TU Berlin
Institut für Mathematik
Institut für Numerische Fluiddynamik

Bachelorarbeit

Lösung der zweidimensionalen Wirbeltransportgleichung auf NVIDIA Grafikkarten

Manuel Baumann
Matrikelnr.: 314394
mbaumann@math.tu-berlin.de

Pavel Buran
Matrikelnr.: 314728
p.buran@gmx.de

Erster Gutachter: Prof. Dr. G. Bärwolff
Zweiter Gutachter: Prof. Dr. J. Sesterhenn

06. Dezember 2010

Inhaltsverzeichnis

Die selbstständige und eigenhändige Anfertigung versichere ich an Eides statt.

Berlin, den

Notations- und Abkürzungstabelle

B	Bandbreite
b_h	Randwertevektor in lexikographischer Anordnung
β	thermischer Ausdehnungskoeffizient
$\vec{c}$	Vektor der Geschwindigkeit
c_w	Wärmekapazität
E_n	Einheitsmatrix der Dimension n
$\vec{f}$	konservative Kräfte
g	allgemeine Vorgabe auf Neumann-Rand
Γ	Rand des Diskretisierungsgebiets
h	Schrittweite (in x_1- und x_2-Richtung)
$\vec{n}$	äußerer Normalenvektor
λ	Wärmeleitfähigkeit
ν	kinematische Viskosität
ω	Wirbelstärke
Ω	Diskretisierungsgebiet (mit Innerem $\overset{\circ}{\Omega} = \Omega \backslash \Gamma$)
P	Rechenleistung in $Flops/s$
p	Druck
Ψ	Stromfunktion
φ	allgemeine Vorgabe auf Dirichlet-Rand
$\dot{Q}$	Wärmestrom
$\dot{q}$	spezifischer Wärmestrom
ρ	Dichte
t	Zeit
δt	Zeitschrittweite
S	Entropie
s	spezifische Entropie
T	Temperatur
u	x_1-Koordinate des Geschwindigkeitsvektors
v	x_2-Koordinate des Geschwindigkeitsvektors
CG	Conjugate Gradients
FVM	Finite-Volumen-Methode
LGS	Lineares Gleichungssystem
pDGL	partielle Differentialgleichung
SPD	symmetrisch positiv definit

Abstract

In the field of Computational Fluid Dynamics (CFD), the Navier-Stokes equation fully describes the flow behavior of fluids and gases. Since the Navier-Stokes equation is a nonlinear partial differential equation (PDE), an analytical solution for arbitrary boundary conditions is unknown.
Therefore, numerical methods are used to approximate the solution on discrete nodes of a mesh.

In this work, we present a two-dimensional flow solver using the Method of Finite Volumes on an equidistant rectangular mesh. Assuming an incompressible plane flow, the Navier-Stokes equation can be written in the coordinates of stream function Ψ and vorticity ω. Furthermore, an equation for the heat distribution T in moving fluids is solved using the same mathematical methods.

After the discretization of a PDE it is usually necessary to solve a high-dimensional linear equation system of the form $Ax = b$. In our application, the matrix A is a symmetric positive definite band matrix for which the well-known Conjugate Gradient Method (CG algorithm) can be used to solve the linear equation system.
Since the dimension of A can amount to a couple of millions in technical applications, Parallel Computing is used to develop an efficient implementation of the CG algorithm.

Hence, the CUDA technology of the NVIDIA corporation shows excellent performance results, the parallel design of Graphics Processing Units (GPU) can be used to combine Parallel Computing on GPUs with a sequential *C++* code as described in [10] and [11].

After the introduction, we will derive the $\Psi - \omega$-formulation from the Navier-Stokes equation in **Chapter 2**. **Chapter 3** consists of a description of the numerical methods which transform the coupled PDE system to a system of linear equations using the Method of Finite Volumes in space and an implicite Euler approximation in time discretization.
Chapter 4 includes a short introduction to the CUDA technology and evaluates the performance of GPUs by means of the example of a sparse matrix vector multiplication as well as the inner product of two vectors. These operations were then used in the CG algorithm.
Our implementation is tested on a NVIDIA *Tesla C1060* with four test set-ups in **Chapter 5**.

1 Einleitung

1.1 Problemstellung und Motivation

Partielle Differentialgleichungen (pDGL) spielen in der mathematischen Modellierung von physikalischen Vorgängen eine zentrale Rolle, da sie das Verhalten einer Funktion $f(\vec{x}, t)$, die von mehreren Variablen abhängt, mit deren zeitlicher und räumlicher Änderung in Beziehung setzt. Im zweidimensionalen Raum ist eine allgemeine Form einer pDGL gegeben durch

$$F\left(\vec{x}, t, f(\vec{x}, t), \frac{\partial f(\vec{x}, t)}{\partial x_1}, \frac{\partial f(\vec{x}, t)}{\partial x_2}, \frac{\partial f(\vec{x}, t)}{\partial t}, \frac{\partial^2 f(\vec{x}, t)}{\partial x_1 x_2}, ...\right) = 0,$$

wobei F eine beliebige Funktion ist.
Treten in einer solchen partiellen Differentialgleichung zusätzlich Nichtlinearitäten auf, so gibt es keine allgemeine mathematische Lösungstheorie, so dass numerische Verfahren angewandt werden um die Lösung der pDGL im Diskreten zu approximieren und den zuvor modellierten Vorgang zu simulieren.

Im Bereich der numerischen Strömungsmechanik stellt die Navier-Stokes Gleichung eine pDGL mit einer Nichtlinearität dar, die das Strömungsverhalten von Flüssigkeiten und Gasen beschreibt.

In dieser Arbeit sollen unter Einführung der Stromfunktion Ψ und der Wirbelstärke ω inkompressible ebene Strömungen berechnet werden. Durch diese Substitution kann die Navier-Stokes Gleichung in eine Poissongleichung und die so genannte **Wirbeltransportgleichung** überführt werden. Darüber hinaus wird die Temperaturverteilung in bewegten Fluiden durch die **Wärmeleitungsgleichung** simuliert, die mit Hilfe der Boussinesq-Approximation in der Wirbeltransportgleichung berücksichtigt wird. Dieses System aus partiellen Differentialgleichungen wird auf einem rechteckigen Gebiet mit Hilfe der Finite-Volumen-Methode im Ort sowie einem impliziten Euler Einschrittverfahren in der Zeit gelöst.

Die Auflösung komplizierter Strömungsphänomene führt typischerweise auf große lineare Gleichungssysteme der Form $Ax = b$, deren Lösung sich durch den Einsatz von Grafikkarten mit Hilfe der CUDA Technologie beschleunigen lassen. CUDA ist eine 2006 von NVIDIA zur Verfügung gestellte Programmierumgebung zur parallelen Programmierung auf Grafikkarten. Im zweiten Teil dieser Arbeit werden daher elementare Operationen der linearen Algebra sowie ein parallelisiertes CG-Verfahren zur Lösung von linearen Gleichungssystemen mit dünnbesetzter Bandmatrix implementiert und analysiert.

1.2 Gliederung

Ausgehend von der Navier-Stokes Gleichung werden in **Kapitel 2** die physikalischen Größen Wirbelstärke ω und Stromfunktion Ψ eingeführt und damit die Wirbeltransportgleichung für ebene inkompressible Strömungen sowie die Wärmeleitungsgleichung für Fluide hergeleitet. Anschließend wird die Temperaturabhängigkeit des Strömungsverhaltens unter Annahme der Boussinesq-Approximation in der Wirbeltransportgleichung berücksichtigt.

In **Kapitel 3** werden die verwendeten numerischen Methoden zur Lösung der Differentialgleichungen erläutert. Hierbei wird nach einer kurzen theoretischen Einführung die Methode der Finiten Volumen zur Ortsdiskretisierung angewandt sowie ein implizites Euler Einschrittverfahren für die Zeitintegration benutzt. Insbesondere wird in diesem Kapitel darauf eingegangen, inwiefern sich die Differentialgleichungen als gekoppeltes System verbinden lassen und wie sich Randbedingungen berücksichtigen lassen. Um die dabei entstehenden großen linearen Gleichungssysteme zu lösen, wird in **Kapitel 4** die CUDA Technologie vorgestellt und beschrieben, wie man damit auf NVIDIA Grafikkarten rechnen kann. Dabei wurden insbesondere das Skalarprodukt und eine Sparsematrix-Vektor Multiplikation implementiert und schrittweise optimiert. Die unterschiedlichen Versionen wurden auf einer *Tesla C1060* miteinander verglichen und für eine effiziente, hoch parallele Implementierung des CG-Verfahrens verwendet.
In **Kapitel 5** werden numerische Tests zur Validierung des implementierten Strömungslösers durchgeführt. Unter anderem am Beispiel des *Driven Cavity* Problems werden hier numerische Simulationsergebnisse präsentiert.
Abschließend wird ein Ausblick auf die Entwicklung der Grafikkarten und deren Einsatz in der numerischen Mathematik gegeben.

2 Strömungsmechanische Grundlagen

In diesem Abschnitt werden die für die spätere Simulation benötigten Differentialgleichungen aus der Navier-Stokes Gleichung unter der Annahme einer inkompressiblen ebenen Strömung hergeleitet. Dabei haben wir uns im Wesentlichen an den Ausführungen in [15] und [1] orientiert.

2.1 Die Navier-Stokes Gleichung

Für inkompressible Newton-Fluide wird die Navier-Stokes Gleichung zum Beispiel in [15] aus dem Impulssatz hergeleitet. Sie ist gegeben durch:

$$\frac{\partial \vec{c}}{\partial t} + \vec{c} \cdot \ grad\ \vec{c} = \vec{f} - \frac{1}{\rho}\ grad\ p + \nu \Delta \vec{c}, \tag{1}$$

wobei $\vec{c} \in \mathbb{R}^{2,3}$ hier der Vektor der Geschwindigkeit ist.

Aus mathematischer Sicht handelt es sich hierbei um eine nichtlineare partielle Differentialgleichung zweiter Ordnung, die zusammen mit geeigneten Randbedingungen grundsätzlich die Berechnung von instationären Strömungen ermöglicht. Dabei ist $\vec{f}$ der Vektor der äußeren Kräfte und die Viskosität ν und die Dichte ρ sind skalare Stoffparameter des betrachteten Fluides.
Vor allem auf Grund der Nichtlinearität des Konvektionsterms $\vec{c}\ (\nabla \vec{c})$ existiert allerdings bisher keine geschlossene analytische Lösung der Navier-Stokes Gleichung, weshalb numerische Verfahren angewandt werden, um die Lösung im Diskreten zu approximieren.

2.2 Stromfunktion und Wirbelstärke für ebene Strömungen

Bei **ebenen Strömungen** handelt es sich um Strömungen, bei denen eine kartesische Koordinate Null ist und die anderen beiden Koordinaten von der dritten Ortskoordinate nicht abhängen. Man kann also schreiben:

$$\vec{c} = \begin{pmatrix} u(x_1, x_2, t) \\ v(x_1, x_2, t) \\ 0 \end{pmatrix}$$

Diese Annahme, zusammen mit der vorausgesetzten Inkompressibilität des betrachteten Fluides, vereinfacht die Kontinuitätsgleichung:

$$\frac{\partial \rho}{\partial t} + div\ (\rho \vec{c}) = 0$$

zu

$$\frac{\partial u}{\partial x_1} + \frac{\partial v}{\partial x_2} = div\ \vec{c} = 0. \tag{2}$$

Die **Wirbelstärke** $\vec{\omega}$ einer Strömung ist definiert als Rotation der Geschwindigkeit $\vec{c}$:

$$\vec{\omega} := rot\,(\vec{c})$$

Wirbelfreie Strömungen bezeichnet man auch als Potentialströmungen, da sich diese durch ein Potential beschreiben lassen; während Strömungen, deren Wirbelstärke nicht verschwindet, als wirbelbehaftete Strömungen bezeichnet werden.

Unter der Annahme einer ebenen Strömung ergibt sich die Wirbelstärke zu:

$$rot\,\vec{c} = \begin{pmatrix} \frac{\partial}{\partial x_1} \\ \frac{\partial}{\partial x_2} \\ \frac{\partial}{\partial x_3} \end{pmatrix} \times \begin{pmatrix} u \\ v \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \frac{\partial v}{\partial x_1} - \frac{\partial u}{\partial x_2} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \omega(x_1, x_2, t) \end{pmatrix}$$

Die Wirbelstärke hat also nur eine von Null verschieden Komponente, die im Folgenden als skalarwertige Größe ω behandelt wird:

$$\omega = \frac{\partial v}{\partial x_1} - \frac{\partial u}{\partial x_2} \tag{3}$$

Darüber hinaus wird die **Stromfunktion** $\Psi(x_1, x_2)$ für ebene Strömungen wie folgt definiert:

$$u := \frac{\partial \Psi}{\partial x_2} \qquad\qquad v := -\frac{\partial \Psi}{\partial x_1} \tag{4}$$

Auf Grund dieser Konstruktion wird die Kontinuitätsgleichung (2) implizit erfüllt. Der Name Stromfunktion rührt daher, dass die Stromfunktion entlang der Stromlinen konstant ist.

2.3 Die Ψ-ω-Formulierung der Navier-Stokes Gleichung

Setzt man die Definition der Stromfunktion (4) in die Gleichung für die Wirbelstärke (3) ein, so ergibt sich:

$$\omega = \frac{\partial v}{\partial x_1} - \frac{\partial u}{\partial x_2} \overset{(4)}{=} -\frac{\partial}{\partial x_1}(\frac{\partial \Psi}{\partial x_1}) - \frac{\partial}{\partial x_2}(\frac{\partial \Psi}{\partial x_2}) = -\Delta\Psi \tag{5}$$

Diese Gleichung $-\Delta\Psi = \omega$ wird als Poissongleichung bezeichnet und stellt den ersten Teil des Differentialgleichungssystems dar.

Im zweiten Teil der Herleitung wird die Navier-Stokes Gleichung (1) komponentenweise betrachtet, so dass gilt:

$$\frac{du}{dt} = \frac{\partial u}{\partial t} + u \cdot \frac{\partial u}{\partial x_1} + v \cdot \frac{\partial u}{\partial x_2} = -\frac{\partial}{\partial x_1}(U + \frac{p}{\rho}) + \nu(\underbrace{\frac{\partial^2 u}{\partial {x_1}^2} + \frac{\partial^2 u}{\partial {x_2}^2}}_{=\Delta u})$$

$$\frac{dv}{dt} = \frac{\partial v}{\partial t} + u \cdot \frac{\partial v}{\partial x_1} + v \cdot \frac{\partial v}{\partial x_2} = -\frac{\partial}{\partial x_2}(U + \frac{p}{\rho}) + \nu(\underbrace{\frac{\partial^2 v}{\partial {x_1}^2} + \frac{\partial^2 v}{\partial {x_2}^2}}_{=\Delta v})$$

Differenziert man anschließend die zweite Gleichung nach x_1 und subtrahiert davon die nach x_2 differenzierte erste Gleichung, so erhält man:

$$\begin{aligned}
&\frac{\partial}{\partial t}\underbrace{(\frac{\partial v}{\partial x_1} - \frac{\partial u}{\partial x_2})}_{=\omega} + \underbrace{u \cdot \frac{\partial^2 v}{\partial {x_1}^2} - u \cdot \frac{\partial^2 u}{\partial x_1 \partial x_2}}_{=u \cdot \frac{\partial \omega}{\partial x_1}} + \underbrace{v \cdot \frac{\partial^2 v}{\partial x_1 \partial x_2} - v \cdot \frac{\partial^2 u}{\partial {x_2}^2}}_{=v \cdot \frac{\partial \omega}{\partial x_2}} \\
&\qquad + \underbrace{\frac{\partial u}{\partial x_1} \cdot \frac{\partial v}{\partial x_1} + \frac{\partial v}{\partial x_1} \cdot \frac{\partial v}{\partial x_2} - \frac{\partial u}{\partial x_2} \cdot \frac{\partial u}{\partial x_1} - \frac{\partial v}{\partial x_2} \cdot \frac{\partial u}{\partial x_2}}_{=0 \text{ (folgt aus der Definition von } \Psi)} \\
&= \underbrace{-\frac{\partial^2}{\partial x_1 \partial x_2}(U + \frac{p}{\rho}) + \frac{\partial^2}{\partial x_2 \partial x_1}(U + \frac{p}{\rho})}_{=0 \text{ (Satz von Schwarz)}} + \underbrace{\nu(\frac{\partial}{\partial x_1}\Delta v - \frac{\partial}{\partial x_2}\Delta u)}_{=\nu\Delta\omega \text{ (Linearität)}}
\end{aligned}$$

Somit folgt aus der obigen Rechnung die **Wirbeltransportgleichung**:

$$\frac{\partial \omega}{\partial t} + \vec{c}\ grad\, \omega = \nu \Delta \omega \tag{6}$$

Insgesamt ergibt sich eine äquivalente Formulierung der Navier-Stokes Gleichung, in der nicht wie üblich Geschwindigkeit und Druck gesucht sind, sondern Gleichungen für Stromfunktion und Wirbelstärke gelöst werden. Die Divergenzfreiheit der Geschwindigkeit wird dabei durch die Definition von Ψ identisch erfüllt. Um später Werte für die Geschwindigkeit zu erhalten, werden die differentiellen Beziehungen (4) geeignet verwendet.

$$\boxed{\begin{array}{c} -\Delta \Psi = \omega \\ \frac{\partial \omega}{\partial t} + \vec{c}\ grad\, \omega = \nu \Delta \omega \end{array}}$$

2.4 Die Wärmeleitungsgleichung

Zusätzlich zur Geschwindigkeitsverteilung soll die Temperaturverteilung simuliert werden. Dazu muss sowohl die Temperatur in die obigen Gleichungen eingebaut werden, als auch die **Wärmeleitungsgleichung im bewegten Körper** erfüllt werden.

2.4.1 Boussinesq-Annahme

Bisher wurde von einer konstanten Dichte ρ_0 ausgegangen, die jetzt allerdings um einen temperaturabhängigen Term $\delta\rho$ erweitert wird. Die Boussinesq-Approximation wird beispielsweise in [1] eingeführt und besagt, dass für Trägheits- und Reibungskräfte weiterhin angenommen werden kann, dass der temperaturinduzierte Dichteunterschied vernachlässigbar ist. Für die Trägheits- und Reibungsterme gilt also:

$$\rho = \rho_0 + \delta\rho \approx \rho_0 \Leftrightarrow \delta\rho \ll 1 \tag{7}$$

2.4.2 Berücksichtigung in der Wirbeltransportgleichung

Die Navier-Stokes Gleichung (1) kann komponentenweise geschrieben werden als:

$$\rho \frac{dc_i}{dt} = -\frac{\partial p}{\partial x_i} + \rho \nu \Delta c_i + \rho g_i, \qquad i = 1, 2$$

wobei als einzige äußere Kraft die Gewichtskraft des Fluids angenommen wird, so dass $\vec{f} = \rho \vec{g} = \rho [g_1, g_2]^T = [0, -\rho g]^T$ gilt.

Da die Dichte nun temperaturabhängig ist, ergibt sich $\rho = \rho_0 + \delta\rho$ mit zugehörigem Druck $p = p_0 + \delta p$. Berücksichtigt man diese Erweiterung und wendet zudem die Boussinesq-Approximation an, so folgt:

$$\rho_0 \frac{dc_i}{dt} = -\frac{\partial p}{\partial x_i} + \rho_0 \nu \Delta c_i + \rho g_i, \qquad i = 1, 2$$

Ergänzt man in dieser Gleichung auf geeignete Weise eine Null, so erhält man:

$$\rho_0 \frac{dc_i}{dt} = -\frac{\partial}{\partial x_i}(p - \rho_0(g_1 x_1 + g_2 x_2)) + (\rho - \rho_0) g_i + \rho_0 \nu \Delta c_i, \qquad i = 1, 2$$

$$\Leftrightarrow \frac{dc_i}{dt} = -\frac{1}{\rho_0}\frac{\partial}{\partial x_i}(p - \rho_0(g_1 x_1 + g_2 x_2)) + \frac{\rho - \rho_0}{\rho_0} g_i + \nu \Delta c_i, \qquad i = 1, 2$$

Geht man weiterhin davon aus, dass die temperaturinduzierte Dichteänderung klein ist, so kann linearisiert werden:

$$\rho = \rho_0 + \left(\frac{\partial \rho}{\partial T}\right)_p \cdot (T - T_0) = \rho_0 - \rho_0 \beta (T - T_0),$$

wobei $\beta := -\frac{1}{\rho_0}\left(\frac{\partial \rho}{\partial T}\right)_p$ als thermischer Ausdehnungskoeffizient in [1] eingeführt wird.

Daraus folgt die Beziehung:

$$\frac{\rho_0 - \rho}{\rho_0} = \beta (T - T_0) \tag{8}$$

Ersetzt man in der obigen Gleichung nun den Dichtequotienten gemäß Gleichung (8), so erhält man:

$$\frac{dc_i}{dt} = -\frac{1}{\rho_0}\frac{\partial}{\partial x_i}(p - \rho_0(g_1 x_1 + g_2 x_2)) - \beta g_i (T - T_0) + \nu \Delta c_i, \qquad i = 1, 2$$

Wenn man nun diese beiden Gleichungen geeignet subtrahiert, so folgt mit der Definition der Wirbelstärke:

$$\frac{d\omega}{dt} = -\beta \left(\frac{\partial}{\partial x_1}(g_2(T - T_0)) - \frac{\partial}{\partial x_2}(g_1(T - T_0)) \right) + \nu \Delta \omega$$

Schreibt man nun das totale Differential auf der linken Seite aus und berücksichtigt, dass in einem kartesischen Koordinatensystem $g_1 = 0$ und $g_2 = -g$ gilt, so erweitert sich die Wirbeltransportgleichung (6) zu:

$$\boxed{\frac{\partial \omega}{\partial t} + \vec{c}\, grad\, \omega = \nu \Delta \omega + \underbrace{\beta g}_{:=C} \frac{\partial T}{\partial x_1}} \tag{9}$$

2.4.3 Wärmeleitung in bewegten Fluiden

Nach dem Zweiten Hauptsatz der Thermodynamik gilt für die Änderung der Entropie eines infinitesimal kleinen Stücks in einem abgeschlossenene System

$$\frac{dS}{dt} = \frac{\dot{Q}}{T} + \dot{S}_{gen} \tag{10}$$

Nimmt man zusätzlich einen reversiblen (umkehrbaren) Prozess an, so gilt $\dot{S}_{gen} = 0$, d.h. es wird keine Entropie durch Reibung erzeugt.

Dabei kann die Entropieänderung durch ein Volumenintegral dargestellt und das totale Differential ausgeschrieben werden:

$$\frac{dS}{dt} = \frac{d}{dt} \int_V \rho s dV = \int_V \left(\frac{\partial \rho s}{\partial t} + c_i \frac{\partial \rho s}{\partial x_i} \right) dV$$

$\dot{Q}$ bezeichnet hierbei die Wärmeänderung, die durch Wärmezufuhr entsteht. Da im Innern unseres Systems keine Quellen oder Senken sind, kann weiter angenommen werden, dass die Wärme nur über die Oberfläche des infitisimal kleinen Stücks übertragen wird:

$$\dot{Q} = \int_A \dot{q} n_i dA_i = - \int_A \dot{q}_i n_i dA$$

Setzt man diese Überlegungen in (10) ein, so folgt mit dem Satz von Gauß:

$$\int_V \frac{\partial \rho s}{\partial t} dV + \int_V c_i \frac{\partial \rho s}{\partial x_i} dV = -\frac{1}{T} \int_V \frac{\partial \dot{q}_i}{\partial x_i} dV$$

und somit

$$\frac{\partial \rho s}{\partial t} + c_i \frac{\partial \rho s}{\partial x_i} = -\frac{1}{T} \frac{\partial \dot{q}_i}{\partial x_i} \tag{11}$$

Darüber hinaus wird für inkompressible Fluide in [17] die kalorische Zustandsgleichung hergeleitet:

$$s = c_w \cdot ln(T)$$

Differenziert folgt:

$$\frac{\partial s}{\partial t} = c_w \frac{1}{T} \frac{\partial T}{\partial t} \qquad \text{sowie} \qquad \frac{\partial s}{\partial x_i} = c_w \frac{1}{T} \frac{\partial T}{\partial x_i}$$

Aufgrund der vorausgesetzten Inkompressibilität ergibt sich (11) somit zu:

$$\rho c_w \frac{\partial T}{\partial t} + \rho c_w c_i \frac{\partial T}{\partial x_i} = -\frac{\partial \dot{q}_i}{\partial x_i}$$

Weiterhin besagt das Grundgesetz der Wärmeleitung (auch FOURIERsches Gesetz), dass sich Temperaturgradient und Wärmestrom proportional verhalten, wobei als Proportionalitätskonstante die ortsunabhängige Wärmeleitfähigkeit λ eingeführt wird. Das Minuszeichen berücksichtigt dabei, dass Wärme in Richtung des Temperaturgefälles strömt.

$$\dot{q}_i = -\lambda \frac{\partial T}{\partial x_i}$$

Ersetzt man auf diese Weise den Term auf der rechten Seite, so erhält man:

$$\rho c_w \frac{\partial T}{\partial t} + \rho c_w c_i \frac{\partial T}{\partial x_i} = \lambda \frac{\partial^2 T}{\partial {x_i}^2} \qquad i = 1, 2$$

Und dies ergibt durch Addition der beiden Gleichungen und Einführung der Konstanten a die **Wärmeleitungsgleichung**:

$$\boxed{\frac{\partial T}{\partial t} + \vec{c}\ grad\ T = \underbrace{\frac{\lambda}{\rho c_w}}_{=:a} \Delta T} \tag{12}$$

3 Numerische Methoden

In diesem Kapitel wird beschrieben, wie das folgende System, bestehend aus Poissongleichung (5), Wirbeltransportgleichung (9) und Wärmeleitungsgleichung (12), mit der Finiten-Volumen-Methode numerisch gelöst wird:

$$\begin{aligned} -\Delta\Psi &= \omega \\ \frac{\partial\omega}{\partial t} + \vec{c}\ grad\,\omega &= \nu\Delta\omega + C\frac{\partial T}{\partial x_1} \qquad (*) \\ \frac{\partial T}{\partial t} + \vec{c}\ grad\ T &= a\Delta T \end{aligned}$$

Die **Finite-Volumen-Methode** (FVM) ist eine Bilanzmethode zur diskreten Approximation partieller Differentialgleichungen, bei der die Flüsse durch ein Simulationsgebiet Ω bilanziert werden. Auf Grund dieses Vorgehens wird beispielsweise die Massenerhaltung trotz Diskretisierungsfehler garantiert, was aus ingenieurwissenschaftlicher Sicht oft von zentralem Interesse ist. Man nennt das Verfahren daher *konservativ*.

Ein wichtiges Hilfsmittel, das bei der FVM häufig benutzt wird, stellt der **Satz von Gauß** dar:

$$\boxed{\int_{\Omega^\nu} div\,\vec{c}\,dV = \int_{\partial\Omega^\nu} \vec{c}\cdot\vec{n}\,dF} \tag{13}$$

wobei über ein Gebiet der Dimension $\nu = 2, 3$ integriert wird.

Im Fall $\nu = 2$ steht auf der rechten Seite ein Linienintegral, wobei $\vec{n}$ der äußere Normalenvektor auf dem (stückweise glatten) Rand $\partial\Omega^\nu =: \Gamma$ ist. Dieses lässt sich in der Regel mit einfachen Methoden approximieren.

3.1 Ortsdiskretisierung

Das Gebiet Ω wird in endlich viele Teilstücke zerlegt, wobei $\Omega = \bigcup_{i\in I, j\in J}\Omega_{i,j}$ gelten soll und $\Omega_{i,j}\cap\Omega_{i',j'}$ für $(i,j)\neq(i',j')$ eine Menge vom Maß Null ist. Hierbei sind I, J geeignete Indexmengen.

In dem hier behandelten Problem wird als Diskretisierungsgebiet stets ein rechteckiges Gebiet $\Omega\subset\mathbb{R}^2$ gewählt und in ein Gitter mit der Schrittweite h äquidistant zerlegt, so dass man N_{x_1} bzw. N_{x_2} Stützstellen in die jeweilige Koordinatenrichtung erhält (siehe Abb. 1). Die Stützstellen befinden sich dann an den Mittelpunkten der auf diese Art entstehenden Teilgebiete $\Omega_{i,j}$. Außerdem werden die zuvor erwähnten äußeren Normalenvektoren besonders einfach.

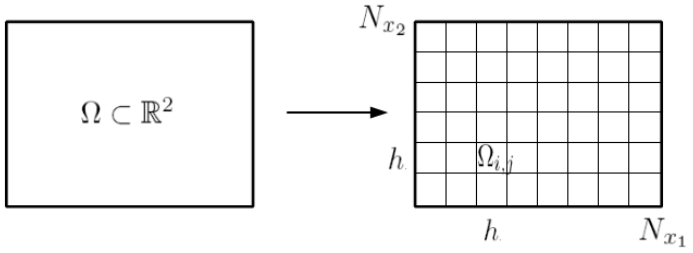

Abbildung 1: Äquidistante Diskretisierung des Gebiets Ω mit Schrittweite h

Um die Werte an den Stützstellen später geordnet in einen Vektor schreiben zu können, wird *lexikographische* Anordnung verwendet, bei der einem Tupel $(i, j) \in \mathbb{N} \times \mathbb{N}$ eine Identifikationsnummer $l \in \mathbb{N}$ via

$$\begin{aligned} lexi : \mathbb{N} \times \mathbb{N} &\to \mathbb{N} \\ (i, j) &\mapsto (i-1) \cdot N_{x_1} + j \end{aligned}$$

zugeordnet wird. Werden im Folgenden Werte in äquidistanter Ordnung in einen Vektor oder eine Matrix geschrieben, so werden die jeweiligen Objekte mit einem h indiziert.

3.1.1 Diskretisierung der Poisson-Gleichung

Das in Kapitel 2 hergeleitete Problem (5) ist mathematisch vollständig beschrieben durch die Vorgabe von Randwerten; etwa

$$\begin{gathered} -\Delta\Psi = \omega \text{ in } \overset{\circ}{\Omega} \\ \Psi = \varphi \text{ auf } \Gamma_D \text{ sowie } \tfrac{\partial\Psi}{\partial\vec{n}} = g \text{ auf } \Gamma_N \end{gathered}$$

wobei $\partial\Omega = \Gamma_N \cup \Gamma_D$ aus Neumann- bzw. Dirichleträndern besteht.

An dieser Stelle wird nur das Innere des Gebietes Ω betrachtet, konkrete Randwerte werden in 3.2 definiert und in der Diskretisierung mit berücksichtigt.

Wendet man nun die FVM auf (5) an, so integriert man zunächst über Ω und erhält:

$$-\int_\Omega \Delta\Psi \, d\Omega = \int_\Omega \omega \, d\Omega$$

Mit der zuvor beschriebenen äquidistanten Diskretisierung ist dies äquivalent zu

$$\sum_{i\in I, j\in J} -\int_{\Omega_{i,j}} \Delta\Psi \, d\Omega_{i,j} = \sum_{i\in I, j\in J} \int_{\Omega_{i,j}} \omega \, d\Omega_{i,j},$$

da Ω disjunkt zerlegt wurde.

Betrachtet man nun ein finites Volumen $\Omega_{i,j}$ (vgl. Abb. 2) und wendet den Satz von Gauß geeignet an, so folgt:

$$
\begin{aligned}
-\int_{\Omega_{i,j}} \Delta\Psi \, d\Omega_{i,j} &= \int_{\Omega_{i,j}} \omega \, d\Omega_{i,j} \\
\Leftrightarrow -\int_{\partial\Omega_{i,j}} grad\,\Psi \cdot \vec{n} \; d\partial\Omega_{i,j} &= \int_{\Omega_{i,j}} \omega \; d\Omega_{i,j}
\end{aligned}
$$

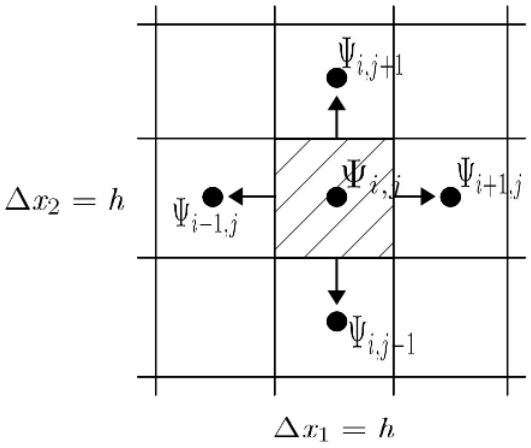

Abbildung 2: FVM mit Stützstellen im Mittelpunkt der Zellen

Da ein Element $\Omega_{i,j}$ bei dieser Diskretisierung ein Quadrat mit der Seitenlänge h ist, kann das Flächenintegral auf der rechten Seite durch

$$\int_{\Omega_{i,j}} \omega \; d\Omega_{i,j} \approx h^2 \omega_{i,j}$$

approximiert werden, wobei $\omega_{i,j}$ der Wert der Wirbelstärke im Mittelpunkt des Teilgebiets $\Omega_{i,j}$ ist.

Zur Diskretisierung des linken Terms stellt man zunächst fest, dass $\partial\Omega_{i,j} = \Gamma_{i,j} = \Gamma^N_{i,j} \cup \Gamma^O_{i,j} \cup \Gamma^S_{i,j} \cup \Gamma^W_{i,j}$ gilt, wobei die hochgestellten Indizes für Himmelsrichtungen stehen. Auf diese Art sieht man in Abbildung 2 leicht, dass für die zugehörigen Normalenvektoren gilt:
$\vec{n}^N_{i,j} = [0,1]^T, \vec{n}^O_{i,j} = [1,0]^T, \vec{n}^S_{i,j} = [0,-1]^T, \vec{n}^W_{i,j} = [-1,0]^T$.

Dies führt zu folgender kanonischen Approximation für die linke Seite der Poisson-Gleichung:

$$
\begin{aligned}
-\int_{\Gamma_{i,j}} \frac{\partial\Psi}{\partial\vec{n}} \, d\Gamma_{i,j} &= -\int_{\Gamma_N} \frac{\partial\Psi}{\partial x_2} \, dx_1 - \int_{\Gamma_O} \frac{\partial\Psi}{\partial x_1} \, dx_2 + \int_{\Gamma_S} \frac{\partial\Psi}{\partial x_2} \, dx_1 + \int_{\Gamma_W} \frac{\partial\Psi}{\partial x_1} \, dx_2 \\
&\approx -\frac{\Psi_{i,j+1} - \Psi_{i,j}}{h} \cdot h - \frac{\Psi_{i+1,j} - \Psi_{i,j}}{h} \cdot h + \frac{\Psi_{i,j} - \Psi_{i,j-1}}{h} \cdot h + \frac{\Psi_{i,j} - \Psi_{i-1,j}}{h} \cdot h \\
&= -\Psi_{i,j+1} - \Psi_{i+1,j} + 4\Psi_{i,j} - \Psi_{i,j-1} - \Psi_{i-1,j} \qquad (14)
\end{aligned}
$$

Insgesamt erhält man also einen **klassischen 5-Punkte-Stern**[1] für die Diskretisierung der Poissongleichung (5) im Innern von Ω:

$$h^{-2} \begin{pmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{pmatrix} \Psi_h = \omega_h$$

Wählt man nun die lexikographische Anordnung, die einem Tupel (i, j) eine Zell-ID $l \in \mathbb{N}$ zuordnet, so erhält man eine komplette Diskretisierung des Laplace Operators auf $\mathring{\Omega}$ durch die Matrix A_h via:

$$A_h = h^{-2} \begin{bmatrix} T_A & -E_{N_{x_1}} & & \\ -E_{N_{x_1}} & T_A & \ddots & \\ & \ddots & \ddots & -E_{N_{x_1}} \\ & & -E_{N_{x_1}} & T_A \end{bmatrix} \in \mathbb{R}^{N,N}, \; T_A = \begin{bmatrix} 4 & -1 & & \\ -1 & 4 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 4 \end{bmatrix} \in \mathbb{R}^{N_{x_1}, N_{x_1}},$$

wobei $N := N_{x_1} \cdot N_{x_2}$.

Es ist also das lineare Gleichungssystem $A_h \Psi_h = \omega_h$ zu lösen, wobei die Berücksichtigung von Randwerten auf der Rechten Seite in Kapitel 3.2 erläutert wird.

Zur Lösung dieses linearen Gleichungssystems wird das in Kapitel 4.4 behandelte CG-Verfahren angewendet. Dieses hat als notwendige Voraussetzung, dass die Matrix A_h symmetrisch positiv definit (SPD) ist. Die Symmetrie von A_h folgt offensichtlich per Konstruktion; es gilt also $A_h = A_h^T$.
Um zu zeigen, dass A_h positiv definit ist, d.h. $x^T A_h x > 0 \; \forall x \in \mathbb{R}^N \backslash 0$ gilt, reicht es nach [6], Satz 4.3.24 aus, zu zeigen, dass

- A_h ist symmetrisch,
- die Diagonalelemente von A_h sind positiv: $a_{i,i} > 0$ für $1 \leq i \leq N$,
- A_h ist irreduzibel diagonaldominant.

Punkt 1 und 2 folgen direkt aus der Konstruktion von A_h. Zu Punkt 3 ist zu sagen, dass für alle Zeilen i von A_h gilt: $\sum_{1 \leq j \leq N} |a_{i,j}| \leq |a_{i,i}|$ und beispielsweise für die erste Zeile von A_h gilt: $\sum_{1 \leq j \leq N} |a_{1,j}| = 2h^{-2} < 4h^{-2} = |a_{1,1}|$. Darüber hinaus ist A_h irreduzibel, falls zwei beliebige Indizes $i, j \in \Omega_h$ miteinander verbunden sind. Offensichtlich sind i und j direkt miteinander verbunden ($\Leftrightarrow a_{i,j} \neq 0$), falls i und j Nachbarn sind. Beliebige Indizes können dann über eine Kette von benachbarten Punkten miteinander verbunden werden.
Somit ist A_h irreduzibel diagonaldominant und folglich auch symmetrisch positiv definit.

[1] Die Bezeichnung klassischer 5-Punkte Stern ergibt sich, da man auch bei einer Finiten Differenzen Diskretisierung und äquidistanter Schrittweite zu dieser Matrix kommt.

3.1.2 Diskretisierung der Transportgleichung

Die Wirbeltransportgleichung (9) und die Wärmetransportgleichung (12) haben die selbe mathematische Struktur und können verallgemeinert geschrieben werden als:

$$\frac{\partial \zeta}{\partial t} = k_1 \Delta \zeta - \vec{c}\ grad\, \zeta + k_2 \frac{\partial T}{\partial x_1}, \tag{15}$$

mit $\zeta = \{\omega, T\}$ und $k_1 = \begin{cases} \nu & \text{für } \zeta = \omega \\ a & \text{für } \zeta = T \end{cases}$ sowie $k_2 = \begin{cases} \mathrm{C} & \text{für } \zeta = \omega \\ 0 & \text{für } \zeta = T \end{cases}$.

Wendet man auf diese partielle Differentialgleichung erneut die Methode der Finiten Volumen an, so ergibt sich nach Integration über Ω zunächst:

$$\int_\Omega \frac{\partial \zeta}{\partial t} d\Omega = \int_\Omega k_1 \Delta \zeta - \vec{c}\ grad\, \zeta + k_2 \frac{\partial T}{\partial x_1} d\Omega.$$

Auf Grund der in 3.1 beschriebenen Diskretisierung ist dies äquivalent zu:

$$\begin{aligned} \sum_{i\in I, j\in J} \int_{\Omega_{i,j}} \frac{\partial \zeta}{\partial t} d\Omega_{i,j} &= \sum_{i\in I, j\in J} \int_{\Omega_{i,j}} k_1 \Delta \zeta - \vec{c}\ grad\, \zeta + k_2 \frac{\partial T}{\partial x_1} d\Omega_{i,j} \\ &= \sum_{i\in I, j\in J} \left(\underbrace{k_1 \int_{\Omega_{i,j}} \Delta \zeta\, d\Omega_{i,j}}_{\text{Laplace-Term}} - \overbrace{\int_{\Omega_{i,j}} \vec{c}\ grad\, \zeta\, d\Omega_{i,j}}^{\text{Konvektionsterm}} + \underbrace{k_2 \int_{\Omega_{i,j}} \frac{\partial T}{\partial x_1} d\Omega_{i,j}}_{\text{Temperatur-Term}} \right) \end{aligned} \tag{16}$$

Da im Folgenden mit versetzten Gittern gearbeitet wird, soll ein Wert $\zeta_{i,j}$ im Kontrollvolumen $\Omega_{i,j}$ zukünftig mit ζ^P bezeichnet werden. Umliegende Werte werden dann gemäß der jeweiligen Himmelsrichtung beispielsweise mit ζ^N bezeichnet. Diese Notation ist an dieser Stelle zweckmäßig, da zum Beispiel benachbarte Geschwindigkeitswerte nur im Abstand von $h/2$ zu ζ^P liegen (vgl. Abb. 3).
Ein einzelnes Kontrollvolumen (finites Volumen) kann dann also geschrieben werden als: $\Omega_{i,j} = [S, N] \times [W, O]$.

Um die Transportgleichung im Innern von Ω zu diskretisieren, werden die drei in (16) benannten Terme separat betrachtet.

Laplace-Term:
Die Diskretisierung des Laplace-Terms erfolgt analog zu der in Kapitel 3.1.1 beschriebenen Vorgehensweise.

$$\begin{aligned} k_1 \int_{\Omega_{i,j}} \Delta \zeta\, d\Omega_{i,j} &= k_1 \int_{\partial\Omega_{i,j}} grad \cdot \zeta\ \vec{n}\ d\partial\Omega_{i,j} \\ &\approx k_1 (\zeta^N + \zeta^O - 4\zeta^P + \zeta^S + \zeta^W) \end{aligned}$$

Ordnet man die Werte lexikographisch an, so kann dieser Teil von (15) durch $B_h := -k_1 h^2 A_h$ diskretisiert werden.

Konvektionsterm:
Zur Diskretisierung des Konvektionsterms wird mit zwei versetzten Gittern für die Geschwindigkeitskomponenten gearbeitet. Das Gitter mit Werten der Geschwindigkeit im Norden und Süden eines Kontrollvolumens wird im Folgenden als *Gitter 1* (in Abb. 3 gestrichelt dargestellt) und das Gitter mit Werten im Osten und Westen als *Gitter 2* (gepunktet dargestellt) bezeichnet. Auf diese Weise liegen die Geschwindigkeitswerte genau an den Rändern des Kontrollvolumens $\Omega_{i,j}$ vor, wo sie in Gleichung (17) benötigt werden.

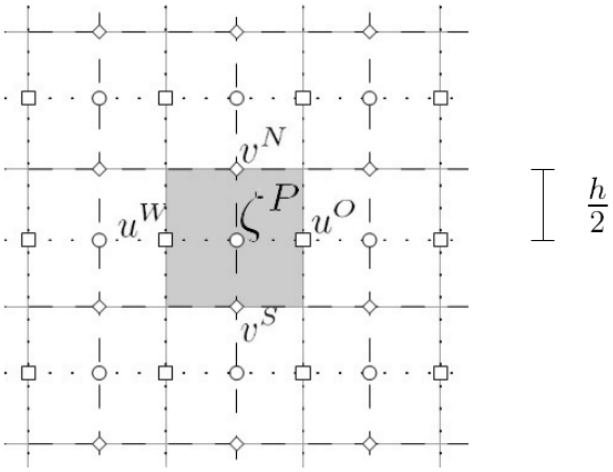

Abbildung 3: Versetztes Gitter mit Kontrollvolumen: Die Geschwindigkeitswerte liegen genau an den Rändern von $\Omega_{i,j}$.

Bei der Diskretisierung ist zu beachten, dass die entsprechenden Werte im Abstand $h/2$ von ζ^P liegen:

$$\begin{aligned}
\int_{\Omega_{i,j}} \vec{c}\ grad\,\zeta\, d\Omega_{i,j} &= \int_{\Omega_{i,j}} u\frac{\partial\zeta}{\partial x_1} + v\frac{\partial\zeta}{\partial x_2}\, d\Omega_{i,j} \\
&= \int_S^N \int_W^O u\frac{\partial\zeta}{\partial x_1}\, dx_1\, dx_2 + \int_W^O \int_S^N v\frac{\partial\zeta}{\partial x_2}\, dx_2\, dx_1 \\
&\approx \int_S^N \frac{h}{2}\left(u^O\frac{\zeta^O-\zeta^P}{h} + u^W\frac{\zeta^P-\zeta^W}{h}\right) dx_2 + \int_W^O \frac{h}{2}\left(v^N\frac{\zeta^N-\zeta^P}{h} + v^S\frac{\zeta^P-\zeta^S}{h}\right) dx_1 \\
&\approx \frac{h}{2}\left(u^O\zeta^O + (u^W-u^O)\zeta^P - u^W\zeta^W\right) + \frac{h}{2}\left(v^N\zeta^N + (v^S-v^N)\zeta^P - v^S\zeta^S\right) \qquad (17)
\end{aligned}$$

Bei dieser ersten, kanonischen Diskretisierung haben numerische Tests ergeben, dass im Innern von Ω Temperaturwerte entstanden, die größer als die vorgegebenen Randwerte waren. Da dieses Verhalten sehr unphysikalisch ist, wurde an dieser Stelle ein UPWIND-Schema implementiert. Dieses wird zum Beispiel in [16] vorgestellt und berücksichtigt, dass der Wert in der betrachteten Zelle lediglich von den in das Gebiet reinfließenden Parametern beeinflusst wird.

Eine kompakte Schreibweise von (17) kann durch Einführung von $[\![x, y]\!] := max(x, y)$ erreicht werden:

$$\begin{aligned}\int_{\Omega_{i,j}} \vec{c}\, grad\, \zeta\, d\Omega_{i,j} \approx\ & h \cdot \left(-[\![-u^O, 0]\!]\zeta^O + ([\![u^W, 0]\!] + [\![-u^O, 0]\!])\zeta^P - [\![u^W, 0]\!]\zeta^W\right) \\ & + h \cdot \left(-[\![-v^N, 0]\!]\zeta^N + ([\![v^S, 0]\!] + [\![-v^N, 0]\!])\zeta^P - [\![v^S, 0]\!]\zeta^S\right) \\ =\ & h \cdot (-[\![-v^N, 0]\!]\zeta^N - [\![-u^O, 0]\!]\zeta^O + [\![v^S, 0]\!]\zeta^S - [\![u^W, 0]\!]\zeta^W \\ & - ([\![u^W, 0]\!] + [\![-u^O, 0]\!] + [\![v^S, 0]\!] + [\![-v^N, 0]\!])\zeta^P) \end{aligned} \tag{18}$$

Dieser Ausdruck lässt sich in Matrixschreibweise an dieser Stelle nicht übersichtlich aufschreiben, aber in (18) wird deutlich, dass eine Diskretisierungsmatrix C_h Werte auf der Hauptdiagonalen, den beiden Nebendiagonalen und auf zwei weitere Diagonalen im Abstand N_{x_1} hat.

Temperatur-Term:
Für den Temperatur-Term muss eine erste Ableitung diskretisiert werden. Dies geschieht via:

$$\begin{aligned} k_2 \int_{\Omega_{i,j}} \frac{\partial T}{\partial x_1} d\Omega_{i,j} &= k_2 \int_S^N \int_W^O \frac{\partial T}{\partial x_1} dx_1\, dx_2 \\ &= \frac{k_2}{2} \int_S^N (T^O - T^W) dx_2 \approx \frac{k_2 h}{2} (T^O - T^W) \end{aligned}$$

und führt in Matrixschreibweise auf:

$$D_h = \frac{k_2 h}{2} \begin{bmatrix} T_D & & & \\ & T_D & & \\ & & \ddots & \\ & & & T_D \end{bmatrix} \in \mathbb{R}^{N^2, N^2}, \qquad T_D = \begin{bmatrix} 0 & -1 & & \\ 1 & 0 & \ddots & \\ & \ddots & \ddots & -1 \\ & & 1 & 0 \end{bmatrix} \in \mathbb{R}^{N_{x_1}, N_{x_1}},$$

wobei $N = N_{x_1} \cdot N_{x_2}$.

Nachdem alle Differentialoperatoren auf der rechten Seite von (15) diskretisiert wurden, kann das Oberflächenintegral auf der linken Seite approximiert werden durch:

$$\int_{\Omega^P} \frac{\partial \zeta}{\partial t} d\Omega^P \approx h^2 \cdot \frac{\partial \zeta}{\partial t}.$$

Will man die Gleichung dann in passender Weise für die Zeitintegration umformen, so ist es hilfreich im Weiteren folgende Matrizen zu definieren:

$$\begin{aligned} \tilde{B}_h &:= h^{-2} B_h \\ \tilde{C}_h &:= h^{-2} C_h \\ \tilde{D}_h &:= h^{-2} D_h \end{aligned}$$

3.2 Anfangswerte und Randbedingungen

Bisher wurden die Differentialgleichungen nur im Innern von Ω betrachtet. Mathematisch eindeutig formuliert sind diese Probleme allerdings erst durch die Vorgabe von Anfangswerten sowie Randbedingungen auf dem Rand von Ω. Diesen bezeichnet man häufig mit Γ und da es sich bei dem hier betrachteten Diskretisierungsgebiet um ein rechteckiges Gitter handelt, kann man den Rand des Gebietes durch $\Gamma = \Gamma_1 \cup \Gamma_2 \cup \Gamma_3 \cup \Gamma_4$, wie in Abbildung 4 angedeutet, schreiben.

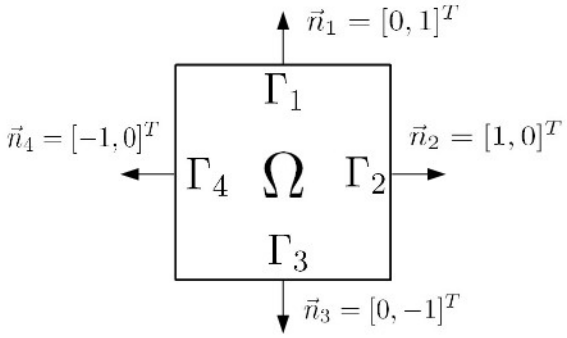

Abbildung 4: Gebiet Ω mit äußeren Normalenvektoren

Während Anfangswerte im Allgemeinen im Innern des Diskretisierungsgebiets vorgegeben werden, müssen die Randbedingungen auf geeignete Weise in die diskretisierten Gleichungen eingebaut werden. Generell unterscheidet man bei den Randbedingungen zwischen so genannten Dirichlet- und Neumann-Randbedingungen, wobei man im Falle von Dirichlet-Randbedingungen den Wert der gesuchten Funktion auf dem Rand explizit vorgibt und bei Neumann-Randbedingungen die äußere Normalenableitung vorgibt.
Im Folgenden nehmen wir an, dass $\Gamma = \Gamma_D \cup \Gamma_N$ sowie $\Gamma_D \cap \Gamma_N = \varnothing$ gelte.

3.2.1 Vorgabe und Berechnung der Anfangswerte und Randbedingungen

Für das System von Differentialgleichungen $(*)$ müssen nun Anfangswerte und Randbedingungen für die Wirbelstärke ω und die Temperatur T sowie Randbedingungen für die Stromfunktion Ψ vorgegeben werden, damit die Wirbeltransportgleichung, die Wärmeleitungsgleichung und die Poissongleichung eine eindeutige Lösung haben.
Während Anfangswerte und Randbedingungen für die Temperatur T in einem physikalischen System im Allgemeinen bekannt sind, müssen diese für die Stromfunktion Ψ und die Wirbelstärke ω durch die Vorgabe von Anfangswerten und Randbedingungen für die Geschwindigkeit $\vec{c} = [u, v]^T$ berechnet werden.
Randwerte für die Stromfunktion Ψ lassen sich, wie Abbildung 5 zeigt, durch Integration über den Rand nach Gleichung (4) berechnen. Hierzu legt man das Potential an einer Stelle fest (in der Zeichnung linke untere Ecke) und approximiert im Folgenden ein einfaches Linienintegral, zum Beispiel mit der Trapezregel. Es ist zu beachten, dass zulässige Randbedingungen für die Geschwindigkeit die Masse im Innern erhalten. Dadurch wird auch die

notwendige Bedingung erfüllt, dass bei der Integration für den Wert der Stromfunktion gilt: $\int_{\Gamma_4} u\ dy - \int_{\Gamma_1} v\ dx \stackrel{!}{=} -\int_{\Gamma_3} v\ dx + \int_{\Gamma_2} u\ dy$.

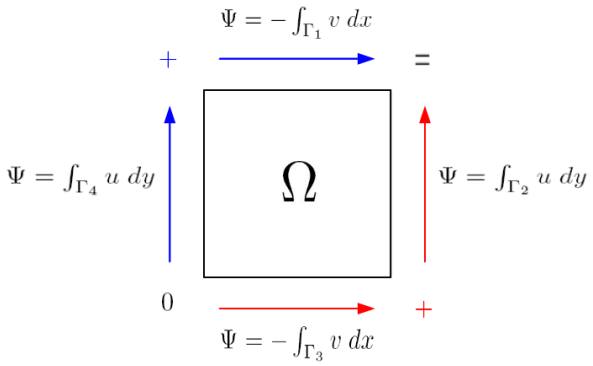

Abbildung 5: Umrechnung von Geschwindigkeitsvorgaben in Vorgaben für die Stromfunktion Ψ auf dem Rand von Ω

Die Anfangswerte und die Randwerte für die Wirbelstärke ω lassen sich nach der Gleichung $\omega = \frac{\partial v}{\partial x_1} - \frac{\partial u}{\partial x_2}$ aus den Geschwindigkeiten berechnen. Für den häufig auftretenden Fall, dass die Geschwindigkeit zu Beginn Null ist, kann man die Anfangswerte für die Wirbelstärke ebenfalls auf Null setzen.
In Abbildung 6 ist zu sehen, dass zusätzlich zu den vorgegebenen Geschwindigkeitswerten auf dem Rand ein Geschwindigkeitswert aus dem Innern und die Tangentialkomponente eines Geschwindigkeitswert außerhalb des Gebiets (*Ghost Node*) zur Berechnung der Randwerte von ω benötigt wird. Dies hat zur Folge, dass erstens die Randwerte von ω in jedem Zeitschritt neu berechnet und zweitens die Tangentialkomponente der Geschwindigkeit in den *Ghost Nodes* vorgegeben werden muss oder durch Interpolation der Randwerte berechnet wird.

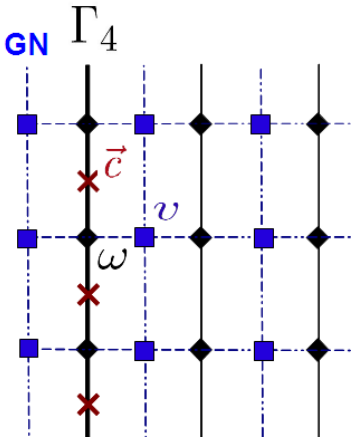

Abbildung 6: Zur Berechnung von ω auf Γ_4 werden neben den Randwerten für $\vec{c} = [u, v]^T$ auch *Ghost Nodes (GN)* für v sowie innere Punkte von v benötigt. Die GN werden in unserer Implementierung vorgegeben und mit Werten belegt als würden sie direkt auf Γ_4 liegen.

3.2.2 Berücksichtigung von Dirichlet Randbedingungen

Um zu demonstrieren wie man Dirichlet-Randbedingungen explizit in seiner Diskretisierung berücksichtigt, sei Gleichung (5) mit Randbedingungen versehen, so dass man ein Problem der Form

$$\begin{aligned} -\Delta\Psi &= \omega \quad \text{in } \overset{\circ}{\Omega} \\ \Psi &= \varphi \quad \text{auf } \Gamma_D \subseteq \Gamma \end{aligned}$$

erhält.
In Kapitel 3.1.1 wurde bereits hergeleitet, dass der Laplace-Operator im Innern von Ω durch eine Matrix A_h diskretisiert werden kann. Schreibt man die Werte von Ψ und φ in lexikographischer Anordnung (vgl. Abb. 7) in Vektoren Ψ_h und φ_h, so ist im Innern von Ω das lineare Gleichungssystem $A_h\Psi_h = \omega_h$ zu lösen.

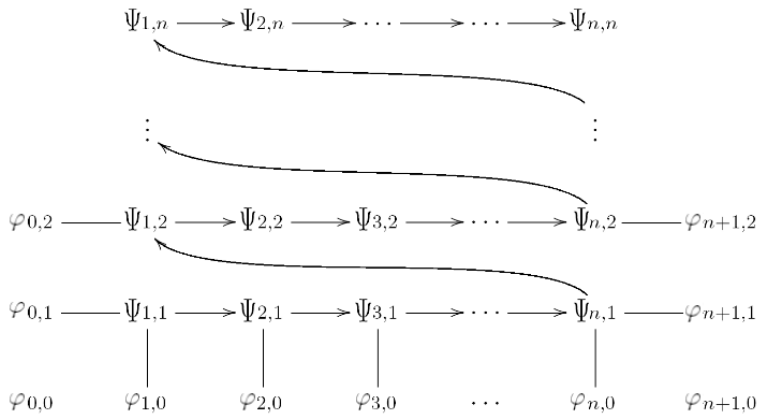

Abbildung 7: Lexikographische Anordnung mit Randbedingungen φ. Hierbei wurde vereinfachend angenommen, dass $N_{x_1} = N_{x_2} =: n$.

Um nun Dirichlet-Randwerte zu berücksichtigen, muss jedesmal, wenn in (14) ein randnaher Wert $\Psi_{i,j}$ berechnet wird auf einen entsprechenden Wert auf dem Rand von Ω zugegriffen werden. Da dieser nicht in der Matrix A_h enthalten ist, muss ein entsprechenden Vektor mit Randwerte auf die Rechte Seite addiert werden.
Man erhält somit:

$$A_h\Psi_h = \underbrace{\omega_h + \overbrace{h^{-2}[\varphi_{0,1} + \varphi_{1,0}, \varphi_{2,0}, ..., \varphi_{n-1,0}, \varphi_{n,0} + \varphi_{n+1,1}, \varphi_{0,2}, ..., \varphi_{n+1,2}, ...]^T}^{\text{Vektor mit Dirichlet-Randbedingungen } b_h^\Psi}}_{=:\tilde{\omega}_h}$$

und hat letztendlich das LGS $A_h\Psi_h = \tilde{\omega}_h$ zu lösen.

Dieses Vorgehen zur Berücksichtigung von Dirichlet-Randbedingungen lässt sich analog auf die übrigen Differentialoperatoren übertragen.

3.2.3 Berücksichtigung von Neumann Randbedingungen

Da in der hier präsentierten Anwendung Neumann-Randbedingungen ausschließlich zur Modellierung von wärmeisolierten Wänden genutzt werden, hat man stets Randbedingungen der Form

$$\frac{\partial \Psi}{\partial \vec{n}} = g = 0,$$

wobei $\vec{n}$ der in Abbildung 4 eingeführte äußere Normalenvektor ist. Auf Grund des rechteckigen Diskretisierungsgebietes ist diese Formulierung auf Γ_3 beispielsweise äquivalent zu:

$$\frac{\partial \Psi}{\partial \vec{n}} \approx \frac{\Psi_{i,1} - \varphi_{i,0}}{h} \stackrel{!}{=} 0 \qquad \Longleftrightarrow \qquad \varphi_{i,0} = \Psi_{i,1}$$

Da auf den anderen Teilrändern analog vorgegangen werden kann, können Neumann-Randbedingungen also genauso wie Dirichlet-Randbedingungen in den Differentialgleichungen berücksichtigt werden. Allerdings müssen die Neumann-Randbedingungen in jedem Zeitschritt in Dirichlet-Randbedingungen umgerechnet werden.

3.3 Zeitintegration

Nachdem in den beiden vorherigen Abschnitten die verallgemeinerte Transportgleichung (15) im Ort diskretisiert wurde, soll in diesem Abschnitt beschrieben werden, wie diese in der Zeit integriert wird. Explizite Zeitintegrationsverfahren, welche besonders schnell sind, da keine Gleichungssysteme gelöst werden müssen, stellten sich leider für unser Problem als nicht geeignet heraus. So ließ sich ein mehrstuffiges Runge-Kutta-Verfahren aufgrund der Kopplung der Differentialgleichungen nicht ohne Weiteres implementieren und das explizite Eulerverfahren lief nicht stabil.
Da beim impliziten Eulerverfahren aufgrund des nichtlinearen Konvektionsterms ein nichtlinerares Gleichungssystem gelöst werden müsste, entschieden wir uns für eine **Variation des impliziten Eulerverfahrens**. Bei diesem wird im Gegensatz zum impliziten Eulerverfahren der nichtlineare Konvektionsterm explizit angegeben. Bei einer gegebenen Zeitschrittweite δt ergibt sich für die Diskretisierung der verallgemeinerten Transportgleichung (15) also:

$$\frac{\zeta_h^{i+1} - \zeta_h^i}{\delta t} = \tilde{B}_h \zeta_h^{i+1} - \tilde{C}_h^i \zeta_h^i + \tilde{D}_h \, T_h^i + b_h^i, \tag{19}$$

Der hochgestellte Index i bedeutet dabei, dass die Werte im aktuellen Zeitschritt verwendet werden, während es sich bei $i+1$ um die Werte im folgenden Zeitschritt handelt. In b_h^i stehen die in Kapitel 3.2 beschriebenen Randwerte des aktuellen Zeitschritts.
Durch elementare Umformungen kann Gleichung (19) in die Form eines linearen Gleichungssystems gebracht werden:

$$\underbrace{(E_n - \delta t \tilde{B}_h)}_{\textit{wieder SPD}} \zeta_h^{i+1} = \underbrace{(E_n - \delta t \tilde{C}_h^i)\zeta_h^i + \delta t \tilde{D}_h \; T_h^i + \delta t \; b_h^i}_{\textit{Rechte Seite}}$$

Dabei ist die Koeffizientenmatrix $E_n - \delta t \tilde{B}_h$ mit der selben Argumentation wie in Abschnitt 3.1.1 wieder symmetrisch positiv definit und somit kann auch für die Lösung dieses linearen Gleichungssystems das später beschriebene CG-Verfahren angewendet werden.

Die Wahl des Zeitschrittes δt ist dabei heuristisch, da sich die gängige Stabilitätstheorie für das implizite Euler Verfahren nur bei linearen Operatoren anwenden lässt.
Eine Orientierung bietet allerdings die *Courant-Zahl* (auch *CFL-Zahl* genannt) $Co := \frac{\hat{c} \cdot \delta t}{h}$, wobei $\hat{c}$ die maximal auftretende Geschwindigkeit ist. Sie gibt an, um wieviele Zellen sich die simulierte Größe in einem Zeitschritt maximal fortbewegt. Damit das Zeitintegrationverfahren stabil läuft, muss gelten:

$$Co < 1$$

Somit wird garantiert, dass die betrachtete Größe in einem Zeitschritt keine Zelle überspringt. Dies ist auch aus physikalischer Sicht eine sinnvolle Forderung, da sonst die Bilanzierung der Simulationsgrößen über die Zellwände nicht mehr funktioniert (der Teil, der aus einer Zelle hinaus fließt, fließt in seine Nachbarzelle hinein).
Aus dieser Ungleichung kann eine Zeitschrittweite δt direkt abgeleitet werden.

3.4 Aufbau des Solvers

Das in Kapitel 2 hergeleitete System von Differentialgleichungen $(*)$ liegt nun in vollständig diskretisierter Form vor:

$$A_h \Psi_h = \underbrace{\omega_h + b_h^{\Psi}}_{=:\tilde{\omega}_h} \tag{20}$$

$$\underbrace{(E_n + \delta t \cdot \nu \cdot A_h)}_{=:B_h^{\omega}} \omega_h = \underbrace{(E_n - \frac{\delta t}{h^2} \cdot C_h)\omega_h + \frac{\delta t}{h^2} \cdot D_h T_h + b_h^{\omega}}_{=:rhs^{\omega}} \tag{21}$$

$$\underbrace{(E_n + \delta t \cdot a \cdot A_h)}_{=:B_h^{T}} T_h = \underbrace{(E_n - \frac{\delta t}{h^2} \cdot C_h) T_h + b_h^{T}}_{=:rhs^{T}} \tag{22}$$

In diesem Abschnitt soll nun der generelle Ablauf des von uns implementierten Strömungslösers beschrieben werden, der in Abbildung 9 in Form eines Struktogramms dargestellt ist. Dabei kann unser Programm generell in zwei Teile aufgeteilt werden: Im **Preprocessing** werden alle Operationen, die nicht von sich zeitlich ändernden Größen abhängen zusammengefasst und einmalig vor der Zeitschleife, in der alle Operationen eines **Zeitschritts** berechnet werden, ausgeführt.

Zu Beginn des Preprocessing werden die Netzparamteter und Stoffkonstanten gesetzt und Anfangswerte für die Wirbelstärke ω_h und die Temperatur T_h sowie Randbedingungen für die Geschwindigkeit und die Temperatur vorgegeben. Aus diesen Daten lassen sich dann die Differentialoperatoren A_h, B_h^{ω}, B_h^{T} und D_h aufstellen. Lediglich der Differentialoperator des Konvektionsterms C_h muss in jedem Zeitschritt neu aufgestellt werden, da dieser von der Geschwindigkeit c_h abhängt, die sich in jedem Zeitschritt ändert. Dabei ist $c_h = (u_h, v_h)$, sowie $u_h = (u_1, u_{Rand}, u_2)^T$ und $v_h = (v_1, v_{Rand}, v_2)^T$, wobei die Indizes für die versetzten Gitter 1 und 2 stehen (vgl. Abb. 8). Am Ende des Preprocessings wird schließlich der Vektor Ψ_{Rand} aus den Geschwindigkeitsvorgaben auf dem Rand, wie in Abschnitt 3.2.1 beschrieben, berechnet und daraus der Randwertevektor b_h^{Ψ} aufgestellt.

In einem Zeitschritt wird zuerst Ψ_h durch Lösen der Poissongleichung berechnet. Die erhaltenen Werte für Ψ_h werden nach Gleichung (4) durch Differenzieren in Werte für die gesuchten Geschwindigkeiten im Innern umgerechnet. Auf Grund der differentiellen Beziehung zwischen der Geschwindigkeit und der Stromfunktion erhält man auf diese Weise die Geschwindigkeitswerte für u auf *Gitter 1* und die Geschwindigkeitswerte für v auf *Gitter 2* (siehe Abb. 8).
Zur Lösung der Wirbeltransportgleichung und der Wärmeleitungsgleichung benötigt man allerdings für den Konvektionsterm die Geschwindigkeiten gerade auf dem jeweils anderen Gitter, weshalb die in Abbildung 8 skizzierte Interpolation notwendig ist. Hierfür bildet man den Mittelwert aus den vier umliegenden Geschwindigkeitswerten.

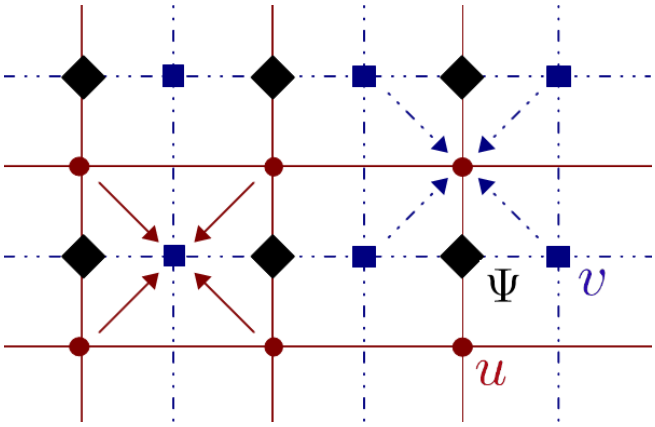

Abbildung 8: Interpolation der Geschwindigkeiten zwischen Gitter 1 (rot) und Gitter 2 (blau)

Aus den neu berechneten Geschwindigkeitswerten lassen sich anschließend die Konvektionsmatrix C_h sowie der Vektor ω_{Rand} aufstellen.
Nach Berechnung von T_{Rand} können nun, wie im Struktogramm zu sehen ist, die Rechten Seiten rhs^{ω} und rhs^{T} für die Wirbeltransportgleichung und Wärmeleitungsgleichung aufgestellt werden.
Durch Lösen der linearen Gleichungssysteme für die Wirbeltransportgleichung (21) und für die Wärmeleitungsgleichung (22) erhält man ω_h und T_h für den nächsten Zeitschritt.
Am Ende eines Zeitschritts können schließlich die Geschwindigkeit und die Temperatur gespeichert werden.

Die gespeicherten Daten lassen sich mit Hilfe von MATLAB aufbereiten und visualisieren.

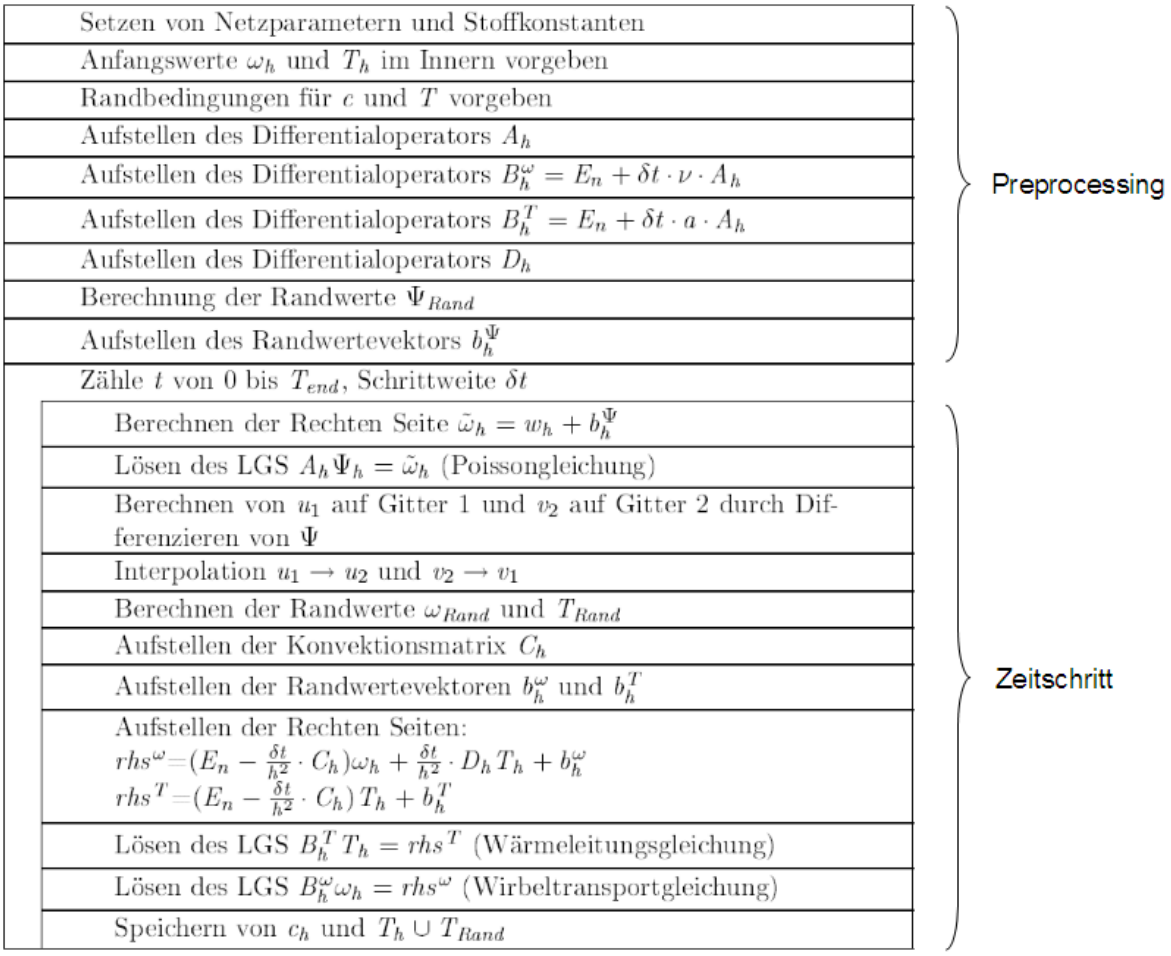

Abbildung 9: Ablauf des Strömungslösers in Struktogrammform

4 Parallele Programmierung mit CUDA

Bei der computerunterstützten Berechnung ingenieurwissenschaftlicher Probleme werden heutzutage typischerweise hohe Rechenleistungen benötigt. Insbesondere bei der numerischen Simulation von strömungsmechanischen Problemen sind in der Praxis Gleichungssyteme mit mehreren Millionen Unbekannten zu lösen. Ein Ansatz um diesem Problem der Berechenbarkeit entgegen zu wirken, ist die Parallelisierung von Algorithmen, sofern das gestellte Problem dies ermöglicht. Im Falle eines Multicore Rechners aus mehreren CPU-Kernen ist dies allerdings mit hohen Kosten verbunden.

Eine weitere Möglichkeit ist daher die Berechnung auf Grafikkarten (GPU), deren Rechenleistung wie in Abbildung (10) dargestellt in den letzten zehn Jahren enorm zugenommen hat.

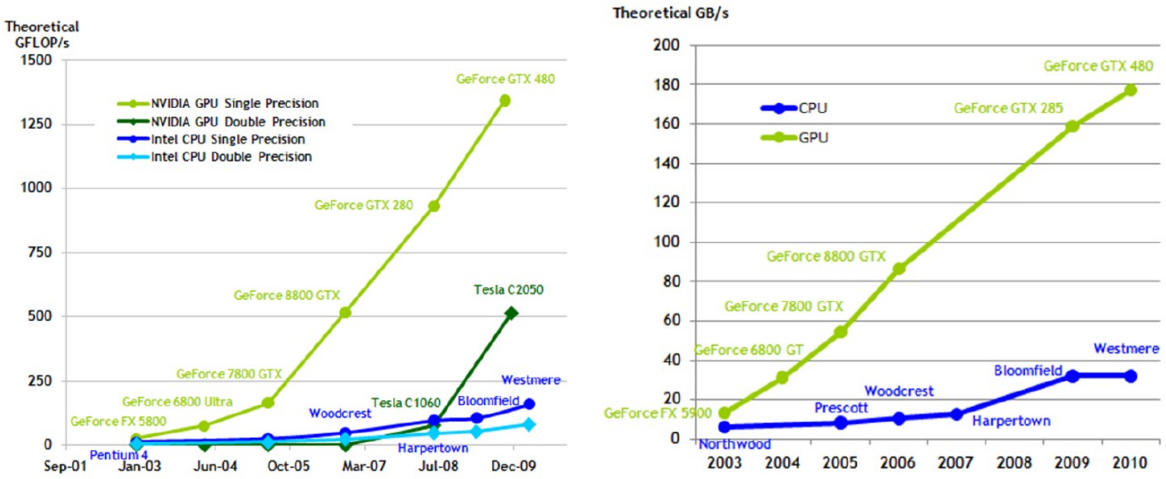

Abbildung 10: Rechenleistung von CPUs und GPUs in den Jahren 2003-2010 (entnommen aus [10])

Da bei Grafikberechnungen die Pixelwerte unabhängig voneinander berechnet werden, eignet sich die Architektur von GPUs grundsätzlich für parallele Berechnungen. Um die Rechenleistung der GPUs auch für Berechnungen der numerischen Mathematik zugänglich zu machen, wurde im November 2006 mit CUDA eine Softwareumgebung geschaffen, die es dem Entwickler ermöglicht, direkt aus einem *C/C++* Code auf der Grafikkarte zu programmieren.

4.1 Verwendete Hardware

Bei der Verwendung von CUDA ist eine genaue Kenntnis der Hardware sehr wichtig, da die Programmierung sehr hardwarenah erfolgt. Hierzu wurde in CUDA die Funktion `cudaGetDeviceProperties()` vorgesehen, die dem Programmierer wichtige technische Daten der verwendeten GPU zur Verfügung stellt. Bei der von uns verwendeten Grafikkarte auf dem Numerikcluster der TU Berlin erhält man beispielsweise diese Ausgabe:

```
CUDA Device Query...
There are 1 CUDA devices.

CUDA Device #0
Major revision number:         1
Minor revision number:         3
Name:                          Tesla C1060
Total global memory:           4294770688
Total shared memory per block: 16384
Total registers per block:     16384
Warp size:                     32
Maximum memory pitch:          2147483647
Maximum threads per block:     512
Maximum dimension 0 of block:  512
Maximum dimension 1 of block:  512
Maximum dimension 2 of block:  512
Maximum dimension 0 of grid:   65535
Maximum dimension 1 of grid:   65535
Maximum dimension 2 of grid:   65535
Clock rate:                    1296000
Total constant memory:         65536
Texture alignment:             256
Concurrent copy and execution: Yes
Number of multiprocessors:     30
Kernel execution timeout:      No
```

Es handelt sich also um eine *Tesla C1060* Grafikkarte der *Compute Capability* 1.3 mit $4GB$ Speicher. Im Folgenden werden wir uns auf Grafikkarten dieser Compute Capability beziehen; das Verhalten von Grafikkarten mit anderen Compute Capabilities kann von diesen abweichen.

Die GPU von Grafikkarten der *Compute Capability* 1.3 besteht im Wesentlichen aus 30 Streaming Multiprozessoren (SM). Jeder SM wiederum besteht aus 8 skalaren Recheneinheiten (Streaming Processors - SP), einer Gleitkommaeinheit für doppeltgenaue Multiplikation und Addition (Double-Einheit), sowie zwei Special Function Units (SFU), die mit Tabellen und quadratischer Interpolation mathematische Standardfunktionen (wie sin, cos, ...) schnell, aber mit geringerer Genauigkeit berechnen. Desweiteren verfügt jeder SM noch über eine große Menge an Registern und einen gemeinsamen Speicher (Shared-Memory) für alle enthaltenen Rechenkerne.

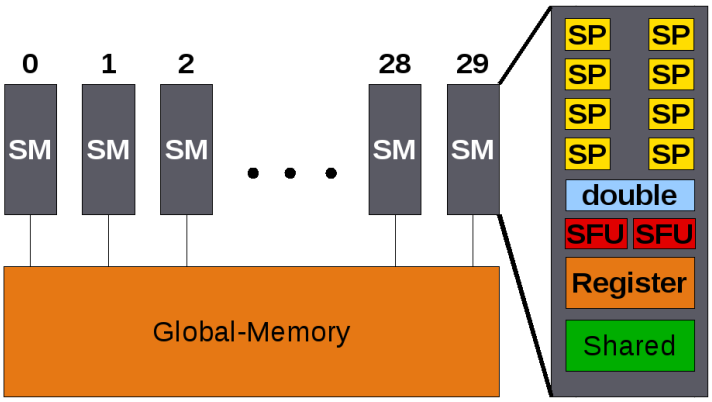

Abbildung 11: Schematischer Aufbau einer GPU

Auf jeder skalaren Recheneinheit werden in 4 Takten jeweils 4 Threads gleichzeitig ausgeführt. Zusammen mit den 8 skalaren Recheneinheiten bedeutet das, dass auf einem Streaming Multiprozessor jeweils 32 Threads parallel abgearbeitet werden. Diese Gruppe aus 32 Threads bezeichnet man als *Warp*. Die parallele Verarbeitung erfolgt dabei nach dem SIMD-Prinzip (Single Instruction - Multiple Data), alle Threads eines Warps müssen also die gleiche Operation mit unterschiedlichen Daten ausführen. Folgen die Threads eines Warps unterschiedlichen Programmzweigen, so ist dies möglich, da die Ergebnisse einer ausgeführten Operation nicht verwendet werden müssen. Allerdings müssen dann alle Programmzweige nacheinander abgearbeitet werden. Dieses als *Divergency* bezeichnete Phänomen ist eine häufige Ursache für Perfomanceverluste und sollte an jeder Programmverzweigung beachtet werden. Würde jeder Thread einer anderen Verzweigung nachgehen, so würde also die Ausführungszeit um das 32-fache steigen.
Die Warps werden vom Warpscheduler verwaltet, welcher auch die verzahnte Ausführung von Warps emöglicht. Diese wird auch intensiv genutzt um beispielsweise die durch Speicheranfragen bedingten Wartezeiten eines Warps mit der Ausführung eines anderen Warps zu füllen.

Anhand dieser Kriterien lassen sich bereits Aussagen über die Geschwindigkeit der Grafikkarte treffen. Im wissenschaftlichen Rechnen wird die Anzahl der Floating Point Operationen pro Sekunde, kurz $Flops/s$, als Maß für die Geschwindigkeit eines Algorithmuses verwendet.
Für doppelte Genauigkeit kann jede der Double-Einheiten auf den 30 Multiprozessoren eine MADD-Anweisung (Multiplikation und Addition) durchführen. Bei einer Taktrate (Clock rate) von $1296 MHz$ erhält man als maximale Rechenleistung also:

$$P_{C1060\_double\_max} = 2 Flops \cdot 1296 \cdot 10^6 Hz \cdot 30 \approx 78.8 GFlops/s$$

Bei Berechnungen, die keine MADD-Anweisung benötigen, halbiert sich diese Rechenleistung allerdings, so dass es sich hierbei nur um eine theoretische Obergrenze handelt.
Für einfache Genauigkeit kann jeder der $8 \cdot 30 = 240$ SPs zusätzlich zu der MADD-Anweisung noch eine weitere Multiplikation ausführen. Bei einer Taktrate von $1296 MHz$ erhält man für die maximale Rechenleistung also:

$$P_{C1060\_single\_max} = 3 Flops \cdot 1296 \cdot 10^6 Hz \cdot 240 \approx 933 GFlops/s$$

Diese zusätzliche Multiplikation kann allerdings nur in den seltensten Fällen genutzt werden.

Ein weiteres wichtiges Maß für die Geschwindigkeit stellt häufig die Speicherbandbreite B zwischen dem Grafikkartenspeicher und der GPU dar. Diese lässt sich aus Informationen über die *Tesla C1060* Grafikkarte aus [12] berechnen:

$$B_{C1060} = 800 \cdot 10^6 Hz \cdot (512/8) Byte \cdot 2 \approx 102 GB/s,$$

wobei in dieser Rechnung der Speichertakt in *Hz* mit der Breite der Speicherschnittstelle in *Byte* multipliziert wurde und auf Grund der doppelten Datenrate mit 2 multipliziert wird.

4.2 Das CUDA-Programmiermodell

Zunächst gibt es in CUDA die CUDA *Runtime API* und die CUDA *Driver API*. Die CUDA *Runtime API* ist dabei eine High-level-API bei der jeder Aufruf einer Runtime-Funktion aus mehreren Funktionsaufrufen der CUDA *Driver API*, welche die Low-Level-API darstellt, besteht. Im Folgenden werden wir uns nur mit der CUDA Runtime API beschäftigen, da diese die gängige CUDA-Programmierumgebung ist.

Ein Programm, welches CUDA verwendet, teilt sich grundsätzlich in zwei Teile auf. Zum einen in den Teil der wie gewohnt auf der CPU läuft und in CUDA als *Host* bezeichnet wird. Der andere Teil wird auf der Grafikkarte ausgeführt, die im Folgenden als *Device* bezeichnet wird. Der Ablauf eines CUDA-Programms sieht dann im Wesentlichen so aus, dass der sequentielle Teil auf dem Host und der parallele Teil auf dem Device bearbeitet wird. Der Wechsel vom Host zum Device erfolgt über spezielle Funktionen, welche man als *Kernel* bezeichnet. Eine Instanz des Kernels läuft dann auf jedem Thread des Device.

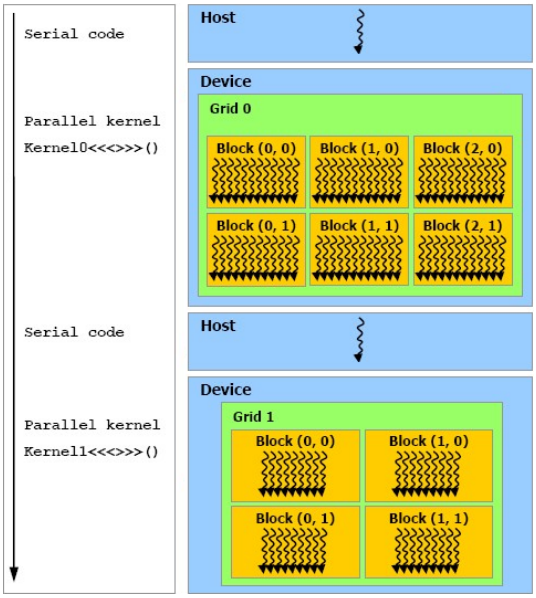

Abbildung 12: Allgemeiner Ablauf eines CUDA-Programms (entnommen aus [10])

4.2.1 Parallelität in Cuda

Auf jedem Thread läuft eine Instanz des Kernels. Threads sind in *Blöcke (Blocks)* gruppiert, welche wiederum in *Gitter (Grids)* gruppiert sind. Dabei erfolgt sowohl die Threadindizierung innerhalb eines Blocks als auch die Blockindizierung innerhalb eines Gitters dreidimensional.

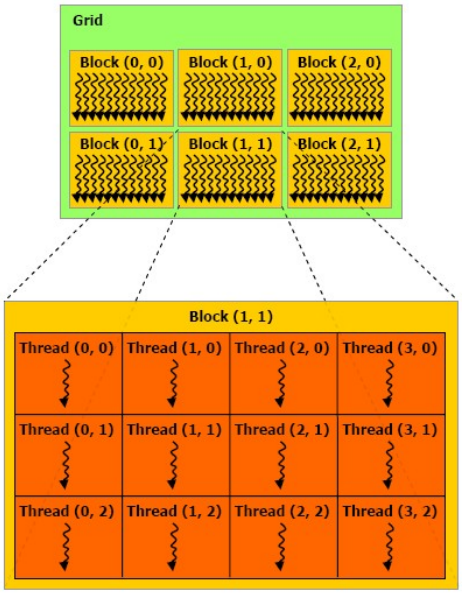

Abbildung 13: Aufteilung in Grid, Blocks und Threads (entnommen aus [10])

Die Parallelität in Cuda erfolgt also auf zwei Hierarchieebenen, zum einen auf der Blockebene und zum anderen auf der Gitterebene.
Die Parallelität auf Blockebene ist dabei relativ eng. Alle Threads innerhalb eines Blocks werden auf dem selben Multiprozessor ausgeführt und können durch den Aufruf der Funktion `__syncthreads()` synchronisiert werden. Außerdem können sie über den gemeinsamen Speicher Daten austauschen.
Die Parallelität auf Gitterbene hingegen ist relativ lose. Man kann hierbei keinerlei Aussagen darüber machen, in welcher zeitlichen Reihenfolge die Blöcke abgearbeitet werden. Die Blöcke kann man sich dabei als ein Art Auftragsvolumen vorstellen, das die Grafikkarte abzuarbeiten hat. Eine weiterentwickelte Grafikkarte könnte dann beispielsweise mehr Blöcke gleichzeitig bearbeiten als eine momentan aktuelle Grafikkarte. Das Programm bleibt so lange skalierbar und eine Verdopplung der Grafikkartenleistung würde die Laufzeit des parallelen Anteils des Programms halbieren.

4.2.2 Verschiedene Speicherarten

Auf der Grafikarte gibt es mehrere verschiedene Speicherarten, welche sich bzgl. Geschwindigkeit und Verfügbarkeit unterscheiden. Eine genaue Kenntnis der Vor- und Nachteile der verschiedenen Speicherarten ist dabei sehr wichtig um die Leistung der Grafikkarte optimal

ausnutzen zu können. In Abbildung 14 sind die Zusammenhänge zwischen den im Folgenden erklärten Speicherarten dargestellt.

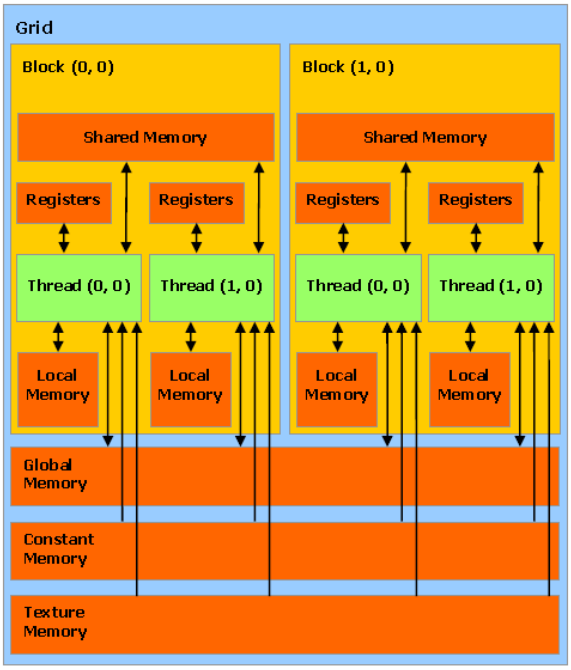

Abbildung 14: Verschiedene Speicherarten (entnommen aus [10])

Arbeitsspeicher (Host-Memory)
Die Bandbreite zwischen Arbeitsspeicher und globalem Speicher ist mit $8GB/s$ (Bandbreite des PCI-E-Anschluss) um mehr als das 10-fache geringer als die Bandbreite zwischen GPU und globalem Speicher ($102GB/s$). Deshalb sollten Kopiervorgänge zwischen Arbeitspeicher der CPU und globalem Speicher auf der GPU minimiert werden, auch wenn das bedeutet, dass man Kernel ausführt, welche langsamer sind als entsprechende Funktionen auf dem Host.

Globaler Speicher (Global-Memory)
Dieser Speicher kann von allen Multiprozessoren gelesen und beschrieben werden. Allerdings ist zu beachten, dass der Zugriff auf den globalen Speicher mit relativ hohen Latenzzeiten verbunden ist und die Bandbreite im Vergleich zur Rechengeschwindigkeit sehr gering ist.
Auf den globalen Speicher kann mit Hilfe der Funktion `cudaMalloc()` vom Host aus Speicher allokiert und mit `cudaFree()` wieder freigegeben werden. Auf den allokierten Speicher kann man dann mit `cudaMemcpy()` Daten vom Host aufs Device und umgekehrt kopieren.
Der Zugriff auf den globalen Speicher erfolgt ungecacht und es werden nur Speicherblöcke von 32, 64 oder 128 Byte gelesen. Die Größe der Segmente hängt dabei von der Größe der Speicheranforderung ab. Die Größe des Segments beträgt bei einer 1-Byte-Anforderung 32 Byte, bei einer 2-Byte-Anforderung 64 Byte und bei einer 4-Byte- oder 8-Byte-Anforderung 128 Byte. Liest jeder Thread also nur 1 Byte aus dem Speicher, so würden für jeden Thread trotzdem mindestens 32 Byte gelesen werden und die restlichen 31 Byte vom Multiprozessor

verworfen werden. Auf diese Weise wird Kapazität der Bandbreite verschwendet. Dieser Effekt wird allerdings dadurch verringert, dass bei gleichzeitigem Zugriff mehrerer Threads eines Warps auf den gleichen Speicherblock die Zugriffe zusammengelegt werden. Dieses bezeichnet man dann als *Coalescing*. Coalescing tritt ein, wenn mehrere Threads der ersten oder zweiten Hälfte eines Warps auf das gleiche Speichersegment zugreifen. Da die Bandbreite - wie später zu sehen - einer der größten Flaschenhälse ist, sind Coalescing und die Minimierung der Zugriffe auf den gemeinsamen Speicher eine der wichtigsten Optimierungsmöglichkeiten.

Register, Local-Memory
Die Register sind der schnellste Speicher, in ihnen werden die lokalen Variablen der Threads gespeichert. Jeder der 30 Multiprozessoren verfügt über 16384 Register, welche jeweils eine Größe von 4 Byte haben. Diese ralativ große Anzahl relativiert sich allerdings dadurch, dass verhältnismäßig viele Threads quasiparallel ausgeführt werden und die Register unter diesen disjunkt verteilt werden. Reichen die Register nicht aus, so wird ein Teil der lokalen Variablen auf den deutlich langsameren Global-Memory ausgelagert, welchen man dann als Local-Memory bezeichnet. Dies führt zu Performanceverlusten und sollte möglichst vermieden werden.

Gemeinsamer Speicher (Shared-Memory)
Jeder der 30 Multiprozessoren verfügt über 16 kB Shared-Memory, auf den alle Threads innerhalb eines Blocks Zugriff haben. Der Shared-Memory kann insbesondere zur Kommunikation von Threads innerhalb eines Blocks und zum Ablegen gemeinsam genutzter Daten genutzt werden, um Zugriffe auf den Global-Memory zu reduzieren.
Der Shared-Memory ist in 16 Speicherbänken organisiert, wobei aufeinander folgende 4-Byte-Wörter jeweils in aufeinander folgendenden Speicherbänken gespeichert sind. Der Zugriff auf den gemeinsamen Speicher wird immer nur für halbe Warps gleichzeitig gestartet. Auf jede Speicherbank kann jeweils nur ein Thread zugreifen. Greifen mehrere Threads eines halben Warps auf die gleiche Speicherbank zu, so bezeichnet man dies als einen *Bankkonflikt* und die Zugriffe werden seriell ausgeführt. Als Ausnahme läuft der lesende Zugriff aller Threads auf eine Speicherbank (Broadcast-Zugriff) konfliktfrei ab. Die Vermeidung von Bankkonflikten ist eine weitere wichtige Optimierungsmöglichkeit in CUDA.
Sofern keine Bankkonflikte bestehen, sind die Zugriffszeiten auf den Shared-Memory die gleichen wie die auf die Register. Shared-Memory muss bei der Deklaration durch ein vorrangestelltes `__shared__` gekennzeichnet werden. Soll die Größe, wie später beschrieben, durch den Kernelaufruf festegelegt werden, so muss vor dem `__shared__` noch ein `extern` stehen.

Texturen
Daten, auf die nur lesend zugegriffen wird, können auf dem Global-Memory auch mittels Texturen gebunden werden. Der Zugriff auf den Global-Memory wird dabei durch die zusätzlich vorhandenen Textur-Caches eventuell beschleunigt. Für Texturen muss bereits zur Compilierzeit eine *Textur-Reference* angelegt werden, welche eine Template-Instanz ist, deren Parameter die Datenstruktur der Textur definieren:
`texture<type, anzDim> texName`
Dabei gibt `type` den Datentyp der Texturelemente an und *anzDim* die Anzahl der Dimensio-

nen.
Zur Laufzeit muss eine Textur während ihrer Verwendung an einem bestimmten Bereich im Global-Memory gebunden sein. Das Binden und Lösen einer Textur erfolgt über die Funktionen `cudaBindTexture()` und `cudaUnbindTexture()`. Der Zugriff auf eine eindimensionale Textur erfolgt über die Funktion `tex1Dfetch()` ähnlich wie bei einem Array.

4.2.3 Kernel und Kernelaufruf

Wie bereits erwähnt, erfolgt der Wechsel vom Host zum Device über den Kernel, wobei eine Instanz des Kernels auf jedem Thread läuft. Ein Kernel wird bei der Deklaration mit einem vorangestelltem `__global__` gekennzeichnet. Im Gegensatz zu gewöhnlichen C-Funktionen darf ein Kernel keinen Rückgabewert und keine statischen Variablen haben und es ist keine Rekursion erlaubt. Innerhalb des Kernels können die vordefinierten Variablen `threadIdx`, `blockIdx`, `blockDim`, und `gridDim` verwendet werden. Jede davon ist vom Typ `dim3`, welche eine Struktur mit den Elementen `x`, `y`, `z` ist. Aus diesen vordefinierten Variablen lässt sich eine globale Thread-ID bestimmen, die sich im eindimensionalen Fall beispielsweise durch `index=threadIdx.x+blockIdx.x*blockDim.x` berechnen lässt.

Beim Kernelaufruf muss im Gegensatz zum Aufruf einer gewöhnlichen C-Funktion noch eine Laufzeitkonfiguration angegeben werden, welche in den spitzen Klammern `<<<` und `>>>` zwischen Kernelname und Paramterliste steht:
`Kernelname<<<dimGrid,dimBlock,Speicherbedarf,Stream>>>(paramter1,...)`
Dabei geben `dimGrid` und `dimBlock` die Größe der Dimensionen des Gitters und der Blöcke an, Speicherbedarf die Größe in Byte des mit `extern` deklarierten Shared-Memory und `Stream` den Stream an. Der dritte und vierte Parameter sind dabei optional.
Eine gute Wahl der Blockgröße und Gittergröße stellt eine weitere Optimierungsmöglichkeit dar. Die richtige Wahl der Blockgröße ist ziemlich schwerig und hängt von sehr vielen Parametern ab und muss daher meistens heuristisch ermittelt werden. Sie sollte allerdings ein Vielfaches von 32 (Warpgröße) und darf nicht größer als 512 (maximale Blockgröße) sein. Eine Blockgröße von 128 hat sich meist als eine gute Wahl herausgestellt.
Die Gittergröße sollte im Falle der Tesla C1060 mindestens 30 betragen, damit jeder Multiprozessor ausgelastet wird. Desweiteren kann die Grafikkarte, je größer die Gittergröße ist, die Arbeit besser auf die Multiprozessoren verteilen.

4.2.4 Ein einfaches Beispiel

Algorithmus 1 stellt ein einfaches Beispiel eines CUDA-Programms dar und soll den üblichen Ablauf eines solchen verdeutlichen. Dieses erste Beispiel berechnet eine SAXPY-Operation (Scalar Alpha X Plus Y).
Zu Beginn des Programms wird der neu definierte Datentyp `Real` eingeführt, welcher durch die Konstante `PRAEZISSION` auf einfache oder doppelte Genauigkeit gestellt werden kann. Dieser Datentyp wird auch in den folgenden Programmen benutzt, damit durch Ändern dieser Konstante das Programm von einfacher auf doppelte Genauigkeit umgestellt werden kann. Als nächstes kommt die Definition des Kernel. Der Kernel ist dabei so implementiert, dass jeder Thread ein Element der SAXPY-Operation berechnet.

Der Ablauf des Hauptprogramms sieht dann so aus, dass zunächst der Speicher für die Vektoren auf dem Host und dem Device allokiert wird. Anschließend werden die Vektoren auf dem Host mit Werten belegt. Damit das Device mit den Vektoren arbeiten kann, werden diese vor dem Kernelaufruf auf das Device kopiert. Der Kernelaufruf erfolgt dann mit einer Blockgröße von 128 und der entsprechend benötigten Gittergröße, die zuvor berechnet wurde. Nach Aufruf des Kernel wird der Ergebnisvektor vom Device auf den Host kopiert, um diesen ausgeben zu können. Am Ende wird der Speicher auf dem Host und dem Device wieder freigeben.

Algorithm 1 Ein einfaches Beispiel eines CUDA-Programmes

```
#include <stdio.h>
#define PRAEZISSION 2
//Datentyp:
#if PRAEZISSION == 2
  typedef double Real;
#else
  typedef float Real;
#endif
__global__ void k_saxpy(Real *erg, Real *x, Real *y, Real alpha, int dim){
  int index=blockIdx.x*blockDim.x+threadIdx.x;  //globale Thread-ID
  if(index<dim)
    erg[index]=alpha*x[index]+y[index];
}
int main (int argc, char * const argv[]){
  int dim=100000;
  //Anlegen der Vektoren auf dem Host
  Real* x=(Real*)malloc(sizeof(Real)*dim);
  Real* y=(Real*)malloc(sizeof(Real)*dim);
  Real* erg=(Real*)malloc(sizeof(Real)*dim);
  Real alpha=5.0;
  //Anlegen der Vektoren auf dem Device
  Real *d_x, *d_y, *d_erg;
  cudaMalloc((void**)&d_x, dim*sizeof(Real));
  cudaMalloc((void**)&d_y, dim*sizeof(Real));
  cudaMalloc((void**)&d_erg, dim*sizeof(Real));
  //Vektoren mit Werten belegen
  for(int i=0; i<dim; i++){
    x[i]=i;
    y[i]=2*i;
  }
  //Vektoren auf Device kopieren
  cudaMemcpy(d_x, x, dim*sizeof(Real), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, dim*sizeof(Real), cudaMemcpyHostToDevice);
  //Kernelaufruf
  dim3 dimBlock(128);
  dim3 dimGrid(dim/dimBlock.x+1);
  k_saxpy<<<dimGrid,dimBlock>>>(d_erg,d_x,d_y,alpha,dim);
  //Ergebnis auf Host kopieren
  cudaMemcpy(erg,d_erg,dim*sizeof(Real),cudaMemcpyDeviceToHost);
  //Ausgabe des Ergebnisvektors
  for (int i=0;i<dim;i++){
    printf("%f\n",erg[i]);
  }
  //Speicherfreigabe
  cudaFree(d_x);
  cudaFree(d_y);
  cudaFree(d_erg);
  free(x);
  free(y);
  free(erg);
  return 0;
}
```

4.3 Parallelisierung und Optimierung elementarer Operationen der Linearen Algebra

Nachdem in Kapitel 4.2.4 bereits die Verwendung von CUDA anhand einer SAXPY Operation demonstriert wurde, sollen in diesem Kapitel die wesentlichen Konzepte und Optimierungsmöglichkeiten unter CUDA anhand des Skalarprodukts und der Sparsematrix-Vektor Multiplikation demonstriert und diskutiert werden. Diese drei Operationen stellen die wesentlichen Rechenoperationen des CG-Verfahrens (Kapitel 4.4) dar, das zur Lösung der entstehenden linearen Gleichungssysteme benutzt wird.

4.3.1 Skalarprodukt und Norm

Algorithmus

Bei der Berechnug des Skalarprodukts müssen grundsätzlich jeweils die Elemente der beiden Vektoren mit gleichem Index miteinander multipliziert werden. Dieser Teil lässt sich ohne Probleme parallelisieren, da all diese Operationen unabhängig voneinander sind. Jedem Thread kann also eine bestimmte Menge der Indizes zugewiesen werden, für die er die Multiplikation ausführt.
Anschließend muss jedoch die Summe über all diese Produkte gebildet werden. Eine baumartige Berechnung ist dabei besonders effektiv, da sich diese parallelisieren lässt. Dabei summiert zunächst jeder Thread die von ihm berechneten Produkte. Anschließend werden alle Threads synchronisiert und jeder zweite Thread summiert seine Teilsumme mit der Teilsumme seines Nachbarn. Anschließend werden wieder alle Threads synchronisiert und so weiter, bis alle Zweige des Baums aufsummiert sind. Die Synchronisation der Threads stellt allerdings ein Problem dar, da nur Threads innerhalb eines Blocks synchronisiert werden können.
Eine Lösung wäre dabei die Verwendung nur eines Blocks, allerdings wird ein Block immer nur jeweils auf einem Multiprozessor ausgeführt, weshalb man dabei nur ein Dreizigstel (30 Multiprozessoren) der Rechenleistung nutzen würde.
Um nun die volle Rechenleistung der Grafikkarte zu nutzen, sind daher mindestens zwei Kernelaufrufe notwendig. Zunächst werden im ersten Kernel alle Produkte berechnet und innerhalb eines Blocks aufsummiert. Anschließend werden in einem zweiten Kernel mit nur einem Block die Teilsummen aller Blöcke aufsummiert.
In Abbildung 15 ist in orange der Teil dargestellt, den der erste Kernel aufsummiert, während in blau der Teil, den der zweite Kernel aufsummieren muss, gekennzeichnet ist.

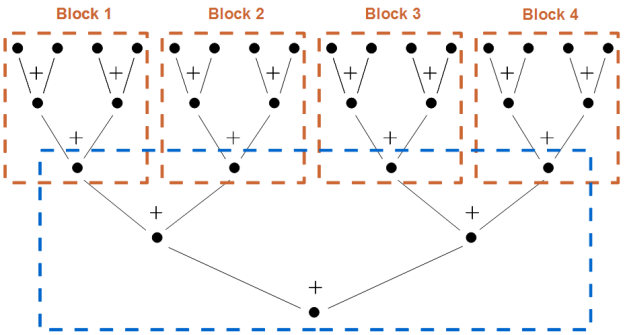

Abbildung 15: Baumartige Berechnung des Skalarprodukts

Implementierung und Optimierung

In diesem Abschnitt werden wir uns ausschließlich mit der Optimierung des ersten Kernel beschäftigen. Der zweite Kernel ergibt sich dann aus dem ersten Kernel, da dieser im Wesentlichen so wie die Bildung der Teilsumme des im Folgenden beschriebenen ersten Kernel funktioniert.

Eine erste Implementierung des ersten Kernel (Algorithmus 2) könnte so aussehen, dass jeder Thread die Produkte für `n` aufeinanderfolgende Indizes berechnet, diese aufsummiert und im Shared-Memory speichert. Dabei wird `n` so berechnet, dass jeder Thread etwa gleich ausgelastet ist. Anschließend wird die Teilsumme des Blocks berechnet und im Global-Memory gespeichert. Zusätzlich muss die Blöckgröße auf eine Potenz von 2 eingeschränkt werden, damit die baumartige Aufsummierung innerhalb des Blocks aufgeht.

Algorithm 2 Erste Skalarproduktimplementierung

```
__global__ void k_skp1(Real *c, Real *a, Real *b, int dim){
  //Anzahl der von jedem Thread zu berechnenden Produkte
  int n=dim/(blockDim.x*gridDim.x+1)+1;
  extern __shared__ Real shared_summands[];
  Real sum=0.0;
  //Berechnung der Produkte und Aufsummation innerhalb des Thread
  int index=(blockIdx.x*blockDim.x+threadIdx.x)*n;   //Startindex
  for(int i=0;i<n;i++){
    if((index+i)<dim)
      sum+=a[index+i]*b[index+i];
  }
  //Abspeicherung der Teilsumme im Shared-Memory
  shared_summands[threadIdx.x]=sum;
  //Berechnung der Teilsumme des gesamten Blockes
  __syncthreads();
  for(int s=1; s<blockDim.x; s*=2){
    if(threadIdx.x%(2*s) == 0){
      shared_summands[threadIdx.x] += shared_summands[threadIdx.x+s];
    }
    __syncthreads();
  }
  //Abspeicherung der Teilsumme des gesamten Blocks im Global-Memory
  if(threadIdx.x==0)
    c[blockIdx.x] = shared_summands[0];
}
```

Bei dieser ersten Implementierung tritt kein *Coalescing* auf, da die Threads beim Laden der Faktoren auf Speicherstellen mit jeweils Abstand `n` im Global-Memory zugreifen. Eine Verbesserung stellt daher Algorithmus 3 dar. Hier berechnet jeder Thread die Produkte für `n` Indizes, die jeweils die Gesamtthreadanzahl als Abstand haben. So greifen aufeinanderfolgende Threads auf aufeinanderfolgende Speicherstellen im Global-Memory zu und es tritt *Coalescing* auf.

Algorithm 3 Skalarproduktimplementierung mit Coalescing

```
__global__ void k_skp2(Real *c, Real *a, Real *b, int dim){
  int n=dim/(blockDim.x*gridDim.x+1)+1;
  extern __shared__ Real shared_summands[];
  Real sum=0.0;

  //Berechnung der Produkte und Aufsummation innerhalb des Thread
  int index=blockIdx.x*blockDim.x+threadIdx.x;
  for(int i=0;i<n;i++){
    if(index<dim)
      sum+=a[index]*b[index];
    index+=blockDim.x*gridDim.x;
  }

  shared_summands[threadIdx.x]=sum;
  //Berechnung der Teilsumme des gesamten Blockes
  __syncthreads();
  for(int s=1; s<blockDim.x; s*=2){
    if(threadIdx.x%(2*s) == 0){
      shared_summands[threadIdx.x] += shared_summands[threadIdx.x+s];
    }
    __syncthreads();
  }
  if(threadIdx.x==0)
    c[blockIdx.x] = shared_summands[0];
}
```

Bei der Bildung der Gesamtteilsumme eines Blocks ist im ersten Durchgang jeder zweite Thread innerhalb eines Blocks beteiligt, im zweiten Durchgang jeder vierte Thread, usw. Es werden also innerhalb eines Warp unterschiedliche Programmzweige abgearbeitet, so dass *Divergency* auftritt. Dies lässt sich, wie in Algorithmus 4 umgesetzt, dadurch vermeiden, dass im ersten Durchgang die erste Hälfte aller Threads eines Blocks an der Aufsummierung beteiligt ist, im zweiten Durchgang das erste Viertel aller Threads eines Blocks, usw.

Algorithm 4 Skalarproduktimplementierung ohne Divergency

```
__global__ void k_skp3(Real *c, Real *a, Real *b, int dim){
  int n=dim/(blockDim.x*gridDim.x+1)+1;
  extern __shared__ Real shared_summands[];
  Real sum=0.0;
  //Berechnung der Produkte und Aufsummation innerhalb des Thread
  int index=blockIdx.x*blockDim.x+threadIdx.x;
  for(int i=0;i<n;i++){
    if(index<dim)
      sum+=a[index]*b[index];
    index+=blockDim.x*gridDim.x;
  }
  shared_summands[threadIdx.x]=sum;
  //Berechnung der Teilsumme des gesamten Blockes
  __syncthreads();

    for(int s = 1; s<blockDim.x; s *= 2){
      index=2*s*threadIdx.x;
      if(index < blockDim.x){
        shared_summands[index] += shared_summands[index+s];
      }
      __syncthreads();
    }

  if(threadIdx.x==0)
    c[blockIdx.x] = shared_summands[0];
}
```

Eine weitere Optimierungsmöglichkeit besteht nun darin, Bankkonflikte zu vermeiden. Betrachtet man nochmal die Aufsummation in Algorithmus 4, so sieht man, dass im ersten Durchgang auf jedes zweite Element im Shared-Memory zugegriffen wird, im zweiten Durchgang auf jedes vierte Element, usw. Bei 16 Speicherbänken bedeutet dies, dass innerhalb eines Halbwarp im ersten Durchgang jeweils zwei Threads auf die gleiche Speicherbank zugreifen, im zweiten Durchgang jeweils 4 Threads, usw. Es treten also Bankkonflikte auf.
Eine Verbesserung stellt Algorithmus 5 dar. Hier greifen bei der Aufsummierung aufeinanderfolgende Threads auf aufeinanderfolgende Speicherstellen zu, daher greifen innerhalb eines Halbwarp alle Threads auf unterschiedliche Speicherbänke zu und es kommt zu keinen Bankkonflikten.

Algorithm 5 Skalarproduktimplementierung ohne Bankkonflikte

```
__global__ void k_skp4(Real *c, Real *a, Real *b, int dim){
  int n=dim/(blockDim.x*gridDim.x+1)+1;
  extern __shared__ Real shared_summands[];
  Real sum=0.0;
  //Berechnung der Produkte und Aufsummation innerhalb des Thread
  int index=blockIdx.x*blockDim.x+threadIdx.x;
  for(int i=0;i<n;i++){
    if(index<dim)
      sum+=a[index]*b[index];
    index+=blockDim.x*gridDim.x;
  }
  shared_summands[threadIdx.x]=sum;
  //Berechnung der Teilsumme des gesamten Blockes
  __syncthreads();

  for(int stride = blockDim.x/2; stride>0; stride /= 2) {
    if(threadIdx.x < stride){
      shared_summands[threadIdx.x] += shared_summands[threadIdx.x+stride];
    }
    __syncthreads();
  }

  if(threadIdx.x==0)
    c[blockIdx.x] = shared_summands[0];
}
```

Betrachtet man in Algorithmus 5 die Schleife, bei der die Produkte berechnet werden, so fällt auf, dass die if-Anweisung in der Schleife bis auf den letzten Durchgang immer erfüllt ist. Man könnte somit sehr viele if-Abfragen dadurch einsparen, dass man die Schleife einmal weniger durchläuft und den letzten Durchgang zusammen mit der if-Anweisung hinter die Schleife verschiebt, wie in Algorithmus 6 zu sehen ist.

Algorithm 6 Skalarproduktimplementierung mit weniger if-Anweisungen

```
__global__ void k_skp5(Real *c, Real *a, Real *b, int dim){
  int n=dim/(blockDim.x*gridDim.x+1)+1;
  extern __shared__ Real shared_summands[];
  Real sum=0.0;

  //Berechnung der Produkte und Aufsummation innerhalb des Thread
  int index=blockIdx.x*blockDim.x+threadIdx.x;
  for(int i=0;i<n-1;i++){
    sum+=a[index]*b[index];
    index+=blockDim.x*gridDim.x;
  }
  if(index<dim)
    sum+=a[index]*b[index];

  shared_summands[threadIdx.x]=sum;
  //Berechnung der Teilsumme des gesamten Blockes
  __syncthreads();
  for(int stride = blockDim.x/2; stride>0; stride /= 2) {
    if(threadIdx.x < stride){
      shared_summands[threadIdx.x] += shared_summands[threadIdx.x+stride];
    }
    __syncthreads();
  }
  if(threadIdx.x==0)
    c[blockIdx.x] = shared_summands[0];
}
```

Eine weitere Optimierungsmöglichkeit besteht nun darin, die letzte Schleife, wie in Algorithmus 7 umgesetzt, *abzurollen*. Dadurch können die Schleifenberechnungen eingespart werden. Dies lässt sich relativ einfach umsetzen, da ein Block maximal 512 Threads umfasst und wir die Blockgröße für die baumartige Aufsummierung auf eine Potenz von 2 eingeschränkt haben. Somit wird diese Schleife maximal 9 mal durchlaufen ($2^9 = 512$) und es müssen nur Zweierpotenzen beachtet werden.
Da die letzten 32 arbeitenden Threads eines Blocks immer in dem selben Warp liegen und sie stets mit den selben Anweisungen beschäftigt sind, kann an diesen Stellen die Synchronisation wegfallen. Zudem ist es unnötig die überzähligen Threads innerhalb dieses Warp stillzulegen, da sie sowieso die entsprechende Arbeit verrichten, unabhängig davon, ob wir die entprechenden Ergebnisse nutzen oder nicht.

Algorithm 7 Skalarproduktimplementierung abgerollt

```
__global__ void k_skp6(Real *c, Real *a, Real *b, int dim){
  int n=dim/(blockDim.x*gridDim.x+1)+1;
  extern __shared__ Real shared_summands[];
  Real sum=0.0;
  //Berechnung der Produkte und Aufsummation innerhalb des Thread
  int index=blockIdx.x*blockDim.x+threadIdx.x;
  for(int i=0;i<n-1;i++){
    sum+=a[index]*b[index];
    index+=blockDim.x*gridDim.x;
  }
  if(index<dim)
    sum+=a[index]*b[index];
  shared_summands[threadIdx.x]=sum;

  //Berechnung der Teilsumme des gesamten Blocks
  __syncthreads();
  Real *ppart_sum=shared_summands+threadIdx.x;
  __syncthreads();
  if (blockDim.x >= 512){
    if(threadIdx.x < 256) ppart_sum[0] += ppart_sum[256];
    __syncthreads();
  }
  if (blockDim.x >= 256){
    if(threadIdx.x < 128) ppart_sum[0] += ppart_sum[128];
    __syncthreads();
  }
  if (blockDim.x >= 128){
    if(threadIdx.x < 64) ppart_sum[0] += ppart_sum[64];
    __syncthreads();
  }
  if (threadIdx.x < 32){
    if(blockDim.x >= 64) ppart_sum[0] += ppart_sum[32];
    if(blockDim.x >= 32) ppart_sum[0] += ppart_sum[16];
    if(blockDim.x >= 16) ppart_sum[0] += ppart_sum[8];
    if(blockDim.x >= 8) ppart_sum[0] += ppart_sum[4];
    if(blockDim.x >= 4) ppart_sum[0] += ppart_sum[2];
    if(blockDim.x >= 2) ppart_sum[0] += ppart_sum[1];
  }

  if(threadIdx.x==0)
    c[blockIdx.x] = shared_summands[0];
}
```

Eine weitere Optimierungsmöglichkeit würde sich noch dadurch ergeben, dass man für jede Blockgröße einen eigenen Kernel schreibt, also insgesamt neun verschiedene Kernel. Dadurch würde man noch einige if-Abfragen einsparen können.
Diesen Ansatz haben wir im Weiteren jedoch nicht verfolgt, da er uns in der Benutzung zu unflexibel erschien.

Häufig wird, insbesondere auch beim CG-Verfahren, die Norm bzw. das Quadrat der Norm benötigt. Diese lässt sich durch das Bilden des Skalarprodukts eines Vektors mit sich selbst berechnen. Diese Berechnung lässt sich allerdings, wie in Algorithmus 8 umgesetzt, dahingehend optimieren, dass die Elemente des Vektors nur einmal, nicht wie beim Skalarprodukt zweimal, geladen werden. Dadurch wird die Anzahl der Zugriffe auf den Global-Memory wesentlich reduziert.

Algorithm 8 Berechnung des Quadrates der Norm

```
__global__ void k_normquad(Real *c, Real *a, int dim){
  int n=dim/(blockDim.x*gridDim.x+1)+1;
  extern __shared__ Real shared_summands[];
  Real sum=0.0;
  //Berechnung der Produkte und Aufsummation innerhalb des Thread
  int index=blockIdx.x*blockDim.x+threadIdx.x;

  for(int i=0;i<n-1;i++){
    sum+=a[index]*a[index];
    index+=blockDim.x*gridDim.x;
  }
  if(index<dim)
    sum+=a[index]*a[index];

  shared_summands[threadIdx.x]=sum;
  //Berechnung der Teilsumme des gesamten Blockes
  __syncthreads();
  Real *ppart_sum=shared_summands+threadIdx.x;
  __syncthreads();
  if (blockDim.x >= 512){
    if(threadIdx.x < 256) ppart_sum[0] += ppart_sum[256];
    __syncthreads();
  }
  if (blockDim.x >= 256){
    if(threadIdx.x < 128) ppart_sum[0] += ppart_sum[128];
    __syncthreads();
  }
  if (blockDim.x >= 128){
    if(threadIdx.x < 64) ppart_sum[0] += ppart_sum[64];
    __syncthreads();
  }
  if (threadIdx.x < 32){
    if(blockDim.x >= 64) ppart_sum[0] += ppart_sum[32];
    if(blockDim.x >= 32) ppart_sum[0] += ppart_sum[16];
    if(blockDim.x >= 16) ppart_sum[0] += ppart_sum[8];
    if(blockDim.x >= 8) ppart_sum[0] += ppart_sum[4];
    if(blockDim.x >= 4) ppart_sum[0] += ppart_sum[2];
    if(blockDim.x >= 2) ppart_sum[0] += ppart_sum[1];
  }
  if(threadIdx.x==0)
    c[blockIdx.x] = shared_summands[0];
}
```

Der zweite Kernel (Algorithmus 9) funktioniert, wie bereits erwähnt, genauso wie die Bildung der Teilsumme des ersten Kernel. Wie beim ersten Kernel muss auch hier die Blockgröße auf eine Potenz von 2 eingeschränkt werden, damit die baumartige Aufsummation aufgeht. Zu beachten ist, dass beim Aufruf des zweiten Kernel die Blockgröße größer oder gleich der Anzahl der Teilsummen, die aufsummiert werden, sein muss. Desweiteren können nicht mehr als 512 Teilsummen aufsummiert werden, da dies, wie bereits in Abschnitt 4.2.3 erwähnt wurde, die maximale Blockgröße ist.

Algorithm 9 Zweiter Kernel

```
__global__ void k_sumup(Real* s, Real* c, int cdim){
    extern __shared__ Real part_sum[];
  //Teilsummen laden
  if(threadIdx.x<cdim)
      part_sum[threadIdx.x] = c[threadIdx.x];
  else
    part_sum[threadIdx.x] = 0.0;
    __syncthreads();
  Real *ppart_sum=part_sum+threadIdx.x;
  __syncthreads();
  if (blockDim.x > 256){
    if(threadIdx.x < 256) ppart_sum[0] += ppart_sum[256];
    __syncthreads();
  }
  if (blockDim.x > 128){
    if(threadIdx.x < 128) ppart_sum[0] += ppart_sum[128];
    __syncthreads();
  }
  if (blockDim.x > 64){
    if(threadIdx.x < 64) ppart_sum[0] += ppart_sum[64];
    __syncthreads();
  }
  if (threadIdx.x < 32){
    if(blockDim.x > 32) ppart_sum[0] += ppart_sum[32];
    if(blockDim.x > 16) ppart_sum[0] += ppart_sum[16];
    if(blockDim.x > 8) ppart_sum[0] += ppart_sum[8];
    if(blockDim.x > 4) ppart_sum[0] += ppart_sum[4];
    if(blockDim.x > 2) ppart_sum[0] += ppart_sum[2];
    if(blockDim.x > 1) ppart_sum[0] += ppart_sum[1];
  }
  if(threadIdx.x == 0)
    *s = ppart_sum[0];
}
```

Messung und Vergleich der einzelnen Skalarproduktversionen

Für eine Testreihe wurden zwei Vektoren der Dimension *dim* mit Zufallszahlen initialisiert, wobei $dim = \{10^2, 10^3, ..., 10^8\}$ gewählt wurde. Beim Skalarprodukt werden $N = 2 \cdot dim$ Floating Point Operations, kurz *Flops*, berechnet (*dim* für die Produkte und *dim* für die Aufsummierung). Zusammen mit der gemessenen Ausführungszeit t (in Sekunden) lassen sich damit die Geschwindigkeiten der unterschiedlichen Kernel nach $P = \frac{2 \cdot dim}{t} \cdot Flops$ berechnen. Die Geschwindigkeiten wurden jeweils für alle Kombinationen aus den Blockgrößen (32, 64, 128, 256, 512) und den Gittergrößen (30, 60, 120, 240, 510) gemessen. Die schnellste ist jeweils in Abbildung 16 aufgetragen.

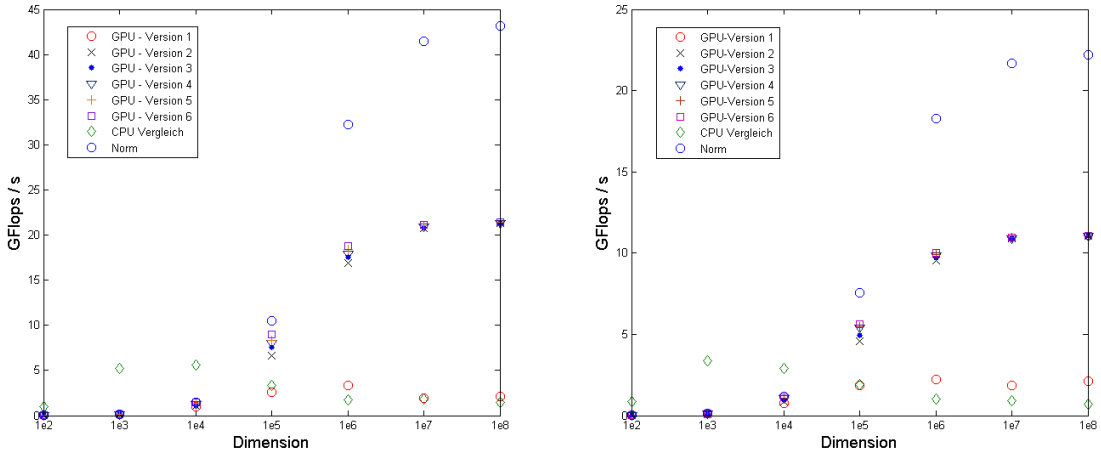

Abbildung 16: Vergleich der Performance der verschiedenen Skalarprodukt Implementierungen für einfache (links) und doppelte (rechts) Genauigkeit.

Zunächst fällt auf, dass die Geschwindigkeiten aller GPU-Implementierungen für kleine Dimensionen sehr gering sind und erst ab einer Dimension zwischen 1.000.000 und 10.000.000 ihr Maximum erreichen. Erst ab einer Dimension im Bereich von 10.000 bis 100.000 sind die GPU-Implementierungen deutlich schneller als die CPU-Implementierung. Deutlich zu erkennen ist weiterhin, dass lediglich die erste Optimierung (die für *Coalescing* sorgt) eine deutliche Performance-Steigerung bewirkt hat. Da dies die einzige Optimierung ist, die für eine bessere Ausnutzung der Speicherbandbreite zum Global-Memory sorgt, liegt es nahe zu folgern, dass gerade hier der entscheidende Flaschenhals liegt. Dies wird nochmal dadurch unterstrichen, dass die Berechnung des Quadrates der Norm deutlich schneller ist als die Skalarproduktberechnungen. Bedenkt man, dass beim Skalarprordukt für jeweils $2Flops$ (eine Multiplikation und eine Addition) zwei Werte aus dem Global-Memory geladen werden müssen, so lassen sich die maximal zu erreichenden Geschwindigkeiten für einfache (4 Byte) und doppelte (8 Byte) Genauigkeit aus der maximalen Bandbreite $B_{C1060} = 102 GByte/s$ berechnen:

$$P_{max\_skp\_single} = \frac{102 GByte/s \cdot 2 Flops}{2 \cdot 4 Byte} = 25.5 GFlops/s$$

$$P_{max\_skp\_double} = \frac{102 GByte/s \cdot 2 Flops}{2 \cdot 8 Byte} = 12.75 GFlops/s$$

Angesichts der Tatsache, dass dabei der Overhead der Kernelaufrufe, das Speichern der Teilsummen der Blöcke und das Laden der Teilsummen der Blöcke vernachlässigt wurde, sind wir mit etwa $22\,GFlops/s$ bzw. $11\,GFlops/s$ dieser maximalen Geschwindigkeit sehr nahe gekommen.

4.3.2 Sparsematrix-Vektor Multiplikation für Bandmatrizen

Für die Sparsematrizen haben wir das in [3] beschriebene DIA- und ELL-Format verwendet. Das DIA-Format eignet sich hervorragend zur Speicherung von Bandmatrizen während das ELL-Format besonders zur Speicherung von Matrizen, die in jeder Zeile gleich viele von Null verschiedene Einträge haben, geeignet ist. Da es sich bei den Koeffizientenmatrizen der hier auftretenden linearen Gleichungssysteme um Bandmatrizen handelt, beschränken wir uns im Folgenden auf die Beschreibung des DIA-Formats. Für das ELL-Format sei auf [3] verwiesen.

Algorithmus

Beim DIA-Format wird jede Diagonale als ein Vektor in einer Matrix `data`, bei der untersten Diagonale beginnend, so gespeichert, dass sich die Elemente der Diagonalen in der Zeile befinden, in der sie auch in der Originalmatrix stehen würden. Zudem speichert man in einem Vektor `offsets` die Abstände der Diagonalen zur Hauptdiagonale, wobei die Hauptdiagonale den Offset 0 hat. Die darunterliegenden Nebendiagonalen haben negative Offsets und die darüberliegenden Nebendiagonalen haben entsprechend positive Offsets. Ein Beispiel für eine 4x4-Matrix ist in Abbildung 17 zu sehen.

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix} \quad \rightarrow \quad \text{data} = \begin{bmatrix} * & 1 & 7 \\ * & 2 & 8 \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix}, \quad \text{offset} = \begin{bmatrix} -2 & 0 & 1 \end{bmatrix}$$

Abbildung 17: Das DIA-Speicherformat am Beispiel

Implementierung

Eine erste Implementierung ist der aus [3] entnommene Algorithmus 10. Jeder Thread berechnet ein Element des Ergebnisvektors `y`, indem eine Zeile der DIA-Matrix mit dem Vektor `x` multipliziert wird. Da die Spalten der `data`-Matrix hintereinander im Speicher liegen, wird zusätzlich ein Parameter `stride` übergeben, welcher angibt nach wie vielen Speicherstellen die nächste Spalte der Matrix beginnt.

Algorithm 10 Erste DIA-Sparsematrix-Vektor Multiplikation

```
__global__ void k_spmv_dia1 (Real* y,          //Ergebnisvektor
                             Real* data,       //data-Matrix
                             int *offsets,     //Offsetvektor
                             Real *x,          //wird an Matrix multipliziert
                             int dim_r ,       //Anzahl der Zeilen der Sparsematrix
                             int dim_c ,       //Anzahl der Spalten der Sparsematrix
                             int dim_offsets, //Anzahl der Diagonalen
                             int stride){
  //Berechnung der Zeile
  int row = blockDim.x * blockIdx.x + threadIdx.x;
  if(row < dim_r ){
    Real dot = 0.0;
    for ( int n = 0; n < dim_offsets ; n++){
      //Berechnung der Spalte
      int col = row + offsets [n];
      if( col >= 0 && col < dim_c)
        dot += data[stride * n + row] * x[col];
    }
    y[row] = dot;
  }
}
```

Bei diesem Algorithmus tritt bereits *Coalescing* beim Zugriff auf die `data`-Matrix auf, während *Divergency* auf ein Minimum reduziert ist. Beim Zugriff auf den `x`-Vektor tritt kein *Coalescing* auf. Dies lässt sich allerdings nicht vermeiden, da sich diese Zugriffe nicht ordnen lassen. Wie bereits beim Skalarprodukt festgestellt, ist auch bei der Sparsematrix-Vektor Multiplikation der entscheidende Flaschenhals der Zugriff auf den Global-Memory, da alle Daten nur für eine Berechnung benötigt werden. Somit ist es sinnvoll die Zugriffe auf den Global-Memory auf ein Minimum zu reduzieren. Betrachtet man den Algorithmus 10 genauer, so fällt auf, dass jeder Thread auf alle Elemente des `offset`-Vektors zugreift. Dies wurde in Algorithmus 11 dahingehend verbessert, dass die ersten Threads jedes Blocks den `offset`-Vektor im Shared-Memory speichern, so dass alle anderen Threads darauf zugreifen können.

Algorithm 11 DIA-Sparsematrix-Vektor Multiplikation mit Shared-Memory

```
__global__ void k_spmv_dia2(Real* y,Real* data,int *offsets,Real *x,
                            int dim_r,int dim_c ,int dim_offsets,int stride){
  int row = blockDim.x * blockIdx.x + threadIdx.x;

  extern __shared__ int shared_offsets[];
  if(threadIdx.x < dim_offsets)
    shared_offsets[threadIdx.x] = offsets[threadIdx.x];
  __syncthreads();

  if(row < dim_r ){
    Real dot = 0.0;
    for ( int n = 0; n < dim_offsets ; n ++){
      int col = row + shared_offsets[n];
      Real val = data[dim_r * n + row];
      if( col >= 0 && col < dim_c)
        dot += val * x[col];
    }
    y[row] = dot;
  }
}
```

Wie bereits erwähnt, tritt beim Zugriff auf den `x`-Vektor kein *Coalescing* auf und es wird somit wertvolle Speicherbandbreite verschwendet. Eine Möglichkeit dieses Problem zu umgehen, ist den `x`-Vektor als Textur zu binden. Dadurch erfolgt der Zugriff auf den Global-Memory gecacht und es wird keine Speicherbandbreite verschwendet. Leider unterstüzt CUDA zur Zeit den Type `double` für Texturen nicht. Allerdings gibt es den Type `int2`, welcher ebenfalls wie `double` 8 Byte groß ist und unterstützt wird. Somit kann man in diesem Fall den `x`-Vektor als `int2`-Textur binden. In Algorithmus 12 muss daher bei der Verwendung von `doubles` der gelesene `int2` Wert in ein `double` umgewandelt werden und mit Hilfe des Präprozessors zwischen `float` und `double` unterschieden werden.

Algorithm 12 DIA-Sparsematrix-Vektor Multiplikation mit Verwendung des Texture-Cache

```
__global__ void k_spmv_dia3(Real* y,Real* data,int *offsets,Real *x,
                            int dim_r,int dim_c ,int dim_offsets,int stride){
  int row = blockDim.x * blockIdx.x + threadIdx.x;
  Real d;
  extern __shared__ int shared_offsets[];
  if(threadIdx.x < dim_offsets)
    shared_offsets[threadIdx.x] = offsets[threadIdx.x];
  __syncthreads();
  if(row < dim_r ){
    Real dot = 0.0;
    for ( int n = 0; n < dim_offsets ; n ++){
      int col = row + shared_offsets[n];
      Real val = data[dim_r * n + ro w];
      if( col >= 0 && col < dim_c){

        #if PRAEZISSION == 2
          int2 temp = tex1Dfetch(Tex,col);
          d = __hiloint2double(temp.y,temp.x);
        #else
          d = tex1Dfetch(Tex,col);
        #endif
        dot += val * d;

      }
    }
    y[row] = dot;
  }
}
```

Messung und Vergleich der einzelnen Versionen

Als Testmatrizen wurden Matrizen der Dimension $dim = \{10^2, 10^3, ..., 10^7\}$ verwendet, die die gleiche Struktur haben wie die Diskretisierungsmatrix des Laplaceoperators und mit Zufallswerten belegt wurden.
Bei der Multiplikation wird jedes Element der `data`-Matrix mit einem Element des `x`-Vektors multipliziert und auf das entsprechende Element des Ergebnisvektors `y` aufaddiert. In unserem Fall werden also etwa $5 \cdot 2 \cdot dim$ Flops berechnet, da die Matrizen fünf Diagonalen haben. Zusammen mit der gemessenen Ausführungszeit der verschiedenen Kernel lassen sich damit die Geschwindigkeiten der unterschiedlichen Kernel in $GFlops/s$ berechen. Die Geschwindigkeiten wurden jeweils für unterschiedliche Blockgrößen (32, 64, 128, 256, 512) gemessen. Die schnellste ist jeweils in Abbbildung 18 dargestellt.

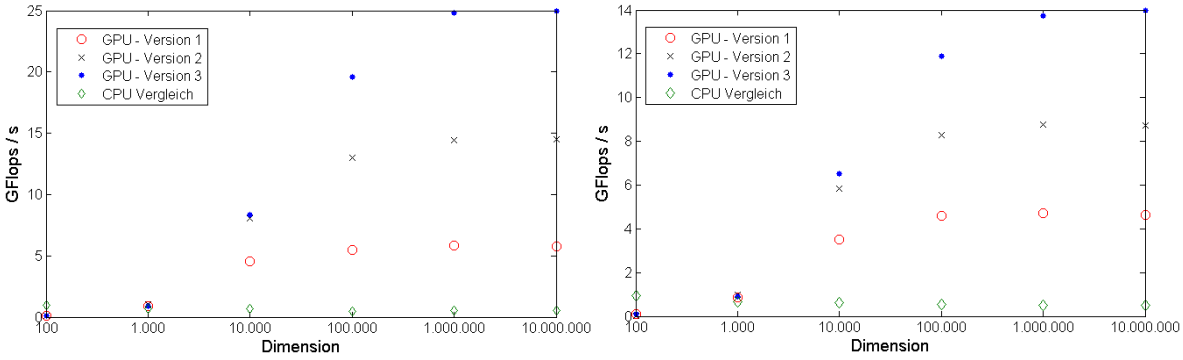

Abbildung 18: Vergleich der Performance der verschiedenen Matrix-Vektor Implementierungen für einfache (links) und doppelte (rechts) Genauigkeit.

Zunächst fällt auf, dass die Geschwindigkeiten ähnlich wie beim Skalarprodukt für kleine Dimensionen sehr gering sind und bei einer Dimension zwischen 100.000 und 1.000.000 ihr Maximum erreichen. Die CPU-Implementierung hingegen bleibt über alle Dimensionen hinweg nahezu konstant. Erst ab einer Dimension im Bereich von 1.000 bis 10.000 sind die GPU-Implementierungen deutlich schneller als die CPU-Implementierung. Wie man deutlich sieht, haben beide Optimierungen die Geschwindigkeit deutlich angehoben. So stieg das Maximum der Geschwindigkeit für einfache Genauigkeit von 5GFlops/s über 15GFlops/s auf 25GFlops/s und für doppelte Genauigkeit von 5GFlops/s über 9GFlops/s auf 14GFlops/s.
Da beide Optimierungen die Ausnutzung der Speicherbandbreite zum Global-Memory verbessert haben, liegt es auch hier nahe, dass die Speicherbandbreite, ähnlich wie beim Skalarprodukt, der entscheidende Flaschenhals ist. Theoretisch werden für eine Sparsematrix-Vektor Multiplikation etwa $5 \cdot dim$ Werte aus den fünf Diagonalen geladen, dim Werte aus dem x-Vektor geladen und dim Werte im y-Vektor gespeichert. Daraus lassen sich wieder, wie beim Skalarprodukt, die maximalen Geschwindigkeiten aus der maximalen Bandbreite B_{C1060} für einfache (4Byte) und doppelte (8Byte) Genauigkeit berechnen:

$$P_{theo\_spmv\_max\_single} = \frac{102 GByte/s \cdot 5 \cdot dim \cdot 2 Flops}{(5+1+1) \cdot dim \cdot 4 Byte} = 36.4 GFlops/s$$

$$P_{theo\_spmv\_max\_double} = \frac{102 GByte/s \cdot 5 \cdot dim \cdot 2 Flops}{(5+1+1) \cdot dim \cdot 8 Byte} = 18.2 GFlops/s$$

Bei unseren Implementierungen werden allerdings die Elemente des x-Vektors fünf mal (für jede Diagonale einmal) geladen, da die Zugriffe ungeordnet ablaufen und wir keine Möglichkeit gefunden haben, diese derart zu ordnen, dass wir sie sinnvoll im Shared-Memory speichern könnten. Für diesen Fall ergeben sich folgende maximale Geschwindigkeiten:

$$P_{spmv\_max\_single} = \frac{102 GByte/s \cdot 5 \cdot dim \cdot 2 Flops}{(5+5+1) \cdot dim \cdot 4 Byte} = 23.2 GFlops/s$$

$$P_{spmv\_max\_double} = \frac{102 GByte/s \cdot 5 \cdot dim \cdot 2 Flops}{(5+5+1) \cdot dim \cdot 8 Byte} = 11.6 GFlops/s$$

Interessant ist nun, dass die dritte Implementierung bei doppelter Genauigkeit mit $14 GFlops$ schneller ist, als die Maximalgeschwindigkeit und zwischen $P_{theo\_spmv\_max\_double}$ und

$P_{spmv\_max\_double}$ liegt. Eine Begründung hierfür ist, dass Daten, die bereits für einen anderen Thread aus dem Global-Memory in den Texture-Cache geladen wurden, nicht nochmal geladen werden müssen, sofern diese noch im Texture-Cache liegen. Bedenkt man, dass wir dabei das Laden der Offsets und den Overhead des Kernelaufrufs vernachlässigt haben, so sieht man, dass wir auch hier dem maximalen theoretischen Geschwindigkeiten sehr nahe gekommen sind.

4.4 CG Verfahren

Um ein lineares Gleichungssystem der Form $Ax = b, A \in \mathbb{R}^{N,N}, x, b \in \mathbb{R}^N$ mit dünnbesetzter Matrix A effizient zu lösen, bietet sich das in [7] vorgestellte und gut bekannte CG-Verfahren an. Hierbei handelt es sich um ein *iteratives* Lösungsverfahren aus der Klasse der Krylov-Unterraum-Verfahren, welches linearen Gleichungsystemen mit symmetrisch positiv definiter Koeffizientenmatrix löst. Wie bereits in Abschnitt 3.1.1 und 3.3 erwähnt, ist diese Forderung für die hier auftretenden linearen Gleichungssysteme erfüllt.

Algorithmus

Beim CG-Cerfahren wird, ausgehend von einem Startvektor x_0, in jeder Iteration ein Vektor x_j berechnet, der gegen die Lösung x des Gleichungssystems konvergiert und spätestens nach N Iterationen die exakte Lösung liefert.
Wie im Pseudocode des CG-Verfahrens (Algorithmus 13) zu sehen ist, besteht der Hauptaufwand im Berechnen von Matrix-Vektor Multiplikationen und Skalarprodukten. Das macht das CG-Verfahren besonders geeignet für die Grafikkarte, da sich diese Operationen, wie in den beiden vorherigen Kapiteln beschrieben, sehr gut parallelisieren lassen.
Da das Verfahren bereits nach wenigen Iterationsschritten eine Lösung mit ausreichender Genauigkeit liefert, wird in der Regel noch ein Abbruchkriterium vorgegeben. Dabei handelt es sich um eine Toleranz *tol*, die die geforderte Genauigkeit der Lösung vorgibt und um eine maximale Anzahl $MAXIT$ an durchzuführenden Iterationschritten.
Dabei hängt die Anzahl der benötigten Iterationen in der Regel von der Konditionszahl $\kappa(A) := \|A\|_2 \left\|A^{-1}\right\|_2$ der Systemmatrix A ab. Eine genaue Abschätzung für den relativen Fehler im j-ten Iterationsschritt durch die Konditionszahl wird zum Beispiel in [8] diskutiert.

Algorithm 13 CG-Verfahren zur Lösung von $Ax = b$ (Pseudocode)

Input: $A \in \mathbb{R}^{N,N}$, $b, x_0 \in \mathbb{R}^N$, $MAXIT \in \mathbb{N}$, $tol \in \mathbb{R}_+$
Output: Lösung $x \in \mathbb{R}^N$

$r_0 = b - Ax_0$
$p_0 = r_0$
for $j = 1$ to $MAXIT$ **do**
  $\gamma_{j-1} = (r_{j-1}^T r_{j-1})/(p_{j-1}^T A p_{j-1})$
  $x_j = x_{j-1} + \gamma_{j-1} p_{j-1}$
  $r_j = r_{j-1} - \gamma_{j-1} A p_{j-1}$
  if $\|r_j\|_2 / \|r_0\|_2 < tol$ **then**
    return x_j
  end if
  $\beta_j = r_j^T r_j / r_{j-1}^T r_{j-1}$
  $p_j = r_j + \beta_j p_{j-1}$
end for

Implementierung

Eine erste Implementierung des CG-Verfahrens stellt die Funktion `CG_GPU_DIA1()` (Algorithmus 14 im Anhang) dar. Dieser werden die Koeffizientenmatrix `A`, die Rechte Seite `b`, die Toleranz *tol*, die maximale Anzahl `MAXIT` an durchzuführenden Interationen, sowie der Vektor `xj`, in dem die Lösung gespeichert wird und der gleichzeitig Startvektor ist, übergeben. Zu Beginn der Funktion wird der benötigte Speicher auf der Grafikkarte allokiert und die Koeffizientenmatrix `A`, die Rechte Seite `b` und der Startvektor `xj` in den Global-Memory geladen. Anschließend wird der CG-Algorithmus wie im obigen Pseudocode ausgeführt, wobei sämtliche Operationen nacheinander durch den Aufruf der entsprechenden Kernel auf der Grafikkarte berechnet werden. Die Lösung wird dann wieder in den Hauptspeicher geladen und der verwendete Grafikkartenspeicher wieder freigegeben.
Wie bereits erwähnt, sind Kopiervorgänge zwischen Grafikkartenspeicher und Hauptspeicher vergleichsweise langsam. Eine mögliche Optimierung ergibt sich daher, wie in der Funktion `CG_GPU_DIA2()` (Algorithmus 15 im Anhang) dargestellt, daraus, dass man der Funktion die Zeiger auf die Daten im Grafikkartenspeicher übergibt. Dadurch muss kein weiterer Speicher auf der Grafikkarte allokiert werden und man spart so die langsamen Kopiervorgänge ein.
Die Funktion `CG_CPU_DIA()` (Algorithmus 16 im Anhang) stellt die sequentielle Implementierung des CG-Verfahrens dar, die für die folgenden Geschwindigkeitstests als Vergleich verwendet wurde.

Messung und Vergleich der einzelnen Versionen

Die Geschwindigkeit der drei Implementierungen des CG-Verfahrens wurde jeweils auf drei unterschiedlich großen Gittern für die in jedem Zeitschritt auftretenden linearen Gleichungssysteme gemessen. Hierbei sind das aus der Diskretisierung der Poissongleichung stammende LGS mit Koeffizientenmatrix A_h sowie die beim impliziten Euler-Verfahren auftretenden linearen Gleichungssysteme zur Lösung der Wirbeltransportgleichung und der Wärmeleitungs-

gleichung mit den Koeffizientenmatrizen B_h^w und B_h^T zu lösen. Die Rechten Seiten wurden für die Messung jeweils durch Vektoren ersetzt, die mit Zufallswerten belegt wurden. Dabei wurde jeweils ein 64×64, ein 256×256 sowie ein 1024×1024 Gitter verwendet.

Gitter	LGS	Iter.	CG_GPU_DIA1		CG_GPU_DIA2		CG_CPU_DIA	
64×64	A_h	225	$22.52ms$	$0.77GF$	$22.06ms$	$0.79GF$	$156ms$	$0.11GF$
64×64	B_h^w	15	$1.63ms$	$0.74GF$	$1.50ms$	$0.80GF$	$10.9ms$	$0.11GF$
64×64	B_h^T	1	$0.29ms$	$0.45GF$	$0.16ms$	$0.81GF$	$0.80ms$	$0.16GF$
256×256	A_h	932	$151.5ms$	$7.94GF$	$149.9ms$	$8.03GF$	$12.57s$	$0.10GF$
256×256	B_h^w	54	$10.34ms$	$6.82GF$	$8.78ms$	$8.04GF$	$631ms$	$0.11GF$
256×256	B_h^T	1	$1.70ms$	$1.30GF$	$0.26ms$	$8.29GF$	$13.29ms$	$0.17GF$
1024×1024	A_h	3786	$4.68s$	$16.9GF$	$4.65s$	$17.0GF$	$11.4min$	$0.12GF$
1024×1024	B_h^w	215	$0.28s$	$16.3GF$	$0.27s$	$17.0GF$	$40.8s$	$0.11GF$
1024×1024	B_h^T	2	$13.6ms$	$4.16GF$	$3.24ms$	$17.4GF$	$0.4s$	$0.13GF$

Tabelle 1: Zeitmessungen CG-Verfahren in einfacher Genauigkeit

Gitter	LGS	Iter.	CG_GPU_DIA1		CG_GPU_DIA2		CG_CPU_DIA	
64×64	A_h	202	$22.75ms$	$0.69GF$	$22.96ms$	$0.68GF$	$137ms$	$0.11GF$
64×64	B_h^w	15	$1.97ms$	$0.61GF$	$1.68ms$	$0.72GF$	$10.8ms$	$0.11GF$
64×64	B_h^T	1	$0.36ms$	$0.36GF$	$0.18ms$	$0.74GF$	$1.31ms$	$0.10GF$
256×256	A_h	801	$182.4ms$	$5.67GF$	$180.5ms$	$5.73GF$	$9011ms$	$0.11GF$
256×256	B_h^w	54	$14.46ms$	$4.88GF$	$12.21ms$	$5.78GF$	$624ms$	$0.11GF$
256×256	B_h^T	1	$2.48ms$	$0.88GF$	$0.36ms$	$6.03GF$	$22.35ms$	$0.10GF$
1024×1024	A_h	3564	$7.45s$	$9.99GF$	$7.39s$	$10.08GF$	$10.1min$	$0.12GF$
1024×1024	B_h^w	215	$0.47s$	$9.67GF$	$0.45s$	$10.04GF$	$40.66s$	$0.11GF$
1024×1024	B_h^T	2	$21.8ms$	$2.59GF$	$5.36ms$	$10.52GF$	$0.57s$	$0.11GF$

Tabelle 2: Zeitmessungen CG-Verfahren in doppelter Genauigkeit

Wie bereits beim Skalarprodukt und der Sparsematrix-Vektor Multiplikation erreichen auch die GPU-Implementierungen des CG-Verfahrens erwartungsgemäß erst mit größer werdender Dimension bzw. Gittergröße ihre maximale Geschwindigkeit. Diese liegt für einfache Genauigkeit bei etwa $17 GFlops/s$ und für doppelte Genauigkeit bei etwa $10 GFlops/s$. Die Geschwindigkeit der CPU-Implementierung ist mit $0.11 GFlops/s$ bis $0.17 GFlops/s$ deutlich langsamer als die GPU-Implementierungen (bis zu 100 mal langsamer). Auch zeigt sich, dass die zweite GPU-Implementierung stets schneller ist als die erste. Besonders deutlich werden die Unterschiede bei nur wenigen Iterationschritten des CG-Verfahrens, da sich dann die Dauer der Kopiervorgänge verhältnismäßig stärker auswirkt. So ist die zweite GPU-Implementierung mit einfacher Genauigkeit bei zwei Iterationschritten mit $17.4 GFlops/s$ auf dem 1024×1024 Gitter mehr als viermal so schnell wie die erste GPU-Implementierung mit $4.16 GFlops/s$. Für eine große Anzahl an Iterationsschritten ist die zweite Implementierung hingegen nur geringfügig schneller als die erste Implementierung. Dabei hängt die Anzahl der benötigten Iterationen, wie zuvor beschrieben, wesentlich von der Konditionszahl der Systemmatrix ab. Wie in [6] dargestellt, wächst die Konditionszahl der Laplace-Matrix mit steigender Dimension N, was

erklärt, weshalb ein großer Anteil der Rechenzeit zum Lösen des LGS mit Koeffizientenmatrix A_h benötigt wird. Insbesondere die Matrix B_h^T hat bei kleiner Schrittweite eine Konditionszahl nahe Eins, da bei ihrer Konstruktion von der Einheitsmatrix eine Matrix mit sehr kleinen Einträgen abgezogen wird. Das CG-Verfahren konvergiert daher schon nach wenigen Iterationen mit der gewünschten Genauigkeit.

4.5 Erzielter Speedup des Solvers

Da bei einer großen Anzahl an Zeitschritten der Aufwand des Preprocessing vernachlässigbar ist, wurden nur Operationen auf die GPU ausgelagert, die innerhalb eines Zeitschritts berechnet werden. Der im Anhang A.2 befindliche Algorithmus 18 stellt die Implementierung zur Berechnung der Zeitschritte auf der CPU dar. Hierbei wird zunächst, wie in Abschnitt 3.4 beschrieben, die Stromfunktion Ψ, durch Lösen der Poissongleichung mit dem CG-Verfahren, berechnet. Aus der Stromfunktion werden dann die Geschwindigkeitswerte ermittelt und auf die entsprechend anderen Gitter interpoliert. Anschließend werden die Wirbelstärke ω und die Temperatur T durch Euler Zeitintegration aktualisiert, wofür zwei weitere lineare Gleichungssysteme mittels CG-Verfahren gelöst werden. Am Ende eines Zeitschritts werden die berechneten Daten für Geschwindigkeit und Temperatur in externen Dateien abgespeichert.
In einer ersten GPU-Variante wurde in der zuvor beschriebenen sequentiellen Version die Funktion `CG_CPU_DIA()` durch die auf der GPU laufende CG-Implementierung `CG_GPU_DIA1()` ersetzt, da das Lösen der drei linearen Gleichungssysteme einen wesentlichen Teil des Aufwands darstellt.
Um unnötige Kopiervorgänge zwischen Host und Device zu vermeiden, wurden in einer verbesserten GPU-Variante die linearen Gleichungssysteme mit der Funktion `CG_GPU_DIA2()` gelöst. Dazu war es allerdings notwendig, alle übrigen Operationen des Zeitschritts als eigene Kernel auf die GPU auszulagern, damit alle Daten auf dem Grafikkartenspeicher vorliegen. Die zugehörige Implementierung ist in Algorithmus 17 im Anhang A.2 zu finden.

Die in Tabelle 3 und 4 präsentierten Messergebnisse vergleichen nun die drei verschiedenen Implementierungen auf Gittern der Größe 64×64, 256×256 und 1024×1024. Dabei wurde die Dauer des gesamten Zeitschritts gemessen und in die Dauer zum Lösen der drei LGSs, des Abspeicherns und der restlichen Operationen unterteilt.

Gitter	Version	Gesamt	LGS lösen	Abspeichern	Rest
64×64	GPU_1	$53.35ms$	$26.06ms$	$26.70ms$	$0.58ms$
64×64	GPU_2	$52.89ms$	$25.41ms$	$27.19ms$	$0.30ms$
64×64	CPU	$204.6ms$	$176.7ms$	$27.40ms$	$0.50ms$
256×256	GPU_1	$0.53s$	$0.17s$	$0.34s$	$6.72ms$
256×256	GPU_2	$0.52s$	$0.17s$	$0.35s$	$0.85ms$
256×256	CPU	$13.02s$	$12.67s$	$0.34s$	$6.83ms$
1024×1024	GPU_1	$12.00s$	$5.27s$	$6.62s$	$0.11s$
1024×1024	GPU_2	$11.74s$	$5.20s$	$6.53s$	$10.35ms$
1024×1024	CPU	$9.13min$	$9.03min$	$5.50s$	$0.13s$

Tabelle 3: Zeitmessungen eines Zeitschritts bei einfacher Genauigkeit

Gitter	Version	Gesamt	LGS lösen	Abspeichern	Rest
64×64	GPU_1	$52.84ms$	$24.98ms$	$27.36ms$	$0.5ms$
64×64	GPU_2	$51.68ms$	$23.97ms$	$27.41ms$	$0.31ms$
64×64	CPU	$178.4ms$	$150.1ms$	$27.76ms$	$0.5ms$
256×256	GPU_1	$0.58s$	$0.21s$	$0.37s$	$7.1ms$
256×256	GPU_2	$0.56s$	$0.20s$	$0.35s$	$0.93ms$
256×256	CPU	$9.90s$	$9.56s$	$0.34s$	$7.06ms$
1024×1024	GPU_1	$16.02s$	$7.63s$	$8.27s$	$0.13s$
1024×1024	GPU_2	$15.67s$	$7.41s$	$8.25s$	$12.3ms$
1024×1024	CPU	$9.15min$	$9.07min$	$5.45s$	$0.13s$

Tabelle 4: Zeitmessungen eines Zeitschritts bei doppelter Genauigkeit

Deutlich zu sehen ist, dass das Abspeichern der Daten in allen Konfigurationen einen erheblichen Anteil am Gesamtaufwand aufweist, weshalb es bei kleinen Schrittweiten ratsam ist, die Daten nicht in jedem Zeitschritt abzuspeichern. Die beiden GPU-Varianten unterscheiden sich in ihrer Performance beim Lösen der drei linearen Gleichungssysteme kaum. Das liegt daran, dass, wie im vorherigen Abschnitt bereits beschrieben, das Lösen der Poissongleichung den wesentlichen Teil des Rechenaufwands beim Lösen der drei linearen Gleichungssysteme darstellt und aufgrund der großen Anzahl an benötigten Iterationsschritten bei beiden GPU-Varianten etwa gleich lange dauert. Die übrigen Operationen (in obigen Tabellen mit „Rest“ bezeichnet) während des Zeitschritts werden bei der zweiten GPU-Variante etwa 10 mal so schnell durchgeführt wie von der ersten GPU-Variante. Gleichwohl ist die erste GPU-Variante nahezu genau so schnell wie die zweite Variante, da Abspeichern der Daten und Lösen der lineraren Gleichungssysteme den Hauptaufwand darstellen und bei beiden Varianten nahezu gleich schnell sind.

Im Vergleich zur CPU-Variante ist auf großen Gittern allerdings ein erheblicher Speedup zu verzeichnen, der bei der Messung auf dem 1024×1024 Netz bei über 35-facher Beschleunigung liegt.

5 Numerische Tests

Zur Validierung der zuvor diskretisierten Gleichungen sollen im Folgenden verschiedene physikalische Experimente simuliert werden. Hierzu werden zunächst alle verwendeten Stoffparameter aus [1] entnommen und auf die entsprechenden Werte von Wasser bei einem Druck von $p = 1bar$ und einer Umgebungstemperatur von $T_0 = 20\ °\mathrm{C}$ gesetzt.

Parameter	Wert
kinematische Viskosität	$\nu = 1000\ m^2 s^{-1}$
Wärmeleitfähigkeit	$\lambda = 0.597\ Wm^{-1}K^{-1}$
Dichte	$\rho = 1000\ kg\ m^{-3}$
Wärmekapazität	$c_w = 4187\ J\ kg^{-1}\ K^{-1}$
therm. Ausdehnungskoeffizient	$\beta = 0.21 \cdot 10^{-3}\ K^{-1}$

Die Abmaße des Simulationsgebiets Ω sowie die Anzahl der Stützstellen in x_1 und x_2 Richtung sind dabei im Strömungslöser variabel gehalten und können somit problemangepasst variiert werden.

5.1 Driven Cavity

Ein Standardtest zur Verifizierung der Ergebnisse eines neu implementierten Strömungslösers ist der so genannte *Driven Cavity* Versuch. Hierbei wird die Geschwindigkeit auf dem Rand auf Null gesetzt mit Ausnahme der x_1-Komponente auf Γ_1. Hier gibt man die Dirichlet-Randbedingung $u_1 = \tilde{u}$ vor (siehe Abbildung 19). Die Berechnung wird isotherm durchgeführt.

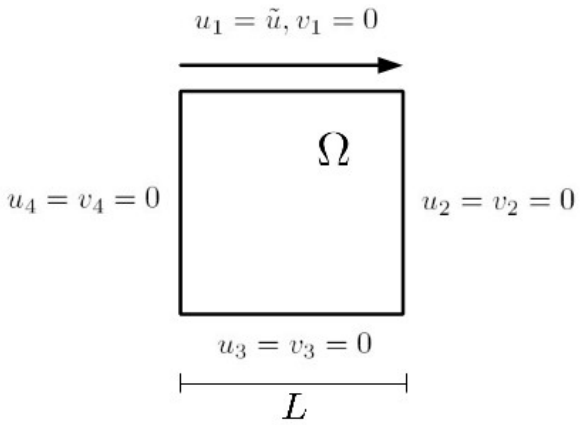

Abbildung 19: Driven Cavity Versuch

Dieser Versuch ist vor allem deshalb von Interesse, da das erwartete Strömungsbild stationär und von der Reynoldszahl abhängig ist, die bei diesem Aufbau via

$$Re = \frac{c^* \cdot d^*}{\nu}$$

definiert ist, wobei $c^* = \tilde{u}$ als charakteristische Geschwindigkeit und $d^* = L$ als charakteristische Länge gewählt wird.
In einem ersten Rechenbeispiel wurde auf einem $100m \times 100m$ Gitter mit 16.384 Gitterpunkten auf dem Hauptnetz (128 Knoten in jede Dimension) gerechnet. Als Geschwindigkeit auf Γ_1 wurde $\tilde{u} = 100m/s$ vorgegeben. Wählt man zusätzlich als Viskosität den zehnfachen Wert von Wasser, so ergibt sich $Re = 1$. Nach weniger als $T_{end} = 1s$ Simulationszeit stellt sich der in (20) gezeigte stationäre Zustand für die Geschwindigkeitsverteilung ein.

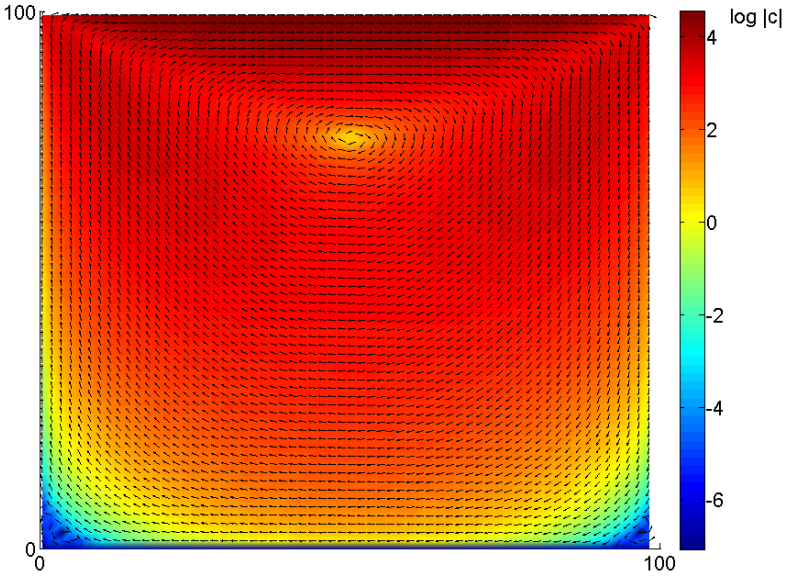

Abbildung 20: Driven Cavity für Re=1. Dargestellt sind die normierten Vektoren der Geschwindigkeit sowie deren logarithmierter Betrag.

Aus der Literatur (zum Beispiel diskutiert in [4]) kann entnommen werden, dass sich bei höherer Reynoldszahl zunächst das Zentrum des Wirbels verschiebt und bei weiterer Erhöhung der Reynoldszahl sich weitere kleinere Wirbel in den unteren Ecken des Simulationsgebiets bilden. Darüber hinaus entsteht im linken oberen Bereich des Diskretisierungsgebiets ein weiterer Wirbel, der erst bei größerer Genauigkeit aufgelöst wird. Die Simulationsergebnisse für die Reynoldszahlen $Re = 100$ und $Re = 3.333$ sind in Abbildung (21) und Abbildung (22) zu finden, wobei ein stationärer Zustand entsprechend später erreicht wurde.

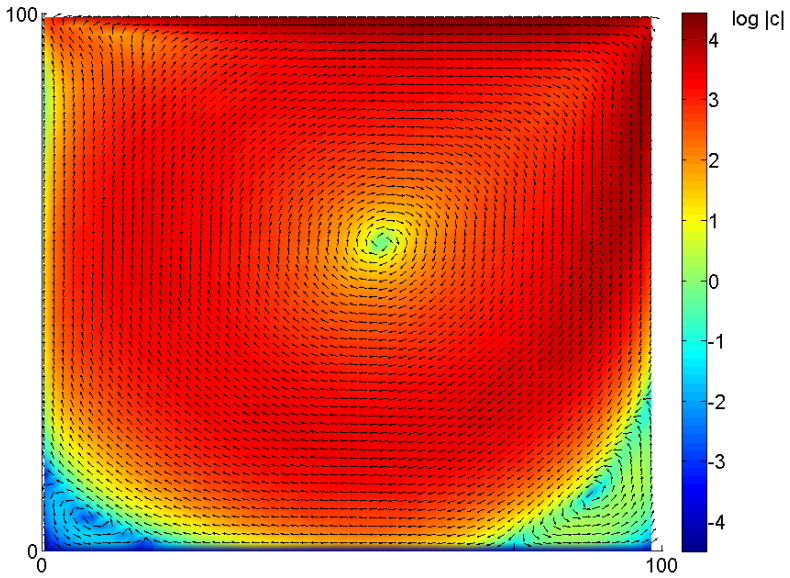

Abbildung 21: Driven Cavity für Re=100. Dargestellt sind die normierten Vektoren der Geschwindigkeit sowie deren logarithmierter Betrag.

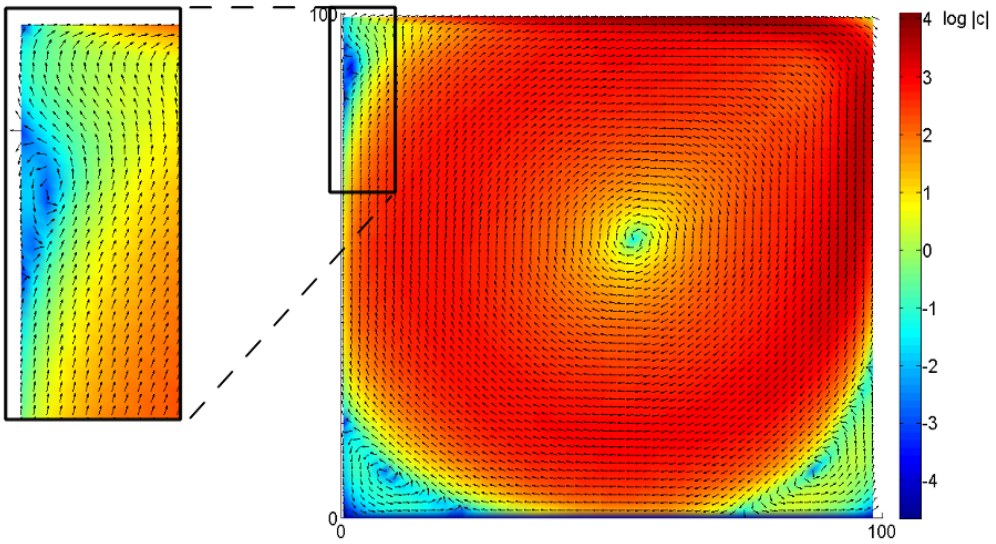

Abbildung 22: Driven Cavity für Re=3.333. Dargestellt sind die normierten Vektoren der Geschwindigkeit sowie deren logarithmierter Betrag.

5.2 Ebene Poisseuille-Strömung

Eine wichtige technische Anwendung ist die ebene Kanalströmung, bei der eine Parallelströmung der Geschwindigkeit $\vec{c}_0$ durch einen Kanal geleitet wird. Hierfür werden erneut Dirichlet-Randbedingungen auf Γ_2 und Γ_4 gesetzt. Darüber hinaus wird die Wandhaftbedingung durch Dirichlet-Randbedingung Null auf Γ_1 und Γ_3 modelliert.

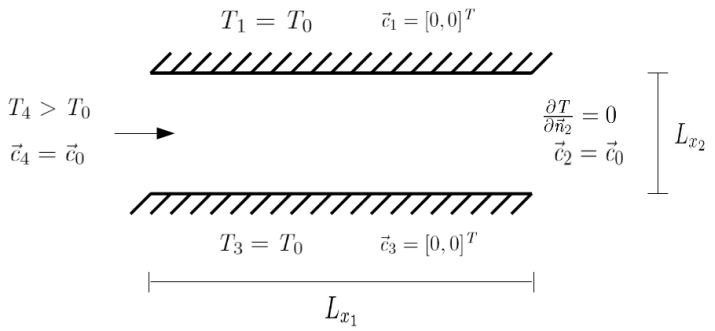

Abbildung 23: Rohrströmung mit einströmender Parallelströmung

In dem hier vorgestellten Beispiel wurde $L_{x_1} = 30m$ und $L_{x_2} = 1,5m$ gewählt sowie die Einströmgeschwindigkeit auf $\vec{c}_0 = [u_0, v_0]^T = [50\ m/s, 0]^T$ gesetzt. Berechnungen auf einem 600×30 Gitter ergeben das in Abbildung (24) gezeigte Geschwindigkeitsprofil. Es ist gut zu erkennen, dass sich das in [18] hergeleitete Profil einer ebenen Poiseuille-Strömung ergibt:

$$u(x_2) = 6u_0 \frac{x_2}{h}\left[1 - \frac{x_2}{h}\right] \tag{23}$$

Dieses besagt, dass sich die Geschwindigkeit im Kanalquerschnitt parabelförmig verhält und sein Maximum in der Kanalachse mit $u_{max} = u(L_{x_2}/2) = \frac{3}{2}u_0$ annimmt.

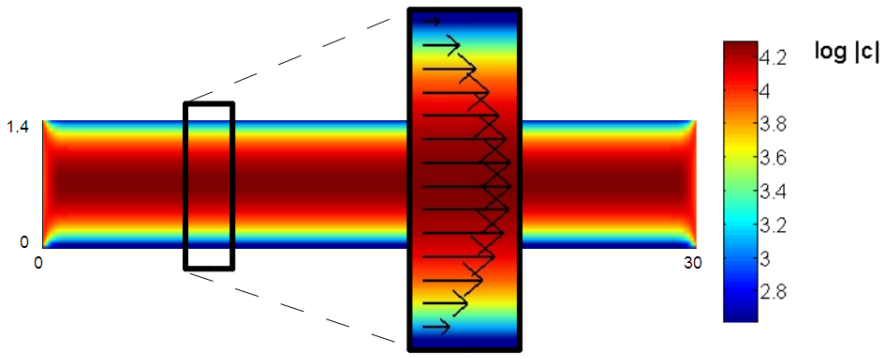

Abbildung 24: Geschwindigkeitsprofil bei der Kanalströmung. Farbig dargestellt ist der logarithmierte Betrag der Geschwindigtkeit sowie ein vergrößerter Ausschnitt mit eingezeichnetem Vektor der Geschwindigkeit.

Für diese Anwendung sollen darüber hinaus die Werte für die Temperatur geplottet werden. Hierfür wurde auf Γ_1 und Γ_3 Umgebungstemperatur vorgegeben und auf Γ_4 eine um 30 K größere Einströmtemperatur gesetzt. Darüber hinaus wählt man als Austrittsbedingung auf Γ_2, dass das Wasser mit konstanter Temperatur aus dem Kanal austritt. Dies kann durch eine Neumann-Randbedingung modelliert werden, indem man vorgibt, dass der Temperaturgradient in Normalenrichtung Null ist.
Abbildung (25) zeigt die Temperaturverteilung im Kanal zu verschiedenen Zeitpunkten. Man sieht deutlich, dass die Temperaturverteilung hauptsächlich durch Wärmekonvektion beeinflusst ist und sich deshalb die Temperaturänderung in der Mitte des Kanals deutlich schneller ausbreitet.

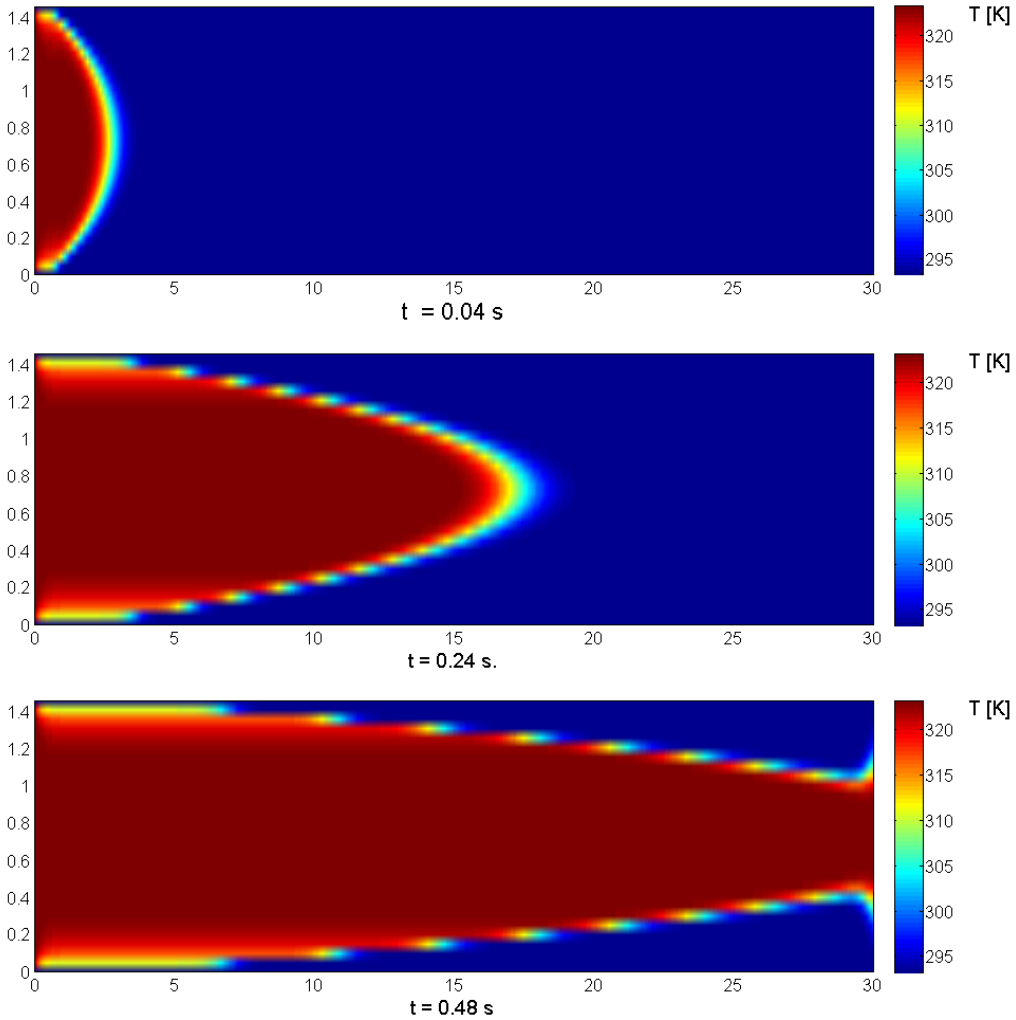

Abbildung 25: Simulation der Temperaturverteilung bei einer Kanalströmung

5.3 Mischung zweier Strömungen

An dieser Stelle soll die Mischung zweier Fluide unterschiedlicher Temperatur simuliert werden. Diese sollen, wie in Abbildung (26) angedeutet, aus zwei Rohren in einen Mischbehälter fließen. Um diesen Aufbau zu simulieren, gibt man an geeigneten Stellen von Γ_2 und Γ_4 Dirichlet-Randbedingungen für die Geschwindigkeit gemäß (23) vor.

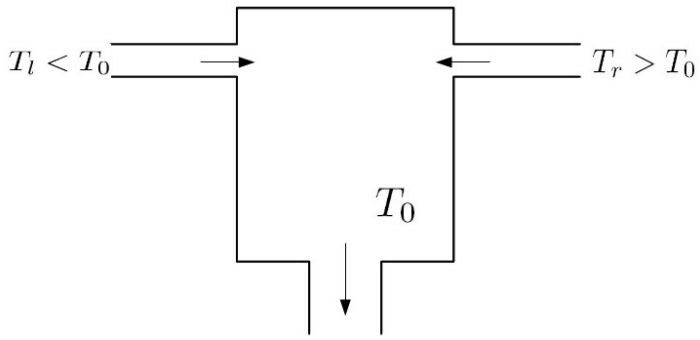

Abbildung 26: Mischung zweier Fluide

Der in Abbildung (28) gezeigte Plot wurde auf einem Gitter der Größe 100×150 berechnet und stellt mit einer Simulationszeit von $T_{end} = 0.1s$ die rechenaufwändigste Berechnung, die wir im Rahmen der Bachelorarbeit durchgeführt haben, dar.
Es ist auffällig, dass keine gegenseitige Beeinflussung der Temperaturen stattfindet, was zunächst erstaunlich erscheint. Allerdings ist die Wärmeleitfähigkeit von Wasser relativ gering und so findet kaum diffusiver Wärmeaustausch statt.

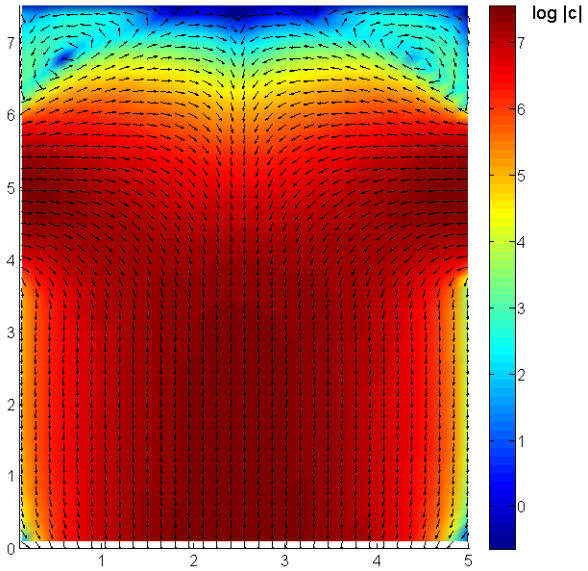

Abbildung 27: Geschwindigkeitsverteilung bei der Mischung zweier Strömungen. Farbig dargestellt ist der logarithmierte Betrag der Geschwindigkeit sowie die normierten Vektoren der Geschwindigkeit.

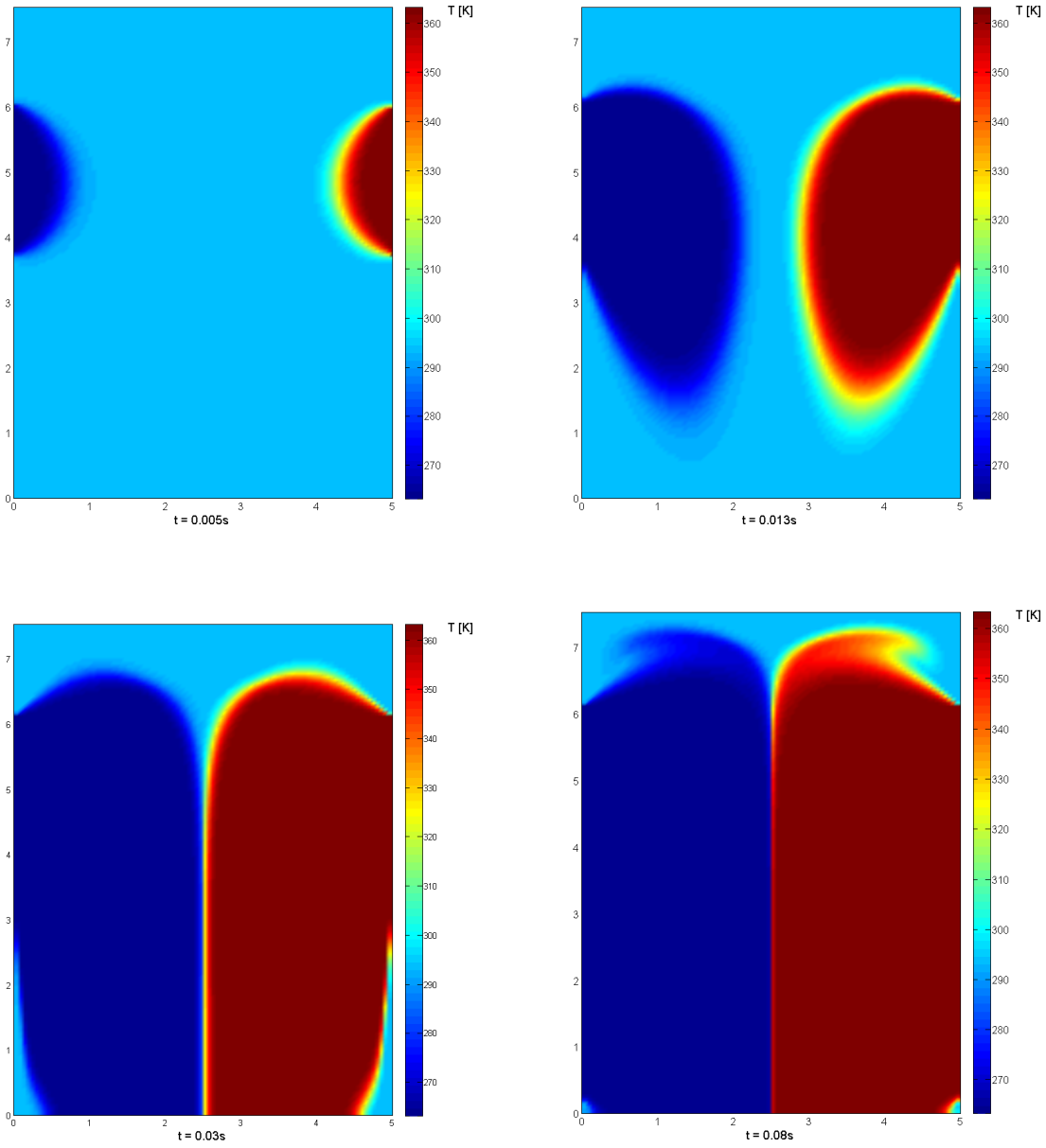

Abbildung 28: Temperaturverteilung bei der Mischung zweier Fluide zu verschiedenen Zeitpunkten

5.4 Wärmediffusion

In den vorangegangenen Versuchen wurde die Temperaturverteilung stets durch die Wärmekonvektion dominiert, so dass der Effekt der Wärmeleitung nicht erkennbar war. Daher soll an dieser Stelle ein Versuch durchgeführt werden, bei dem das Fluid zu Beginn in Ruhe ist, so dass der Einfluss der Wärmekonvektion gering ist. Betrachtet wird nun ein Fluid in einem wärmeisolierten Behälter, das in der linken Hälfte eine geringere Temperatur hat als in der rechten Hälfte (siehe Abbildung 29). Um den Effekt der Wärmeleitung zu verstärken, wurde für die Simulation eine wesentlich größere Wärmeleitfähigkeit λ gewählt.

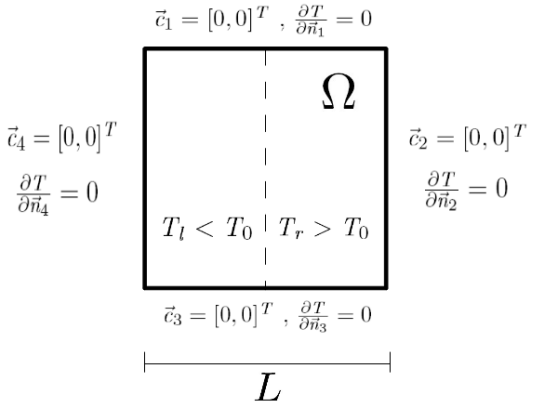

Abbildung 29: Versuchsaufbau mit Temperaturgradient in x_1-Richtung

Das in Abbildung 30 gezeigte Geschwindigkeitsfeld zeigt, dass durch die Boussinesq-Annahme der Einfluss der Temperaturverteilung auf die Geschwindigkeit gut modelliert wird: Das Fluid strömt erwartungsgemäß in der wärmeren Hälfte nach oben und auf der kälteren Seite nach unten.

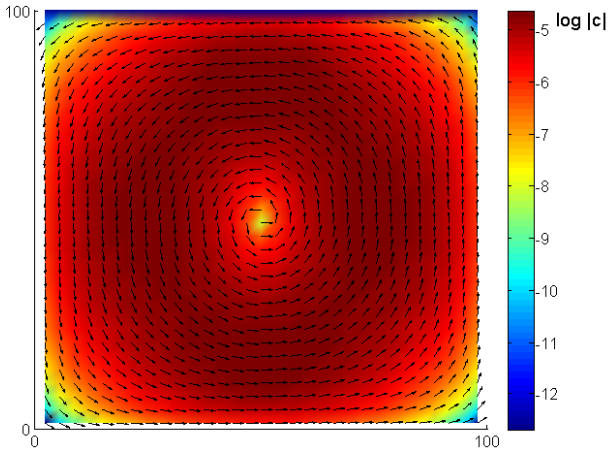

Abbildung 30: Geschwindigkeitsfeld durch Diffusion. Farbig dargestellt ist der logarithmierte Betrag der Geschwindigkeit sowie die normierten Geschwindigkeitsvektoren.

Der zeitliche Verlauf der Temperaturverteilung zeigt deutlich, dass zunächst ausschließlich Wärmeleitung von rechts nach links zu beobachten ist. In späteren Zeitschritten lässt sich auch der Einfluss der Wärmekonvektion durch die von der Temperaturverteilung induzierten Geschwindigkeit erkennen. Dies führt zu einer asymmetrischen Temperaturverteilung, da der wärmere Teil des Fluides im oberen Bereich nach links und der kältere Teil des Fluids im unteren Bereich nach rechts fließt.

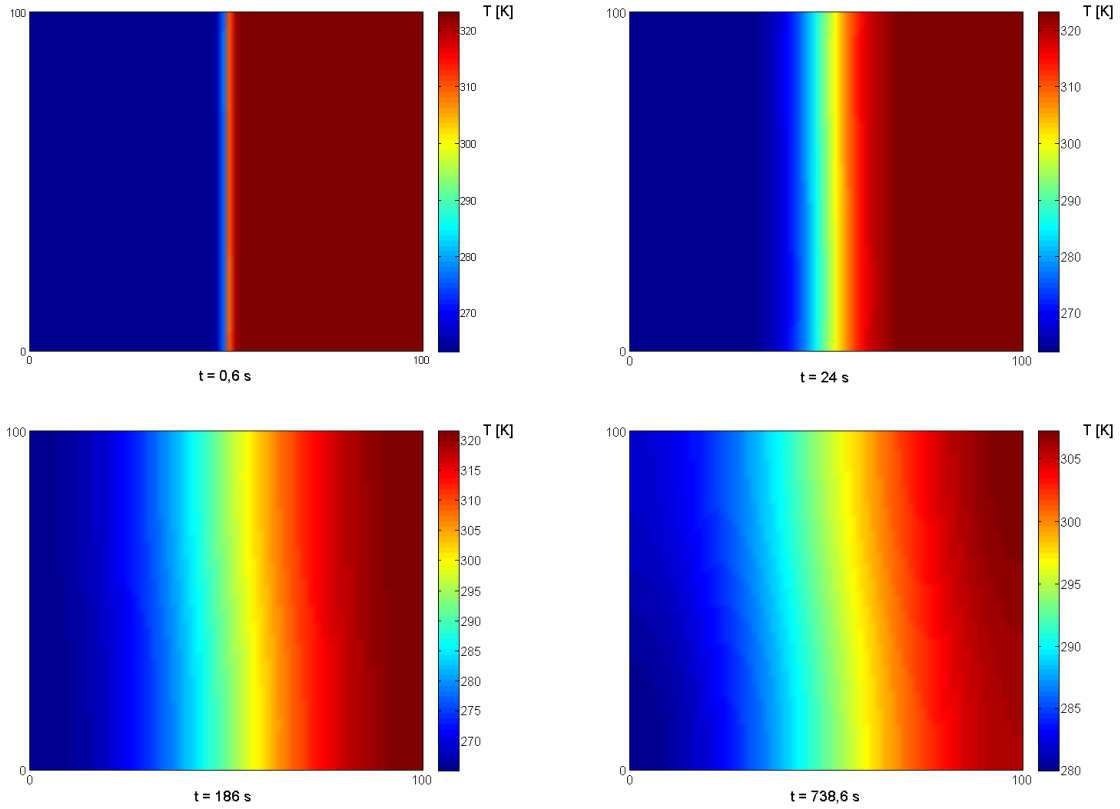

Abbildung 31: Temperaturverteilung durch Diffusion

6 Zusammenfassung und Ausblick

In dieser Bachelorarbeit wurden in Kapitel 2 zunächst die zu lösenden Wirbeltransportgleichung und Wärmeleitungsgleichung hergeleitet. In Kapitel 3 wurde dann eine vollständige Finite-Volumen-Diskretisierung dieser Differentialgleichungen vorgestellt, in der allgemeine Randbedingungen berücksichtigt werden. Dabei wurde das Programm so implementiert, dass Randbedingungen für die Geschwindigkeit und die Temperatur vorgegeben werden und eine Umrechnung in Randwerte für die Stromfunktion und die Wirbelstärke intern geschieht.
Zur instationären Lösung dieser gekoppelten Gleichungen war es in der hier präsentierten Implementierung in jedem Zeitschritt notwendig, drei lineare Gleichungssysteme der Form $Ax = b$ mit dünnbesetzter Bandmatrix A zu lösen. Dies stellte, wie Messungen ergeben haben, bei hochdimensionalen Problemen den wesentlichen Rechenaufwand unseres Programms dar.

Hierfür wurde in Kapitel 4 ein effizientes, hoch paralleles CG-Verfahren entwickelt, das mit Hilfe der CUDA Technologie einen Speedup von bis zu 35-facher Beschleunigung gegenüber der sequentiellen *C++* Variante des Strömungslösers erzielte. Um diesen Speedup zu erreichen, wurden zuvor Optimierungen bei der parallelen Berechnung des Skalarprodukts und der Sparsematrix-Vektor Multiplikation durchgeführt. Dies setzt eine genaue Kenntnis der verwendeten Hardware voraus. Es hat sich herausgestellt, dass bei der Rechnung auf Grafikkarten die Performance meistens entscheidend von der Speicherbandbreite abhängt, so dass die beiden zuvor genannten Operationen die theoretisch maximale Rechenleistung der Grafikkarten von bis zu $933\,GFlops/s$ bei einfacher und $78\,GFlops/s$ bei doppelter Genauigkeit nicht vollständig ausnutzen konnten.

Wie numerische Tests in Kapitel 5 gezeigt haben, werden physikalische Phänomene auf rechteckigem Simulationsgebiet mit guter Genauigkeit berechnet. Insbesondere der *Driven Cavity* Versuch zeigte bei variierender Reynoldszahl die aus der Literatur bekannten Resultate.

Auf dieser Bachelorarbeit aufbauende Arbeiten würden sicherlich von der Verwendung der neu entwickelten FERMI Grafikkarten von NVIDIA profitieren. Diese neue Generation Grafikkarten ist noch stärker auf ein Einsatzgebiet im Bereich wissenschaftliches Rechnen ausgelegt und erzielt für double precision Rechnungen weit höhere Performance als die in dieser Arbeit verwendete TESLA Karte. Darüber hinaus stellt sie einen größeren Grafikkartenspeicher mit einer schnelleren Speicheranbindung zur Verfügung, was bei komplexeren Anwendungen erforderlich ist, um alle benötigten Daten während der kompletten Rechnung auf dem Global-Memory zu halten.
Interessante Anwendungen von CUDA würden insbesondere Berechnungen darstellen, bei denen das Verhältnis der benötigten Daten zur Anzahl der Rechenoperationen besonders klein ist, so dass die maximale Rechenleistung besser ausgenutzt werden kann. Dies ist beispielsweise bei einer Matrixmultiplikation gegeben.
Darüber hinaus könnte man die Lösung der auftretenden linearen Gleichungssysteme mit dem CG-Verfahren durch die Wahl von geeigneten Vorkonditionierern weiter beschleunigen, wie beispielsweise in [14] vorgestellt. Hierbei wäre allerdings zu evaluieren, inwiefern sich diese parallelisieren lassen.

A Codebeispiele

An dieser Stelle sind ausgewählte Teile unserer Implementierung gezeigt, die in den Kapiteln 4.4 und 4.5 zur Zeitmessung verwendet wurden.
Das komplette Programm liegt dieser Bachelorarbeit auf CD bei. Hierauf ist auch eine *ReadMe* Datei enthalten, die dem Benutzer beschreibt, wie Compilierbefehle und das Setzen von Randbedingungen durchzuführen sind.

A.1 CG-Verfahren

In Abschnitt 4.4 wurden zwei verschiedene Varianten einer Implementierung des CG-Verfahrens auf der Grafikkarte mit einer sequentiellen Variante auf der CPU verglichen. Die genaue Implementierung kann in diesem Teil des Anhangs nachvollzogen werden, wobei die Funktionsnamen mit den in Kapitel 4.4 verwendeten Bezeichnungen übereinstimmen.

Algorithm 14 Erste Version des CG-Verfahrens

```
1  int CG_GPU_DIA1(Vector &xj, const DIA &A,const Vector &b, int MAXIT, Real tol){
2    Real *d_A_data, *d_b, *d_xj, *d_rTr, *d_Ap, *d_p, *d_pAp, *d_beta, *d_r, *d_temp;
3    Real normr0,normr,rTr,rTr_neu,pAp,beta,gamma;
4    int *d_A_offsets;
5    int dim=b.dim;
6    int j;
7    //Allokiere Speicher fuer benoetigte Daten
8    cudaMalloc((void**)&d_A_data, A.stride*A.dim_offsets*sizeof(Real));
9    cudaMalloc((void**)&d_A_offsets, A.dim_offsets*sizeof(int));
10   cudaMalloc((void**)&d_b, dim*sizeof(Real));
11   cudaMalloc((void**)&d_xj, dim*sizeof(Real));
12   cudaMalloc((void**)&d_r, dim*sizeof(Real));
13   cudaMalloc((void**)&d_temp, dim*sizeof(Real));
14   cudaMalloc((void**)&d_rTr, sizeof(Real));
15   cudaMalloc((void**)&d_Ap, A.dim_c*sizeof(Real));
16   cudaMalloc((void**)&d_p, dim*sizeof(Real));
17   cudaMalloc((void**)&d_pAp, sizeof(Real));
18   cudaMalloc((void**)&d_beta, sizeof(Real));
19   //Kopiere Matrix, Startvektor und Rechte Seite auf Grafikkartenspeicher
20   cudaMemcpy(d_A_data, A.data, A.stride*A.dim_offsets*sizeof(Real),
         cudaMemcpyHostToDevice);
21   cudaMemcpy(d_A_offsets, A.offsets, A.dim_offsets*sizeof(int),
         cudaMemcpyHostToDevice);
22   cudaMemcpy(d_b,b.data, dim*sizeof(Real), cudaMemcpyHostToDevice);
23   cudaMemcpy(d_xj,xj.data, dim*sizeof(Real), cudaMemcpyHostToDevice);
24
25   //Berechne r=b+(-1)*A*x0;
26   bindTexture(d_xj, dim*sizeof(Real));
27   k_spmv_dia3<<<(dim/BSIZE_DIA+1), BSIZE_DIA, A.dim_offsets*sizeof(int)>>>(d_Ap,
         d_A_data, d_A_offsets, d_xj, dim, dim, A.dim_offsets, A.stride);
28   unbindTexture();
29   k_VecAddVecMulSkal<<<(dim/BSIZE_SUBADD+1),BSIZE_SUBADD>>>(d_r,d_Ap,-1,d_b,dim);
30
31   //Berechne rTr=r*r;
32   k_normquad<<<30, BSIZE_SKP, BSIZE_SKP*sizeof(Real)>>>(d_temp,d_r,dim);
33   k_sumup<<<1,32,32*sizeof(Real)>>>(d_rTr, d_temp,30);
34
35   //Berechne nr0=r.norm(2);
36   cudaMemcpy(&rTr,d_rTr,sizeof(Real),cudaMemcpyDeviceToHost);
37   normr0=sqrt(rTr);
```

```
38
39      //Setze p=r
40      k_vecInitVec<<<(dim/BSIZE_SUBADD+1),BSIZE_SUBADD>>>(d_p,d_r,dim);
41      if(normr0>tol){
42
43        for (j=0;j<MAXIT;j++){
44          //Berechne Ap=A*p;
45          bindTexture(d_p, dim*sizeof(Real));
46          k_spmv_dia3<<<(dim/BSIZE_DIA+1), BSIZE_DIA, A.dim_offsets*sizeof(int)>>>(d_Ap,
                d_A_data, d_A_offsets, d_p, dim ,dim ,A.dim_offsets,A.stride);
47          unbindTexture();
48
49          //Berechne pAp=p*Ap;
50          k_skp7<<<120, BSIZE_SKP, BSIZE_SKP*sizeof(Real)>>>(d_temp,d_p,d_Ap,dim);
51          k_sumup<<<1,128,128*sizeof(Real)>>>(d_pAp, d_temp, 120);
52          cudaMemcpy(&pAp,d_pAp,sizeof(Real),cudaMemcpyDeviceToHost);
53
54          gamma=rTr/pAp;
55          //Berechne xj=xj+gamma*p
56          k_VecAddVecMulSkal<<<(dim/BSIZE_SUBADD+1),BSIZE_SUBADD>>>(d_xj,d_p,gamma,d_xj,
                dim);
57          //r=r-gamma*Ap
58          k_VecAddVecMulSkal<<<(dim/BSIZE_SUBADD+1),BSIZE_SUBADD>>>(d_r,d_Ap,-1*gamma,
                d_r,dim);
59
60          //Berechne rTr=r*r;
61          k_normquad<<<120, BSIZE_SKP, BSIZE_SKP*sizeof(Real)>>>(d_temp,d_r,dim);
62          k_sumup<<<1,128,128*sizeof(Real)>>>(d_rTr, d_temp,120);
63
64          cudaMemcpy(&rTr_neu,d_rTr,sizeof(Real),cudaMemcpyDeviceToHost);
65          normr=sqrt(rTr_neu);
66          if (normr/normr0 < tol)
67            break;
68
69          beta=rTr_neu/rTr;
70          rTr=rTr_neu;
71
72          //Berechne p=beta*p+r;
73          k_VecAddVecMulSkal<<<(dim/BSIZE_SUBADD+1),BSIZE_SUBADD>>>(d_p,d_p,beta,d_r,dim
                );
74        }
75      }
76      //Loesung von Grafikkarte runterkopieren
77      cudaMemcpy(xj.data,d_xj,xj.dim*sizeof(Real),cudaMemcpyDeviceToHost);
78      //Speicher wieder freigeben
79      cudaFree(d_A_data);
80      cudaFree(d_A_offsets);
81      cudaFree(d_b);
82      cudaFree(d_xj);
83      cudaFree(d_r);
84      cudaFree(d_temp);
85      cudaFree(d_rTr);
86      cudaFree(d_Ap);
87      cudaFree(d_p);
88      cudaFree(d_pAp);
89      cudaFree(d_beta);
90      return j;
91  }
```

Algorithm 15 Zweite Version des CG-Verfahrens

```
1  int CG_GPU_DIA2(Real *d_xj, Real *d_A_data, int* d_A_offsets, Real *d_b, int MAXIT,
       Real tol, int dim, int dim_offsets, int stride, int flag){
2    static Real *d_p, *d_Ap, *d_temp, *d_r, *d_rTr, *d_pAp;
3    Real normr0,normr,rTr,rTr_neu,beta,gamma,pAp;
4    int j;
5    //Allokiere Speicher fuer temporaere Daten auf Grafikkartenspeicher
6    if (flag == -1 || flag==2){
7      cudaMalloc((void**)&d_r, dim*sizeof(Real));
8      cudaMalloc((void**)&d_temp, dim*sizeof(Real));
9      cudaMalloc((void**)&d_rTr, sizeof(Real));
10     cudaMalloc((void**)&d_pAp, sizeof(Real));
11     cudaMalloc((void**)&d_Ap, dim*sizeof(Real));
12     cudaMalloc((void**)&d_p, dim*sizeof(Real));
13   }
14   //Berechne r=b+(-1)*A*x0;
15   bindTexture(d_xj, dim*sizeof(Real));
16   k_spmv_dia3<<<(dim/BSIZE_DIA+1), BSIZE_DIA, dim_offsets*sizeof(int)>>>(d_Ap,
       d_A_data, d_A_offsets, d_xj, dim, dim, dim_offsets, stride);
17   unbindTexture();
18   k_VecAddVecMulSkal<<<(dim/BSIZE_SUBADD+1),BSIZE_SUBADD>>>(d_r,d_Ap,-1,d_b,dim);
19
20   //Berechne rTr=r*r;
21   k_normquad<<<30, BSIZE_SKP, BSIZE_SKP*sizeof(Real)>>>(d_temp,d_r,dim);
22   k_sumup<<<1,32,32*sizeof(Real)>>>(d_rTr, d_temp,30);
23
24   //Berechne nr0=r.norm(2);
25   cudaMemcpy(&rTr,d_rTr,sizeof(Real),cudaMemcpyDeviceToHost);
26   normr0=sqrt(rTr);
27
28   //Setze p=r
29   k_vecInitVec<<<(dim/BSIZE_SUBADD+1),BSIZE_SUBADD>>>(d_p,d_r,dim);
30   if(normr0>tol){
31     for (j=0;j<MAXIT;j++){
32       //Berechne Ap=A*p;
33       bindTexture(d_p, dim*sizeof(Real));
34       k_spmv_dia3<<<(dim/BSIZE_DIA+1), BSIZE_DIA, dim_offsets*sizeof(int)>>>(d_Ap,
           d_A_data, d_A_offsets, d_p, dim ,dim ,dim_offsets,stride);
35       unbindTexture();
36
37       //Berechne pAp=p*Ap;
38       k_skp7<<<120, BSIZE_SKP, BSIZE_SKP*sizeof(Real)>>>(d_temp,d_p,d_Ap,dim);
39       k_sumup<<<1,128,128*sizeof(Real)>>>(d_pAp, d_temp, 120);
40       cudaMemcpy(&pAp,d_pAp,sizeof(Real),cudaMemcpyDeviceToHost);
41
42       gamma=rTr/pAp;
43       //Berechne xj=xj+gamma*p
44       k_VecAddVecMulSkal<<<(dim/BSIZE_SUBADD+1),BSIZE_SUBADD>>>(d_xj,d_p,gamma,d_xj,
           dim);
45       //r=r-gamma*Ap
46       k_VecAddVecMulSkal<<<(dim/BSIZE_SUBADD+1),BSIZE_SUBADD>>>(d_r,d_Ap,-1*gamma,
           d_r,dim);
47
48       //Berechne rTr=r*r;
49       k_normquad<<<120, BSIZE_SKP, BSIZE_SKP*sizeof(Real)>>>(d_temp,d_r,dim);
50       k_sumup<<<1,128,128*sizeof(Real)>>>(d_rTr, d_temp,120);
51
52       cudaMemcpy(&rTr_neu,d_rTr,sizeof(Real),cudaMemcpyDeviceToHost);
53       normr=sqrt(rTr_neu);
54       if (normr/normr0 < tol)
55         break;
56
57       beta=rTr_neu/rTr;
58       rTr=rTr_neu;
```

```
59
60          //Berechne p=beta*p+r;
61          k_VecAddVecMulSkal<<<(dim/BSIZE_SUBADD+1),BSIZE_SUBADD>>>(d_p,d_p,beta,d_r,dim
                );
62        }
63      }
64      //Temporaere Daten auf Grafikkartenspeicher wieder freigeben
65      if (flag == 1 || flag==2){
66        cudaFree(d_r);
67        cudaFree(d_temp);
68        cudaFree(d_rTr);
69        cudaFree(d_pAp);
70        cudaFree(d_Ap);
71        cudaFree(d_p);
72      }
73      return j;
74    }
```

Algorithm 16 Sequentielle Version des CG-Verfahrens

```
1   void CG_CPU_DIA(Vector &xj,const DIA &A,const Vector &b, int MAXIT, Real tol){
2     int j;
3     //Berechne r=b-A*x0
4     Vector r(b-A*xj);
5     Real nr0=r.norm(2);
6     Vector p(r);
7     Real rTr=r*r;
8     if (nr0<tol)
9       return;
10    Vector Ap(xj.dim);
11    Real gamma,rTr_neu,beta;
12    for (j=0;j<MAXIT;j++){
13      //Berechne Ap=A*p;
14      DIAMulVec(Ap,A,p);
15      gamma=rTr/(p*Ap);
16      //Berechne xj+=gamma*p;
17      VecAddVecMulSkal(xj,p,gamma,xj);
18      //Berechne r-=gamma*Ap;
19      VecAddVecMulSkal(r,Ap,-1*gamma,r);
20      if (r.norm(2)/nr0 < tol)
21        return; //Abbruch nach Erreichen der tol
22      rTr_neu=r*r;
23      beta=rTr_neu/rTr;
24      rTr=rTr_neu;
25      //Berechne p=r+beta*p;
26      VecAddVecMulSkal(p,p,beta,r);
27    }
28    return j;
29  }
```

A.2 Einzelner Zeitschritt

An dieser Stelle ist der in Kapitel 4.5 verglichene einzelne Zeitschritt des Programms dargestellt. Dieser stellt bei längerer Simulationszeit den eigentlichen Aufwand des Programms dar, da zuvor einmalig Speicher allokiert wird und die in Kapitel 3 hergeleiteten Matrizen aufgestellt werden, die mit Ausnahmen von C_h zeitlich konstante Einträge haben. Darüber hinaus werden die gesetzten Randbedingungen für die Geschwindigkeit und die Temperatur vor der Zeitschleife geeignet umgerechnet.

Algorithm 17 Zeitschleife des CUDA Programms

```
1  for (Real t=0;t<=G_Tend;t+=G_dt){
2    //Loesen der Poissongleichung fuer die Stromfunktion
3    k_VecAddVecMulSkal<<<(G_dim/BSIZE_SUBADD+1),BSIZE_SUBADD>>>(d_b,d_rhsA,1.0,d_w,
         G_dim);  //Rechte Seite
4    CG_GPU_DIA2(d_psiInnen,d_Ah_data,d_Ah_offsets,d_b,G_MAXIT,G_tol, G_dim, Ah.
         dim_offsets, Ah.dim_r, ((t==0)?(-1):(0)));
5
6    //Geschwindigkeiten aus psi berechnen
7    bindTexture(d_psi, psi.dim*sizeof(Real));
8    k_spmv_ell<<<(psi2u1.dim_r/BSIZE_ELL+1), BSIZE_ELL>>>(d_u1Innen,d_psi2u1_data,
         d_psi2u1_indices,d_psi, psi2u1.dim_r, psi2u1.dim_c, psi2u1.dim_ind, psi2u1.
         dim_r);
9    k_spmv_ell<<<(psi2v2.dim_r/BSIZE_ELL+1), BSIZE_ELL>>>(d_v2Innen,d_psi2v2_data,
         d_psi2v2_indices,d_psi, psi2v2.dim_r, psi2v2.dim_c, psi2v2.dim_ind, psi2v2.
         dim_r);
10   unbindTexture();
11
12   //Interpolieren der Geschwindigkeiten auf entspr. anderes Gitter
13   bindTexture(d_tempu, tempu.dim*sizeof(Real));
14   k_spmv_ell<<<(u1intu2.dim_r/BSIZE_ELL+1), BSIZE_ELL>>>(d_u2Innen,d_u1intu2_data,
         d_u1intu2_indices,d_tempu, u1intu2.dim_r, u1intu2.dim_c, u1intu2.dim_ind,
         u1intu2.dim_r);
15   unbindTexture();
16   bindTexture(d_tempv, tempv.dim*sizeof(Real));
17   k_spmv_ell<<<(v2intv1.dim_r/BSIZE_ELL+1), BSIZE_ELL>>>(d_v1Innen,d_v2intv1_data,
         d_v2intv1_indices,d_tempv, v2intv1.dim_r, v2intv1.dim_c, v2intv1.dim_ind,
         v2intv1.dim_r);
18   unbindTexture();
19
20   //Randbedingungen fuer die Temperatur. Dabei werden Neumann-RB in Dirichlet-RB
         umgerechnet.
21   k_NeumannRB<<<TDirich.dim/32+1,32>>>(d_TDirich, d_TRlexi, d_T, G_Nx, G_Ny);
22
23   //Geschwindigkeit und Temperatur von Grafikkarte runterladen und in Datei
         speichern. Dies geschieht nur in jedem 60tem Zeitschritt.
24   if (int(t/G_dt)%60==0){
25     cudaMemcpy(C.data,d_C,C.dim_r*C.dim_c*sizeof(Real),cudaMemcpyDeviceToHost);
26     cudaMemcpy(T.data,d_T,T.dim*sizeof(Real),cudaMemcpyDeviceToHost);
27     cudaMemcpy(TDirich.data,d_TDirich,TDirich.dim*sizeof(Real),
           cudaMemcpyDeviceToHost);
28     mergeT(Tplot,T,TDirich);
29
30     char filename[20];
31     sprintf(filename,"output/c_%i.dat",l);
32     C.save(filename);
33     sprintf(filename,"output/T_%i.dat",l);
34     Tplot.save(filename);
35     l++;
36   }
37
38   //w auf Rand neu berechnen
```

```
39      k_wRandUpdate<<<G_dimR/32+1,32>>>(d_wRand, d_wRproto, 1.0, d_u1Innen, d_v2Innen,
            G_h, G_Nx, G_Ny);
40
41      //Aufstellen der Rechten Seiten fuer Zeitintegration
42      k_vecInitSkal<<<(G_dim/BSIZE_SUBADD+1),BSIZE_SUBADD>>>(d_rhsw,0.0, G_dim);
43      k_vecInitSkal<<<(G_dim/BSIZE_SUBADD+1),BSIZE_SUBADD>>>(d_rhsT,0.0, G_dim);
44      k_RHSx<<<G_dimR/32+2,128>>>(d_rhsw, d_rhsT, d_wRand, d_TDirich, d_v1Innen, G_nu*(
            G_dt/pow(G_h,2)), G_lamda/G_rho/G_cp*(G_dt/pow(G_h,2)), G_h*(G_dt/pow(G_h,2)),
             G_Nx, G_Ny);
45      k_RHSy<<<G_dimR/32+2,128>>>(d_rhsw, d_rhsT, d_wRand, d_TDirich, d_u2Innen, G_nu*(
            G_dt/pow(G_h,2)), (G_beta*G_g)*G_h/2.0*(G_dt/pow(G_h,2)), G_lamda/G_rho/G_cp*(
            G_dt/pow(G_h,2)), G_h*(G_dt/pow(G_h,2)), G_Nx, G_Ny);
46
47      //Aufstellen der Konvektions-Matrix aus aktuellen Geschwindigkeiten
48      cGradw<<<5*(G_dim/128+1),128>>>(d_Ch_data,d_v1Innen, d_u2Innen, G_h*(G_dt/pow(G_h
            ,2)), Ch.stride, G_Nx, G_Ny);
49
50      //impl. EULER Zeitintegration
51      bindTexture(d_T, G_dim*sizeof(Real));
52      k_vecaddspmv_dia<<<(G_dim/BSIZE_DIA+1), BSIZE_DIA, Ch.dim_offsets*sizeof(int)>>>(
            d_rhsT, d_Ch_data, d_Ch_offsets, d_T, d_rhsT, G_dim ,G_dim ,Ch.dim_offsets,Ch.
            stride);
53      unbindTexture();
54
55      bindTexture(d_T, G_dim*sizeof(Real));
56      k_vecaddspmv_dia<<<(G_dim/BSIZE_DIA+1), BSIZE_DIA, Dhw.dim_offsets*sizeof(int)>>>(
            d_rhsw, d_Dhw_data, d_Dhw_offsets, d_T, d_rhsw, G_dim ,G_dim ,Dhw.dim_offsets,
            Dhw.stride);
57      unbindTexture();
58
59      bindTexture(d_w, G_dim*sizeof(Real));
60      k_vecaddspmv_dia<<<(G_dim/BSIZE_DIA+1), BSIZE_DIA, Ch.dim_offsets*sizeof(int)>>>(
            d_rhsw, d_Ch_data, d_Ch_offsets, d_w, d_rhsw, G_dim ,G_dim ,Ch.dim_offsets,Ch.
            stride);
61      unbindTexture();
62
63      //Loest Gleichung in T
64      CG_GPU_DIA2(d_T,d_BhT_data,d_BhT_offsets,d_rhsT,G_MAXIT,G_tol, G_dim, BhT.
            dim_offsets, BhT.dim_r);
65      //Loest Gleichung in w
66      CG_GPU_DIA2(d_w,d_Bhw_data,d_Bhw_offsets,d_rhsw,G_MAXIT,G_tol, G_dim, Bhw.
            dim_offsets, Bhw.dim_r, ((t==G_Tend)?(1):(0)));
67  }
```

Algorithm 18 Zeitschleife des *C++* Programms

```
1  for (Real t=0;t<=G_Tend;t+=G_dt){
2    //Loesen der Poissongleichung fuer die Stromfunktion
3    //CG_CPU_DIA(psiInnen,Ah,w+rhsA,G_MAXIT,G_tol);
4    CG_GPU_DIA1(psiInnen,Ah,w+rhsA,G_MAXIT,G_tol);
5
6    //Geschwindigkeiten aus psi berechnen
7    ELLMulVec(u1Innen,psi2u1,psi);
8    ELLMulVec(v2Innen,psi2v2,psi);
9    ELLMulVec(u2Innen,u1intu2,tempu);
10   ELLMulVec(v1Innen,v2intv1,tempv);
11
12   //Randbedingungen fuer die Temperatur. Dabei werden Neumann-RB in Dirichlet-RB
         umgerechnet.
13   NeumannRB(TDirich,TRlexi,T);
14
15   //w auf Rand neu berechnen
16   wRand=wRproto;
17   for (i=0;i<=G_Nx-3;i++){
18     //Gamma3
19     wRand(i+1)=wRand(i+1)-u1Innen(i)/G_h;
20     //Gamma1
21     wRand(G_Nx+2*G_Ny-3+i)=wRand(G_Nx+2*G_Ny-3+i)+u1Innen((G_Nx-2)*(G_Ny-2)+i)/G_h;
22   }
23   for (i=0;i<=G_Ny-3;i++){
24     //Gamma4
25     wRand(G_Nx+2*i)=wRand(G_Nx+2*i)+v2Innen(i*(G_Nx-1))/G_h;
26     //Gamma2
27     wRand(G_Nx+2*(i+1)-1)=wRand(G_Nx+2*(i+1)-1)-v2Innen((i+1)*(G_Nx-1)-1)/G_h;
28   }
29
30   //Aufstellen der Rechten Seiten fuer Zeitintegration
31   rhsw*=0;
32   rhsT*=0;
33   LaplacewRHS(rhsw,wRand);
34   LaplaceTRHS(rhsT,TDirich);
35   dxTRHS(rhsw,TDirich);
36
37   //Aufstellen der Konvektions-Matrix aus aktuellen Geschwindigkeiten
38   cGradw(Ch,rhsw,rhsT,v1Innen,u2Innen,wRand,TDirich);
39
40   //impl EULER Zeitintegration
41   //CG_CPU_DIA(T,BhT,rhsT+Ch*T,G_MAXIT,G_tol);
42   //CG_CPU_DIA(w,Bhw,rhsw+Ch*w+Dhw*T,G_MAXIT,G_tol);
43   CG_GPU_DIA1(T,BhT,rhsT+Ch*T,G_MAXIT,G_tol);
44   CG_GPU_DIA1(w,Bhw,rhsw+Ch*w+Dhw*T,G_MAXIT,G_tol);
45
46   //Abspeichern der Daten in jedem 60ten Zeitschritt
47   if (int(t/G_dt)%60==0){
48     mergeT(Tplot,T,TDirich);
49     char filename[20];
50     sprintf(filename,"output_seq/c_%i.dat",l);
51     C.save(filename);
52     sprintf(filename,"output_seq/T_%i.dat",l);
53     Tplot.save(filename);
54     l++;
55   }
56 }
```

Literatur

[1] H.D. Baehr, K. Stephan. *Wärme- und Stoffübertragung.* Springer, 6. Auflage (1993)

[2] G. Bärwolff. *Numerik für Ingenieure, Physiker und Informatiker.* Elsevier, 1. Auflage (2006)

[3] N. Bell, M. Garland. *Efficient Sparse Matrix-Vector Multiplication on* CUDA. NVIDIA Technical Report, NVR-2008-004 (2008)

[4] M. Griebel, T. Dornseifer, T. Neunhoeffer. *Numerical Simulation in Fluid Dynamics. A Practical Introduction.* Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1997

[5] J.H. Ferzinger, M. Perić. *Numerische Strömungsmechanik.* Springer, 1. Auflage (2008)

[6] W. Hackbusch. *Theorie und Numerik elliptischer Differentialgleichungen.* Teubner, 1. Auflage (2005)

[7] W. Hackbusch. *Iterative Lösung großer schwachbesetzter Gleichungssysteme.* Teubner, 2. Auflage (1993)

[8] J. Liesen, P. Tichý. *Convergence analysis of Krylov subspace methods.* GAMM-Mitteilungen, vol. 27, no. 2, pp. 153-173 (2004)

[9] J. Liesen. *Numerik 2 für Ingenieure.* Vorlesungsskript, TU Berlin (WS 2009/2010)

[10] NVIDIA Corporation. NVIDIA CUDA *Programming Guide.* Version 3.0 (2010)

[11] NVIDIA Corporation. NVIDIA CUDA *C Best Practice Guide.* Version 3.0 (2010)

[12] NVIDIA Homepage. `http://www.nvidia.com`

[13] T. Rauber, G. Rünger. *Parallele Programmierung.* Springer, 2. Auflage (2006)

[14] Y. Saad. *Iterative Methods for Sparse Linear Systems.* SIAM, Philadelphia, 2. Auflage (2003)

[15] H. Schade, E. Kunz. *Strömungslehre.* de Gruyter, 3. Auflage (2007)

[16] F. Thiele et al. *Finite-Volumen-Methode in der Numerischen Thermofluiddynamik* Vorlesungsskript, TU Berlin, 5. Auflage (2006)

[17] G. Tsatsaronis. *Thermodynamik* Vorlesungsskript, TU Berlin, 7. Auflage (2008)

[18] R. Wille *Strömungslehre* Vorlesungsskript, TU Berlin, 8. Auflage (2005)