5] `http://www.nvidia.de/object/cuda_home_new_de.html`
[6] die Kernel sind nicht zu verwechseln mit mathematischen Kernelfunktionen
[7] Central Processing Unit - Hauptprozessor
[8] während der Compiler den Quellcode übersetzt und in ein ausführbares Programm umwandelt
[9] Beispiel: int a=2

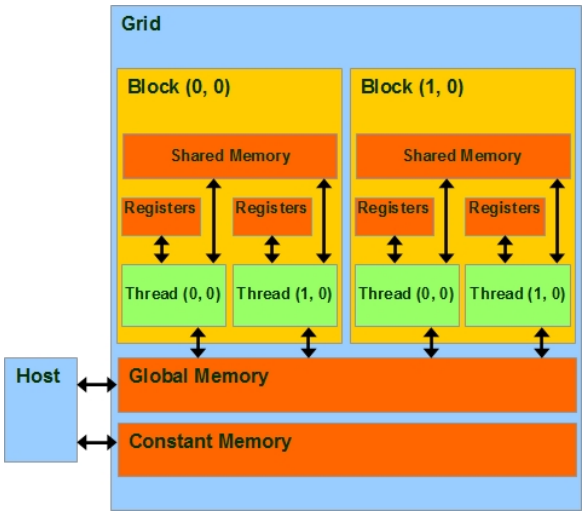


Abbildung 3.2: Cuda Memory Modell [13]

Shared Memory

Auf den Shared Memory können alle Threads eines Blocks zugreifen. Je nach Modell sind das bis zu 48KB (entspricht 12288 Gleitkommazahlen). Die Shared Memory-Performanz entspricht der Register-Performanz sofern keine Speicherkonflikte (gleichzeitiges Lesen oder Schreiben) auftreten.

Global Memory

Je nach Modell steht allen Threads aller Blöcke ein gemeinsamer Speicher von bis zu 4 Gigabyte zur Verfügung. Dieser ist allerdings wesentlich langsamer als die Register oder der Shared Memory (ca. 150x langsamer).

Local Memory

Jedem Thread steht ein lokaler Speicher zur Verfügung. Dieser ist allerdings kein echter eigener Speicher, sondern ist Teil des Global Memory und hat daher auch dieselbe schlechte Performanz.

Constant Memory

Es steht ein bis zu 64 Kilobyte großer Speicher zur Verfügung, der vor Ausführung beschrieben wird und während der Kernelausführung nur lesend, nicht schreibend zur Ver-

fügung steht. Dieser Speicher ist zwar ebenfalls so langsam wie der globale Speicher, allerdings besitzt er einen Cache[10]. Greift man also mehrfach auf einen Speicherbereich zu, so ist der Zugriff insgesamt schneller.

Texture Memory

Auf den Texturspeicher kann wie beim Constant Memory nur lesend zugegriffen werden. Hier können Texturdaten hinterlegt werden. Auch dieser Speicher ist Teil des Global Memory, allerdings auch mit Cache.

3.1.2 CUDA C

CUDA C ist eine Erweiterung der Programmiersprache C. Es werden neue Deklarationen hinzugefügt, damit der Compiler, der das Programm übersetzt, die neuen Funktionalitäten unterscheiden kann. Hierzu werden verschiedene Marker eingeführt.

- `__global__` Diese Funktion ist eine Kernelfunktion
- `__host__` Funktion kann von der CPU ausgeführt werden
- `__device__` Funktion kann nur von der GPU bzw. von einer Kernelfunktion ausgeführt werden
- `__shared__` Definiert eine Variable/Array als im Shared Memory liegend
- `__constant__` Definiert eine Variable/Array als im Constant Memory liegend

Ein Kernelfunktionsaufruf hat zudem eine andere Syntax als normale C-Funktionsaufrufe

```
C: function(parameter)
Kernel: function<<<nBlocks,nThreads>>>(parameter)
```

Die Variablen nBlocks,nThreads können entweder vom Datentyp *int*, *dim*2 oder *dim*3 sein (letztere sind eigene CUDA-Vektordatentypen). Somit kann die Block/Thread-Topologie für jeden Kernel definiert werden.
Da jeder Thread das gleiche Programm ausführt, kann anhand der Nummer des Threads das Programm verzweigt werden. Von CUDA selbst werden die Variablen threadIdx, blockIdx und blockDim (drei-dimensionale Vektoren) zur Verfügung gestellt.

[10] Pufferspeicher

Beispiel einer Kernelfunktion

```
__global__ void add_source(float * src,float * dst,float dt,int n) {
   int id = blockIdx.x*blockDim.x+threadIdx.x;
   if(id<(n+2)*(n+2)*(n+2)) {
       dst[id]+=src[id];
   }
}
```

Die CPU und der normale Arbeitsspeicher werden als Host bezeichnet, die Grafikkarte mit ihrem eigenen Speicher als Device. Der Host kann nicht auf den Device-Speicher direkt zugreifen. Um Daten von Host->Device zu schicken, müssen diese Daten durch spezielle Befehle in den Speicher der Grafikkarte kopiert werden (cudaMemCpy). Auch kann die Kernelfunktion direkt nichts zurückgeben, die Kernelfunktionen bekommen alle die Deklaration `__global__ void`, somit liefern sie keinen Rückgabewert. Um das Ergebnis der Kernelfunktion auf dem Host zu betrachten, müssen die Daten von der Grafikkarte wieder zurückkopiert werden. Dies geschieht auch wieder mit dem cudaMemCpy-Befehl.

3.2 Ray Casting

Der grundlegende Algorithmus für das Ray Casting lautet:

```
Bekannt: Kameraposition, Pixelpositionen, Anzahl der maximalen Samples S
Für jede Pixelposition p der Fläche
    Strahlursprung r=Kameraposition
    Strahlrichtung d=normiert(p-Kameraposition)
    getroffen=schneide-strahl-mit-box(r,d,Box,Schnitt1,Schnitt2)
    wenn(getroffen)
        Startwert=Schnitt1
        Führe S mal aus:
            Datenwert=Sample(Schnitt1)
            RGBA-Farbwert=Transferfunktion(Datenwert)
            RGBA-Farbwert=Shading(RGBA-Farbwert)
            Kommulierter Farbwert=Composition(RGBA-Farbwert,Kommulierter Farbwert)
        SetzeFarbwert(P,Kommulierter Farbwert)
    SetzteFarbwert(P,Hintergrundfarbe)
```

3.3 Simulation

Der Algorithmus ist an [2] angelehnt, erweitert um Auftrieb (Buoyancy) und Vorticity Confinement [3] . Die Simulation wird auf einem Gitter der Größe $(N+2)^3$ ausgeführt, um die Ränder gesondert zu behandeln. Alle Daten werden als ein-dimensionale Felder gespeichert. Die einzelnen Geschwindigkeitskomponenten werden getrennt und als die Felder X,Y,Z gespeichert. Die Werte des zu transportierenden Skalars wird im Feld D gespeichert. Für alle Felder existieren Hilfsfelder (X_0, Y_0, Z_0, D_0), um Schreibkonflikten vorzubeugen. Man operiert also immer auf einem Lese- und einem Schreibgitter. Der Abstand h zwischen den Zellen wird mit $h = \frac{1}{N}$ definiert. Der Zeitschritt wird mit dt bezeichnet.

Algorithmus

Im Folgenden wird die grundlegende Struktur des Simulationscodes in Pseudocode dargestellt.

```
Fluid() {
  velocitystep(x,y,z,x0,y0,z0,d,dt,N) // Geschwindigkeitsfeldänderung
  scalarstep(x,y,z,d,d0,dt,N) // Skalarfeldänderung
}
```

Die Simulation besteht aus dem Aktualisieren des Geschwindigkeitsfeldes und der darauf folgenden Aktualisierung des Skalarfeldes. Die Funktion *velocitystep* bekommt die aktuellen Geschwindigkeitskomponentenfelder x, y, z, sowie die Felder des letzten Zeitschritts $x0, y0, z0$ neben Informationen zum Zeitschritt und die Größe der Felder übergeben. Die Funktion *scalarstep* bekommt das Skalarfeld vom letzten Zeitschritt $d0$ und aktuellen Zeitschritt d sowie das aktuelle Geschwindigkeitsfeld, den Zeitschritt dt und die Größe des Skalarfeldes N übergeben.
Betrachten wir nun den Ablauf der Funktionen im einzelnen.

```
velocitystep(x,y,z,x0,y0,z0,d,dt,N) {
    add_bouyancy(y,d,dt,N) // thermaler Autrieb
    add_vorticity(x0,y0,z0,x,y,z,N) // Wirbelstärkeerhaltung
    Swap(x0,x) Swap(y0,y) Swap(z0,z) // Tauschen der Gitter
    diffuse(1,x0,x,visc,dt,iter,N) // Diffusion für x-Werte
    diffuse(2,y0,y,visc,dt,iter,N) // Diffusion für y-Werte
    diffuse(3,z0,z,visc,dt,iter,N) // Diffusion für z-Werte
    Swap(x0,x) Swap(y0,y) Swap(z0,z) // Tauschen der Gitter
    project(x0,y0,z0,x,y,z,N) // Projektionsschritt
    Swap(x0,x) Swap(y0,y) Swap(z0,z) // Tauschen der Gitter
    advect(1,x0,x,dt,N) // Advektion der x-Werte
```

```
    advect(2,y0,y,dt,N) // Advektion der y-Werte
    advect(3,z0,z,dt,N) // Advektion der z-Werte
    Swap(x0,x) Swap(y0,y) Swap(z0,z) // Tauschen der Gitter
    project(x0,y0,z0,x,y,z,N) // Projektionsschritt
}
```

Zunächst wird der thermale Auftrieb (siehe Abschnitt 2.2.7) durch die Funktion *add_buoyancy* der y-Komponente des Geschwindigkeitsfeldes hinzugefügt. Dann folgt durch die Funktion *add_vorticity* eine Wirbelstärkeerhaltung, die bereits im Abschnitt 2.2.8 beschrieben wurde. Durch die Funktion *Swap* folgt das Vertauschen der beiden Geschwindigkeitsfelder. Dies geschieht, um zu erreichen, dass aus den Feldern $x0, y0, z0$ nur gelesen wird und die neuen Werte in die Felder x, y, z geschrieben werden ohne überlagernde Lese-, bzw. Schreibkonflikte. Für jede Geschwindigkeitskomponente wird nun ein Diffusionsschritt durchgeführt. Der erste Parameter dient der Identifikation der Komponente für die in diesem Schritt eingesetzten Randbedingungen. Nach erfolgreichem Schreiben werden wieder die Felder durch die *Swap*-Funktion getauscht. Da der Advektionsschritt genauer arbeitet, wenn das Fluid vorher divergenzfrei ist, folgt nun ein erster Projektionsschritt durch die Funktion *project* mit darauffolgendem Feldtausch. Nun erfolgt für jede Geschwindigkeitskomponente der Advektionsschritt durch die Funktion *advect*. Der erste Parameter ist wie beim Diffusionsschritt für die Identifikation der Komponente für die Randbedingungen zuständig. Nach einem erneuten Feldtausch erfolgt der finale Projektionsschritt womit die Aktualisierung des Geschwindigkeitsfeldes abgeschlossen ist.

```
scalarstep(x,y,z,d,d0,dt,N) {
    add_source(src,dst,dt,N) // externe Skalarwertänderung
    Swap(d0,d) // Tauschen der Gitter
    diffuse(0,d0,d,diff,dt,iter,N) //Diffusion des Skalar
    Swap(d0,d) // Tauschen der Gitter
    advect(0,x0,x,dt,N) // Advektion des Skalar
}
```

Die Aktualisierung des Skalarfeldes erfolgt ähnlich, allerdings wird hier nur ein Feld im Gegensatz zu den drei Geschwindigkeitskomponentenfeldern aktualisiert. Auch findet hier kein Projektionsschritt statt. Somit beginnt der *scalarstep* mit dem benutzerdefinierten Ändern des Skalarfelds, beispielsweise der Kühlung der Temperatur in bestimmten Feldzellen. Es folgt wieder ein Tauschen der Felder. Daraufhin findet ein Diffusionsschritt für das Skalarfeld statt. Nach einem weiteren Feldtausch endet die Funktion mit dem Advektionsschritt des Skalarfelds.

3.3.1 Externe Kräfte

Sowohl im Geschwindigkeits-, als auch im Skalar-Schritt können externe Kräfte bzw. Skalarfeldeinflüsse berücksichtigt werden. Hierbei werden die externen Kräfte auf das bestehende Gitter hinzuaddiert.

```
add_source(src,dst,dt,N) {
for(i=0;i<N*N*N;i++)
      dst[i]=dst[i]+src[i]*dt;
}
```

Die Funktion *add_source* bekommt ein Quell und ein Zielfeld übergeben und addiert diese komponentenweise (in Abhängigkeit vom Zeitschritt) auf.

3.3.2 Diffusion

Der Diffusionsschritt beschreibt die Ausbreitung des Skalars oder der Geschwindigkeiten auf die Nachbarzellen.

```
diffusestep(b,x0,x,diff,dt,iter,N) {
  a=dt*diff*N*N;
  for(iteration=0;iteration<iter;iteration++) {
    lin_solve(b,x,x0,a,1+6*a,N); // Gleichungssystem lösen
    set_bnd(b,x,N); // Randbedingungen setzen
  }
}
```

Wie in Abschnitt 2.2.2. beschrieben wird im Diffusionsschrit ein dünnbesetztes lineares Gleichungssystem gelöst. Die Variable a entspricht dem Koeffizienten $\frac{\nu \Delta t}{h^2}$ mit $h = \frac{1}{N}$. Da das lineare Gleichungssystem mit einem iterativen Löser gelöst wird, kann mit der Variable *iter* die Anzahl der Iterationen festgelegt werden. Die Funktion *set_bnd* setzt die Randbedingungen.

3.3.3 Advektion

```
advect(b,x0,x,dt,N){
  for(i=1;i<=N;i++)
    for(j=1;j<=N;j++)
      for(k=1;k<=N;k++)
        ...
        // Gehe einen Zeitschritt zurück
        prex=i-dt0*velocX[I(i,j,k)];
```

```
        prey=j-dt0*velocY[I(i,j,k)];
        prez=k-dt0*velocZ[I(i,j,k)];
        // Am Rand anpassen - 1. Koordinate
        if(prex<0.5f) prex=0.5f;
        if(prex>N+0.5f) prex=N+0.5f;
        // Eckpunktkoordinaten der Zelle bestimmen
        i0=(int)prex;
        i1=i0+1;
        // Am Rand anpassen - 2. Koordinate
        if(prey<0.5f) prey=0.5f;
        if(prey>N+0.5f) prey=N+0.5f;
        // Eckpunktkoordinaten der Zelle bestimmen
        j0=(int)prey;
        j1=j0+1;
        // Am Rand anpassen - 3. Koordinate
        if(prez<0.5f) prez=0.5f;
        if(prez>N+0.5f) prez=N+0.5f;
        // Eckpunktkoordinaten der Zelle bestimmen
        k0=(int)prez;
        k1=k0+1;
        // normierte Koordinaten in der Zelle
        sx1=prex-i0;
        sx0=1-sx1;
        sy1=prey-j0;
        sy0=1-sy1;
        sz1=prez-k0;
        sz0=1-sz1;
        // Trilineare Interpolation
        t1=linear_interp(sy0,x0[I(i0,j0,k0)],x0[I(i0,j1,k0)]);
        t2=linear_interp(sy0,x0[I(i1,j0,k0)],x0[I(i1,j1,k0)]);
        t3=linear_interp(sx0,t1,t2);
        t1=linear_interp(sy0,x0[I(i0,j0,k1)],x0[I(i0,j1,k1)]);
        t2=linear_interp(sy0,x0[I(i1,j0,k1)],x0[I(i1,j1,k1)]);
        t4=linear_interp(sx0,t1,t2);
        x[I(i,j,k)]=linear_interp(sz0,t3,t4); // Werte setzen
 set_bnd(b,x,N);
}
```

Der Advektionsschritt wird nun für alle inneren Zellen durchgeführt. Die Randzellen werden separat gesetzt. Da das Feld die Größe $(N+2)^3$ besitzt, beginnt die Indizierung für

die inneren Elemente in den Schleifen bei 1 und endet bei N. Es folgt das Backtracing, indem die Geschwindigkeit um einen Zeitschritt rückwärts in der Zeit verfolgt wird (für jede Komponente). Landet man hierbei außerhalb des gültigen Gitters, so wird die Position entsprechend auf den Rand projeziert. Dies wird auch Clamping genannt. Für jede Koordinate wird nun ermittelt welche Eckpunkte die Zelle hat, in der sich der zurückverfolgte Punkt befindet. Die Abstände zu den Eckpunkten werden ausgerechnet und eine Interpolation angewandt. Zuletzt werden die Randbedingungen gesetzt.

3.3.4 Projektion

```
project(x0,y0,z0,x,y,z,N)
  // Für jede Zelle (ohne Rand)
  for(i=1;i<=N;i++)
    for(j=1;j<=N;j++)
      for(k=1;k<=N;k++)
        Nr=-1.0f/(2.0f*N);
        // Divergenz ausrechnen
        div[I(i,j,k)]=(x0[I(i+1,j,k)]-x0[I(i-1,j,k)]+y0[I(i,j+1,k)]-y0[I(i,j-1,k)]+z0[I(i,
        p[I(i,j,k)]=0; // Druckfeld initialisieren mit 0
  // Randbedingungen setzen
  set_bnd(0,div,N);
  set_bnd(0,p,N);
  for(iteration=0;iteration<iter;iteration++)
    lin_solve(0,p,div,1.0f,6.0f,N); // Gleichungssystem lösen
    set_bnd(0,p,N); // Randbedingungen setzen
  float Nr=(1.0f/2.0f)*N;
  // Für jede Zelle (ohne Rand)
  for(i=1;i<N+1;i++)
    for(j=1;j<N+1;j++)
      for(k=1;k<N+1;k++)
        // Druckfeld subtrahieren
        x[I(i,j,k)]=x0[I(i,j,k)]-Nr*((p[I(i+1,j,k)]-p[I(i-1,j,k)]));
        y[I(i,j,k)]=y0[I(i,j,k)]-Nr*((p[I(i,j+1,k)]-p[I(i,j-1,k)]));
        z[I(i,j,k)]=z0[I(i,j,k)]-Nr*((p[I(i,j,k+1)]-p[I(i,j,k-1)]));
```

Im Projektionsschritt wird zunächst (siehe 3.3.5) für jede innere Zelle die Divergenz ausgerechnet und das Druckfeld mit Null initialisiert. Dann werden die Randzellen durch die Randbedingungen (*set_bnd*) gesetzt. Dann wird das lineare Gleichungssystem gelöst und das Druckfeld vom Geschwindigkeitsfeld subtrahiert.

3.3.5 Linearer Gleichungslöser

Da sowohl im Diffusionsschritt, als auch im Projektionsschritt ein dünnbesetztes lineares Gleichungssystem zu lösen ist, wird ein entsprechender Löser benötigt. Um Speicher zu sparen, wählt man hier einen iterativen Löser, beispielsweise das Gauss-Seidel-, das Jacobi- oder das CG-Verfahren[11]. Aufgrund des sparsamen Speicherbedarfs wird in der Implementierung das Gauss-Seidel-Verfahren verwendet. Folgende Gleichungssysteme müssen gelöst werden:

Diffusion

$$X_{i,j,k}^{n+1} - \frac{kdt}{h^2}\left(X_{i-1,j,k}^{n+1} + X_{i,j-1,k}^{n+1} + X_{i,j,k-1}^{n+1} + X_{i+1,j,k}^{n+1} + X_{i,j+1,k}^{n+1} + X_{i-1,j,k+1}^{n+1} - 6X_{i,j,k}^{n+1}\right) = X_{i,j,k}^{n} \tag{3.1}$$

Projektion

$$\begin{aligned} P_{i-1,j,k} + P_{i,j-1,k} + P_{i,j,k-1} + P_{i+1,j,k} + P_{i,j+1,k} + P_{i,j,k+1} - 6P_{i,j,k} = \\ \frac{h}{2}(U_{i+1,j,k} - U_{i-1,j,k} + V_{i,j+1,k} - V_{i,j-1,k} + W_{i,j,k+1} - W_{i,j,k-1}) \end{aligned} \tag{3.2}$$

Gauss-Seidel-Verfahren für Diffusion und Projektion

Das Gauss Seidel-Verfahren entspricht nun [4]:

$$X_{i,j,k}^{n+1} = \frac{(X_{i,j,k}^{n} + a(X_{i-1,j,k}^{n+1} + X_{i,j-1,k}^{n+1} + X_{i,j,k-1}^{n+1} + X_{i+1,j,k}^{n+1} + X_{i,j+1,k}^{n+1} + X_{i,j,k+1}^{n+1}))}{b} \tag{3.3}$$

$a = \frac{kdt}{h^3}$ (Diffusion)
$a = 1$ (Projektion)
$b = 1 + \frac{kdt}{h^3}$ (Diffusion)
$b = 6$ (Projektion)
Hinweis: Im Projektionsschritt ist $X_{i,j,k}^{n} = -\frac{h}{2}\nabla \cdot U$ mit $U = (u, v, w)_{i,j,k}$

```
lin_solve(b,x,x0,a,c,N) {
  cR=1.0/c;
  // Für jede Zelle (ohne Rand)
  for(i=1;i<=N;i++) {
    for(j=1;j<=N;j++) {
      for(k=1;k<=N;k++)
        // Gauss Seidel
```

[11] Conjugate Gradient

```
        x[I(i,j,k)]=cR*(x0[I(i,j,k)]+a*(x[I(i+1,j,k)]+x[I(i-1,j,k)]+x[I(i,j+1,k)]+
                                        x[I(i,j-1,k)]+x[I(i,j,k+1)]+x[I(i,j,k-1)]));
}
```

3.4 Parallelisierbarkeit

Die im vorherigen Abschnitt angegebenen Algorithmen werden nun im folgenden Abschnitt parallelisiert und als CUDA-Quellcode besprochen. An einigen Stellen wurden auch schon bereits diverse Optimierungen durchgeführt, um Operationen zu sparen bzw. durch günstigere Bitoperationen zu ersetzen.

3.4.1 Ray Casting

Aufgrund der Unabhängigkeit jedes konstruierten Strahls kann jeder Thread einen Strahl bearbeiten. Einzig das gleichzeitige Lesen der Volumendaten kann den Performanzvorteil durch die Parallelisierung dämpfen.

```
__global__ void calc_pixel(int* pixels,float* data,int N)
{
  int idx=blockIdx.x*blockDim.x+threadIdx.x; // Nummer des Threads
  int x=idx & (1024 -1); // entspricht idx mod 1024
  int y=idx >> 10; // entspricht idx/1024
  renderthis(pixels,data,x,y,idx,N); // Ray Casting
}
```

Die Funktion *calc_pixel* bekommt eine Referenz auf das zu schreibende 2D-Bild, eine Referenz auf das Volumendatenfeld aus der Simulation und die Größe des Feldes N. Da nun für jeden Pixel der virtuellen Bildfläche ein Strahl konstruiert werden muss, kann jeder Thread die Farbe eines dieser Strahlen berechnen. Die Variablen x und y geben die lokalen Koordinaten innerhalb der Fläche an. In diesem Beispiel wird eine Auflösung von $1024x1024$-Bildpunkten verwendet. Die Funktion *renderthis* führt das Ray Casting durch und bestimmt die Farbe des durch die Koordinaten x, y definierten Bildpunktes der virtuellen Bildfläche.. Da die Modulo-Operation und die Division aufwendig sind wurden diese durch schnellere Bitoperationen ausgetauscht.

```
__host__ __device__ void renderthis(int* pixels,float *d,float x, float y,
                                    int idx, int N)
{
  // Kameraposition setzen
  float3 eye=make_float3(-1024,512,-1024.0);
  // Größe der Volumenumgebungsbox
```

```
float boxwidth=1024.0f;
float rbwidth=1.0f/1024.0f;
// Linke untere Ecke der Box
float3 o1=make_float3(0,0,20);
// Rechte obere Ecke der Box
float3 o2=make_float3(boxwidth,boxwidth,boxwidth)+o1;
// Punkt auf der virtuellen Bildfläche
float3 w=make_float3(x-512,y,512-x);
// Strahlkonstruktion von Kamera zum Bildpunkt
ray_t r;
r.s=eye;
r.d=w-eye;
r.d=normalize(r.d);
// Schnittpunkte initialisieren
float tnear=0.0f,tfar=0.0f;
// Schnitt Ray-Box
bool hit=intersect(r,o1,o2,tnear,tfar);
// Schrittweite als Faktor zur Strahlrichtung
float tstep=boxwidth/N;
// Schrittweite
int steps=floor((tfar-tnear)/tstep);
// Farben und Opazität initialisieren
float3 c={0,0,0};
float c_s_a=0.0f;
// Volumenwert initialisieren
float s=0.0f;
// Test auf Schnitt
if(hit==true) {
  if(tnear<0.0f) tnear=0.0f;
  // an erstem Schnitt starten
  float t=tnear;
  // Punkt ausrechnen
  float3 pos=r.s+t*r.d;
  // Schrittweite als Vektor
  float3 step=tstep*r.d;
  float3 col;
  // Transfer+Kompositionsschritt
  for(int i=0;i<steps;i++) {
    // Wert bestimmen
    s=sample3D(d,N,(pos-o1)*rbwidth);
```

```
      // Transferfunktion
      transfer(s,col,c_s_a);
      // Farbe nach Average-Komposition aufsummieren
      c+=(col*c_s_a);
      t+=tstep;
      // Frühzeitiger Abbruch
      if(t>tfar) { steps=max(1,i); break; }
      pos+=step;
    }
    // Mittelwert bilden
    c=(255.0f*c)/(steps);
  }
  // Farbe setzen
  unsigned char dr, dg, db;
  dr = (unsigned char)(c.x);
  dg = (unsigned char)(c.y);
  db = (unsigned char)(c.z);
  // RGB-Wert -> Integer Transformation
  pixels[idx] = dr|(dg<<8)|(db<<16);
}
```

Im eigentlichen Ray Casting Schritt wird zunächst die Kameraposition gesetzt. $float3$ bezeichnet hierbei einen CUDA-eigenen Vektordatentyp. Die Größe der Volumenumgebungsbox wird gesetzt. *rbwidth* dient der Vorab-Division, um später in der Schleife nur schnellere Multiplikationen zu verwenden. Die Eckpunkte der Box werden gesetzt. Der Punkt auf der virtuellen Bildfläche wird errechnet und der Strahl r wird konstruiert. Der Ursprung ($r.s$) soll an der Kameraposition liegen, die Richtung ($r.d$) von Kameraposition zum Punkt der virtuellen Bildfläche zeigen. Dann folgt der Schnitttest zwischen dem Strahl und der Box. *tnear* und $tfar$ geben durch $r_s + tnear/tfar \cdot r_d$ den Schnittpunkt an. Daher existieren auch zwei Schrittweiten. *step* bezeichnet die Schrittweite als Vektor, *steps* die Anzahl der Schritte innerhalb des Volumens. Tritt nun ein Schnitt auf, so startet man an dem ersten Schnittpunkt, rechnet über die Funktion $sample3D$ den Wert des jeweiligen Volumendatenpunkts aus, bestimmt über die $transfer$-Funktion die Farbe und kommuliert die Farben nach einem Kompositionsverfahren (Average). Die Farben werden aufsummiert und am Ende gemittelt. Zum Schluss wird die Farbe ausgehend von den RGB-Daten in einen Integerwert umgewandelt, da dies für die Grafikausgabe speziell benötigt wird.

3.4.2 externe Kräfte

Die Daten eines Gitters werden einem anderen hinzuaddiert. Es existieren keine Abhängigkeiten.

```
__global__ void add_source(float * src,float * dst,float dt,int n) {
  int id=blockIdx.x*blockDim.x+threadIdx.x;
  if(id<(n+2)*(n+2)*(n+2)) {
    dst[id]=dst[id]+src[id]*dt;
  }
}
```

Da jeder Thread eine eindeutige Identifikation hat können die Daten komponentenweise hinzuaddier werden. Jeder Thread übernimmt dabei genau einen Index und somit eine Operation.

3.4.3 Advektion

Der folgende Auszug stammt aus der Datei *advect.h* aus der Funktion *advectstep*.

```
...
int numBlocks=N*N;
int numThreads=N;
advect<<<numBlocks,numThreads>>>(b, x0, x,vx, vy, vz, dt, N);
...
```

Es werden folglich N^2 Blöcke erzeugt mit je N Threads.

```
__global__ void advect(float *x0,float *x,float *velocX, float *velocY,
                       float *velocZ, float dt, int N)
{
  int gid=blockIdx.x*blockDim.x+threadIdx.x;
  int k=threadIdx.x+1;
  int j=(blockIdx.x/N)+1;
  int i=(blockIdx.x%N)+1;
  if(gid<(N*N*N)) {
  ...bisheriger Advektionscode
  }
}
```

Statt nun drei Schleifen zu verwenden wird zunächst die Identifikationsnummer des Threads festgestellt. Durch die Konfiguration des Kernels liegt $threadIdx.x$ zwischen 0 und N und

blockIdx.x zwischen 0 und $N \cdot N$. Die Indizes i, j, k ergeben sich wie oben angegeben direkt aus den genannten Variablen. Es wird hierbei jeweils Eins aufaddiert, um nur die Inneren Zellen (Größe der Felder $(N+2)^3$) zu betrachten und um ansonsten auftretende If-Abfragen zu reduzieren.

3.4.4 Linearer Gleichungslöser

```
__global__ void lin_solve(int b, float *x,float *x0,float a,float c,int N,int N1,int N2) {
  int i=threadIdx.x;
  int j=blockIdx.x;
  int ijk=0;
  if(i!=0 && i!=N1 && j!=0 && j!=N1) { // Nicht am Rand
    float tmp=x[I(i,j,0,N2)];
    for(int k=1;k<N1;k++) {
      ijk=I(i,j,k,N2);
      if(((i+j) & 1) == 0) { // entspricht (i+j)%==0
        tmp=(x0[ijk]+a*(x[I(i-1,j,k,N2)]+x[I(i+1,j,k,N2)]+x[I(i,j-1,k,N2)]+
                                        x[I(i,j+1,k,N2)]+x[I(i,j,k+1,N2)]+tmp))*c;
        x[ijk]=tmp;
      }
      __syncthreads(); // Thread synchronisieren
      if(((i+j) & 1) == 1) { // entspricht (i+j)%2==1
        tmp=(x0[ijk]+a*(x[I(i-1,j,k,N2)]+x[I(i+1,j,k,N2)]+x[I(i,j-1,k,N2)]+
                                        x[I(i,j+1,k,N2)]+x[I(i,j,k+1,N2)]+tmp))*c;
        x[ijk]=tmp;
      }
    }
  }
}
```

Die parallele GPU-Variante des Red-Black-Gauss-Seidel Kernels basiert auf [4]. Die Indizes i, j werden wie vorher im Advektionsschritt durch die Variablen *threadIdx.x* und *blockIdx.x* bestimmt. Jeder Kernel bearbeitet eine komplette Zeile in z-Richtung, daher die innere Schleife. Da auf den Index $I(i, j, k, N2)$ mehrfach zugegriffen wird, wird dieser in der Variable *ijk* zwischengespeichert. Auch werden direkt die Variablen N, $N+1$ und $N+2$ als Parameter N, $N1$, $N2$ übergeben um Additionen zu sparen. Außerdem wird statt durch c zu dividieren mit c multipliziert ($\frac{1}{c}$ wird bereits als Parameter c übergeben), um eine Division zu vermeiden. Im Gauss-Seidel-Kernel werden die 6 Nachbarzellen benötigt, sowie den Wert an der aktuellen Position im vorherigen Zeitschritt ($x0$). Somit würde man 7 Leseoperationen aus dem Global Memory und eine Schreiboperation in den Global Me-

mory benötigen. Da die Schleife über alle z-Komponenten geht und jedesmal $I(i,j,k-1)$ benötigt wird, kann eine Leseoperation gespart werden, indem das letzte Ergebnis (tmp) zwischengespeichert wird. Außerdem wurden die Modulo-Operationen durch Bitoperationen ersetzt. An dieser Stelle sei gesagt, dass das Gauss-Seidel-Verfahren offenbar robust ist gegen Schreib bzw. Lesekonflikte. Da bei CUDA die Reihenfolge der Abarbeitung der Blöcke nicht vorhergesagt werden kann, liegt beim Zugriff auf das Feld x entweder schon bearbeitete Werte vor oder unbearbeitete. Dies wird weder in [4] erwähnt, noch wird dort überhaupt die Konvergenz des Verfahrens bewiesen. Während der Erstellung dieser Arbeit wurde jedoch zumindest getestet, dass die verschiedenen Varianten bei entsprechender Iterationszahl alle gegen dieselbe Lösung konvergieren. Über die Konvergenzgeschwindigkeit kann jedoch im Rahmen dieser Arbeit keine Aussage getroffen werden.

3.4.5 Diffusion

```
void diffusestep(int b,float *x0,float *x,float diff,float dt,int iter,int N) {
  float a=dt*diff*N*N;
  float c=1.0f/(1+6*a);
  int N2=N+2;
  int numBlocks=N2*N2;
  for(int iteration=0;iteration<iter;iteration++) {
    lin_solve<<<numBlocks,N2>>>(b,x,x0,a,c,N);
    set_bnd<<<N2,N2>>>(b,x,N);
    set_bnd2<<<1,N>>>(b,x,N);
  }
}
```

Im Diffusionsschritt wird nur die GPU-Variante des Gleichungslösers und die der Randbedingungen aufgerufen.

3.4.6 Randbedingungen

```
__global__ void set_bnd(int b,float *x,int N) {
  int gid=threadIdx.x+blockIdx.x*blockDim.x;
  int j=threadIdx.x;
  int i=blockIdx.x;
  int N2=N+2;
  int N1=N+1;
  // Flächen
  if(gid<(N2*N2) && i!=0 && j!=0 && i!=N1 && j!=N1) {
    x[I(0,i,j,N2)]=b==1?0.0f:x[I(1,i,j,N2)];
```

```
    x[I(N1,i,j,N2)]=b==1?0.0f:x[I(N,i,j,N2)];
    x[I(i,0,j,N2)]=b==2?0.0f:x[I(i,1,j,N2)];
    x[I(i,N1,j,N2)]=b==2?0.0f:x[I(i,N,j,N2)];
    x[I(i,j,0,N2)]=b==3?0.0f:x[I(i,j,1,N2)];
    x[I(i,j,N1,N2)]=b==3?0.0f:x[I(i,j,N,N2)];
  }
}
__global__ void set_bnd2(int b,float *x,int N) {
  int i=threadIdx.x+1;
  int N2=N+2;
  int N1=N+1;
  // Kanten
  x[I(0,0,i,N2)]=(b==1||b==2)?0.0f:0.5f*(x[I(0,1,i,N2)]+x[I(1,0,i,N2)]);
  x[I(0,i,0,N2)]=(b==1||b==3)?0.0f:0.5f*(x[I(0,i,1,N2)]+x[I(1,i,0,N2)]);
  x[I(i,0,0,N2)]=(b==2||b==3)?0.0f:0.5f*(x[I(i,0,1,N2)]+x[I(i,1,0,N2)]);
  x[I(0,N1,i,N2)]=(b==1||b==2)?0.0f:0.5f*(x[I(0,N,i,N2)]+x[I(1,N1,i,N2)]);
  x[I(0,i,N1,N2)]=(b==1||b==3)?0.0f:0.5f*(x[I(0,i,N,N2)]+x[I(1,i,N1,N2)]);
  x[I(i,0,N1,N2)]=(b==2||b==3)?0.0f:0.5f*(x[I(i,0,N,N2)]+x[I(i,1,N1,N2)]);
  x[I(N1,N1,i,N2)]=(b==1||b==2)?0.0f:0.5f*(x[I(N1,N,i,N2)]+x[I(N,N1,i,N2)]);
  x[I(N1,i,N1,N2)]=(b==1||b==3)?0.0f:0.5f*(x[I(N1,i,N,N2)]+x[I(N,i,N1,N2)]);
  x[I(i,N1,N1,N2)]=(b==2||b==3)?0.0f:0.5f*(x[I(i,N1,N,N2)]+x[I(i,N,N1,N2)]);
  x[I(N1,0,i,N2)]=(b==1||b==2)?0.0f:0.5f*(x[I(N1,N,i,N2)]+x[I(N,N1,i,N2)]);
  x[I(N1,i,0,N2)]=(b==1||b==3)?0.0f:0.5f*(x[I(N1,i,N,N2)]+x[I(N,i,N1,N2)]);
  x[I(i,N1,0,N2)]=(b==2||b==3)?0.0f:0.5f*(x[I(i,N1,N,N2)]+x[I(i,N,N1,N2)]);
  __syncthreads();
  // Eckpunkte
  if(b==0) {
    switch(threadIdx.x) {
      case 0:x[I(0,0,0,N2)] = (x[I(1,0,0,N2)]+x[I(0,1,0,N2)]+
                                              x[I(0,0,1,N2)])*0.33f;break;
      case 1:x[I(0,N1,0,N2)] = (x[I(1,N1,0,N2)]+x[I(0,N,0,N2)]+
                                                x[I(0,N1,1,N2)])*0.33f;break;
      case 2:x[I(N1,0,0,N2)] = (x[I(N,0,0,N2)]+x[I(N1,1,0,N2)]+
                                              x[I(N1,0,1,N2)])*0.33f;break;
      case 3:x[I(N1,N1,0,N2)] = (x[I(N,N1,0,N2)]+x[I(N1,N,0,N2)]+
                                                 x[I(N1,N1,1,N2)])*0.33f;break;
      case 4:x[I(0,0,N1,N2)] = (x[I(1,0,N1,N2)]+x[I(0,1,N1,N2)]+
                                              x[I(0,0,N,N2)])*0.33f;break;
      case 5:x[I(0,N1,N1,N2)] = (x[I(1,N1,N1,N2)]+x[I(0,N,N1,N2)]+
                                                x[I(0,N1,N,N2)])*0.33f;break;
```

```
      case 6:x[I(N1,0,N1,N2)] = (x[I(N,0,N1,N2)]+x[I(N1,1,N1,N2)]+
                                                 x[I(N1,0,N,N2)])*0.33f;break;
      case 7:x[I(N1,N1,N1,N2)] = (x[I(N,N1,N1,N2)]+x[I(N1,N,N1,N2)]+
                                                   x[I(N1,N1,N,N2)])*0.33f;break;
    }
  } else {
    switch(threadIdx.x) {
      case 0:x[I(0,0,0,N2)]=0.0f;break;
      case 1:x[I(0,N1,0,N2)]=0.0f;break;
      case 2:x[I(N1,0,0,N2)]=0.0f;break;
      case 3:x[I(N1,N1,0,N2)]=0.0f;break;
      case 4:x[I(0,0,N1,N2)]=0.0f;break;
      case 5:x[I(0,N1,N1,N2)]=0.0f;break;
      case 6:x[I(N1,0,N1,N2)]=0.0f;break;
      case 7:x[I(N1,N1,N1,N2)]=0.0f;break;
    }
  }
}
```

Das Setzen der Randbedingungen erfolgt über zwei Kernelfunktionen, dies ist nötig, da das Setzen der Randflächen abgeschlossen sein muss, bevor das Setzen der Randkanten und Randeckpunkte erfolgen kann. Die Indizierung erfolgt bei beiden auch nicht über die Schleifen sondern über die Thread-Id bzw. Block-Id.
Exemplarisch sei hier gesagt, dass
`x[I(0,i,j,N2)]=b==1?0.0f:x[I(1,i,j,N2)];`
eine schnellere Variante des gleichbedeutenden Codes
`if(b==1) { x[I(0,i,j,N2)]=0.0f; } else { x[I(0,i,j,N2)]=x[I(1,i,j,N2)] }`
darstellt. Die Randzellen an den Eckpunkten sollen bei Skalarfeldern ($b == 0$) als Mittelwert der drei Nachbarzellen gebildet werden (Neumann-Randbedingung), bei Geschwindigkeitsfeldern haben die Eckpunkte die Geschwindigkeit Null (No Penetration-Randbedingung). Die Zellen der Randkanten sollen ebenfalls als Mittelwert zweier Nachbarzellen gebildet werden bzw. Null gesetzt werden.

3.4.7 Auftriebskraft

```
void buoyancystep(float *x,float *d,float dt,float N) {
  sumarr<<<((N+2)*(N+2)*(N+2))/512,512>>>(d,g_t4,(N+2)*(N+2)*(N+2));
  buoyancy<<<((N+2)*(N+2)*(N+2))/512,512>>>(x,d,g_t4,dt,N);
  set_bnd<<<N+2,N+2>>>(2,x,N);
  set_bnd2<<<1,N>>>(2,x,N);
```

```
}
```

Bei der Auftriebskraft wird zunächst der Mittelwert aller Skalare gebildet. Dies geschieht mit einer parallelen Reduktion (*sumarr*). Danach wird die Auftriebskraft berechnet und die Randbedingungen gesetzt (für die y-Komponente des Geschwindigkeitsfeldes).

```
__global__ void sumarr(float *x,float *t,int N) {
  int gid=threadIdx.x+blockIdx.x*blockDim.x;
  __shared__ float s[512];
  if(gid<N) {
    s[threadIdx.x]=x[gid];
    __syncthreads();
    if(threadIdx.x<256) { s[2*threadIdx.x]+=s[threadIdx.x*2+1]; __syncthreads(); }
    if(threadIdx.x<128) { s[4*threadIdx.x]+=s[4*threadIdx.x+2]; __syncthreads(); }
    if(threadIdx.x<64) { s[8*threadIdx.x]+=s[8*threadIdx.x+4]; __syncthreads(); }
    if(threadIdx.x<32) { s[16*threadIdx.x]+=s[16*threadIdx.x+8]; __syncthreads(); }
    if(threadIdx.x<16) { s[32*threadIdx.x]+=s[32*threadIdx.x+16]; __syncthreads(); }
    if(threadIdx.x<8) { s[64*threadIdx.x]+=s[64*threadIdx.x+32]; __syncthreads(); }
    if(threadIdx.x<4) { s[128*threadIdx.x]+=s[128*threadIdx.x+64]; __syncthreads(); }
    if(threadIdx.x<2) { s[256*threadIdx.x]+=s[256*threadIdx.x+128]; __syncthreads(); }
    if(threadIdx.x==0) { s[0]+=s[256]; __syncthreads(); t[blockIdx.x]=s[0];}
  }
}
```

Diese Funktion führt eine parallele Reduktion durch, sie addiert alle Skalarwerte zusammen mithilfe des Shared Memory, indem ein Teil des Feldes in den Shared Memory geladen wird und dann die Threads erst jeden zweiten Wert aufsummieren, danach jeden vierten, usw. Am Ende werden die aufsummierten Werte in ein Hilfsfeld zurückgeschrieben (sie müssen später in der *buoyancy*-Funktion noch weiter aufaddiert werden.

```
__global__ void buoyancy(float *x,float *d,float *t,float dt,float N) {
  int gid=threadIdx.x+blockIdx.x*blockDim.x;
  int N2=N+2;
  int k=(gid%(N2))
  int j=((gid-k)/(N2))%(N2);
  int i=(gid-k)/(N2*N2);
  if(gid<N2*N2*N2) {
    float Tamb = 0.0f;
    float a = 0.0f; // Gravitation
    float b = 0.25f; // Auftriebsfaktor
    int ijk=I(i,j,k,N2);
```

```
    float dens=d[ijk];
    // letzte Aufsummierung nach Reduktion
    for(int c=0;c<((N2*N2*N2)/512);c++) {
      Tamb+=t[c];
    }
    // Temperaturmittelwert
    Tamb = Tamb/(N2*N2*N2);
    // Auftriebskraft
    x[ijk] += (-a * dens + b * (dens - Tamb))*dt;
  }
}
```

In der Auftriebskraft-Funktion werden zunächst die Indizes i, j, k bestimmt, die Temperatur schlussendlich aufaddiert und der Mittelwert gebildet. Die Auftriebskraft wird dann wie in Abschnitt 2.2.7 beschrieben zur y-Komponente des Geschwindigkeitsfeldes hinzuaddiert. In der Implementierung der Auftriebskraft kann bei Bedarf auch die Gravitation berücksichtigt werden, sofern das Skalarfeld beispielsweise die Dichte von Rauch beschreibt.

3.4.8 Vorticity Confinement

```
__host__ __device__ float3 curl(float *x,float *y,float *z,int i,int j,int k,int N) {
  int N2=N+2;
  float d1=0.5f*(z[I(i,j+1,k,N2)]-z[I(i,j-1,k,N2)]-y[I(i,j,k+1,N2)]+y[I(i,j,k-1,N2)]);
  float d2=0.5f*(x[I(i,j,k+1,N2)]-x[I(i,j,k-1,N2)]-z[I(i+1,j,k,N2)]+z[I(i-1,j,k,N2)]);
  float d3=0.5f*(y[I(i+1,j,k,N2)]-y[I(i-1,j,k,N2)]-x[I(i,j+1,k,N2)]+x[I(i,j-1,k,N2)]);
  return make_float3(d1,d2,d3);
}
__host__ __device__ float vabs(float3 vec) {
  return sqrtf(vec.x*vec.x+vec.y*vec.y+vec.z*vec.z);
}
__global__ void vorticityconfinement(float *vx0,float *vy0,float *vz0,float *vx,
                             float *vy,float *vz,float dt,float vort_eps,int N) {
  int gid=blockIdx.x*blockDim.x+threadIdx.x;
  int N2=N+2;
  if(gid<(N2)*(N2)*(N2)) {
    int k=(gid)%(N2);
    int j=((gid-k)/(N2))%(N2);
    int i=(gid-k)/((N2)*(N2));
```

```
    if(i>=2 && i<N && j>=2 && j<N && k>=2 && k<N) {
      int ijk=I(i,j,k,N2);
      float dt0=dt*vort_eps;
      // Rotation berechnen
      float Nx = (vabs(curl(vx0,vy0,vz0,i+1,j,k,N))
                - vabs(curl(vx0,vy0,vz0,i-1,j,k,N))) * 0.5f;
      float Ny = (vabs(curl(vx0,vy0,vz0,i,j+1,k,N))
                - vabs(curl(vx0,vy0,vz0,i,j-1,k,N))) * 0.5f;
      float Nz = (vabs(curl(vx0,vy0,vz0,i,j,k+1,N))
                - vabs(curl(vx0,vy0,vz0,i,j,k-1,N))) * 0.5f;
      // Wirbelstärke-Ortsvektor
      float len1 = 1.0f/(sqrtf(Nx*Nx+Ny*Ny+Nz*Nz)+0.00000001f);
      Nx *= len1;
      Ny *= len1;
      Nz *= len1;
      float3 w=curl(vx0,vy0,vz0,i,j,k,N);
      vx[ijk] = vx0[ijk]+(Ny*w.z - Nz*w.y) * dt0;
      vy[ijk] = vy0[ijk]+(Nz*w.x - Nx*w.z) * dt0;
      vz[ijk] = vz0[ijk]+(Nx*w.y - Ny*w.x) * dt0;
    }
  }
}
```

Die Erhaltung der Wirbelstärke wird durch die folgende Gleichung $f_{vorticity} = \epsilon h(N \times \omega)$, wie in Abschnitt 2.2.8 beschrieben, mit Hilfe der Rotation $\omega = \nabla \times u$ und dem normalisierten Ortsvektors $N = \frac{\nabla|\omega|}{|\nabla|\omega||}$ errechnet. Hier ist es nötig, die $(N-1)^3$ innersten Zellen zu betrachten.
Die Addition `float len1 = 1.0f/(sqrtf(Nx*Nx+Ny*Ny+Nz*Nz)+0.00000001f);` mit 10^{-8} verhindert eventuelle Null-Divisionen.

4 Ergebnisse

In diesem Abschnitt erfolgt für das implementierte Programm zunächst eine algorithmische Komplexitätanalyse, gefolgt von der Speicherbedarfsermittlung, der Beschreibung des Testsystems und den Ergebnissen der Laufzeitmessungen der einzelnen Teilschritte bzw. der Kernelfunktionen bei verschiedenen Datengrößen. Hierbei werden CPU- und GPU-Laufzeit auf dem Testsystem gegenübergestellt und eventuelle Auffälligkeiten diskutiert. Zuletzt folgt eine Auswertung der produzierten Bilder und ein Vergleich der Gesamtlaufzeit.

4.1 Rechenzeit- und Speicherplatzanalyse

Im Folgenden wird die algorithmische Komplexität der Implementierung beschrieben. Die Komplexität jeder Operation wird der Einfachheit halber mit $O(1)$ festgelegt. Da sich die tatsächlichen Beschleunigungsfaktoren durch die Parallelisierung nicht im Vorhinein bestimmen lassen, treten diese Faktoren auch nicht in den algorithmischen Komplexitäten auf. Somit wird der Rechenaufwand pro Thread betrachtet.

4.1.1 Externe Kräfte

Der Kernel *add_source* zum Hinzufügen externer Kräfte führt in jedem Thread folgende Operationen aus:

- Addition: 5
- Multiplikation: 4
- Global Memory Load/Store: 3

Die algorithmische Komplexität entspricht $O(12 \cdot N^3)$.

4.1.2 Randbedingungen

Beim Setzen der Randbedingungen durch die Funktionen *set_bnd* und *set_bnd*2 fallen folgende Operationen pro Thread an:

- Additionen: 20

- Multiplikationen: 15
- Global Memory Load/Store: 52

Die algorithmische Komplexität entspricht $O(87 \cdot N^2)$.

4.1.3 Linearer Gleichungslöser

Im Gauss-Seidel-Kernel *lin_solve* wird jeweils eine komplette Datenreihe in z-Richtung gerechnet, somit fallen folgende Operationen pro Thread an:

- Additionen: 7*N
- Multiplikationen: 2*N
- Global Memory Load/Store: 7*N+1

Die algorithmische Komplexität entspricht $O(98 \cdot N^3 + N^2)$.

4.1.4 Diffusion

Im Diffusionsschritt $diffuse$ finden neben den lin_solve folgende Operationen pro Thread statt:

- Additionen: 2
- Multiplikationen: 5
- Divisionen: 1

Die algorithmische Komplexität entspricht $O(8 \cdot N^3)$.
Hinzu kommt die Komplexität der *iter* Gleichungslöser-Aufrufe und der Randbedingungen: $iter \cdot (O(98 \cdot N^3 + N^2) + O(87 \cdot N^2))$.
Die Gesamtkomplexität entspricht $O((iter \cdot 98 + 8) * N^3 + iter \cdot 88N^2)$.

4.1.5 Advektion

Im Advektionsschritt *advect* fallen folgende Operationen pro Thread an:

- Additionen: 19 (+2)
- Multiplikationen: 7 (+1)
- Modulo-Operationen: 1
- Global Memory Load/Store: 4 (+8)

Die Werte in Klammern entsprechen den Operationen einer trilinearen Interpolation. Somit ergibt sich eine Gesamtkomplexität von $O(42 \cdot N^3)$.

4.1.6 Projektion

Neben den *lin_solve*-Aufrufen benötigt der Projektionsschritt (*project*, *project*2) folgende Operationen pro Thread:

- Additionen: 15 +11 +13
- Multiplikationen: 4 +7 +9
- Modulo-Operationen: 1 +1
- Global Memory Load/Store: 8 +12

Dies entspricht einer Komplexität von $O(81 \cdot N^3)$.
Hinzu kommt die Komplexität der *iter* Gleichungslöseraufrufe und die $(iter + 4)$ Randbedingungen: $iter \cdot O(98 \cdot N^3 + N^2) + (iter + 4) \cdot O(87 \cdot N^2)$
Somit ergibt sich eine Gesamtkomplexität von: $O(iter + 87 \cdot N^2 + (iter \cdot 179) \cdot N^3)$.

4.1.7 Vorticity Confinement

Beim *vorticity_confinement*-Kernel fallen folgende Operationen pro Thread an:

- Additionen: 188
- Multiplikationen: 61
- Division: 3
- Wurzel: 7
- Modulo-Operationen: 2
- Global Memory Load/Store: 88

Dies ergibt eine Komplexität von $O(349 \cdot N^3)$
Somit ist das Vorticity Confinement die rechenintensivste Funktion im Einzelvergleich.

Sowohl intuitiv als auch aus den obigen Darstellungen ist ersichtlich, dass die Simulation eine Gesamtkomplexität von $O(N^3)$ hat.

4.1.8 Speicherbedarf

Es werden die insgesamt 9 Felder der Größe N^3 benötigt. Bei Verwendung des Datentyps 32bit-Float ergibt sich daher ein Speicherbedarf von $9 \cdot 4 \cdot N^3$-Byte.

- N=16 : 144 Kilobyte
- N=32 : 1152 Kilobyte
- N=64 : 9216 Kilobyte
- N=128 : 73728 Kilobyte = 72 Megabyte
- N=256 : 576 Megabyte
- N=512 : 4608 Megabyte (passt nicht mehr in den Speicher bisheriger Grafikkarten)

4.2 CPU vs. GPU

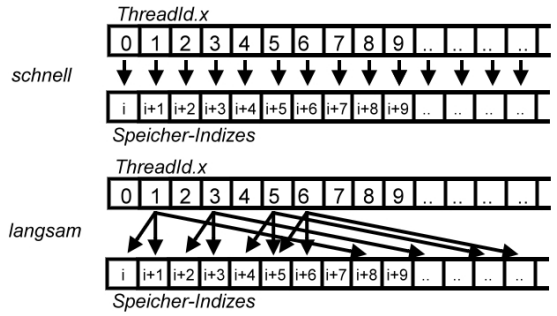


Abbildung 4.1: Performanzabhängiger Speicherzugriff

Bei allen Vergleichstests ist zu beachten, dass die Leistungsfähigkeit eines Grafikkarten-Kerns nicht mit der Leistung eines CPU-Kerns gleichzusetzen ist. Zwar ist der Speicher der Grafikkarte (GDDR5[1]) schneller als der normale Arbeitsspeicher (DDR3[2]), jedoch besitzt die CPU bessere Cache-Mechanismen, wodurch häufig benutzte Daten schneller verfügbar sind. Läuft ein Programm auf der CPU, so steht die Leistung der CPU nicht vollständig dem Programm zur Verfügung, sondern alle Programme des Rechners laufen quasi nebenläufig und bekommen Zeitscheiben. Läuft ein Kernel auf einer Grafikkarte, so steht dem

[1] Graphics Double Data Rate 5-Speicher bezeichnet den aktuell schnellsten Arbeitsspeicher für Grafikkarten
[2] Double Data Rate 3-Speicher bezeichnet den aktuellen Arbeitsspeicherstandard

Kernel im Normalfall die komplette Grafikkarte zur Verfügung. Ein GPU-Kern eines Multiprozessors alleine ist jedoch langsamer als ein regulärer CPU-Kern. Auch zu beachten ist, dass die Performanz eines GPU-Kerns maßgeblich von den Speicherzugriffen abhängt. Erfolgt der Zugriff geordnet (koaleszierend), so kann maximale Performanz erzielt werden. Bei einem ungeordneten Zugriff (unkoaleszierend) kann der Speicherzugriff bis zu 10x langsamer sein (siehe Abbildung 4.1).

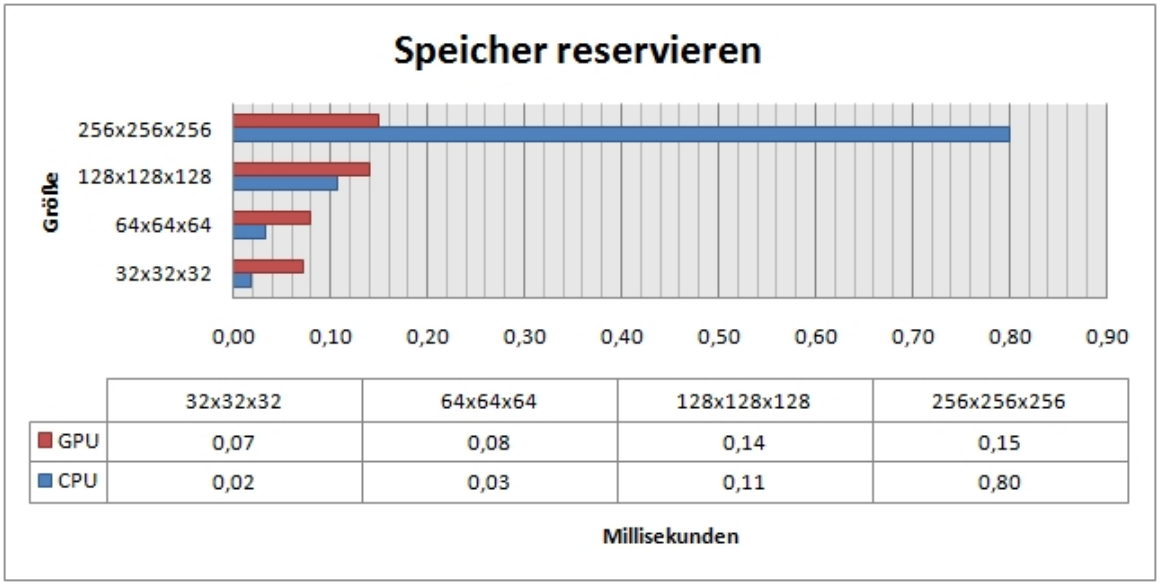
4.2.1 Testumgebung

Die im Folgenden durchgeführten Laufzeittests wurden alle auf der nachfolgend angegebenen Hardware-Konfiguration durchgeführt.

- Computer: Dell Precision Workstation T7500 (C3)
- Prozessor: Intel Xeon E5520 @ 2.27 Ghz (Quadcore)
- Arbeitsspeicher: 24 GB
- Grafikkarten: 2x NVIDIA Tesla C2070
- Betriebssystem: Ubuntu v10.04 LTS
- CUDA-Version: v4.0

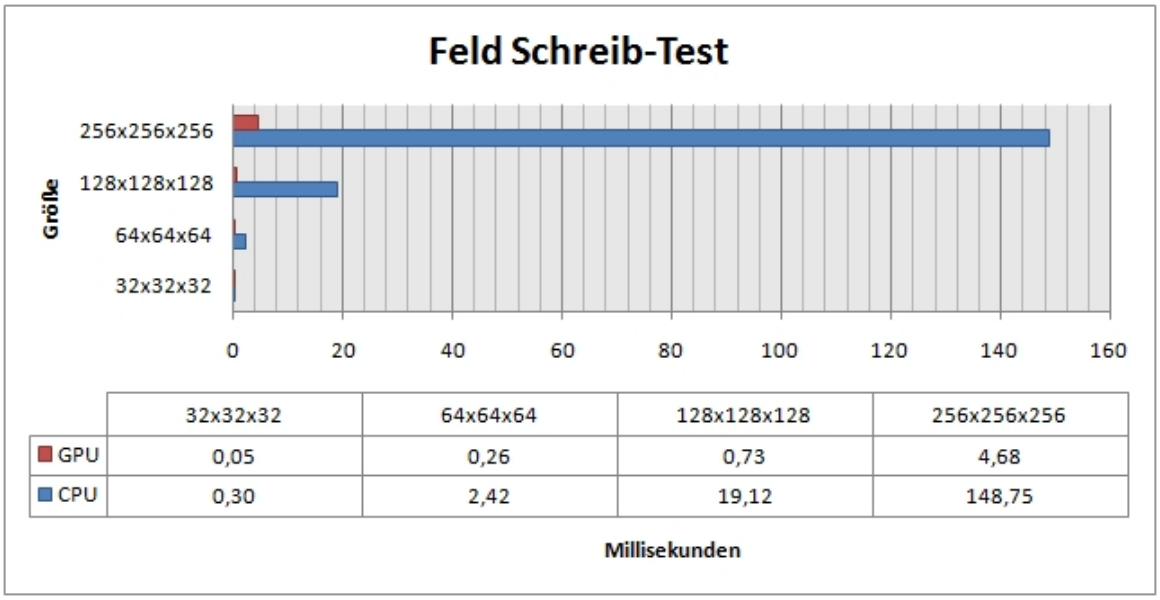
4.2.2 Speichertests

Zunächst wurden einige Vergleichstests einfacher Speicheroperationen durchgeführt. Abbildung 4.2 stellt die Ergebnisse eines einfachen Speicherreservierungstests mit verschiedenen Feldgrößen (32^3 bis 256^3) dar. Bis auf den Fall $N = 256^3$ sind dauert die Speicherreservierung auf der GPU länger als auf CPU, was nicht weiter überrascht. Der Fall $N = 256^3$ stellt allerdings eine (reproduzierbare) Anomalie dar, da hier die CPU ca. die fünffache Zeit benötigt gegenüber der GPU. Insgesamt liegen die Anforderungszeiten alle unter einer Millisekunde.
Abbildung 4.3 stellt die Ergebnisse eines Feld-Schreib-Tests dar. Während bei der CPU-Variante das Schreiben der Daten sequentiell erfolgt, so ist das Schreiben der Daten in der GPU-Variante parallelisiert. Daher ist es wenig überraschend, dass die parallelisierte GPU-Variante bei allen Größen schneller ist.
Das Kopieren von Daten zwischen CPU und der Grafikkarte wird in Abbildung 4.4 dargestellt. Es ist zu entnehmen, dass das der Datenaustausch zwischen CPU und GPU meist gleichlang dauert und auch proportional anwächst. Pro Feld wird maximal $\frac{1}{10}$ Sekunde benötigt.



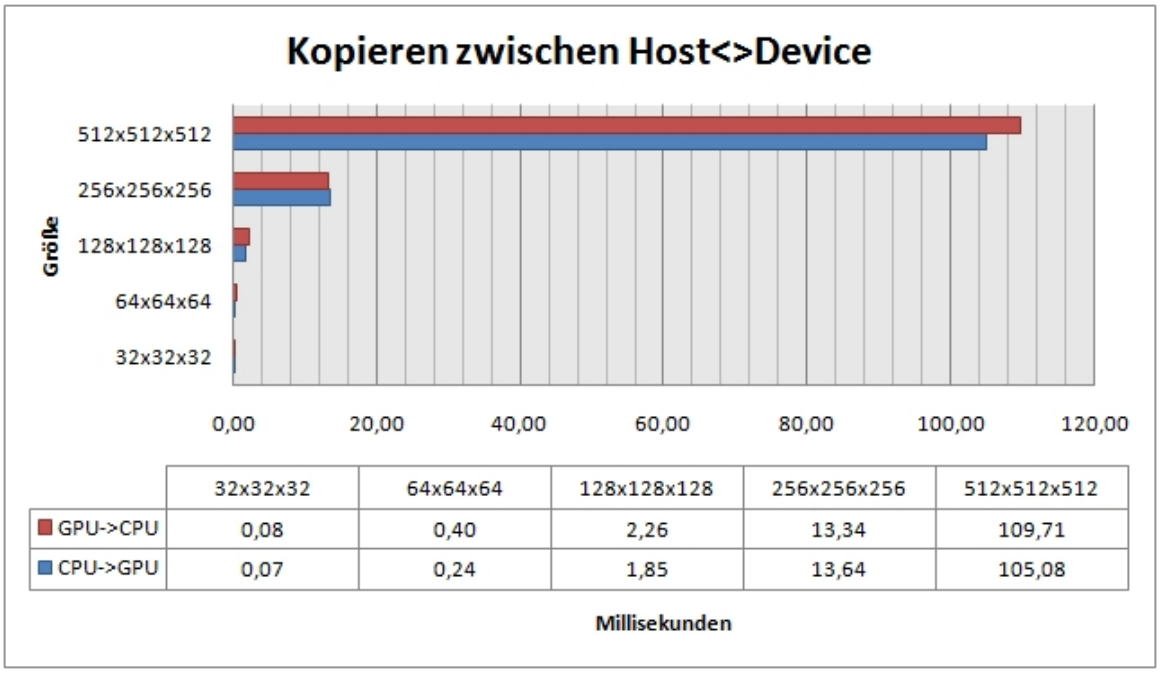
	32x32x32	64x64x64	128x128x128	256x256x256
GPU	0,07	0,08	0,14	0,15
CPU	0,02	0,03	0,11	0,80

Abbildung 4.2: Speicher reservieren



	32x32x32	64x64x64	128x128x128	256x256x256
GPU	0,05	0,26	0,73	4,68
CPU	0,30	2,42	19,12	148,75

Abbildung 4.3: Schreib-Test

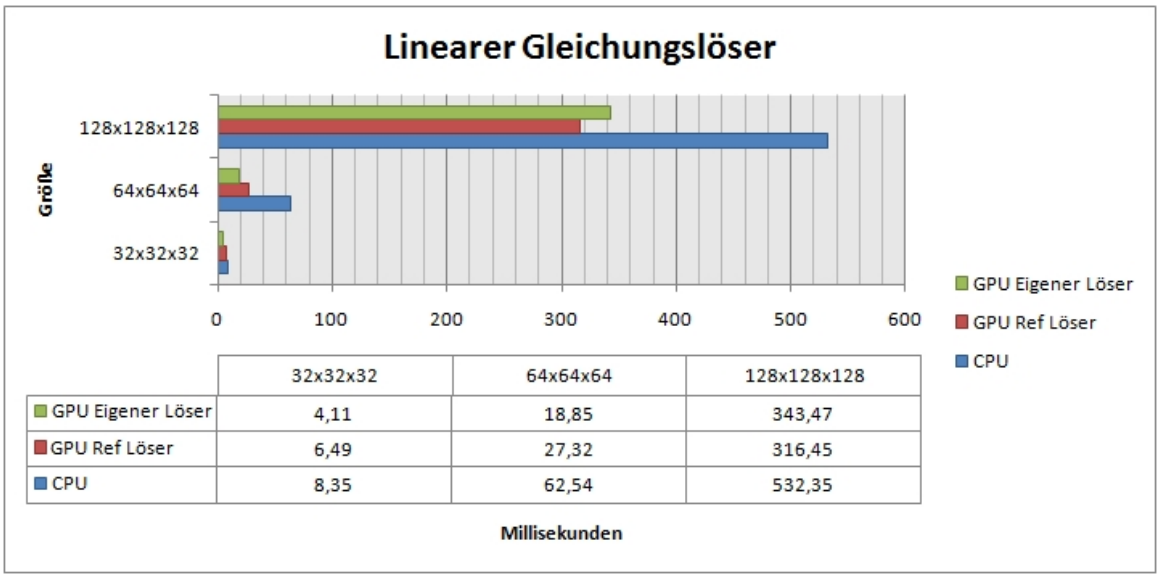


	32x32x32	64x64x64	128x128x128	256x256x256	512x512x512
GPU->CPU	0,08	0,40	2,26	13,34	109,71
CPU->GPU	0,07	0,24	1,85	13,64	105,08

Abbildung 4.4: Kopieren zwischen CPU<>GPU

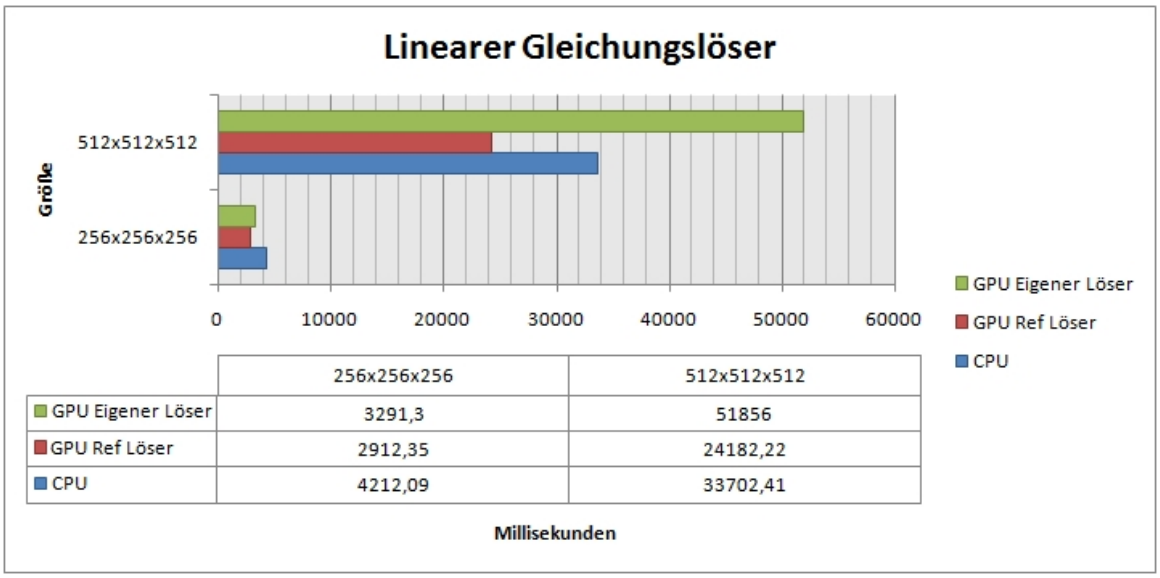
4.2.3 Gleichungslöser

Zum Lösen des Projektions- und Diffusionsschritts wurde ein Red-Black-Gauss-Seidel-Kernel verwendet. Sowohl eine sequentielle CPU-Variante, als auch die GPU-Variante aus [4] und eine leicht abgeänderte eigene Version wurden implementiert. Die Abbildungen 4.5 und 4.6 zeigen die Ergebnisse eines Leistungstests mit den besagten Varianten bei verschiedenen Feldgrößen. Es wurden jeweils 20 Gauss-Seidel-Iterationen durchgeführt und die Zeit gemessen. Der Referenzlöser aus [4] erreicht bis auf den Fall $N = 32^3$ und den Fall $N = 64^3$ die besten Zeiten. In den eben genannten Fällen ist die eigene Implementierung etwas schneller und ist bei $N = 64^3$ mehr als doppelt so schnell wie die CPU-Variante. Da beim Gauss-Seidel-Verfahren viele Speicherzugriffe aus dem langsamen Global Memory erfolgen, die sich teilweise blockieren durch den parallelen Zugriffsversuch, können keine sehr großen Beschleunigungswerte erzielt werden. Außerdem verfügt die CPU über bessere Cache-Mechanismen, so dass Speicherzugriffe schneller erfolgen können, was den Beschleunigungsfaktor der GPU weiter dämpft.



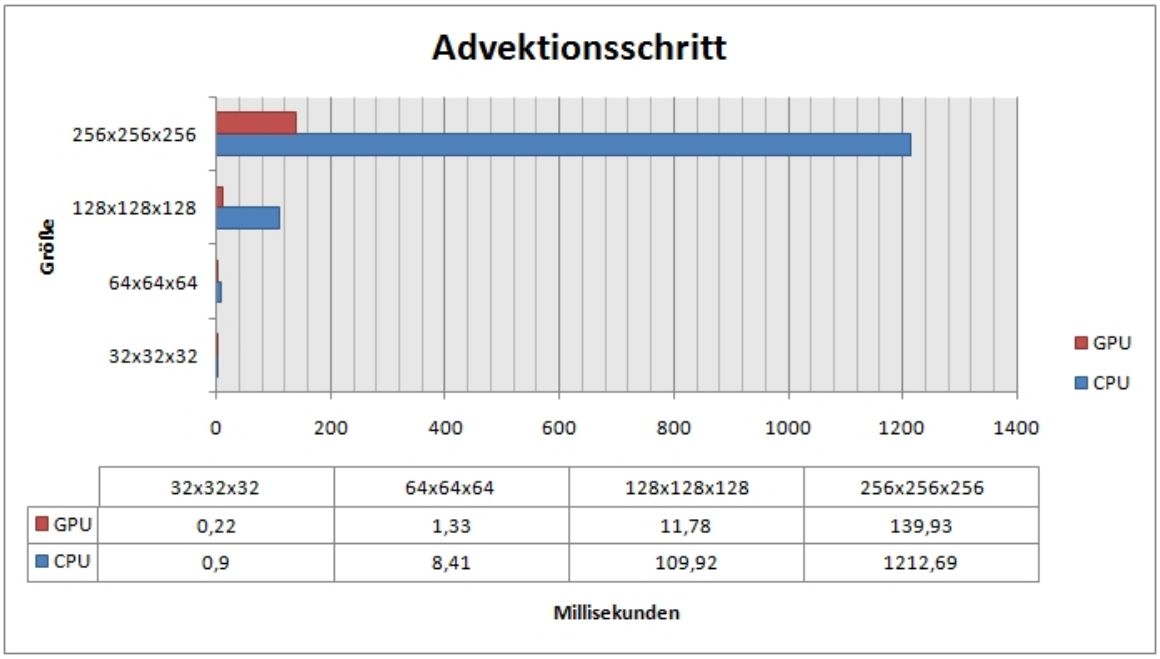
	32x32x32	64x64x64	128x128x128
GPU Eigener Löser	4,11	18,85	343,47
GPU Ref Löser	6,49	27,32	316,45
CPU	8,35	62,54	532,35

Abbildung 4.5: Rechenzeit: Linearer Gleichungslöser(1)



	256x256x256	512x512x512
GPU Eigener Löser	3291,3	51856
GPU Ref Löser	2912,35	24182,22
CPU	4212,09	33702,41

Abbildung 4.6: Rechenzeit: Linearer Gleichungslöser(2)



	32x32x32	64x64x64	128x128x128	256x256x256
GPU	0,22	1,33	11,78	139,93
CPU	0,9	8,41	109,92	1212,69

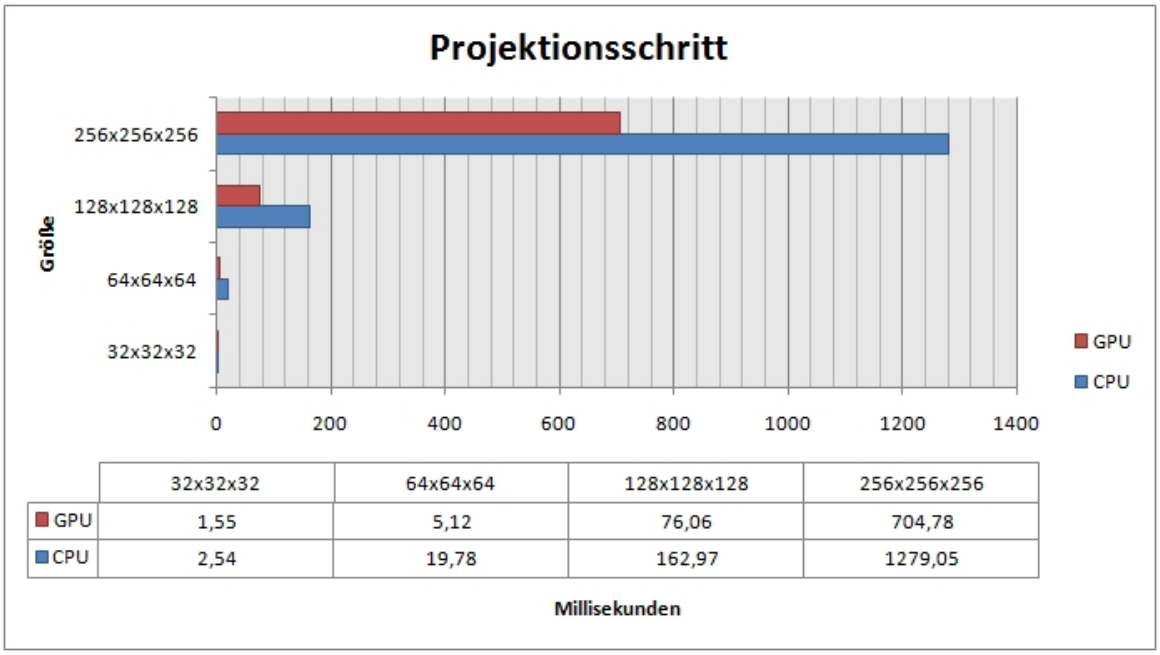
Abbildung 4.7: Rechenzeit: Advektionsschritt

4.2.4 Advektion

Im Advektionsschritt sind, wie aus der Abbildung 4.7 zu entnehmen ist, Beschleunigungsfaktoren zwischen 5-7 möglich. Da pro Thread zwölf Global-Memory-Anfragen stattfinden, die ungeordnet sind (kein blockweiser Zugriff) können offenbar keine höheren Raten erreicht werden.

4.2.5 Projektion

Sowohl der Projektionsschritt, als auch der Diffusionsschritt hängen maßgeblich von der Geschwindigkeit des Gleichungslösers ab. Da im Diffusionsschritt praktisch nur der Löser aufgerufen wird und die Randbedingungen gesetzt werden wurden keine Leistungsvergleiche für den Diffusionsschritt erstellt. Stattdessen soll nun der Projektionsschritt betrachtet werden, welcher allerdings auch zu großem Teil vom Löser abhängt. In Abbildung 4.8 sind die Ergebnisse dargestellt und zeigen, dass die Beschleunigung maximal den Faktor drei beträgt (bei $N = 64^3$). Die GPU-Variante erzielt trotzdem bei jeder Größe bessere Zeiten auch wenn der Beschleunigungsfaktor begrenzt ist.



	32x32x32	64x64x64	128x128x128	256x256x256
GPU	1,55	5,12	76,06	704,78
CPU	2,54	19,78	162,97	1279,05

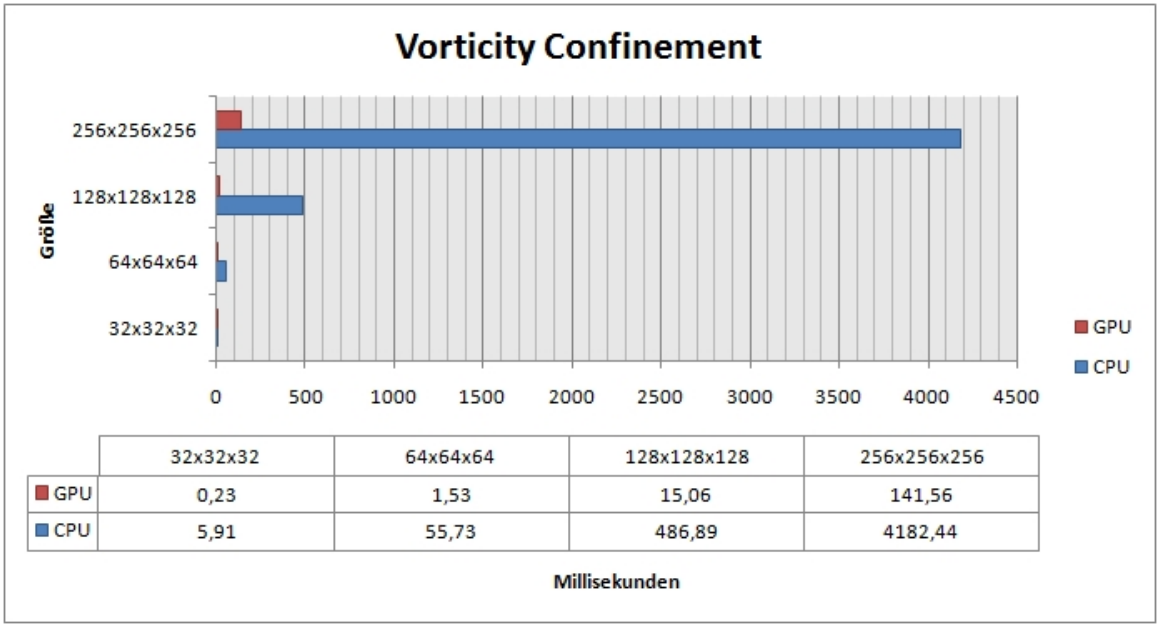
Abbildung 4.8: Projektionssschritt

4.2.6 Vorticity Confinement

In diesem Schritt ist ein Beschleunigungsfaktor von ca. 38 gegenüber der CPU-Variante messbar (siehe Abbilung 4.9). Zwar werden wie beim Gleichungslöser viele Daten ungeordnet gelesen, aufgrund der Verwendung der Rotation, jedoch sind die Schreibzugriffe geordnet.

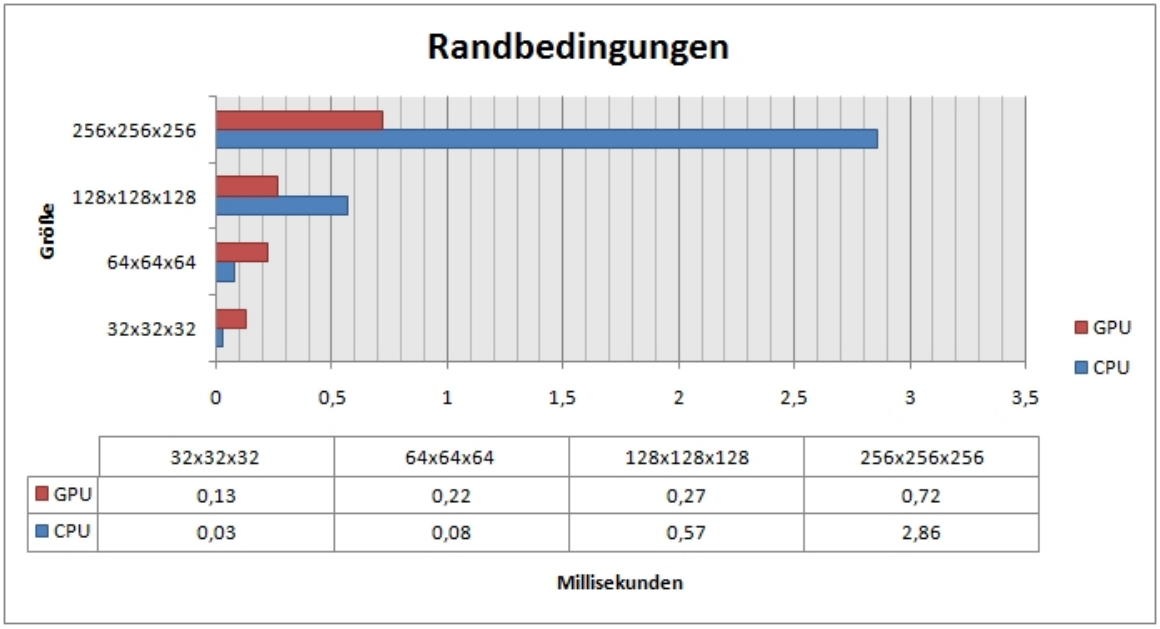
4.2.7 Randbedingungen

Aus Abbildung 4.10 geht hervor, dass beim Setzen der Randbedingungen die GPU-Variante erst ab einer Größe von $N = 128^3$ punkten kann. Da nur die Ränder betrachtet werden ist die Anzahl der Daten für kleinere N zu gering, hier erzielt die CPU-Variante ein besseres Ergebnis. Zwar werden die Randbedingungen am häufigsten ausgeführt von allen Funktion, da die Laufzeit jedoch unterhalb einer Millisekunde liegt, wirkt sich dies kaum auf die Gesamtperformanz der Simulation aus (vgl. Abbildung 4.20).



	32x32x32	64x64x64	128x128x128	256x256x256
GPU	0,23	1,53	15,06	141,56
CPU	5,91	55,73	486,89	4182,44

Abbildung 4.9: Rechenzeit: Vorticity Confinement



	32x32x32	64x64x64	128x128x128	256x256x256
GPU	0,13	0,22	0,27	0,72
CPU	0,03	0,08	0,57	2,86

Abbildung 4.10: Rechenzeit: Randbedingungen

4.2.8 Ray Casting

Im Folgenden wurde die Leistung des Ray Casters gemessen. Die Abbildung 4.11 zeigt die Ergebnisse bei einer Auflösung von 1024x1024-Pixel und verschiedenen Volumengrößen. Zunächst scheint die Performanz zwar von der Volumengröße abzuhängen, jedoch findet bei einer Volumengröße unter $N = 32^3$ keine Geschwindigkeitserhöhung mehr statt. Maßgeblich für die Performanz scheinen die Auflösung und somit die Anzahl der Strahlen zu sein. Sowohl die GPU als auch die CPU-Variante wurden ohne Simulation des Fluids durchgeführt, so dass die Ergebnisse in Abbildung 4.8 nur die Werte des Ray Casters alleine beinhalten, nicht die Geschwindigkeit des vollständigen Programms. Der Ray Caster kann für Volumengrößen bis $N = 128^3$ noch Echtzeitausführung ermöglichen.

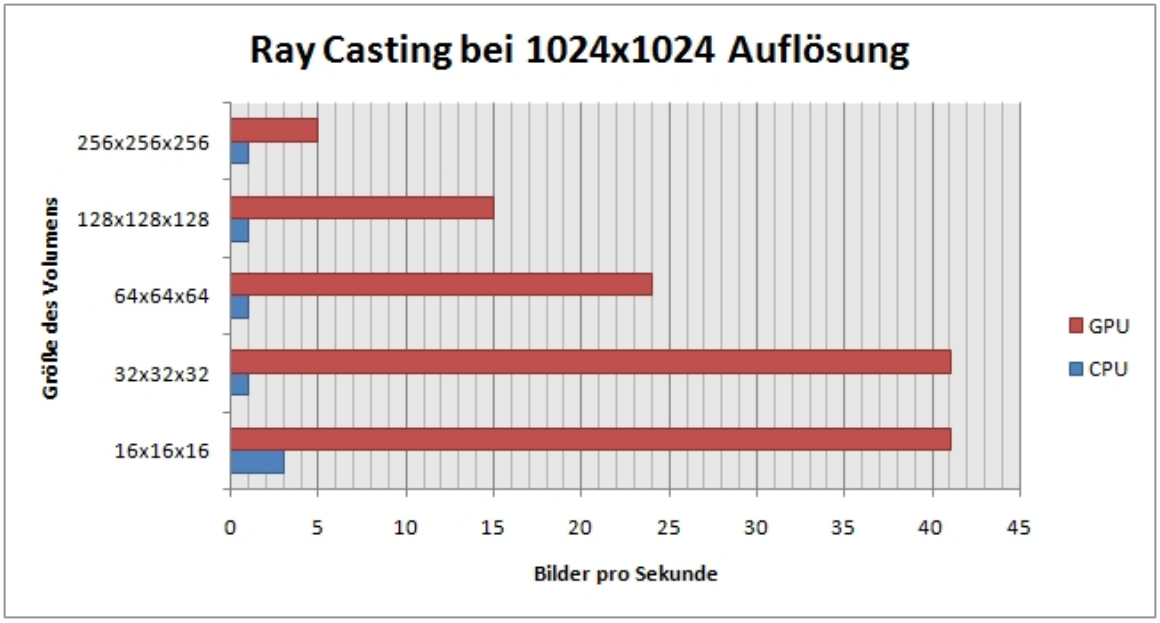


Abbildung 4.11: Rechenzeit: Ray Casting

4.3 Test-Szenarien

Es wurden zwei Testszenarien implementiert.

- Simulation der Temperatur eines Raumes mit einer Kühlung am Boden und einem Heizelement an der Decke
- Simulation von aufsteigendem Rauch

Beide Szenarien wurden bei einer Volumendatengröße von $N = 64^3$, einer Auflösung von $1024x1024$-Bildpunkten und einem Zeitschritt von $\Delta t = 0.1$ durchgeführt.

4.3.1 Temperaturmodellierung mit Heiz- und Kühlelement

Im ersten Szenario wird die Temperatur als Skalarfeld verwendet, die Umgebungstemperatur wird mit $20°C$ festgelegt. Das Kühlelement befindet sich eine Schicht unterhalb der Decke, hat eine Ausdehnung von $62x1x62$-Zellen und ist in der Lage, die Temperatur um $20°C$ pro Sekunde bis auf $0°C$ in diesem Bereich abzukühlen, das Heizelement, welches sich eine Schicht oberhalb des Bodens befindet, erwärmt die Luft um $20°C$ pro Sekunde bis auf maximal $40°C$. Es hat ebenfalls eine Ausdehnung von $62x1x62$-Zellen. Dieses Szenario ähnelt den Standard-Versuchen zur Rayleigh-Bénard-Konvektion, dort wird allerdings die Temperatur über die Randbedingungen fest gewählt, während im implementierten Beispiel die aktuelle Temperatur relativ heruntergekühlt, bzw. erhitzt wird.
Das Geschwindigkeitsfeld wird zum Zeitpunkt t_0 mit Null, das Temperaturfeld mit $20°C$ initialisiert.
In den produzierten Bildern ergibt sich die Farbe aus dem Unterschied zur Raumtemperatur ($20°C$). Rote Bereiche entsprechen wärmerer Luft, blaue Bereiche kälterer Luft. Aufgrund der Veränderungen der Temperatur durch das Heiz- und das Kühlelement (Abbildung 4.12) bildet sich zunächst eine Kalt- und eine Warmfront (Abbildung 4.13). Nachdem die beiden Fronten aufeinandertreffen, durchmischen sich kalte und warme Luft und es bilden sich Konvektionszellen, die ansich zwar kaum zu erkennen sind, jedoch sind "Temperaturschläuche", welche bei klassichen Rayleigh-Bénard-Konvektionsexperimenten auftreten, zu sehen (Abbildung 4.14). Beim Voranschreiten der Simulation bilden sich irgendwann zwei Hauptströme, ein Warmstrom, der vom Heizelement über die rechte Wandseite nach oben fließt, und ein Kaltstrom, der vom Kühlelement an der linken Wand herunterfließt (Abbildung 4.15).

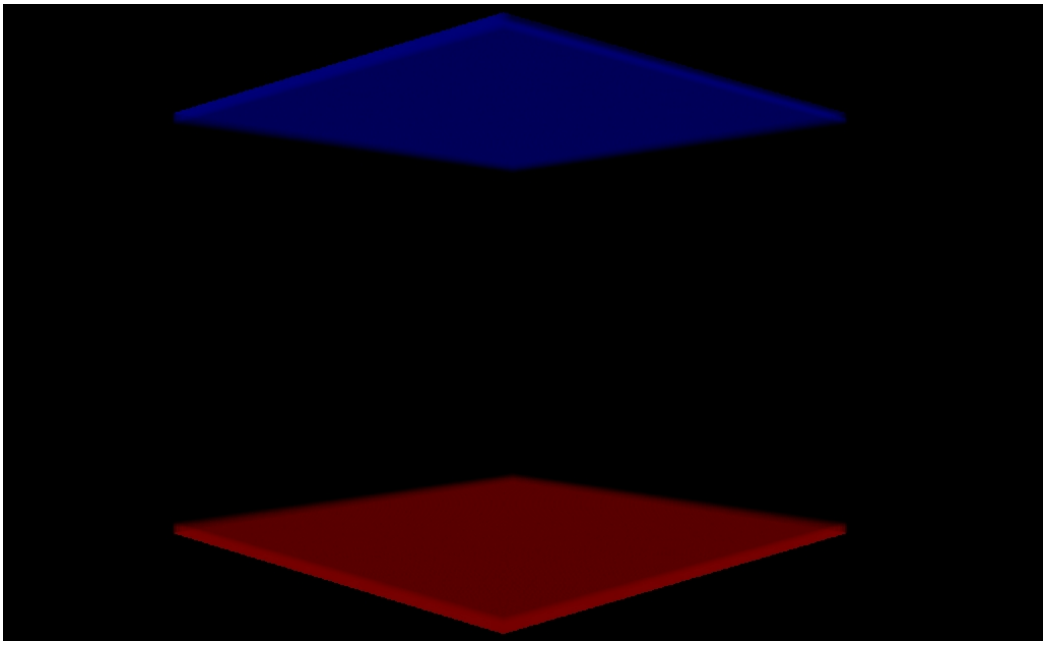


Abbildung 4.12: Heiz- und Kühlelement (t_1)

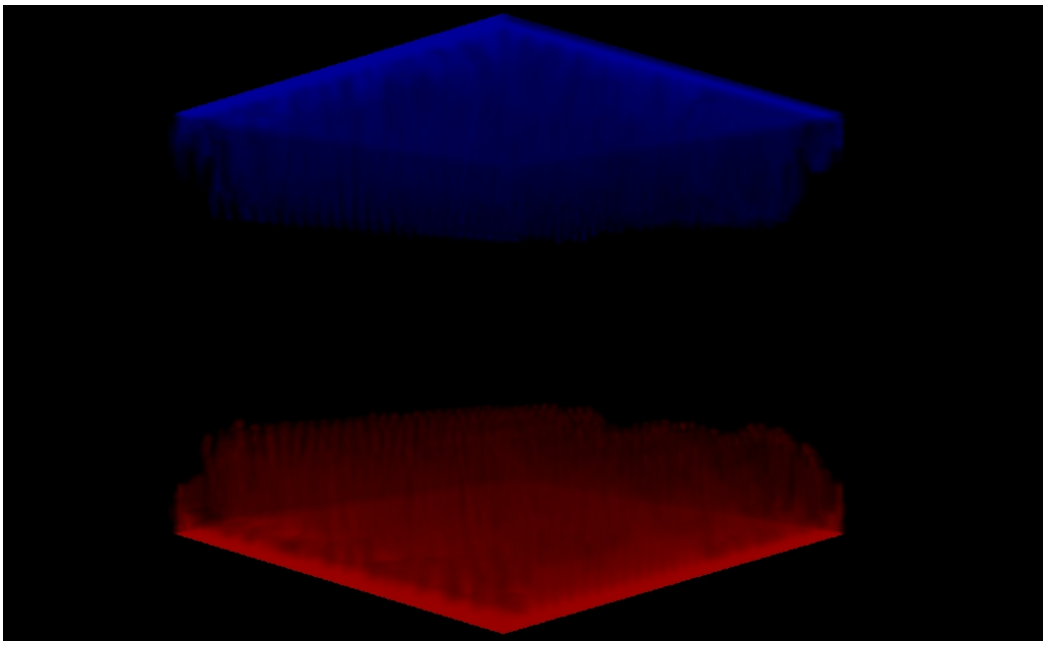


Abbildung 4.13: Temperaturfronten (t_2)

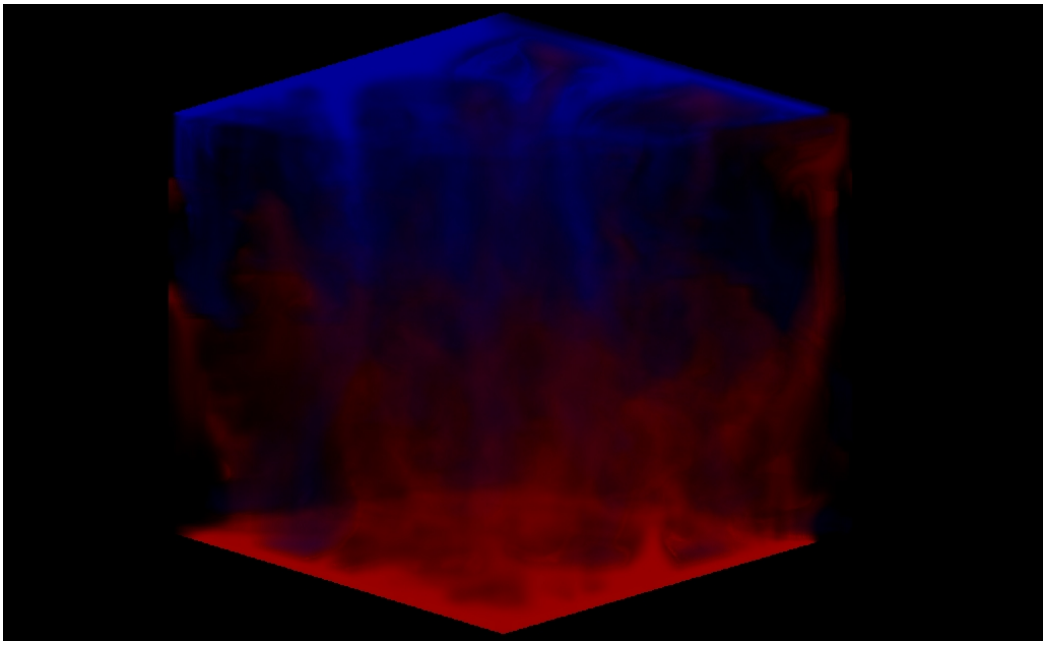


Abbildung 4.14: Konvektionszellen und Temperaturschläuche (t_{10})

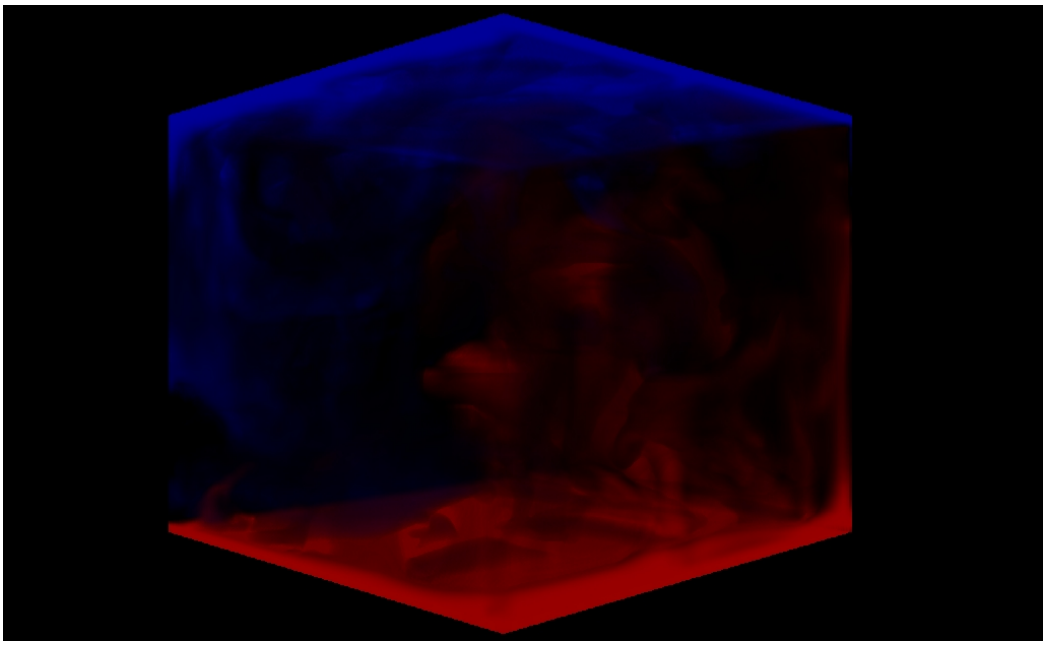


Abbildung 4.15: Zwei Hauptströme (t_{80})

Die Visualisierung des Volumendatensatzes funktioniert, die Darstellung ist jedoch sehr abhängig von der verwendeten Transferfunktion. Als Kompositionsschema wurde eine Mittelwertsfunktion verwendet. Das Phong-Beleuchtungsmodell wurde zwar implementiert, jedoch deaktiviert, da für die Temperaturmodellierung die Verwendung des Gradienten als Normale eher ungeeignet ist. Möchte man sowohl kalte, als auch heiße Bereiche visualisieren, so zeigt der Gradient für einen der beiden Fälle in die falsche Richtung. Abbildung 4.16 zeigt die Gradienten und die (korrekten) Normalen für drei Fälle. Das Beleuchtungsmodell kann anhand der Normalen Oberflächen sichtbar machen. Im ersten Fall zeigt der Gradient am Punkt P zur Warmfront hin, statt von ihr weg. Im zweiten Fall ist der Gradient äquivalent zur Normalen. Im dritten Fall ist der Gradient Null. Dieser Fall würde beim Normalisieren zu einer Nulldivision führen und muss daher gesondert behandelt werden. Diese Problematik führt bei der Temperaturmodellierung mit Beleuchtungsmodell zu enormen Darstellungsfehlern, da die Beleuchtung falsch berechnet wird. Als Lösung bleibt in diesem Fall entweder der Verzicht auf die Beleuchtung oder der Verzicht auf die gleichzeitige Visualisierung beider Temperaturbereiche.

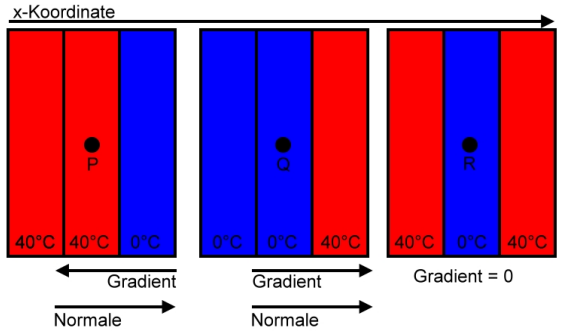


Abbildung 4.16: Probleme bei Nutzung des Gradienten als Normale

4.3.2 Simulation von aufsteigendem Rauch

Im zweiten Szenario wird die Dichte von Rauchpartikeln simuliert. Es existiert eine Quelle, die dem System in jedem Zeitschritt Rauchpartikel hinzufügt, ähnlich einer glimmenden Zigarette. Die Rauchquelle befindet sich eine Schicht über dem Boden und hat ein Ausmaß von $1x1x1$-Zellen. Auch hier wird das Geschwindigkeitsfeld, sowie das Dichtefeld des Rauchs zum Zeitpunkt t_0 mit Null initialisiert.
In den produzierten Bildern ergibt sich die Farbe aus der Dichte des Rauchs. Weiße Bereiche entsprechen einer hohen Rauchdichte. In diesem Szenario wird die Temperatur jedoch nicht explizit modelliert bzw. transportiert. Da der Rauch jedoch entsprechend seiner

Dichte aufsteigen bzw. fallen soll, wird angenommen, dass die Temperatur proportional zur Dichte ist. Daher wird im Auftriebsterm $f_{buoyancy}$, statt der Temperatur der Zelle und der durchschnittlichen Temperatur, die Dichte der Zelle und die durchschnittliche Dichte (über alle Zellen) verwendet.
In Abbildung 4.17 ist die Rauchentwicklung an den Zeitpunkten $t_{1.5}$, $t_{2.5}$, $t_{3.5}$ und $t_{4.5}$ unter Verwendung von trilinearer Interpolation im Advektionsschritt. In Abbildung 4.18 wurde alternativ im Advektionsschritt eine Kosinusinterpolation verwendet. Erkennbar ist, dass bei Verwendung der Kosinusinterpolation mehr Details zum Vorschein kommen. Die trilineare Interpolation scheint eher einen glättenden Effekt zu haben.

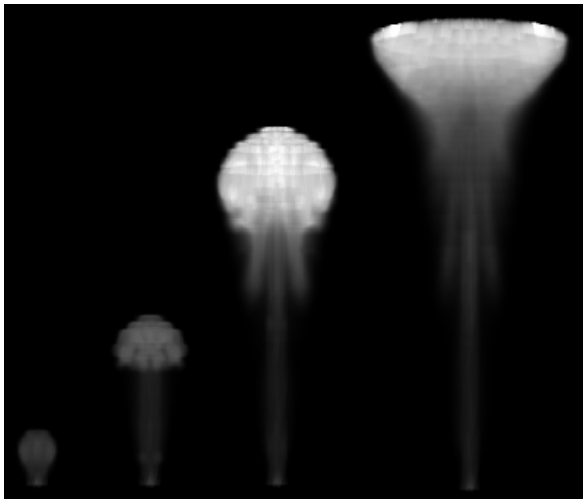


Abbildung 4.17: Advektion mit Trilinearer Interpolation bei $t_{1.5}, t_{2.5}, t_{3.5}, t_{4.5}$

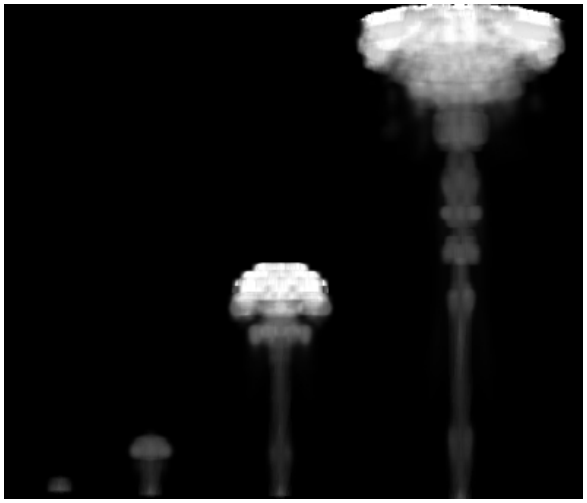
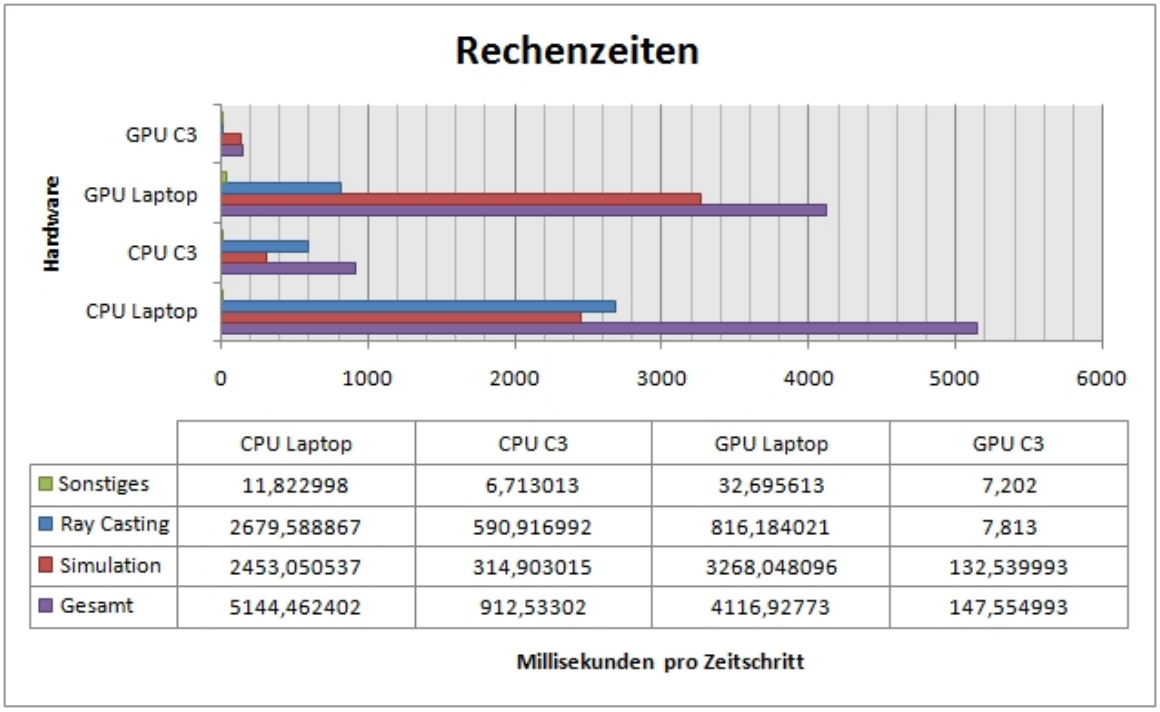


Abbildung 4.18: Advektion mit Kosinusinterpolation bei $t_{1.5}, t_{2.5}, t_{3.5}, t_{4.5}$

4.3.3 Rechenzeiten

Zuletzt wurde die Gesamtperformanz des Programms getestet. Gemessen wurden die Laufzeit für das Ray Casting, für die Simulation und die Gesamtlaufzeit des Programms pro Zeitschritt. Das Programm wurde auf der in 4.2.1 angegebenen Hardware-Umgebung und auf einem handelsüblichen Laptop (CPU: Intel Pentium T4500, 2,3Ghz, GPU: NVIDIA Geforce G105M) durchgeführt. Die Ergebnisse sind in Abbildung 4.19 und 4.20 aufgeführt. In Abbildung 4.19 ist erkennbar, dass das Programm auf einem Laptop circa fünf Sekunden pro Zeitschritt benötigt (CPU-Variante). Hierbei benötigen Simulation und Ray Casting ungefähr gleiche Zeit. Das Ausführung auf der Grafikkarte beschleunigt zwar den Ray Casting Schritt, verlangsamt allerdings sogar den Simulationsschritt. Dies ist damit zu erklären, dass die Laptop-Grafikkarte nur sehr wenige Kerne besitzt und ein GPU-Kern langsamer ist als ein CPU-Kern. Somit verwirkt der Gauss-Seidel-Löser seinen Vorsprung, da weniger Schritte parallelisiert ausgeführt werden.
Da die Grafikkarte des in 4.2.1 angegebenen Rechners (C3 genannt) über 512 Kerne verfügt, wird sowohl der Ray Casting Schritt, als auch die Simulation gut beschleunigt. Insgesamt fällt die Laufzeit von 0,9 Sekunden auf 0,15 Sekunden. Die GPU-Variante ist also circa sechs mal schneller als die CPU-Variante und es können mehrere Zeitschritte pro Laufzeitsekunde gerechnet werden.
In Abbildung 4.20 sind zusätzliche Informationen zu den einzelnen Kernelfunktionen aufgeführt. Erkennbar ist, welche Funktion die meiste Zeit auf der Grafikkarte in Anspruch nimmt, wieviele Aufrufe innerhalb von 30 Sekunden Programmlaufzeit stattfinden, wieviele Aufrufe pro Zeitschritt stattfinden und die Beschleunigungsfaktoren zusammengefasst aus den vorherigen Grafiken. Die *Buoyancy*-Funktion wurde nicht explizit bei verschiedenen Volumengrößen getestet, da diese in der Implementation speziell auf die Größe $N = 64^3$ optimiert wurde. Der *add_source*-Kernel wurde auch nicht explizit getestet, da der Feld Schreib-Test (Abbildung 4.3) äquivalent zu diesem ist. Die Tabelle zeigt, dass das Gesamtprogramm von den Kernels *lin_solve*, *calc_pixel*, *set_bnd* und *set_bnd*2 abhängt. Der Löser ist, wie bereits mehrfach erwähnt, der Flaschenhals des Programms, das Ray Casting spielt bei der Performanz kaum eine Rolle. Die Randbedingungen werden am häufigsten aufgerufen. Da dort jedoch nur $O(N^2)$ Daten verarbeitet werden, ist die Verlangsamung gegenüber der CPU-Variante verkraftbar.



	CPU Laptop	CPU C3	GPU Laptop	GPU C3
Sonstiges	11,822998	6,713013	32,695613	7,202
Ray Casting	2679,588867	590,916992	816,184021	7,813
Simulation	2453,050537	314,903015	3268,048096	132,539993
Gesamt	5144,462402	912,53302	4116,92773	147,554993

Abbildung 4.19: Rechenzeiten für die Testszenarien

Beschreibung	Kernel	% GPU-Zeit	Aufrufe (30s)	Aufrufe (pro Zeitschritt)	Beschleunigung
Gleichungslöser	lin_solve	79,14	21960	120	3,3
Ray Casting	calc_pixel	13,22	184	1	24
Advektion	advect	3,08	732	4	6,3
Randbedingungen	set_bnd	1,96	24705	131	0,36
	set_bnd2	1,89	24705	131	0,36
Vorticity Confinement	vorticityconfinement	0,24	183	1	36,42
Projektion	project2	0,16	366	2	3,8
	project	0,11	366	2	3,8
externe Kräfte	add_source	0,09	732	4	9,24
Buoyancy	sumarr	0,05	366	2	5,4
	buoyancy	0,05	183	1	5,4

Abbildung 4.20: Kernel-Informationstabelle

5 Fazit und Ausblick

In dieser Arbeit wurde eine 3D-Strömungssimulation auf Basis eines Navier-Stokes-Lösers von Jos Stam mittels der Programmiersprache CUDA auf Grafikkarten umgesetzt und parallelisiert. Die Visualisierung wurde mithilfe des Volumen Ray Casting Verfahrens ebenfalls auf der Grafikkarte umgesetzt und parallelisiert. Während das Visualisierungsverfahren durch die relative Datenunabhängigkeit der einzelnen Strahlen sich sehr gut parallelisieren ließ und prinzipiell interaktive Bildraten bei Darstellung von Volumendatensätzen erlaubt, so hängt die Geschwindigkeit der Simulation maßgeblich von der Geschwindigkeit des Lösers ab, der in der vorliegenden Implementation nur eine Beschleunigung um Faktor drei erlaubt, was auf die unkoordinierten (langsamen) Speicherzugriffe zurückzuführen ist. Außerdem ist die Leistung eines Grafikkarten-Kerns schwächer gegenüber der Leistung eines reinen CPU-Kerns.
Ein weiterer Vorteil liegt in der direkten Visualierung während der Simulation. Bei traditionellen Visualisierungsprogrammen wie beispielsweise ParaView [18] müssen die Daten zunächst in ein entsprechendes Dateiformat gespeichert werden, um sie dann einzulesen und anzuzeigen.
Leider kann das Volumen Ray Casting nur Skalarfelder visualisieren. Traditionelle Verfahren können hier durchaus auch die Geschwindigkeitsvektoren als Ganze visualisieren und nicht nur die einzelnen Komponenten.
Das Phong-Beleuchtungsmodell ist leider eher ungeeignet für das Volumenrendering.Für realistischere, aber auch rechenaufwendigere Beleuchtung auf Basis des Volumen Rendering Integrals sei an dieser Stelle auf [5] verwiesen.
Eine sinnvolle Erweiterung wäre das zusätzliche Herausschneiden einer 2D-Ebene, um eine Hybrid-Darstellung zu erhalten, sowie die Implementierung flexiblerer Transferfunktionen.
Der Navier-Stokes-Löser nach Jos Stam ist relativ leicht zu implementieren, die Idee kann auch ohne tiefere Kenntnisse über die Lösung partieller Differentialgleichungen nachvollzogen und intuitiv verstanden werden. Die Simulation kann durchaus physikalisch sinnvolle Ergebnisse erzielen, auch wenn die Qualität der Ergebnisse durchaus besser sein könnte. Ebenfalls ist fraglich, welchen Einfluss das Konvergenzverhalten des GPU-Gauss-Seidel-Kernels auf die Ergebnisse hat. Der größte Vorteil ist die Verwendung größerer Zeitschritte, leider kommt es aber genau deswegen auch zum Auftreten von numerischer Dissipation durch das verwendete Advektionsverfahren, was einen großen Nachteil darstellt. Advektionsverfahren höherer Ordnung sind beispielsweise das MacCormack-Schema [15]

oder das BFECC-Verfahren [17].
Auch durch bessere Interpolationsverfahren kann die Genauigkeit des Advektionsschritts gesteigert werden, siehe dazu [3]. Hierbei muss jedoch ein eventuelles “Overshooting“ der Daten beachtet bzw. korrigiert werden, um beispielsweise keine negativen Dichten zu erhalten.

Insgesamt haben Simulation und Visualisierung gut zusammengearbeitet, nicht zuletzt, da beide auf einem regelmäßigen kartesischen Gitter arbeiten. Wie bereits erwähnt kann das Programm technisch sinnvoll erweitert werden. Auch die Flexibilität ist durchaus verbesserungswürdig. Eine Erweiterung der grafischen Oberfläche, die bereits erwähnte 2D-Hybrid-Darstellung, das Setzen von Hindernissen und Objekten sowie die Anpassung an (nicht-kubische) Gittergrößen wären weitere sinnvolle Verbesserungen (letzteres jedoch auf Kosten der Performanz).

Abbildungsverzeichnis

Literaturverzeichnis

[1] Jos Stam, Stable Fluids, In SIGGRAPH 99 Conference Proceedings, Annual Conference Series, August 1999, 121-128

[2] Jos Stam, Real-Time Fluid Dynamics for Games. Proceedings of the Game Developer Conference, März 2003

[3] Ronald Fedkiw, Jos Stam and Henrik Wann Jensen, Visual Simulation of Smoke, In SIGGRAPH 2001 Conference Proceedings, Annual Conference Series, August 2001

[4] G. Amador,A.Gomes, Linear Solvers for Stable Fluids: GPU vs CPU, Proceedings of the 17th Encontro Portugeês de Computação Gráfica (EPCG 09),145-153,2009

[5] M. Wrenninge, Nafees Bin Zafar, Volumetric Methods in Visual Effects, SIGGRAPH 2010 Course Notes

[6] Marc Levoy, Display of Surfaces from Volume Data, IEEE Computer Graphics and Applications Vol. 8 Nr. 3, März 1988

[7] Bui Tuong Phong, Illumination for Computer Generated Pictures, Communications of the ACM 18-6,311-317, Juni 1975

[8] T. L. Kay, J. T. Kayjia, Ray tracing complex scenes, SIGGRAPH '86 Proceedings, Vol. 20, 1996

[9] Keenan Crane, Ignacio Llamas, Sarah Tariq, Real-Time Simulation and Rendering of 3D Fluids, GPU Gems 3, August 2007

[10] Thanh Vi Bach, GPU-Based Volume Ray Casting with Advanced Illumination, 2009, `http://ganymed.imib.rwth-aachen.de/deserno/seminare/bv_2009-07.pdf`, Abrufdatum 16.09.2011

[11] NVIDIA CUDA Programming Guide 4.0, `http://developer.download.nvidia.com/compute/cuda/4_0/docs/NVIDIA_CUDA_Programming_Guide_4.0.pdf`, Abrufdatum 16.09.2011

[12] http://www.geeks3d.com/20100606/gpu-computing-nvidia-cuda-compute-capability-comparative-table, Abrufdatum 16.09.2011

[13] David Kirk, Wen-mei Hwu, Lecture Notes of Course "ECE 498 AL: Applied Parallel Programming" at Universiy Of Illinois, 2010

[14] Prof. Dr. Winfried Kurth, Skript Praktikum Computergrafik, http://www.uni-forst.gwdg.de/~wkurth/cg09_v14.pdf, 2009, Abrufdatum 16.09.2011

[15] Selle, A., R. Fedkiw, B. Kim, Y. Liu, and J. Rossignac., An Unconditionally Stable MacCormack Method, Journal of Scientific Computing, 2007

[16] Wikipedia-Artikel über Volumen Ray Casting,http://en.wikipedia.org/wiki/Volume_ray_casting, Abrufdatum 16.09.2011

[17] Byungmoon Kim, Yingjie Liu, Ignacio Llamas, Jarek Rossignac,FlowFixer: Using BFECC for Fluid Simulation,Eurographics Workshop on Natural Phenomena, 2005

[18] ParaView - Open Source Scientific Visualization, http://www.paraview.org/, Abrufdatum 16.09.2011