Institut für Visualisierung und Interaktive Systeme
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3025

GPU-basierte Simulation und in-situ Visualisierung von 3D Fluiden in porösen Medien

Bertram Thomaß

Studiengang: Informatik

Prüfer: Prof. Dr. Thomas Ertl

Betreuer: Dipl.-Inf. Markus Üffinger, Dipl.-Phys. oec. Steffen Müthing

begonnen am: 23. Juni 2010

beendet am: 23. November 2010

CR-Klassifikation: D.1.3 Concurrent Programming, G.1.8 Partial Differential Equations

Zusammenfassung

In der vorliegenden Arbeit wird die Simulation eines speziellen Problems der Strömungssimulation auf massiv parallel arbeitende Grafikhardware übertragen. Bei dem zu betrachtenden Problem handelt es sich um die Simulation einer Zweiphasenströmung in einem porösen Medium. Das mathematische Modell zu einer solchen Simulation wird durch diverse, nichtlineare Differentialgleichungen beschrieben. Bei der Klasse dieser Probleme steht das *Darcy-Gesetz* als Formulierung der Impulserhaltung zusammen mit der Massenerhaltung im Mittelpunkt der Betrachtung. Das Problem der Zweiphasenströmung wird mit einer *IMPES*[1]-Lösungsstrategie auf Basis des *CUDA*[2]-Frameworks mit einem *Finite-Volumen*-Verfahren auf eine stark parallel arbeitende Grafikhardware (GPU[3]) ausgelagert. Das Ziel ist ein Geschwindigkeitsvorteil gegenüber rein CPU-basierten Lösungen. Außerdem werden die Simulationsergebnisse interaktiv visualisiert.

[1] *Im*plicit *P*ressure *E*xplicit *S*aturation.
[2] *C*ompute *U*nified *D*evice *A*rchitecture, eine Parallel-Architektur von Nvidia.
[3] *G*raphics *P*rocessing *U*nit.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Fragestellung . . . 1

1.2 Gang der Arbeit . . . 2

2 Grundlagen **3**

2.1 Die Kontinuitätsgleichung . . . 3

2.2 Strömung in porösen Medien . . . 4

2.2.1 Darcy-Gesetz . . . 4

2.2.2 Permeabilität . . . 5

2.3 Mehrphasenströmung . . . 6

2.3.1 Zweiphasen-Strömung . . . 7

2.3.2 Effektive Sättigung . . . 9

2.3.3 Effektive und relative Permeabilität . . . 10

2.3.4 Mobilität und *fractional flow* . . . 11

2.4 Das Buckley–Leverett-Modell . . . 12

2.4.1 Schwache Kopplung mit IMPES . . . 14

2.5 Diskretisierung . . . 15

2.5.1 Upwind Approximation . . . 18

2.5.2 Randbedingungen . . . 20

2.5.3 CFL Bedingung . . . 21

3 Parallelisierung **23**

3.1 Die CUDA-Plattform . . . 24

3.2 Speicher und Datentypen in CUDA . . . 27

3.3 Sprachkonstrukte in CUDA . . . 29

4 Implementierung **31**

4.1 Transport-Berechnung . . . 32

4.2 Das Druckgleichungssystem . . . 36

4.2.1 Dünnbesetzte Matrix . . . 36

4.2.2 Aufbau des Gleichungssystems . . . 37

4.2.3 Gleichungssystem-Löser in CUDA . . . 40

4.3 Die in-situ Visualisierung . . . 44

5 Ergebnisse **47**
5.1 Genauigkeit/Fehlertoleranz 49
5.1.1 Probleme bei einfacher Gleitkomma-Genauigkeit 52
5.2 Leistungsanalyse 54

6 Zusammenfassung und Ausblick **57**

A Bedienungsanleitung `porSimGPU` **59**
A.1 Steuerung der in-situ Visualisierung 60

Literatur **61**

Abbildungsverzeichnis

1 Poröse Medien. 6
2 Die Oberflächenspannung. 7
3 Die effektive Sättigung. 9
4 Die relative Permeabilität $k_{r\alpha}$. 10
5 Die *fractional flow*-Funktion. 12
6 Die Sättigungs-Front in eindimensionaler Ausbreitung. 13
7 Finites Volumen. 15
8 Dünnbesetze Konnektivitätsmatrix 18
9 Zelle mit Nachbarzellen im kartesischen Gitter. 20
10 Parallele Architekturen im Vergleich. 23
11 Die CUDA-Architektur. 25
12 Speicher-Hierarchie in CUDA. 29
13 *Konjugierte Gradienten* vs. *Gradient-Abstiegsverfahren* 42
14 Gitter-Visualisierung von `porSimGPU`. 44
15 Volumen-Visualisierung. 45
16 Sättigungsverteilung in einem $240\times16\times16$ Gitter bei $t = 5 \cdot 10^7$. . . . 47
17 Randbedingungen des Berechnungsgitters. 48
18 Druckverteilung nach einer Simulationszeit von $t = 5 \cdot 10^7$. 48
19 Veränderung der Randbedingung. 49

20 Zeitlicher Verlauf der Sättigungsfront. 50
21 Numerische Oszillation. 52
22 Druckverteilung im Vergleich. 53
23 Leistungsverhalten. 55

Listings

1 Grundstruktur eines CUDA-Kernels. 25
2 Speichervorbereitung und Aufruf eines Kernels. 26
3 SaturationUpdateKernel. 33
4 CRS Matrix-Vektor Multiplikations-Kernel. 36
5 PressureMatrixAssemble-Kernel. 39
6 Klasse `DeviceWriteMatrix`. 40
7 SolvePressureEqSystemDevice. 43

Algorithmenverzeichnis

1 Das IMPES-Verfahren. 14
2 Der Simulationsablauf. 31
3 TransportStep. 32
4 TransportStep – CFL-Bedingung. 34
5 TransportStep – Randbedingung. 34
6 AssemblePressEqSystem. 37
7 AssemblePressEqSystem-Randbedingung. 38
8 SolvePressEqSystem als PCG Verfahren 41

Tabellenverzeichnis

1 Relativer Fehler der Simulation. 51
2 Vergleich der Berechnungszeiten. 54

1 Einleitung

In der Natur spielen sie oft eine Rolle, aber auch in Naturwissenschaft und Technik: Mehrphasen-Strömungen sowie Transportprozesse in porösen Medien. Das klassische Beispiel hierfür sind Simulationen von Öl-Reservoiren, in welche an einer Stelle Wasser oder Gas eingeleitet wird, um das Öl an einer anderen Stelle aus dem Erdreich zu gewinnen. Da sich Wasser und Öl chemisch nicht mischen lassen, spricht man hier von unterschiedlichen *Phasen*. Ein weiterer konkreter Anwendungsfall ist die Ausbreitung von Schadstoffen im Erdreich. Die Vorhersage der voranschreitenden Kontamination des Grund- beziehungsweise Trinkwassers ist von hoher Relevanz. Hier bilden der Schadstoff und das Grundwasser die Phasen und das Erdreich das poröse Medium. Auch andere Bereiche wie CO_2-Einlagerung im Erdreich, geothermale Prozesse oder der Blutfluss durch das Kapillarnetz sind Anwendungsgebiete der Simulation von Fluss- und Transportprozessen, denen das Medium ein Widerstand entgegen setzt.

1.1 Fragestellung

In realistischen Anwendungen für derartige Simulationen handelt es sich um größere Zeiträume die simuliert werden, sowie um eine große räumliche Ausdehnung des zu betrachtenden Gebiets. Dies führt in einem Simulationsprogramm zu hohem Rechenaufwand und bietet sich zur Optimierung durch parallele Berechnung an. In letzter Zeit ist die Grafikhardware als Zielplattform für massiv parallele Berechnungen in den Vordergrund gerückt. Dies liegt unter anderem in der breiteren Verfügbarkeit und in der besseren hard- und softwareseitigen Unterstützung begründet. In einem langen Prozess hat sich die Hardware zur Bildberechnung aufgrund ihrer Bauweise immer mehr zu einer separaten Einheit entwickelt, auf der beliebige Berechnungen ausgeführt werden können. Das gegebene Problem der Strömungs- und Transportprozesse wird üblicherweise über reguläre Gitterstrukturen berechnet, die sich gut zur parallelen Berechnung auf der Grafikhardware abbilden lassen. Vereinfacht ausgedrückt, lässt sich jede Gitterzelle durch eine der vielen Recheneinheiten der GPU[4], dem Herzstück moderner Grafikkarten, berechnen.

Die mathematischen Modelle solcher Simulationen sind üblicherweise Differenzialgleichungssysteme, die jeden Punkt im Raum in Abhängigkeit zu seiner räumlichen und zeitlichen Umgebung stellen. Viele dieser Gesetzmäßigkeiten sind empirisch bestimmt oder der Realität so angepasst, dass zutreffende Berechnungen und Vorhersagen möglich werden. Für reale Szenarien sind derartige Probleme nicht analytisch lösbar. Lediglich mit Hilfe von Computerprogrammen kann man sich einer Lösung numerisch annähern. Dazu muss das Problem räumlich und zeitlich diskretisiert, sowie die Lösungen der Differenzialgleichungssysteme numerisch angenähert werden. Dabei kann man gekoppelte Abhängigkeiten in separat berechenbare Teile aufgliedern. Dies führt zu einer Reduzierung der Berechnungskomplexität.

[4] *G*raphics *P*rocessing *U*nit.

Die theoretischen Grundlagen, wie sie in Kapitel 2 vorgestellt werden, basieren auf dem Vorlesungsskript von Prof. Dr. Helmig [7], dem Buch von Zhangxin Chen et al. [4] sowie teilweise auf der Masterarbeit von Asbjørn Bydal [3].

Als Referenz für das Simulationsprogramm "`porSimGPU`", dessen Implementierung in Kapitel 4 beschrieben wird, dient die Software-Plattform DuMu^x[6][5]. DuMu^x ist für die Simulation von Fluss- und Transportprozessen in porösen Medien konzeptioniert und basiert auf der FE-Toolbox DUNE[5][9].

1.2 Gang der Arbeit

In Kapitel 2 werden zunächst ausführlich die mathematischen Grundlagen behandelt und darauf aufbauend das Modell für die betrachtete Zweiphasen-Strömung vorgestellt. Im Unterkapitel 2.5 wird die Grundlage für die Simulation geschaffen und das mathematische Modell in eine diskrete Form gebracht. Diese bildet die Basis für die spätere Implementierung in Kapitel 4.

Kapitel 3 gibt eine Einführung in parallele Architekturen und insbesondere in das verwendete CUDA-Framework. Dabei wird unter anderem auf die Besonderheiten der parallelen Programmierung eingegangen.

Kapitel 4 beschreibt die Implementierung des Simulationsprogramms, welches den praktischen Teil dieser Arbeit ausmacht. Hier werden die wichtigsten Algorithmen erläutert. Die Implementierung in CUDA wird vorgestellt und mit Auszügen aus dem Quellcode veranschaulicht.

Kapitel 5 stellt die Simulationsergebnisse vor. Es werden Genauigkeit und Leistung des Simulationsprogramms mit CPU- und GPU-Ansatz für unterschiedliche Szenarien evaluiert. Auch ein Vergleich zur Referenz-Implementierung DuMu^x[6] wird vorgenommen.

Abgeschlossen wird diese Diplomarbeit in Kapitel 6 mit der Zusammenfassung und einem Ausblick.

[5] *D*istributed and *U*nified *N*umerics *E*nvironment, `http://www.dune-project.org`.

2 Grundlagen

Partielle Differenzialgleichungen (kurz *PDG*, engl. *PDE*[6]) bilden die Grundlage für die Modellierung vieler physikalischer Vorgänge. Es werden räumliche oder zeitliche Änderungen verschiedener Größen in Bezug zueinander gesetzt. Dabei spielen physikalische Erhaltungsgesetze eine wichtige Rolle. Beispielweise bleibt die Energie in einem geschlossenen System konstant - man spricht in diesem Fall von Energieerhaltung. Dasselbe lässt sich auf Masse, Impuls, Ladung etc. übertragen. Wird nun eine räumliche oder zeitliche Änderung einer solchen Zustandsgröße mit einer *PDE* beschrieben, so muss diese in ein Gleichgewicht mit dem Umfeld gebracht werden, um für das Gesamtsystem die Erhaltungsgesetze nicht zu verletzen.

Die Formulierung eines Erhaltungsgesetzes durch eine *PDE* beschreibt ein Problem jedoch nicht vollständig, denn es wird keine Aussage über den Anfangszustand gemacht. Es können viele Lösungen eine gegebene Gleichung erfüllen. Um eine eindeutige Lösung zu erhalten, müssen *Startwerte* beziehungsweise *Randbedingungen* definiert werden. Diese Bedingungen gelten am *Rand* der Berechnung, das heißt entweder räumlich am Rand eines endlichen Gebietes oder zeitlich als Ausgangswert der Betrachtung. Als die beiden wichtigsten Typen von Randbedingungen gelten die *Dirichlet-* und die *Neumann-Randbedingung.*

Eine *Dirichlet-Randbedingung* definiert einen konkreten Wert, den eine Variable der *PDE* am Rand einnimmt. Dieser Wert kann konstant oder beispielweise eine Funktion des Ortes sein.

Eine *Neumann-Randbedingung* definiert für den Rand der Berechnung einen konkreten Ableitungswert, der an der jeweiligen Stelle in die PDE einzusetzen ist. Auf diese Weise kann ein Austausch mit der Umgebung, beispielsweise das Einströmen von Masse oder Energie in ein System, beschrieben werden.

Treten in einem Differentialgleichungssystem mehrere Erhaltungsgrößen auf, so können für jede Größe und für jeden Teil des Randgebiets unterschiedliche Randbedingungen definiert werden.

2.1 Die Kontinuitätsgleichung

Für die Simulation jeder Art von Strömung sind grundsätzlich die physikalischen Erhaltungsgesetze zu beachten - im Einzelnen die Erhaltung der Masse, des Impulses und der Energie. Insbesondere die Erhaltung der Masse ist hierbei hervorzuheben. Dieses Gesetz besagt, dass Masse nicht aus dem Nichts entsteht oder verschwindet. Mathematisch wird das durch die *Divergenzfreiheit* der bewegten Masse ausgedrückt. Wird also das Geschwindigkeitsfeld einer Strömung berechnet, so muss darauf geachtet werden, dass dieses Vektorfeld keine Quellen oder Senken aufweist -

[6] *P*artial *D*ifferential *E*quations.

zumindest dort, wo dies nicht ausdrücklich durch einen Quellterm q vorgesehen ist. Die Divergenzfreiheit wird im Bezug auf Fluide auch *Kontinuitätsgleichung* genannt:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \vec{v}) = q. \tag{2.1}$$

Der verwendete *Divergenz*operator "$\nabla \cdot$" ist als das Skalarprodukt des Operanden mit dem räumlichen Ableitungsvektor definiert: $\nabla \cdot \vec{v} = \frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z}$.

Die Kontinuitätsgleichung (2.1) setzt die zeitliche Änderung des Dichtefelds ρ und die Divergenz des Geschwindigkeitsfeldes $\vec{v}$ mit den Quellen und Senken q gleich. Der Quellterm q ist am Ort einer Senke negativ und an einer Quelle positiv. Werden keine Quellen/Senken angenommen und die Dichte ρ als konstant betrachtet, so vereinfacht sich (2.1) zur *Divergenzfreiheit*:

$$\nabla \cdot \vec{v} = 0.$$

2.2 Strömung in porösen Medien

Ein poröses Medium besteht aus einem festen Material mit Hohlräumen, sogenannten *Poren*, durch die Flüssigkeit hindurchströmen kann. Die Anordnung und Ausdehnung der Poren bestimmt, wie durchlässig das Medium für ein Fluid ist - das Maß hierfür ist die *Permeabilität* k. Der Anteil des Hohlvolumens zum Gesamtvolumen ist die *Porösität* ϕ. Der Anteil des Hohlvolumens, der durch eine Flüssigkeit ausgefüllt ist, wird *Sättigung* S genannt. Eine wichtige Größe für die Betrachtung von Fluiden ist die *Viskosität* μ. Sie ist ein Maß für die *Zähflüssigkeit* eines Fluids und wird über den Widerstand bestimmt, den die einzelnen Moleküle des Fluids einer Scherungskraft entgegensetzen. Diese Scherspannung verhält sich im Normalfall linear zur Schergeschwindigkeit - man spricht dann von einem *Newton'schen Fluid.*

In dieser Arbeit werden poröse Medien nicht mikroskopisch auf der Ebene der einzelnen Poren, sondern makroskopisch auf der Ebene der Materialeigenschaften betrachtet (*Porösität* und *Permeabilität*). Die bereits vorgestellte Kontinuitätsgleichung (2.1) wird für poröse Medien um die *Porösität* ϕ erweitert:

$$\phi \frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \vec{v}) = q. \tag{2.2}$$

2.2.1 Darcy-Gesetz

Für Strömungen von Flüssigkeiten in porösen Medien gilt das sogenannte *Darcy-Gesetz* (Darcy's Law) - eine spezielle Lösung der allgemeinen Navier-Stokes-Gleichungen.

Diese Gleichung ist eine Beschreibung der Impulserhaltung und ist analog zum Ohmschen Gesetz oder der Wärmeausbreitung zu sehen. All diese Gleichungen beschreiben die Ausbreitung einer Kraft, der durch das Ausbreitungsmedium ein Widerstand entgegensetzt wird.

Die Gesetzmäßigkeit des *Darcy-Gesetzes* wurde im 19. Jahrhundert von Henry Darcy empirisch ermittelt. Es stellt das Geschwindigkeitsfeld mit dem Druck-Gradienten in linearen Zusammenhang. Das Fluid bewegt sich also abhängig vom Druckunterschied im Raum durch ein poröses Medium - abhängig von den Materialeigenschaften des Mediums:

$$\vec{v} = -\frac{1}{\mu} k \nabla p. \tag{2.3}$$

Mathematisch (vereinfacht, ohne Gravitation) wird das Darcy-Gesetz in Formel (2.3) dargestellt, wobei p für den Druck und $\vec{v}$ für das Geschwindigkeitsfeld steht. Der Faktor k steht für die *Permeabilität* beziehungsweise der *Durchlässigkeitsbeiwert* des Mediums und μ für die *Viskosität*.

Das Geschwindigkeitsfeld $\vec{v}$ ist hier jedoch nicht als die Geschwindigkeit der einzelnen Fluid-Partikel zu betrachten, sondern als ein Maß für den Massentransport durch ein Volumen. Daher wird das Darcy-Gesetz auch Filtergesetz genannt. Für die reale Geschwindigkeit der Fluid-Partikel in einem porösen Medium gilt:

$$\vec{v}_{real} = \frac{\vec{v}}{\phi}.$$

Theoretisch könnte man Strömungen in porösen Medien anstatt mit dem Darcy-Gesetz auch mit den allgemeinen Navier-Stokes-Gleichungen beschreiben. Die genaue Beschaffenheit des Mediums ist jedoch in unserem makroskopischen Ansatz unbekannt und wäre außerdem sehr aufwendig zu modellieren. Das Darcy-Gesetz ist daher eine gute empirische Näherung der tatsächlichen Verhältnisse.

2.2.2 Permeabilität

Das Maß für die Durchlässigkeit eines porösen Mediums, die *Permeabilität* k, besteht im einfachsten Fall aus einer materialabhängigen Konstante. Beispielsweise würde dies bei einem Medium aus gleichmäßigen Kieselsteinen oder Sand zutreffen (siehe Bild a) in Abbildung 1). Dann nennt man das poröse Medium *isotrop*.

Es gibt aber auch Situationen, in denen die Permeabilität richtungsabhängig ist, beispielsweise wenn es sich um längliche Gesteinsbrocken mit einer Ausrichtung handelt (siehe Bild b) in Abbildung 1). Ein Fluid erfährt hier horizontal weniger Widerstand als in vertikaler Flussrichtung. Hier spricht man von einem *anisotropen* Medium und der Durchlässigkeitsbeiwert k wird zum Tensor:

$$k = \begin{pmatrix} k_{xx} & & \\ & k_{yy} & \\ & & k_{zz} \end{pmatrix}. \tag{2.4}$$

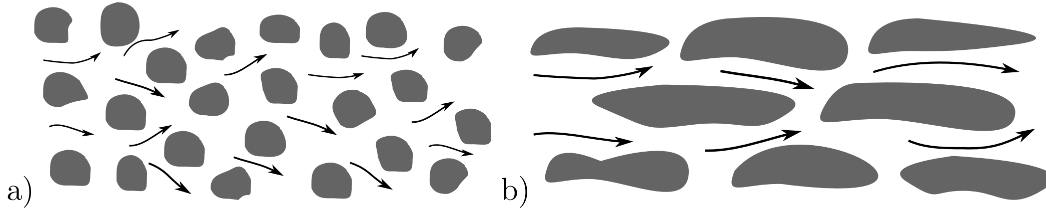


Abbildung 1: Poröse Medien: a) isotrop, b) anisotrop.

Formel (2.4) zeigt k als *Diagonaltensor* - auf diese Weise kann in jeder Raumachse eine unterschiedliche Permeabilität abgebildet werden. Es sind selbstverständlich auch nicht-triviale Tensoren k möglich.

Die Formeln für die Impuls- und Massenerhaltung fasst man in einer Gleichung zusammen. Dazu wird das Darcy-Gesetz (2.3) in die erweiterte Kontinuitätsgleichung (2.2) eingesetzt und man erhält:

$$\phi \frac{\partial \rho}{\partial t} - \nabla \cdot \left(\frac{\rho}{\mu} k \nabla p \right) = q. \tag{2.5}$$

In Gleichung (2.5) wurde durch Einsetzen die unabhängige Variable $\vec{v}$ eliminiert. Wird die Dichte ρ als konstant über die Zeit betrachtet, so ist neben der Massen- und der Impulserhaltung auch die Energieerhaltung gewährleistet. Auf diese Weise ist die eine Einphasen-Strömung vollständig charakterisiert.

2.3 Mehrphasenströmung

Besteht das zu simulierende Fluid aus mehreren, chemisch nicht interagierenden, beziehungsweise nicht mischbaren Flüssigkeiten, so spricht man von mehreren *Phasen*. Im Folgenden wird davon ausgegangen, dass das gesamte Hohlvolumen des zu betrachtenden Mediums mit Flüssigkeit gefüllt ist. Für die Gesamt-Sättigung gilt daher immer $S = 1$. Jede Phase nimmt für sich genommen einen Teil des Volumens ein - alle Phasenvolumina addiert ergeben das gesamte ausfüllbare Volumen. Den Anteil einer Phase in einem Teilvolumen nennt man *Phasen-Sättigung*. Für die Sättigung einer Phase S_α in einem Medium mit den Phasen $\alpha \in P$ gilt in jedem Teilvolumen:

$$\sum_{\alpha \in P} S_\alpha = 1. \tag{2.6}$$

Ebenso werden der globale Druck p und das globale Geschwindigkeitsfeld $\vec{v}$ innerhalb des Fluids aus den einzelnen Phasen-Drücken p_α respektive Geschwindigkeiten $\vec{v}_\alpha$ zusammengesetzt:

$$p = \sum_{\alpha \in P} p_\alpha, \tag{2.7}$$

$$\vec{v} = \sum_{\alpha \in P} \vec{v}_\alpha. \tag{2.8}$$

Zwei Phasen können zueinander unterschiedliche Oberflächenspannungen haben, was insbesondere das Verhalten an der Grenzschicht beeinflusst. Die Phase mit der größeren Oberflächenspannung nennt man "benetzende Phase" (*wetting phase*), da diese an der Grenze im porösen Medium die andere Phase verdrängt. Ein anschauliches Beispiel für diesen Effekt sind die Phasen Wasser und Luft in einem dünnen Glasrohr, wobei sich das Wasser mit deutlich größerer Oberflächenspannung am Rand nach oben wölbt.

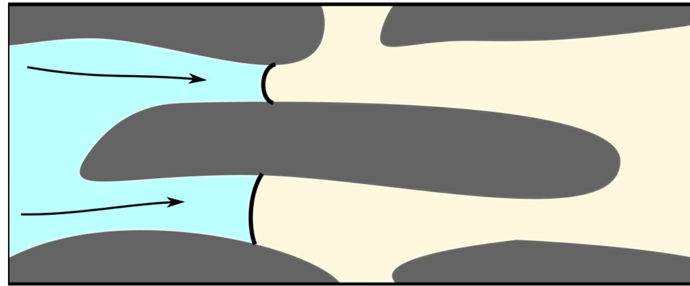


Abbildung 2: Oberflächenspannung an der Phasen-Grenze zwischen *wetting phase* (blau) und *non-wetting phase* (gelb).

Der Effekt der Oberflächenspannung wird in Abbildung 2 verdeutlicht: Die Grenzschicht zwischen der *wetting phase* und der *non-wetting phase* bildet eine Wölbung aus. Je feiner die Poren des Mediums, desto stärker der Effekt.

2.3.1 Zweiphasen-Strömung

Im Folgenden wird von einer Problemstellung mit zwei Phasen ausgegangen, nämlich einer *wetting phase* und einer *non-wetting phase*. Für den Phasenindex α bei der *wetting phase* gilt hier $\alpha = w$, entsprechend gilt $\alpha = n$ für die *non-wetting phase*. Das charakteristische Beispiel für eine Zweiphasen-Strömung ist ein Wasser-Öl-Modell mit Wasser in der Rolle der *wetting phase*. Für die Sättigung gilt wegen Gleichung (2.6) in einer Zweiphasen-Strömung:

$$S_n + S_w = 1. \tag{2.9}$$

Aus der Gleichung (2.9) folgt die einfache Abhängigkeiten zwischen den Phasen-Sättigungen: $S_w = 1 - S_n$ und $S_n = 1 - S_w$. Das ist deshalb erwähnenswert, da dieser Zusammenhang häufig bei der Kopplung der Phasen und für das Umformulieren durch Substitution benötigt wird.

Aufgrund der Oberflächenspannung ist der Druck der *wetting phase* (p_w) nahe der Grenzschicht geringer als der Druck in der *non-wetting phase* (p_n). Es entsteht eine

Stufe in der Druckverteilung entlang der Grenzschicht zwischen den Phasen. Diese Druckdifferenz wird *Kapillardruck* p_c genannt. Der Kapillardruck beeinflusst indirekt die Sättigung. Für p_c gilt folgender Zusammenhang:

$$p_n - p_w = p_c. \tag{2.10}$$

Für die Impulserhaltung in Form des Darcy-Gesetz (2.3) gilt für jede Phase $\alpha \in \{w, n\}$ jeweils:

$$\vec{v}_\alpha = -\frac{1}{\mu_\alpha} k_\alpha \left(\nabla p_\alpha\right), \qquad \alpha \in \{w, n\}. \tag{2.11}$$

Die Permeabilität einer Phase k_α wird später noch genauer erläutert.

Für die Kontinuitätsgleichung (2.2) muss beachtet werden, dass für die Massenerhaltung die Konzentration pro Phase, d.h. die Sättigung S_α, mit berücksichtigt wird (Masse bleibt in jeder Phase erhalten):

$$\phi \frac{\partial \left(\rho_\alpha S_\alpha\right)}{\partial t} + \nabla \cdot \left(\rho_\alpha \vec{v}_\alpha\right) \;=\; q_\alpha, \qquad \alpha \in \{w, n\}. \tag{2.12}$$

Die Massenerhaltung (2.12) für eine Phase lässt sich analog zu (2.5) mit der Impulserhaltung für eine Phase (2.11) in eine Formel einsetzen:

$$\phi \frac{\partial \left(\rho_\alpha S_\alpha\right)}{\partial t} - \nabla \cdot \left(\frac{\rho_\alpha}{\mu_\alpha} k_\alpha \left(\nabla p_\alpha\right)\right) = q_\alpha, \qquad \alpha \in \{w, n\}. \tag{2.13}$$

Dadurch erhält man ein Differenzialgleichungssystem, welches die ortsabhängigen Variablen Sättigung S und den Druck p direkt in Verbindung bringt. Übliche vereinfachende Annahmen zur Simulation sind unter anderem, dass die Dichte ρ und die Viskosität μ als konstant über die Zeit und der Quellterm q mit 0 angenommen werden. Analog zur Einphasen-Strömung macht die Annahme der konstanten Phasen-Dichte ($\frac{\partial \rho_\alpha}{\partial t} = 0$) eine zusätzliche Formulierung der Energieerhaltung überflüssig.

Mit diesen Annahmen und der Formulierung der Massen- und Impulserhaltung für beide Phasen $\alpha \in \{w, n\}$ in (2.13) sowie den Zusammenhängen zwischen Sättigung (2.9) und Druck (2.10) sind vier Formeln gegeben, welche die Unbekannten S_w, S_n, p_w, p_n und p_c koppeln. Im Modell dieser Arbeit wird schließlich noch die Annahme $p_c = 0$ getroffen, wodurch mit den vorgestellten Formeln ein abgeschlossenes Differentialgleichungssystem für zwei Phasen in einem porösen Medium beschrieben wird.

2.3.2 Effektive Sättigung

Im Folgenden wird insbesondere für die Berechnung der Permeabilität k_α nicht mit der absoluten Sättigung einer Phase S_α gerechnet, sondern mit der *effektiven Sättigung* $\overline{S}_\alpha$.

Die *effektive Sättigung* $\overline{S}_\alpha$ ist mit der absoluten Sättigung S_α eng verbunden - man erhält sie durch Herausrechnen einer *Residual-Sättigung* der jeweiligen Phase. Die *Residual-Sättigung* stellt einen minimalen Grenzwert der Sättigung einer Phase dar. Dies ist der Tatsache geschuldet, dass in unserem Zweiphasen-Modell eine Phase niemals vollständig die andere verdrängen kann. In einem realen Modell bleiben immer Reste einer Flüssigkeit in besonders kleinen oder undurchlässigen Poren des Mediums zurück. Auch die Oberflächenspannung kann dafür sorgen, dass Reste eines Fluids kleine Partikel einschließen und deshalb nicht mehr am Transportprozess teilnehmen können.

Bei unseren zwei Phasen *wetting* und *non-wetting* werden Sättigungswerte S_{wr} und S_{nr} definiert, die für die jeweilige Phase nicht unterschritten werden dürfen. Soll beispielsweise für beide Phasen eine Residual-Sättigung von $S_{wr} = S_{nr} = 0.2$ gelten, so können sich wegen $S_w + S_n = 1$ (2.9) die Sättigungen für die Phasen S_w und S_n im Wertebereich von $[0.2 \ldots 0.8]$ bewegen. Die effektive Sättigung $\overline{S}_w$ und $\overline{S}_n$ wird jedoch immer auf den Bereich von $[0 \ldots 1]$ skaliert.

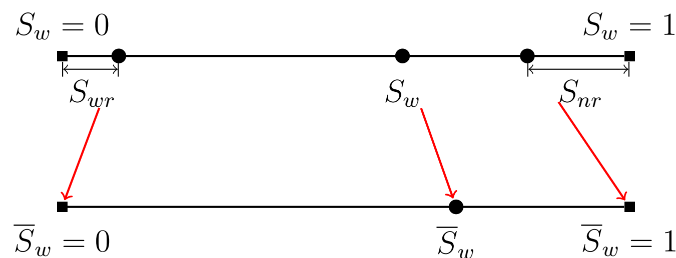


Abbildung 3: Die effektive Sättigung.

Grafisch ist dieser Zusammenhang in Abbildung 3 dargestellt.

Absolute und effektive Sättigung hängen wie folgt zusammen:

$$\overline{S}_w(S_w) = \frac{S_w - S_{wr}}{1 - S_{nr} - S_{wr}}. \tag{2.14}$$

Die Formel (2.14) lässt sich selbstverständlich auch für $\overline{S}_n$ formulieren. Es gilt analog zu Gleichung (2.9):

$$\overline{S}_w + \overline{S}_n = 1.$$

2.3.3 Effektive und relative Permeabilität

Die Phasen-Permeabilität k_α, im Folgenden *effektive Permeabilität* genannt, gibt das Maß für die Durchlässigkeit des Mediums für die Phase α an. Sie ist einerseits von materialabhängigen Parametern bestimmt und hängt andererseits von der effektiven Phasen-Sättigung $\overline{S}_\alpha$ ab:

$$k_\alpha = k_{r\alpha}\left(\overline{S}_\alpha\right) \cdot k, \qquad \alpha \in \{w, n\}\,. \tag{2.15}$$

Die *relative Permeabilität* $k_{r\alpha}$ gibt die Tendenz der Phase α an, wie stark diese in das Medium weiter vordringen kann. Die Permeabilität des porösen Mediums geht durch die *absolute Permeabilität* k in die Formel (2.15) ein. Aufgrund der Interaktion der Phasen kann die *effektive Permeabilität* k_α einer Phase nicht größer sein als die *absolute Permeabilität* k des porösen Mediums. Daher gilt für die *relative Permeabilität*: $k_{r\alpha} \leq 1$.

Die *relative Permeabilität* $k_{r\alpha}$ ist eine empirische Funktion und hängt stark vom verwendeten Modell ab.

Wie in Formel (2.15) skizziert, wird in dieser Arbeit die Modellannahme getroffen, dass die *relative Permeabilität* $k_{r\alpha}$ einer Phase α direkt von der *effektiven Sättigung* $\overline{S}_\alpha$ abhängt.

Im Folgenden wird für das Zweiphasen-Modell die relative Permeabilität $k_{r\alpha}$ in Abhängigkeit von der Sättigung der *wetting phase* S_w definiert - also $k_{rw}\left(\overline{S}_w\right)$ und $k_{rn}\left(\overline{S}_n\right) = k_{rn}\left(1 - \overline{S}_w\right)$.

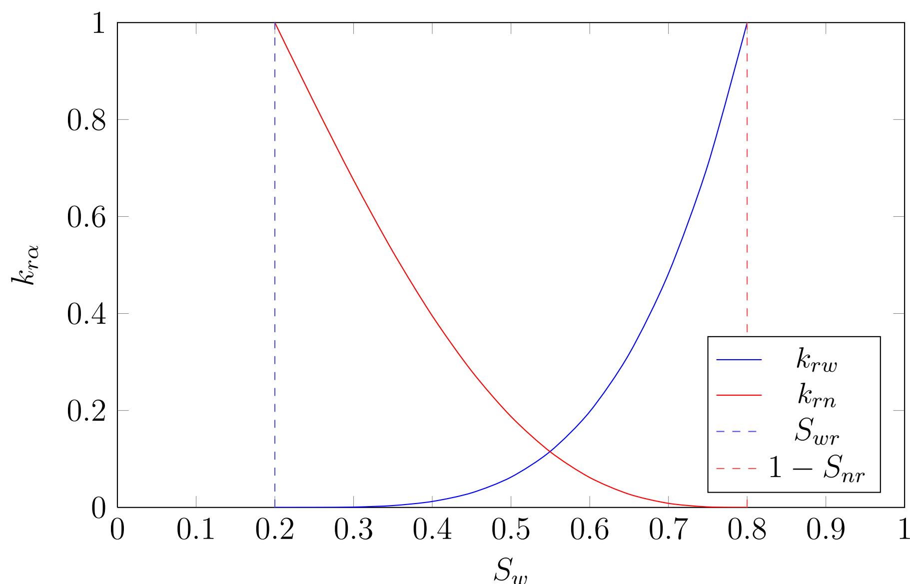


Abbildung 4: Die relative Permeabilität $k_{r\alpha}$.

Im Modell für diese Arbeit wird für die relative Permeabilität $k_{r\alpha}$ die Formulierung nach Brooks-Corey verwendet. In die Formel für $k_{r\alpha}$ geht dabei neben Materialkonstanten die Phasen-Sättigung S_α quadratisch ein. Ein Diagramm für k_{rw} und k_{rn} für ein Wasser-Öl Modell hat ein typisches Schaubild wie in Abbildung 4 zu sehen. In unserem Modell wird nun folgende Formulierung verwendet:

$$k_{rw}\left(\overline{S}_w\right) = \left(\overline{S}_w{}^2\right)^{\frac{2}{a}+1}, \tag{2.16}$$

$$k_{rn}\left(\overline{S}_w\right) = \left(1 - \overline{S}_w\right)^2 \left(1 - \overline{S}_w{}^{\frac{2}{a}+1}\right). \tag{2.17}$$

Die Konstante a ist ein Materialparameter. Für das Modell in dieser Arbeit wird $a = 2$ angenommen - wie auch im DuMu$^\mathrm{x}$-Programm[6].

2.3.4 Mobilität und *fractional flow*

Das vorgestellte Modell von Massenerhaltung (Kontinuitätsgleichung) und Impulserhaltung (Darcy-Gesetz) führt zu einer starken Kopplung der Hauptvariablen Sättigung S und Druck p - wie in Formel (2.13) dargestellt. Um dies zu entkoppeln und handlicher zu formulieren, wird die Größe *Mobilität* ($\lambda = \frac{k}{\mu}$) als das Verhältnis von Permeabilität k zur Viskosität μ eingeführt. Dieses Verhältnis tritt bei den Formulierungen des Darcy-Gesetz auf.

Im vorgestellten Modell der Mehrphasen-Umgebung gilt für die Mobilität einer Phase λ_α:

$$\lambda_\alpha\left(\overline{S}_\alpha\right) = \frac{k_{r\alpha}\left(\overline{S}_\alpha\right)}{\mu_\alpha}, \qquad \alpha \in \{w, n\}. \tag{2.18}$$

Für die *globale Mobilität* gilt allgemein und im Zweiphasen-Modell entsprechend:

$$\lambda = \sum_{\alpha \in P} \lambda_\alpha = \lambda_w + \lambda_n \tag{2.19}$$

Die *Mobilität* ist eine direkte Angabe, wie gut eine Phase abhängig von ihrer Sättigung transportiert wird. Man kann diese Größe auch analog zum Leitwert der Elektrizität sehen, also dem Kehrwert des elektrischen Widerstandes.

Weiter definieren wir eine Funktion *fractional flow* f_α:

$$f_\alpha\left(\overline{S}_\alpha\right) = \frac{\lambda_\alpha}{\lambda} = \frac{\lambda_\alpha\left(\overline{S}_\alpha\right)}{\lambda_w\left(\overline{S}_\alpha\right) + \lambda_n\left(1 - \overline{S}_\alpha\right)}, \qquad \alpha \in \{w, n\}. \tag{2.20}$$

Die *fractional flow*-Funktion gibt den Anteil des Massentransportes einer Phase am totalen Massentransport wieder. Bei einem Zweiphasen-Modell ergibt demnach die

Summe beider Mobilitäts-Anteile, die globale Mobilität und es gilt wie bei der Sättigung ein linearer Zusammenhang zwischen dem *fractional flow* der Phasen w und n:

$$f_w + f_n = 1. \tag{2.21}$$

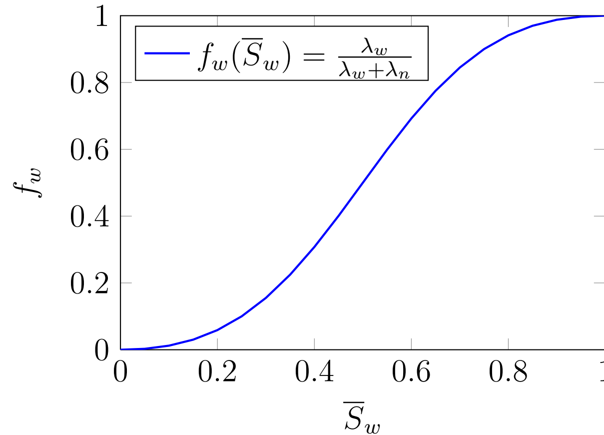


Abbildung 5: Die *fractional flow*-Funktion.

Abbildung 5 zeigt einen typischen (symmetrischen) Verlauf der *fractiontal flow* Rate für die *wetting phase* f_w in Abhängigkeit von der effektiven Sättigung $\overline{S}_w$.

2.4 Das Buckley–Leverett-Modell

Für das Modell dieser Arbeit werden eine Reihe von vereinfachenden Annahmen getroffen. Wie bereits erwähnt, werden die *Dichte* ρ sowie die *Porösität* ϕ räumlich und zeitlich als konstant betrachtet. Auch werden in dieser Arbeit nur *Newton'sche Fluide* betrachtet, was zu Folge hat, dass die *Viskosität* μ ebenfalls konstant ist.

Des weiteren wird der Kapillardruck vernachlässigt: $p_c = 0$. Daher ergibt sich wegen (2.10) und $p_c = 0$ dann: $p_n = p_w$. Genauer gesagt wird $\frac{\partial p_c}{\partial S} = 0$ angenommen, wodurch $\nabla p_n = \nabla p_w$ gilt - für unsere Formulierung ist das jedoch gleichbedeutend. Somit gilt:

$$\vec{v} = -k\lambda\left(\nabla p_n\right). \tag{2.22}$$

Mit Hilfe der vorgestellten *fractional flow*-Funktion f_α lässt sich ein Zusammenhang zwischen Phasengeschwindigkeit und totaler Geschwindigkeit herstellen (hier am Beispiel von $\alpha = w$):

$$\vec{v}_w = f_w\vec{v}. \tag{2.23}$$

Für die Umsetzung auf dem Computer wird das Modell für diese Arbeit auf die zwei (Haupt-)Variablen zusammengeführt: S_w und p_n. Wegen (2.9) sind S_w und S_n ohnehin beliebig gegeneinander substituierbar. Anstatt S_w und p_n wäre durchaus auch eine andere Kombinationen der vier Hauptvariablen S_w, S_n und p_w und p_n denkbar.

Mit Gleichung (2.23) wird unsere *Transportgleichung* analog zu (2.12) wie folgt definiert:

$$\phi \frac{\partial \overline{S}_w}{\partial t} + \nabla \cdot \left(f_w \left(\overline{S}_w \right) \vec{v} \right) = q. \tag{2.24}$$

Die Transportgleichung (2.24) vereint die auf dem Kapillardruck basierenden mikroskopischen Effekte eines porösen Mediums mit dem Darcy-Gesetz. Diese Formulierung wird auch *Buckley-Leverett* Gleichung genannt. Abbildung 6 zeigt beispielhaft einen eindimensionalen Verlauf der Sättigungsverteilung S_w - wie in (2.24) beschriebenen - entlang der x-Raumachse. Hierbei fließt, durch eine *Dirichlet-Randbedingung* bestimmt, an der Stelle $x = 0$ die *wetting phase* in das Medium hinein. Es breitet sich eine sogenannte Sättigungs-Front beziehungsweise Schockwelle der Sättigung mit steilem Gradient aus, gefolgt von einem Anstieg bis zur Residualsättigung.

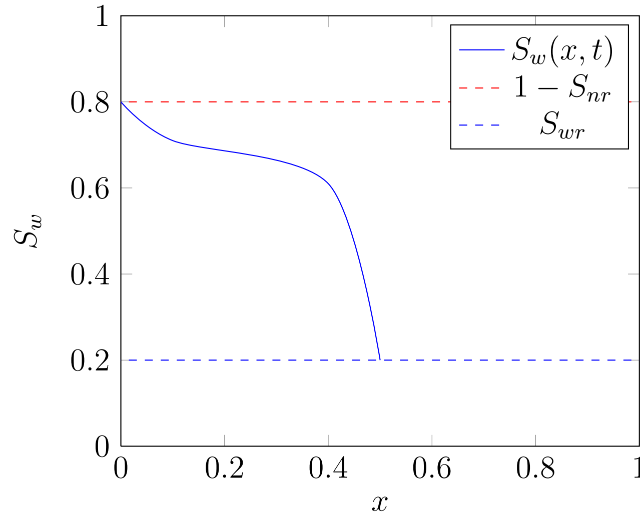


Abbildung 6: Die Sättigungs-Front in eindimensionaler Ausbreitung.

Wegen (2.8) gilt für die *globale Geschwindigkeit* $\vec{v} = \vec{v}_w + \vec{v}_n$ und somit haben wir mit dem Darcy-Gesetz (2.11) eine Abhängigkeit zu p_w und p_n in unserer Transportgleichung (2.24). Da im Modell der Kapillardruck vernachlässigt wird ($p_n = p_w$) lässt sich die globale Geschwindigkeit $\vec{v}$ wie folgt umformen:

$$\vec{v} = -k\lambda \left(\nabla p_n \right). \tag{2.25}$$

Transportgleichung (2.24) setzt so nur noch unsere Hauptvariablen S_w und p_n in Zusammenhang und kann nun mit der Phasenmobilität λ_w wie folgt formuliert werden:

$$\phi \frac{\partial \overline{S}_w}{\partial t} - \nabla \cdot \left(\lambda_w \left(\overline{S}_w \right) \, k \left(\nabla p_n \right) \right) = q. \tag{2.26}$$

2.4.1 Schwache Kopplung mit IMPES

Klassischerweise würden die Variablen für ein abgeschlossenes Gleichungssystem in einer stark gekoppelten Formulierung simultan berechnet werden. In dieser Arbeit wird jedoch die starke Kopplung der Hauptvariablen S_w und p_n in (2.26) in eine schwächer gekoppelte Form gebracht. Dazu wird die Sättigung S_w nach dem Euler-Verfahren mit *direkten* Zeitschritten über die zeitliche Änderung bestimmt:

$$S_w^{t+\Delta t} = S_w^t + \Delta t \cdot \frac{\partial S_w^t}{\partial t}. \tag{2.27}$$

Diese Berechnung des Sättigungs-Schrittes (2.27) bildet die Basis eines Gleichungssystems für die Druckverteilung:

$$\nabla \cdot \left(-\lambda(\overline{S}_w^t) k \left(\nabla p_n^t \right) \right) = q. \tag{2.28}$$

Durch die Lösung von (2.28) erhält man die Druckverteilung p_n zum Zeitpunkt t, die wiederum in die Neuberechnung des Sättigungs-Schrittes (2.27) eingeht. Dieses Schema wird auch IMPES[7] genannt - die implizite Bestimmung des Druckes im Wechsel mit der expliziten Bestimmung der Sättigung.

Algorithmus 1 Das IMPES-Verfahren.

$t \leftarrow 0$ ▷ mit Zeitschritt 0 starten
$S_w^t \leftarrow initialSaturation$ ▷ Sättigung initialisieren
while $t < t_{end}$ **do**
  solve: $\nabla \cdot \left(-\lambda(\overline{S}_w^t) k \left(\nabla p_n^t \right) \right) = q$ ▷ aktuellen Druck p_n^t *implizit* bestimmen
  $S_w^{t+\Delta t} = S_w^t + \Delta t \frac{\partial S_w^t}{\partial t} (p_n^t)$ ▷ neue Sättigung $S_w^{t+\Delta t}$ *explizit* bestimmen
  $t \leftarrow t + \Delta t$

Algorithmus 1 zeigt das IMPES-Verfahren als Algorithmus, der später die Grundlage für das Simulationsprogramm bildet.

Die schwache Kopplung der Variablen S_w und p_n durch das IMPES-Verfahren vereinfacht das Problem. Sowohl die Menge an Daten, die verwaltet werden müssen, als

[7] *Im*plicit *P*ressure *E*xplicit *S*aturation.

auch die Komplexität der Berechnung verringert sich dadurch. Die Genauigkeitseinbußen sind dabei vertretbar.

2.5 Diskretisierung

Die Diskretisierung unseres Problems erfolgt über *Finite-Volumen-Methoden* (kurz *FV*). Das bedeutet, dass das Berechnungsgebiet in endlich große (finite) Volumina unterteilt wird. Für jedes dieser Volumina, im Folgenden *Zellen* genannt, wird der Wert der Variablen im Mittelpunkt durch die Simulation berechnet. Der Wert dieses Mittelpunktes ist für die gesamte Zelle als konstant anzusehen. Dieser Ansatz wird auch *block-centered grid* genannt. Sonstige Raumpunkte, insbesondere an den Grenzflächen der Zellen, können mit Interpolation bestimmt werden. Die Differenz-Operation beziehungsweise der Gradient wird durch die lineare Interpolation der Werte-Differenz zu den Nachbar-Zellen berechnet. Dies führt insbesondere zu Verwaltungsaufwand an den Zellen am Rand des Berechnungsgebietes - doch dazu später mehr in Abschnitt 2.5.2.

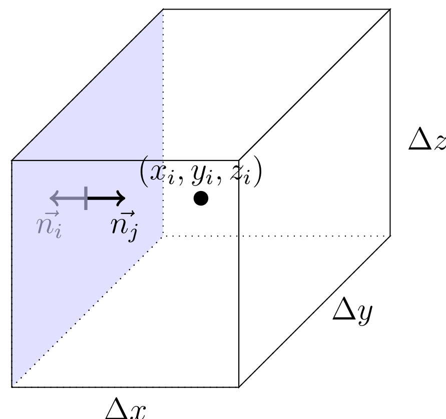


Abbildung 7: Finites Volumen / Zelle eines *block-centered grid*.

Um die Parallelisierbarkeit des Programms zu vereinfachen, verwenden wir in unserem Modell strukturierte, *kartesische Gitter*, in welchen alle Zellkanten parallel zu den Raumachsen liegen. Dadurch vereinfacht sich insbesondere die Betrachtung von zwei benachbarten Zellen, da deren Grenzflächen triviale/achsenparallele Oberflächen-Normalen $\vec{n}$ besitzen. Die Differenzen-Berechnung und somit das Lösen der Transport-Gleichung vereinfachen sich dadurch deutlich. In Abbildung 7 wird eine Zelle eines kartesischen Gitters dargestellt. Dem zu berechnenden Zellwert wird im Folgenden der Index i zugewiesen, dem Wert einer angrenzenden Nachbarzelle der Index j.

Um unsere Transportgleichung auf ein Finite-Volumen-Schema umzusetzen, betrachtet man die Masse, die einer einzelnen Zelle mit Index i zufließt oder diese verlässt. Dies wird mit dem Fluss über die Oberfläche bestimmt - also einem Gauß-Integral über die Oberfläche des Volumen Ω_i:

$$\int_{\partial\Omega_i} \langle \rho\vec{v}, \vec{n} \rangle \, \mathrm{d}s. \tag{2.29}$$

Der $\langle \cdot, \cdot \rangle$-Operator bezeichnet das Skalarprodukt zwischen zwei Vektoren, $\vec{v}$ die Geschwindigkeit und $\vec{n}$ den nach außen gerichteten Normalenvektor der Oberfläche. Die Integrationsvariable für die Oberfläche (ds) sollte hier nicht mit der später eingeführte Sättigungs-Variable s_i für die Diskretisierung des Transportprozesses verwechselt werden.

Mit Hilfe des *Divergenz-Theorems*, das den Fluss über eine geschlossene Oberfläche mit der Divergenz gleichsetzt, gilt:

$$\int_{\partial\Omega_i} \langle \vec{v}, \vec{n} \rangle \, \mathrm{d}s = \int_{\Omega_i} \nabla \cdot \vec{v} \, \mathrm{d}\mathbf{x}. \tag{2.30}$$

Durch Einsetzen von (2.25) in (2.30) erhält man:

$$\int_{\partial\Omega_i} \langle \vec{v}, \vec{n} \rangle \, \mathrm{d}s = -\int_{\Omega_i} \nabla \cdot (k\lambda \nabla p_n) \, \mathrm{d}\mathbf{x}. \tag{2.31}$$

Durch die Formulierung (2.30) beziehungsweise (2.31) wird die Divergenz in unserer Transportgleichung (2.24) zum Oberflächenintegral über die einzelnen *Finite-Volumen*-Zellen. Da wir in einem kartesischen Gitter arbeiten, wird das Oberflächenintegral (2.29) in jeder Zelle über die Summe der Transporte der sechs Seitenflächen bestimmt:

$$\int_{\partial\Omega_i} \langle \vec{v}, \vec{n} \rangle \, \mathrm{d}s = \sum_{j \in N(i)} |\Omega_{ij}| \left\langle \vec{v}_{ij}, \vec{n}_{ij} \right\rangle. \tag{2.32}$$

In (2.32) durchläuft der Index j alle Indizes der angrenzenden Nachbarzellen N von Zelle i. Mit $|\Omega_{ij}|$ wird das Volumen der gemeinsamen Fläche zwischen den Zellen mit Index i und j bezeichnet.

Die Transportgleichung (2.24) wird nun als Integral über ein infinitesimales Volumen Ω betrachtet:

$$\int_\Omega \phi \frac{\partial \overline{S}_w}{\partial t} \mathrm{d}V + \int_\Omega \nabla \cdot \left(f_w \left(\overline{S}_w \right) \vec{v} \right) \mathrm{d}V = \int_\Omega q \mathrm{d}V.$$

Wird nun zeitlich mit endlich großem Δt und räumlich mit endlich großem Volumen Ω_i diskretiesiert, so erhält man ($s_i = \overline{S}_w(\Omega_i)$, n ist der aktueller Zeitschritt, i der Zellenindex):

$$|\Omega_i| \, \phi \frac{s_i^{n+\Delta t} - s_i^n}{\Delta t} + \int_{\Omega_i} \nabla \cdot (f_w (s) \, \vec{v}) \mathrm{d}V = |\Omega_i| \, q_i, \qquad \forall i \in I. \tag{2.33}$$

Mit $|\Omega_i|$ wird das Volumen der Zelle i bezeichnet. Der Wert q_i gibt den Term für Quellen/Senken für jede Zelle i konstant an. I steht für die Menge aller Zellen im Berechnungsgitter.

Nun kann das verbleibende Integral in (2.33) durch den Fluss über die Volumen-Oberfläche (2.32) ersetzt werden und man erhält ($p = p_n(\Omega_i)$):

$$|\Omega_i|\,\phi\frac{s_i^{n+\Delta t} - s_i^n}{\Delta t} + \sum_{j\in N(i)} |\Omega_{ij}|\, f_w(s_{ij}^n)\,\langle \vec{v}_{ij}, \vec{n}_{ij}\rangle = |\Omega_i|\, q_i, \qquad \forall i \in I \tag{2.34}$$

Mit $f_w(s_{ij}^n)$ wird die *fractional-flow*-Funktion an der Oberfläche zwischen den Zellen i und j zum Zeitpunkt n bezeichnet. Da wir jedoch nur Werte an den Mittelpunkten der FV-Zellen berechnen, muss die Sättigung s_{ij}^n an der Zellgrenze noch gesondert definiert werden. Dies wird nicht durch einfache Interpolation, sondern durch *upwind-approximation* realisiert - siehe Abschnitt 2.5.1.

Nun kann ein Gauß'sches Iterations-Schema nach dem Schema (2.27) gebildet werden:

$$s_i^{n+\Delta t} = s_i^n + \frac{\Delta t}{\phi}\underbrace{\left(q_i - \frac{1}{|\Omega_i|}\sum_{j\in N(i)} |\Omega_{ij}|\, f_w(s_{ij}^n)\,\langle \vec{v}_{ij}, \vec{n}_{ij}\rangle\right)}_{\frac{\partial S_w}{\partial t}}, \qquad \forall i \in I \tag{2.35}$$

Mit der Formel (2.35) lässt sich der Transport von Masse durch Druckunterschiede ermitteln. $|\Omega_i|$ steht dabei für das Volumen der Zelle i, $|\Omega_{ij}|$ wie gehabt für die Fläche zwischen Zelle i und j.

Nach einem *expliziten* Sättigungsschritt muss durch Lösen der Kontinuitätsgleichung $\nabla \cdot \vec{v} = q$ die neue Druckverteilung *implizit* berechnet werden. Dazu diskretisiert man die Formulierung der Druckverteilung (2.28) wie folgt:

$$\sum_{j\in N(i)} |\Omega_{ij}|\,\langle \vec{v}_{ij}, \vec{n}_{ij}\rangle = -\sum_{j\in N(i)} |\Omega_{ij}|\, k\lambda(s_{ij})\,\langle \nabla p_i, \vec{n}_{ij}\rangle = q_i, \qquad \forall i \in I \tag{2.36}$$

Die Gleichung (2.36) ist ein lineares Gleichungssystem der Form $\mathbf{A}\vec{x} = \vec{b}$, wobei die Anzahl der Variablen der Anzahl der Zellen des Gitters entspricht. Der gesuchte Lösungsvektor $\vec{x}$ ist die neue Druckverteilung für jede Zelle p_i.

Die Matrix $\mathbf{A}$ ist eine symmetrische $m \times m$ Matrix, wobei $m = |I|$ für die Anzahl der Zellen steht. Diese Matrix spiegelt die *Konnektivität* der Zellen des Gitters wieder. Hat also eine Zelle mit Index i eine gemeinsame Fläche mit einer Zelle j, so besitzt die Matrix an den Stellen $\mathbf{A}_{ij}$ und $\mathbf{A}_{ij}$ Einträge ungleich Null. In unserem kartesischen Gitter hat eine Zelle maximal sechs Nachbarn, diese haben einen räumlich ähnlichen Index. Daher ist $\mathbf{A}$ für große Gitter eine dünnbesetzte Matrix, die hauptsächlich in der Nähe der Diagonalen Einträge ungleich Null besitzt.

Um das Gleichungssystem (2.36) direkt als Matrix zu formulieren, wird ein m-Dimensionaler Vektorraum U angenommen, zu welchem die Vektoren $w_i \in U$, $0 \leq i < m$ für U eine Basis bilden. So lässt sich jeder Zelle mit Index i ein Basisvektor w_i zuordnen. Die Summe über die Nachbarzellen in (2.36) wird über den Gradient gebildet: $\nabla w_i w_j$ bezeichnet die gemeinsame Oberfläche von Zelle i mit Zelle j, normalisiert durch die Distanz der Zellmittelpunkte (h): $\nabla w_i w_j = \frac{|\Omega_{ij}|}{h}$. Dadurch lässt sich (2.36) für ein kartesisches Gitter wie folgendes Gleichungssystem formulieren:

$$-\begin{pmatrix} k\lambda\nabla w_1 w_1 & k\lambda\nabla w_1 w_2 & \cdots & k\lambda\nabla w_1 w_m \\ k\lambda\nabla w_2 w_1 & k\lambda\nabla w_2 w_2 & \cdots & k\lambda\nabla w_2 w_m \\ \vdots & \vdots & \ddots & \vdots \\ k\lambda\nabla w_m w_1 & k\lambda\nabla w_m w_2 & \cdots & k\lambda\nabla w_m w_m \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_m \end{pmatrix} = \begin{pmatrix} q_1 \\ q_2 \\ \vdots \\ q_m \end{pmatrix} \quad (2.37)$$

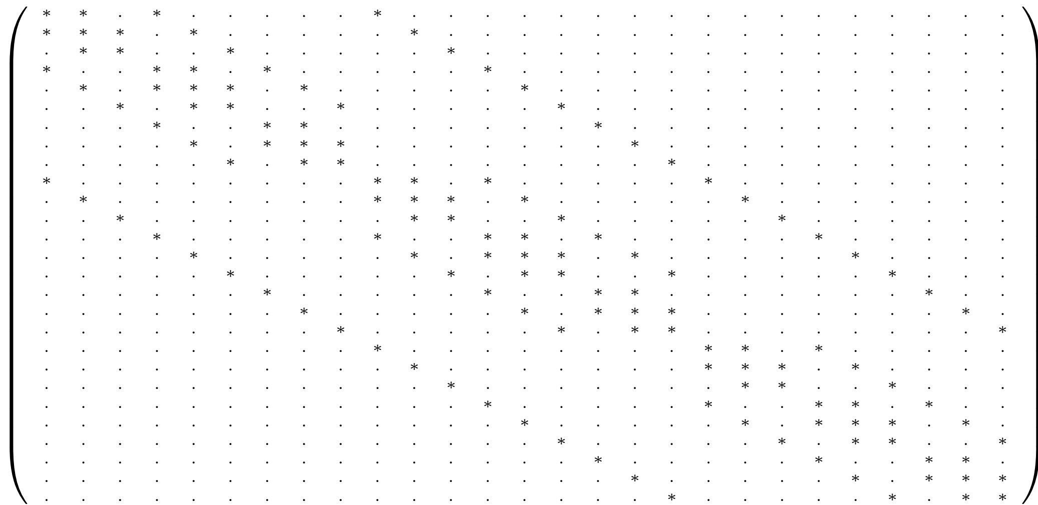


Abbildung 8: Dünnbesetze Konnektivitätsmatrix eines 3x3x3 Berechnungsitters.

Abbildung 8 zeigt, an welchen Stellen die dünnebsetze Matrix aus Gleichung (2.37) für ein 3x3x3-Berechnungsgitter Einträge ungleich Null besitzt. Wie zu erwarten ist, sammeln sich Einträge in der Nähe der Diagonalen, da Zellnachbarn in einem kartesischen Raumgitter ähnliche Indices besitzen. Die Unregelmässigkeiten im Muster der Einträge liegt im Rand des 3D-Gitters begründet, da hier Zellen nicht an allen Oberflächen an Nachbazellen angrenzen.

2.5.1 Upwind Approximation

Wie bereits erwähnt, existiert das Problem, dass an den Oberflächen zwischen den FV-Zellen Berechnungswerte beziehungsweise deren Gradienten definiert sein müssen. Da wir in einem *block-centered grid* arbeiten, werden Werte nur für den Mittelpunkt der Zelle berechnet und gelten als konstant im Zellvolumen. An der Grenze

zwischen den Zellen tritt deshalb eine Unstetigkeit der Werteverteilung auf. Eine einfache Lösung des Problems bestünde darin, an der Zellgrenze linear zwischen den Zell-Mittelpunkten zu interpolieren. Es ist jedoch allgemein bekannt, dass ein solcher Ansatz bei einem expliziten Simulationsverfahren schnell zu numerischen Instabilitäten führen kann und dann keine sinnvollen Ergebnisse mehr produziert. Numerisch stabil bedeutet, dass kleine Änderungen der Startparameter ebenfalls nur kleine Änderungen im Berechnungsergebnis hervorrufen.

Ein besserer Ansatz für dieses Problem stellt die sogenannte *upwind-approximation* dar. Sie ist eine spezielle Methode zur numerischen Diskretisierung partieller Differenzialgleichungen und eignet sich insbesondere bei *expliziten* Finite-Volumen-Verfahren, um diese numerisch *stabil* zu halten.

Die Idee ist dabei, den Gradienten zwischen Zellen als Massenfluss zu interpretieren. Dazu betrachtet man zwischen den benachbarten Zellen, in welche Richtung die Masse an der Grenzfläche fließt. Die Zelle, aus welcher der Massenfluss kommt, wird als Referenz für den neu zu berechnenden Wert verwendet.

Dazu kann einfach die Richtung des Geschwindigkeitsvektors an der Zelloberfläche herangezogen werden. In unserem Modell wird dazu jedoch das *Druckpotential* verwendet - was mathematisch im Modell für diese Arbeit gleichbedeutend ist. Das Druckpotential ist dabei einfach die Druckdifferenz zwischen den beiden benachbarten Zellen: $p_i - p_j$. Das Vorzeichen des Druckpotentials gibt die Flussrichtung der Masse an der Oberfläche $|\Omega_{ij}|$ an. Die Zelle, aus welcher der Massenfluss über $|\Omega_{ij}|$ strömt, liefert den Referenzwert. In unserem Fall ist das die Sättigung an der Grenzfläche s_{ij}^n, welche je nach Vorzeichen von $p_i - p_j$ über s_i^n oder s_j^n definiert wird. Aus Formel (2.35) und $\vec{v} = -k\lambda\nabla p$ folgt mit *upwind-approximation*:

$$s_i^{n+\Delta t} = s_i^n + \frac{\Delta t}{\phi}\left(q_i + \frac{1}{|\Omega_i|}\sum_{j\in N(i)}\frac{|\Omega_{ij}|}{h_{ij}}\left[k\lambda(s_i^n)\max(p_i - p_j, 0) + k\lambda(s_j^n)\min(p_i - p_j, 0)\right]\right). \tag{2.38}$$

In (2.38) wird jedoch nicht direkt die Sättigung s_{ij}^n, sondern die von der Sättigung abhängige Mobilität $\lambda(s_{ij}^n)$ einer upwind-approximation unterzogen. Mit h_{ij} wird der Abstand der Mittelpunkte der benachbarten Zellen i und j bezeichnet.

In Abbildung 9 wird eine Zelle mit Druckpotential $p_i - p_j$ zu einer Nachbarzelle schematisch dargestellt.

Analog wird auch bei der Gleichung für die Druckverteilung (2.36) *upwind-approximation* angewendet:

$$-\sum_{j\in N(i)}\frac{|\Omega_{ij}|}{h_{ij}}\left[k\lambda(s_i^n)\max(p_i - p_j, 0) + k\lambda(s_j^n)\min(p_i - p_j, 0)\right] = q_i \tag{2.39}$$

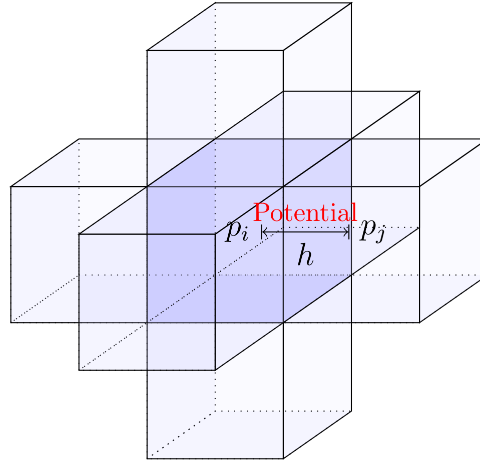


Abbildung 9: Zelle mit Nachbarzellen im kartesischen Gitter.

2.5.2 Randbedingungen

Bei *Finite-Volumen*-Methoden gibt es immer sogenannte *Ränder*, also Zellen, welche an einer oder mehr Oberflächen keine Nachbarzellen besitzen. Da wir insbesondere Differenzen beziehungsweise einen Fluss zwischen den Zellen berechnen, müssen hier Regeln aufgestellt werden, wie zu verfahren ist, wenn es keine Nachbarzellen gibt. Wie bereits beschrieben, gibt es hierzu zwei verschiedene Verfahren, nämlich die *Dirichlet-* oder die *Neumann-Randbedingung*. Bedingungen am Rand unseres Gitters entsprechen Rand- oder Startwertproblem gewöhnlicher Differentialgleichungen.

Die *Dirichlet-Randbedingung* besagt, dass am Berechnungsrand ein vorgegebener Wert oder eine Funktion gelten soll. Für die nicht vorhandene Nachbarzelle wird also durch eine Dirichlet-Funktion ein Wert vorgegeben, der anstatt des Wertes einer Nachbarzelle angenommen wird. Will man beispielsweise einen Druck-Gradienten einer Rand-Oberfläche in einer Zelle bestimmen, so berechnet man die Differenz zwischen dem Druckwert in der Zelle und dem Druckwert, der durch die Dirichlet-Bedingung für die Rand-Oberfläche definiert ist:

$$\langle \nabla p_i, \vec{n} \rangle = \frac{p_i - p_{dirichlet}}{h/2}. \tag{2.40}$$

Da bei einer Zelloberfläche am Rand keine Nachbarzelle vorhanden ist, kann nicht der Abstand zwischen zwei Zellmittelpunkten verwendet werden. Es wird statt dessen der Abstand vom Zellmittelpunkt zur Oberfläche am Rand ($\frac{h}{2}$) verwendet.

Die *Neumann-Randbedingung* gibt im Gegensatz zur Dirichlet-Randbedingung einen konkreten Wert für die Ableitung beziehungsweise den Gradient an:

$$\langle \nabla p_i, \vec{n} \rangle = v_{neumann}. \tag{2.41}$$

Für die Berechnung des expliziten Transportprozess-Schrittes mit Gleichung (2.35) hat das zur Folge, dass für Zellen am Rand neben den Summanden aus den direkten Nachbarzellen auch Terme für Dirichlet und Neumann-Randbedingung mit einbezogen werden müssen. Dabei wird im Wesentlichen der Term "$\langle \nabla p_i, \vec{n}_{ij} \rangle$" für den Dirichlet-Fall durch (2.40) beziehungsweise beim Neumann-Fall durch (2.41) ersetzt.

Analog wird beim Aufstellen des Druck-Gleichungssystems (2.36) verfahren.

Für die *upwind-approximation* wird anstatt des Druck-Potentials $p_i - p_j$ für die Dirichlet-Bedingung $p_i - p_{dirichlet}$ beziehungsweise im Neumann-Fall $v_{neumann}$ zur Bestimmung der Flussrichtung herangezogen.

2.5.3 CFL Bedingung

Für die zeitliche Diskretisierung ist zu beachten, dass die Zeitschrittgröße Δt sorgfältig ausgewählt wird. Ist diese zu groß, so läuft die Simulation Gefahr instabil zu werden und unbrauchbare Ergebnisse zu produzieren. Ist Δt klein, so ist die Simulation nur wenig performant und benötigt viel Berechnungszeit für wenig Simulationsfortschritt. Daher sollte Δt so groß wie möglich gewählt werden, damit noch eine stabile Simulation möglich ist.

Um ein maximal großes aber dennoch stabiles Δt zu wählen, betrachten wir die Stabilität in einer einzelnen Zelle. Die Berechnung wird dann instabil, wenn eine Zelle andere Zellen als ihre direkten Nachbarzellen beeinflusst, da in den Gleichungen nur die direkten Nachbarzellen einbezogen werden. Dieser Fall liegt dann vor, wenn die Masse einer Zelle weiter transportiert wird als der Abstand zwischen den Zellen (h). Das macht sich die sogenannte CFL[8]-Bedingung für die maximale Zeitschrittgröße zu eigen:

$$\frac{\|\vec{v}\| \, \Delta t}{h} < C \tag{2.42}$$

Der Massenzufluss wird in (2.42) durch die Norm der Geschwindigkeit ($\|\vec{v}\|$) repräsentiert. In Kombination mit Δt bildet der Massenzufluss eine Längenangabe, welche in ein Verhältnis zum Zellabstand gebracht wird. C bildet hier ein Stabilitätsmaß - rein mathematisch gesehen gilt $C = 1$. Um jedoch in einer realen Simulation auf der sicheren Seite zu sein, werden Werte kleiner 1 verwendet. Übliche Werte für diesen *CFL-Faktor* liegen im Bereich: $0.9 < C < 1$. Die *CFL-Bedingungen* garantiert, dass Δt klein genug ist, um mit einem expliziten Zeitschritt nur die direkten Nachbarn einer Zelle zu beeinflussen.

Um den Zeitschritt für ein ganzes Gitter zu bestimmen, muss das kleinste Δt aller Zellen mit der CFL-Bedingung bestimmt werden - so kann die Instabilität des Verfahrens für das gesamte Gitter verhindert werden.

[8] *C*ourant–*F*riedrichs–*L*ewy

Das grundsätzliche Problem bei der CFL-Bedingung besteht darin, dass aus kleineren Zellgrößen (h) auch kleinere Zeitschritte (Δt) resultieren. Möchte man also die räumliche Genauigkeit erhöhen, ist man gezwungen, die zeitliche Genauigkeit ebenfalls zu erhöhen.

3 Parallelisierung

Der grundsätzliche Gedanke hinter der Parallelisierung ist die Aufteilung eines Problems in Teilprobleme, welche parallel abgearbeitet und zum Schluss zu einer Gesamtlösung zusammengesetzt werden. Da insbesondere die serielle Abarbeitung von Befehlen in der modernen Prozessortechnologie langsam aber sicher an ihre Grenzen stößt, gewinnt das Thema Parallelisierung immer mehr an Bedeutung. Es lassen sich jedoch nicht alle Aufgaben beliebig gut parallelisieren - insbesondere dann, wenn Abhängigkeiten zwischen den Teilproblemen bestehen. Ein Beispiel für gute Parallelisierbarkeit sind viele Algorithmen zur Bildbearbeitung. Hier kann oft jeder Bildpunkt separat berechnet werden, ohne die Ergebnisse der anderen laufenden Berechnungen zu benötigen.

Es sind verschiedene Klassen von Parallelisierungen zu differenzieren. Das sind zum einen einfache Mehrprozessor-Systeme, bei denen bestehende Software mit relativ überschaubarem Änderungsaufwand beschleunigt werden kann. Solche Systeme werden auch *Shared-Memory-Systeme* genannt, da alle Prozessoren auf einen gemeinsamen Hauptspeicher zugreifen. Komplizierter wird die Sache bei sogenannte *Clustern* mit verteiltem Hauptspeicher. Solche auf Parallelität ausgelegte Systeme erfordern es, Software speziell für die Parallelisierung teilweise neu zu programmieren - oft auf Basis von populären Bibliotheken wie beispielsweise MPI[9]. Dies führt bei Berechnungen mit guter Parallelisierbarkeit im Vergleich zu einfachen *Shared-Memory-Systemen* zu starken Performance-Zuwächsen.

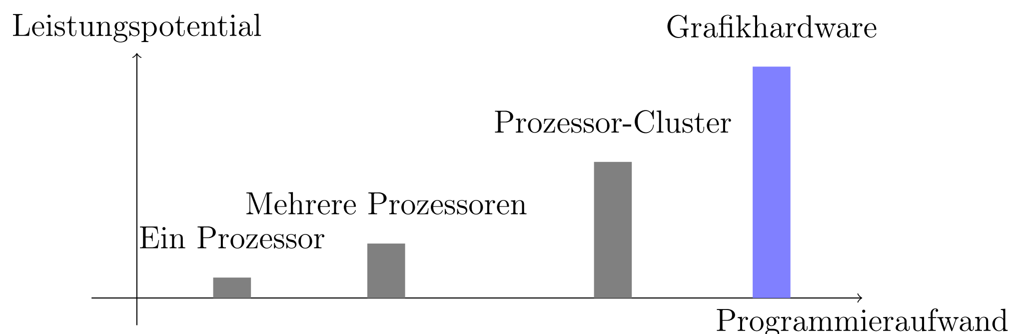


Abbildung 10: Parallele Architekturen im Vergleich.

Die Portierung von Software auf extrem parallele Architekturen beziehungsweise *Many-Core-Architekturen* wie etwa moderne Grafikhardware, bietet das größte Potential an Geschwindkeitszuwächsen. Dies ist jedoch auch mit dem größten Aufwand an die Portierung beziehungsweise das Design der Software verbunden. Hier muss oft auf die speziellen Gegebenheiten der Hardware Rücksicht genommen und beispielsweise Speicherzugriffe, Kontrollfluss, Daten-Abhängigkeiten etc. nach spezifischen Regeln gestaltet werden, um das volle Leistungspotential nutzen zu können. Der

[9] *M*essage *P*assing *I*nterface.

Nachteil liegt hier auch in der Bindung an spezielle Architekturen. Nvidia stellt eine eigene Plattform für die Programmierung ihrer Hardware zur Verfügung - es gibt aber auch allgemeinere Ansätze, die nicht herstellergebunden sind wie beispielsweise OpenCL[10]. Derzeit ist jedoch insbesondere im wissenschaftlichen Bereich die CUDA-Architektur von Nvidia sehr verbreitet und wird die Grundlage für diese Arbeit sein. CUDA und OpenCL sind sich in vielen Aspekten ähnlich und ein Wissenstransfer ist durchaus möglich.

3.1 Die CUDA-Plattform

Die Compute Unified Device Architecture, kurz *CUDA* genannt, ist die Plattform, welche Nvidia für die Programmierung ihre Grafikhardware zur Verfügung stellt. Es handelt sich hierbei um spezielle Treiber sowie um eine Sprache mit Compiler und Laufzeitumgebung. Damit lassen sich auf modernen Nvidia-Grafikkarten allgemeine, nicht auf Grafik spezialisierte, Berechnungen durchführen. Die Programmiersprache CUDA ist ein teilweise auf C++ aufbauendes Konstrukt mit speziellen Schlüsselwörtern, Befehlen und Konstanten. In CUDA wird grundsätzlich zwischen Programmteilen mit zugehörigem Speicher auf der Grafikkarte/GPU[11] (*device*) und Programmcode mit Speicher auf CPU-Seite (*host*) unterschieden.

Das Programmier-Paradigma basiert darauf, rechenintensive und gut parallelisierbare Teile eines Programms auf die Grafikhardware auszulagern. Ein Programmteil, der auf der Grafikhardware beziehungsweise dem *device* ausführbar ist, wird *Kernel* genannt. Die CPU/*host*-Seite steuert dabei den Ablauf des Programms. Die *Kernel*-Prozeduren auf dem *device* werden bis zu Synchronisationspunkten grundsätzlich parallel zum *host*-Programm ausgeführt. Dieses Programmier-Paradigma wird auch *accelerator programming model* genannt. Als Synchronisations-Zeitpunkt zwischen *device* und *host* fungiert in CUDA Programmen typischerweise das Kopieren von Daten zwischen *device* und *host*. Es ist aber auch explizit durch eine CUDA-Funktion möglich.

Ein *Kernel* besteht üblicherweise aus einem kleinen Stück Programmcode, der massiv parallel ausgeführt wird, wobei jeder Kernel für sich auf Basis seiner Identifikationsnummer einen kleinen Teil zur Gesamtberechnung beiträgt. Die Ausführung eines Kernels mit eigener Identifikationsnummer und eigenem Speicher/Kontrollfluss wird auch *Kernel-Instanz* oder *Thread* genannt.

Aus der Sicht des Programmierers können mehrere tausend Instanzen eines Kernels gleichzeitig ablaufen. Wie viele davon tatsächlich gleichzeitig ausgeführt werden, ist stark Hardwareabhängig. Damit entsprechende Programme portabel und effizient auf verschiedenen Hardware-Architekturen ausführbar sind, werden die einzelnen Kernel-Instanzen in sogenannte *Blöcke* zusammengefasst. Alle Blöcke bilden ihrerseits eine Einheit als sogenanntes *Grid*. Die Hierarchie wird in Abbildung 11 dar-

[10] *Open* *C*omputing *L*anguage.

[11] *G*raphics *P*rocessing *U*nit, das Herzstück moderner Grafikkarten.

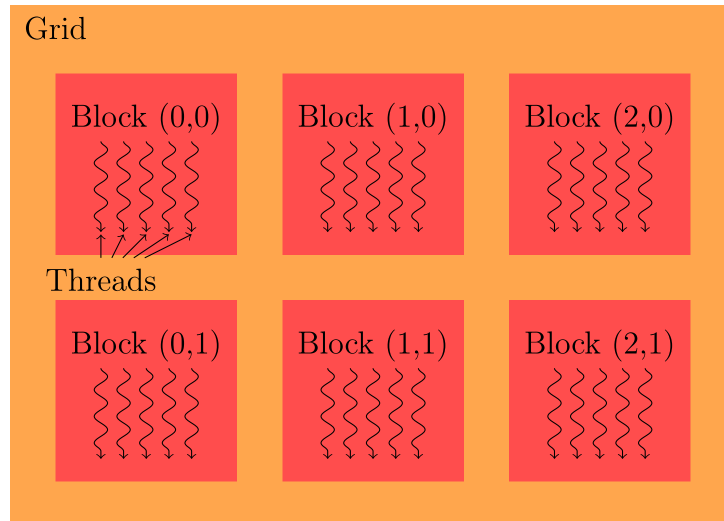


Abbildung 11: Die CUDA-Architektur.

gestellt. Je nach Architektur kann die verwendete Hardware entscheiden, wie viele Blöcke tatsächlich gleichzeitig ausgeführt werden sollen. Ein CUDA-Programm kann auf diese Weise adaptiv von zukünftiger leistungsfähigerer Hardware ohne Redesign/Neukompilierung profitieren oder mit Geschwindigkeitseinbußen auf schwächerer Hardware lauffähig bleiben.

Sowohl Blöcke innerhalb eines Grids als auch Threads innerhalb eines Blocks bekommen eindeutige Identifikationsnummern beziehungsweise einen Index (`blockIdx` und `threadIdx`) um zu entscheiden, welchen Teil der Aufgabe sie abarbeiten sollen. Dabei können Blöcke innerhalb eines Grids sowohl eindimensional als auch zweidimensional organisiert sein, wie in Abbildung 11 grafisch angedeutet. Threads können sogar dreidimensional organisiert sein. Daher sind die Index-Variablen `blockIdx`, `threadIdx` sowie die Variablen für die Block- und Gittergrößen `blockDim`, `gridDim` vom CUDA-Datentyp `uint3` und haben die Komponenten `x`, `y` und `z`. Im eindimensionalen Fall sieht eine Index-Zusammensetzung in einem Kernel wie folgt aus:

```
1 __global__ myKernel(const float *input, float *result, float factor
    ) {
2   int index = blockDim.x*blockIdx.x + threadIdx.x;
3
4   result[index] = input[index] * factor;
5 }
```

Listing 1: Grundstruktur eines CUDA-Kernels.

Das Schlüsselwort `__global__` zeigt dem CUDA-Compiler an, dass diese Funktion von der *host*-Seite aufgerufen werden kann und auf dem *device* ausgeführt wird. Für den Index wird zuerst der Block-Offset berechnet und anschließend der Thread-Index innerhalb des Blocks addiert. Sind Blöcke oder Threads zwei- beziehungsweise dreidimensional organisiert, muss die Index-Logik mit `y` und `z` Komponenten ent-

```
1  int main() {
2      float *deviceMemory, *hostMemory;
3      int numBytes = sizeof(float)*100;
4      hostMemory = (float*)malloc(numBytes)
5
6      // 'hostMemory' mit Daten belegen ..
7
8      /*device-Speicher allokieren*/
9      cudaMalloc(&deviceMemory, numBytes);
10     /* Daten in device-Speicher kopieren*/
11     cudaMemcpy(deviceMemory, hostMemory, numBytes,
           cudaMemcpyHostToDevice);
12     /* Kernel aufrufen mit 5 Blöcken und je 20 Threads */
13     myKernel<<<5,20>>>(deviceMemory, deviceMemory, 2.0f);
14     /* Ergebnis zurück in host-Speicher kopieren */
15     cudaMemcpy(hostMemory, deviceMemory, numBytes,
           cudaMemcpyDeviceToHost);
16
17     // mit 'hostMemory' weiterarbeiten
18
19     return 0;
20 }
```

Listing 2: Speichervorbereitung und Aufruf eines Kernels.

sprechend erweitert werden. Für den Programmierer ist es durchaus möglich, aus eindimensional organisierten Indizes weitere Dimensionen abzuleiten, jedoch würde dies zu größerer Index-Berechnungs-Logik am Anfang des Kernels führen. Da CUDA bisher auf Block-Ebene (im Gegensatz zur Thread-Ebene) leider keine dreidimensionale Organisierung zulässt, muss für die Berechnung von dreidimensionalen Gittern ein besonderes Augenmerk auf die Indexlogik gelegt werden.

Wie in Listing 1 zu sehen, nimmt der *Kernel* Zeiger auf Arrays als Parameter entgegen. In CUDA kann jedoch ein Kernel nicht einfach auf den Hauptspeicher der CPU (*host*) zugreifen. Der Programmierer muss hier Speicher auf dem *device* allokieren und für den Kernel notwendige Daten vom *host* auf das *device* kopieren. Wenn der Kernel ausgeführt wurde, müssen die Ergebnisdaten vom *device* zurück auf Speicher im *host* kopiert werden, um diese dann weiterverwenden zu können.

Im Beispiel von Listing 2 wird Speicher auf *host*- und *device*-Seite angelegt und die notwendigen Daten werden kopiert. Anschließend folgt der Kernel-Aufruf von `myKernel` und zwar mit 5 Blöcken, die jeweils 20 Threads beinhalten. Die Variable `index` in Listing 1 würde in diesem Beispiel je nach Kernel-Instanz (Thread) die Werte von 0 bis 99 annehmen und alle Elemente des Arrays `deviceMemory` in Listing 2 bearbeiten.

Ein CUDA-Kernel kann neben Zeigern auf Speicher auch direkte Call-By-Value Parameter entgegen nehmen wie beispielsweise das letzte Argument des Kernel in Listing 1: `myKernel(..., float factor)`. Hier muss der Programmierer nicht für

den Datentransfer zwischen GPU und CPU sorgen, da CUDA dies übernimmt.

Neben der vorgestellten Organisation von Threads in Blöcken organisiert die Grafikhardware die einzelnen Threads in einem sogenannten *Warp*. Ein *Warp* ist die Menge an Threads, welche gleichzeitig auf einer einzelnen Funktionseinheit der GPU ausgeführt werden. Eine solche Funktionseinheit, auch *Multiprozessor* genannt, führt je nach Hardware typischerweise einen Warp mit 32 Threads aus.

Für den Programmierer ist dies deshalb relevant, da ein *Multiprozessor* nur über eine Einheit zum Steuern des Kontrollflusses und zum Zugriff auf den Speicher besitzt. Wenn einzelne Threads innerhalb eines Warps bedingt durch Kontrollfluss-Anweisungen wie `if`, `for`, `while` etc. unterschiedliche Programmabläufe nehmen, so wirkt sich dies nachteilig auf die Ausführungsgeschwindigkeit aus. Die unterschiedlichen Pfade des Programmablaufs werden in einem Warp seriell ausgeführt.

Um Kontrollfluss-Anweisungen zu minimieren, kann man beispielsweise unterschiedliche Kernel für die jeweiligen Kontrollflusspfade schreiben. Kleinere if-else Konstrukte lassen sich jedoch oftmals vereinfachen. Am Beispiel der `min()`-Funktion lässt sich die if-else Anweisung wie folgt umgehen:

```
1  int min(int a, int b) { // standard Implementierung
2      if(a > b) return b;
3      else return a;
4  }
5
6  // Version ohne Kontrollfluss-Divergenz
7  __device__ int min(int a, int b) {
8      int greater = a > b;
9      return (1-greater)*a + greater*b;
10 }
```

Das Schlüsselwort `__device__` bedeutet, dass eine Funktion auf dem *device* ausführbar ist und somit von einem Kernel aufgerufen werden kann.

Auch für die Größe von Blöcken ist die Warp-Größe relevant. Die Anzahl der Threads pro Block sollte idealerweise ein Vielfaches der Warp-Größe betragen. Maximal sind auf aktueller CUDA-Hardware 512 Threads pro Block möglich.

3.2 Speicher und Datentypen in CUDA

In CUDA werden mehrere verschiedene Speicherarten unterschieden, welche sich in Zugriffseigenschaften und Latenz unterscheiden:

Global memory: Großer Hauptspeicher der Grafikkarte mit viel Bandbreite aber auch hoher Latenz. Das ist der Standard-Speicher, welcher beispielsweise mit `cudaMalloc()` allokiert wird. Wird auch *device memory* genannt.

GPU registers: Schnelle Datenspeicher innerhalb eines *Multiprozessors*. Diese werden unter anderem für lokale Variablen in Threads verwendet.

texture memory: Spezielle Form von Speicher, welcher physikalisch im globalen Speicher lokalisiert ist. Der Textur-Speicher zeichnet sich insbesondere durch schnelle Lesezugriffe mit Hilfe eines eigenen Cache aus. Dieser Cache beschleunigt insbesondere Zugriffe, welche in der Textur-Dimension (1D, 2D oder 3D) in räumlicher Nähe zueinander erfolgen.

constant memory: Kleiner schneller Speicher, welcher nur gelesen werden kann. In CUDA wird dieser Speicher durch das Schlüsselwort `__constant__` in Zusammenhang mit Variablen zugewiesen. Diese Speicherart häufig für Konstanten verwendet, welche sich nur selten ändern.

shared memory: Schneller Speicher innerhalb eines *Multiprozessors*, auf welchen innerhalb eines *Blocks* zugegriffen werden kann. Dieser Speicher ist so schnell wie *GPU register* und kann zum schnellen Datenaustausch innerhalb eines Blocks verwendet werden. Um *shared memory* zu verwenden wird innerhalb eines Kernels eine lokale Variable/Array mit dem Schlüsselwort `__shared__` deklariert:

```
1 __global__ void myKernel() {
2     __shared__ float data[size]; // shared memory
3     ...
4 }
```

Die verschiedenen Speicherarten mit den wechselseitigen Abhängigkeiten werden in Abbildung 12 dargestellt. Hier wird außerdem der Zusammenhang zwischen Speicherhierarchie und den Multiprozessoren eines CUDA-*device* deutlich.

Für Speicherzugriffe in CUDA gelten ähnliche Regeln wie für den Kontrollfluss: sind die Zugriffe innerhalb eines *Warps* weit gestreut, so kostet es deutlich mehr Performance als wenn alle Threads auf einen zusammenhängenden Speicherbereich zugreifen. Wird in einem Warp ein zusammenhängender Speicherbereich von den enthaltenen Threads abgefragt, so kann dieser mit einer kombinierten Speicher-Transaktion (*coalesced access*) gelesen werden.

Die *Multiprozessoren* der CUDA-Hardware beherrschen eine Technik, um die Latenzzeiten des globalen Speichers zu maskieren (*latency hiding*). Dazu kann ein Multiprozessor mehrere Warps in einem Multitaskingverfahren ausführen. Sobald ein Warp auf einen Speicherzugriff wartet, schaltet der Multiprozessor zu einer anderen Warp-Ausführung um.

Neben den üblichen C-Standard Datentypen wie `short`, `int`, `unsigned int` beziehungsweise `uint` beherrscht CUDA-Hardware auch die Fließkommatypen `float` und in neueren Generationen auch `double`. Da die Hardware vorwiegend auf einfache Fließkomma-Genauigkeit ausgelegt ist, ist die Performance für `double` Operationen meist deutlich schlechter. Auf älterer "Tesla" Hardware und auf Consumer "Fermi"-Grafikkarten (GTX 4xx) ist die Addition und Multiplikation von `double` Werten um

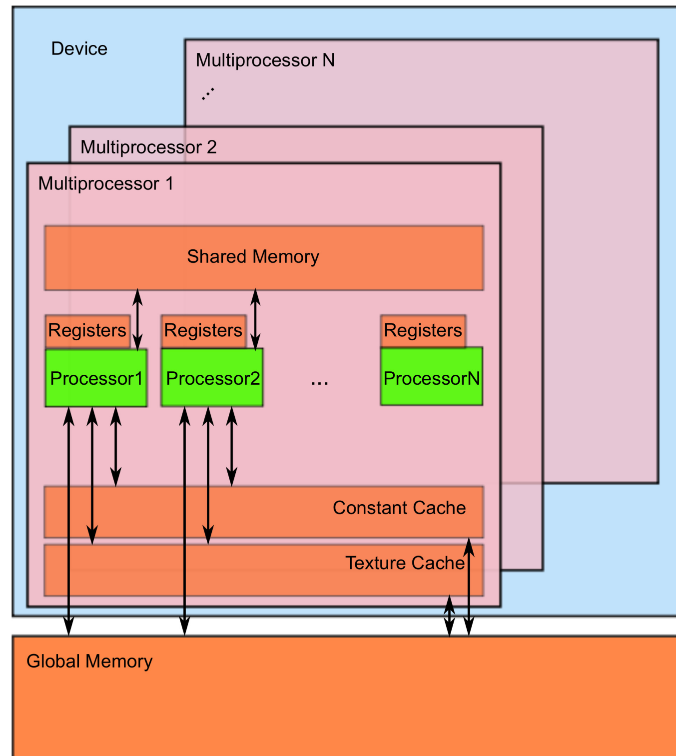


Abbildung 12: Speicher-Hierarchie in CUDA.

Faktor 8 langsamer als für `float`. Auf aktuellen HPC Fermi-Grafikkarten beträgt die Differenz jedoch nur noch Faktor 2.

Zusätzlich beherrscht CUDA auch zusammengesetzte Datentypen wie beispielsweise `uint2`, `float3`, `double4` etc. inklusive der zugehörigen arithmetischen Operationen. Die Zahl hinter im Namen steht dabei für die Dimension. Diese Datentypen erleichtern den Umgang mit Vektoren, Koordinaten und Farbangaben.

3.3 Sprachkonstrukte in CUDA

In CUDA werden Kernel-Prozeduren, die auf dem *device* ausgeführt und von *host*-Seite aus aufgerufen werden, mit dem bereits vorgestellten Schlüsselwort `__global__` deklariert. Weiter lassen sich mit dem Schlüsselwort `__device__` Subroutinen deklarieren, welche ebenfalls auf dem *device* ausgeführt werden und von einem Kernel aus aufrufbar sind. Diese Subroutinen sind jedoch nicht von *host*-Seite aufrufbar. Soll eine Funktion jedoch explizit auf *host*-Seite ausführbar sein, so kann das Schlüsselwort `__host__` verwendet werden. Um Code-Verdopplungen zu vermeiden, ist auch eine Kombination von `__host__` und `__device__` möglich. Technisch gesehen wird dann für eine entsprechende Routine sowohl für die *device*-Seite als auch für die *host*-Seite ausführbarer Code erzeugt. Für Speicherzugriffe auf *device*-Seite ist darauf zu achten, dass nicht auf *host*-seitigen Speicher zugegriffen werden kann.

CUDA unterstützt auch teilweise C++ Klassen. Dazu können Methoden in einer Klasse mit `__device__` und/oder mit `__host__` deklariert werden, so dass Teile der Funktionalität einer Klasse auch von einem CUDA-Kernel verwendet werden können.

Auf diese Art werden beispielweise die Materialparameter für das Simulationsprogramm verwaltet. Um nicht Code separat für *host*- und *device*-Seite zu schreiben, kann beispielsweise für die Berechnung von k_{rw} aus Formel (2.16) wie folgt vorgegangen werden:

```
1 class Cuda2pImpesParameter {
2 public:
3   ...
4   static inline __device__ __host__ scalar_ relPermeabilityWetting(
        scalar_ S) {
5     scalar_ S_ = effSaturationWetting(S);
6     return pow(S_*S_, 2.0/alpha + 1);
7   }
8   ...
9 }
```

Polymorphe Klassen mit virtuellen Methoden sowie Rekursion sind derzeit jedoch nicht in CUDA möglich. Template-Parameter, das beliebte Konstrukt aus C++ für variable Datentypen, sind hingegen für Prozeduren auf *device*-Seite möglich.

4 Implementierung

Für die Implementierung der Zweiphasen-Strömung als Computersimulation wird, wie im Abschnitt 2.4.1 beschrieben, ein diskretisierter IMPES-Ansatz mit den schwach gekoppelten Variablen für Sättigung S_w und Druck p_n verwendet. Schematisch ist das Programm wie im Algorithmus 2 aufgebaut. Die Variablen S_w, p_n, rhs, $\frac{dS_w}{dt}$ und $dtVec$ sind dabei Felder, die für jede Zelle des Berechnungsgitters jeweils ein Element reservieren.

Algorithmus 2 Der Simulationsablauf.

procedure IMPES-SIMULATION(t_{end})
- $S_w \leftarrow initialSaturation$ ▷ Feld für die Sättigung jeder Zelle mit Anfangswert
- $(\mathbf{A}, rhs) \leftarrow$ ASSEMBLEPRESSEQSYSTEM(S_w, p_n) ▷ Gl.System $A\ p_n = rhs$ aufstellen
- $p_n \leftarrow$ SOLVEPRESSEQSYSTEM($\mathbf{A}, rhs, p_n$) ▷ Gleichungssystem nach p_n auflösen
- $t \leftarrow 0$
- **while** $t < t_{end}$ **do**
  - $(\mathbf{A}, rhs) \leftarrow$ ASSEMBLEPRESSEQSYSTEM(S_w, p_n)
  - $p_n \leftarrow$ SOLVEPRESSEQSYSTEM($\mathbf{A}, rhs, p_n$)
  - $\left(\frac{dS_w}{dt}, dtVec\right) \leftarrow$ TRANSPORTSTEP(S_w, p_n) ▷ $\frac{dS_w}{dt}$ Zeitschritte bestimmen
  - $\Delta t \leftarrow \min(dtVec)\ CFLfactor$ ▷ max. stabilen Zeitschritt Δt finden
  - $S_w \leftarrow S_w + \Delta t\ \frac{dS_w}{dt}$ ▷ neues S_w explizit bestimmen
  - $t \leftarrow t + \Delta t$

Die in Algorithmus 2 verwendete Funktion TRANSPORTSTEP(S_w, p_n) implementiert die Berechnung von $\frac{dS_w}{dt}$ aus Gleichung (2.35) auf Seite 17 beziehungsweise mit *upwinding* aus (2.38). Die Funktion ASSEMBLEPRESSEQSYSTEM(S_w, p_n) erstellt die Matrix und die rechte Seite des Divergenz-Gleichungssystems, wie in Gleichung (2.36) beziehungsweise (2.39) beschrieben.

Die Funktion SOLVEPRESSEQSYSTEM($\mathbf{A}, rhs, p_n$) ist ein allgemeiner Lösungsalgorithmus für lineare Gleichungssysteme und löst in unserem Fall die Gleichung $\mathbf{A}p_n = rhs$. Diese Funktion nutzt die Stetigkeit der Druckverteilung über die Zeit aus und nimmt den Wert der Lösung eines früheren Zeitschrittes (p_n) als Start-Parameter entgegen. Die Lösung des vorangegangen Zeitschrittes dient als Näherung der neu zu berechnenden Lösung, was zu einer schnelleren Konvergenz der Berechnung in SOLVEPRESSEQSYSTEM($\mathbf{A}, rhs, p_n$) führt.

Im praktischen Teil dieser Arbeit besteht die Hauptaufgabe darin, diese vorgestellten Funktionen auf die CUDA-Plattform zu portieren. In den folgenden Kapiteln wird das Vorgehen dazu genauer geschildert und mit Code-Auszügen verdeutlicht.

Als Vorlage für die Implementierung lag das DuMu^x[6]-Programm im Quellcode vor. Die DuMu^x-Plattform implementiert die Zweiphasenströmung in porösen Medien sehr allgemein. Hier sind beispielsweise irreguläre Gitter und beliebige Kombinationen der Hauptvariablen Sättigung und Druck, sowie Modelle, in denen Kapillardruck und Gravitation berücksichtigt werden, möglich. In dieser Arbeit wird

das Modell, wie bereits erwähnt, in vielerlei Hinsicht vereinfacht, wodurch das Programm leichter auf die CUDA-Plattform übertragbar ist. Zum Vergleich sind alle CUDA-Programmteile (*device kernel*) auch als CPU-basierte Routinen (*host side*) implementiert worden. Dadurch soll es später möglich werden, den Vorteil der GPU-basierten Implementierung genauer zu analysieren. Dieses Kapitel widmet sich jedoch hauptsächlich der CUDA-seitigen Implementierung.

Ursprünglich war es geplant, für das Lösen des Gleichungssystems die externe Bibliothek OpenNL[12] zu verwenden. Diese Programmbibliothek lässt sich so konfigurieren, dass Sie die Berechnungen mit CUDA auf der Grafikkarte ausführt [2]. Da OpenNL jedoch seine Daten nicht direkt aus dem Speicher der Grafikkarten beziehen kann, wurde in der Arbeit zusätzlich direkt ein CUDA-basierter Löser für Gleichungssysteme für dünnbesetzte Matrizen nach dem *konjugierten Gradienten*-Prinzip implementiert.

4.1 Transport-Berechnung

Die Transportgleichung in der diskretisierten Form als Iterations-Schema in Gleichung (2.38) auf Seite 19 wird mit *upwinding* in Algorithmus 3 umgesetzt. Dabei muss für alle Zelloberflächen der Massenfluss über den Druckgradient zwischen den Zellen bestimmt werden. Dabei wird das bereits erwähnte *upwinding* auf Basis der Druckunterschiede zwischen den benachbarten Zellen eingesetzt.

Algorithmus 3 TransportStep.

```
procedure TRANSPORTSTEP(S, p)                                          ▷
    for all c_i ∈ GridCells do
        (dS_w/dt)_i ← 0
        for all surf ∈ Surfaces(c_i) do
            if surf is not a boundary surface then
                j ← NeighbIndex(surf)
                if (p_i − p_j) ≥ 0 then                    ▷ upwinding mit Druckpotential
                    λ_upwind ← λ(S̄_i)
                else
                    λ_upwind ← λ(S̄_j)
                (dS_w/dt)_i ← (dS_w/dt)_i + |surf| kλ_upwind (pi − pj) / (φ |c_i| CellDistance(i, j))
                ...                                                  ▷ CFL-Bedingung
            else
                ...                             ▷ Randbedingung muss evaluiert werden
    return (dS_w/dt, dtVec)
```

In die Änderungsrate der Sättigung $\frac{dS_w}{dt}$ geht neben dem $k\lambda$-Term auch die aktuell betrachtete Zelloberfläche $|surf|$, das Zellvolumen $|c_i|$, die Porösität ϕ sowie die Distanz zwischen den Nachbarzellen $CellDistance(i, j)$ ein.

[12] *O*pen *N*umerical *L*ibrary, http://alice.loria.fr/index.php/software/4-library/

```
__global__ void SaturationUpdateKernel(scalar_* SwUpdateVec,
    scalar_ *dtValues, scalar_ dt , const scalar_ *Sw, const scalar_
     *pn) {
    uint x, y, z, indexI;
    /* Index-Logik: erstellt aus threadIdx, blockIdy etc die
       Koordinaten (x,y,z) und den indexI */
    indexI = getGridCoordinatesForThread(x, y, z);
    /* bestimmt anhand von (x,y,z) welche Seiten der Zelle an einem
        Rand des Gitters liegen */
    BorderType cellBorder = getBorderType(x, y, z);

    scalar_ SwI = Sw[indexI]; /*Saettigung der aktuellen Zelle*/
    scalar_ pnI = pn[indexI]; /*Druck der aktuellen Zelle*/
    scalar_ lambdaI = Cuda2pImpesParameter::lambdaWetting(SwI);

    SwUpdateVec[indexI] = 0;
    scalar_ timestepFactorIn = 0; scalar_ timestepFactorOut = 0;

    /* Iteration ueber die 6 Seiten der akuellen Zelle */
    for(int neighbIndex = 0; neighbIndex < 6; neighbIndex++) {
        /* Variablen fuer Rand, Seitenflaeche und Zellabstand */
        BorderType faceBorder = (cellBorder & (1<<neighbIndex));
        scalar_ faceArea = cellSurfaceArea[neighbIndex>>1];
        scalar_ centerDist = cellMetrics[neighbIndex>>1];

        if (!faceBorder) { /* Normalfall: Zelle hat Nachbarn */
            const int3& nOffs = cellNeighbOffset[neighbIndex];
            uint indexJ = getGridIndexForCoordinates(x+nOffs.x, y+
                nOffs.y, z+nOffs.z);
            scalar_ SwJ = Sw[indexJ]; /*Saettigung Nachbarzelle*/
            scalar_ pnJ = pn[indexJ]; /*Druck Nachbarzelle*/
            scalar_ lambdaJ = Cuda2pImpesParameter::lambdaWetting(
                SwJ);
            scalar_ potential = pnI - pnJ;
            uint upwindGreaterZero = potential > 0;
            scalar_ upwindLambda = upwindGreaterZero*lambdaI  +
                (1-upwindGreaterZero)*lambdaJ;

            factor = Cuda2pImpesParameter::IntrinsicPermeability*
                potential*upwindLambda*faceArea/(centerDist*
                cellVolume*Cuda2pImpesParameter::Porosity);
            timestepFactorIn += ...;
            timestepFactorOut -= ...;
        } else {
            /* Randbedingung muss evaluiert werden */
        }
        SwUpdateVec[indexI] -= factor;
    }
    dtValues[indexI] = evaluateTimeStepTotalFlux(timestepFactorIn,
       timestepFactorOut);
}
```

Listing 3: SaturationUpdateKernel.

Der Vektor für die Bestimmung des Zeitschrittes mit CFL-Bedingung ($dtVec$) wird ähnlich wie $\frac{dS_w}{dt}$ berechnet - jedoch wird hier separat der Ein- und Ausstrom von Sättigung über dt_{in} und dt_{out} verwaltet. In Algorithmus 4 wird der fehlende Teil von Algorithmus 3 ergänzt. In die Berechnung von $dt_{in/out}$ gehen außerdem die relative Permeabilität $k_{rw/rn}$ und das Viskositätsverhältnis zwischen den Phasen μ_{ratio} mit ein.

Algorithmus 4 TransportStep – CFL-Bedingung.

$dt_{in} \leftarrow 0$, $dt_{out} \leftarrow 0$
for all $surf \in Surfaces(c_i)$ **do**
  ...
  $x \leftarrow |surf|\, k\lambda_{upwind}(p_i - p_j)/\,(\phi\,|c_i|\, CellDistance(i,j))$
  if $(p_i - p_j) \geq 0$ **then**
    $dt_{out} \leftarrow dt_{out} + x/((k_{rw} + k_{rn})\mu_{ratio})$
  else
    $dt_{in} \leftarrow dt_{in} - x/((k_{rw} + k_{rn})\mu_{ratio})$
  ...
$dtVec_i = \min(\frac{1-S_{wr}-S_{nr}}{dt_{in}}, \frac{1-S_{wr}-S_{nr}}{dt_{out}})$

Für den Fall, dass ein Teil der Oberfläche einer Zelle zum Rand des Berechnungsgitters gehört, muss der Massentransport entsprechend der definierten Randbedingungen gesondert bestimmt werden. Dieser in Algorithmus 3 ausgesparte Teil wird in Algorithmus 5 näher beschrieben. Es wird dabei zwischen den beiden bereits erwähnten Fällen der *Dirichlet-* und der *Neumann-Bedingung* unterschieden. Für den Dirichlet-Fall wird anstatt des Drucks einer Nachbarzelle der definierte Dirichlet-Druck für die Bestimmung des Druckpotentials herangezogen. Falls für die Oberfläche die Neumann-Bedingung definiert wurde, wird der Neumann-Fluss selbst als Gradient ausgewertet. Auf welchen Rändern des Berechnungsgitters welche Bedingung mit welchen Werten gilt, muss für die Simulation je nach Anwendungsfall definiert werden.

Algorithmus 5 TransportStep – Randbedingung.

if Dirichlet-condition set for $surf$ **then** ▷ Dirichlet Randbedingung
  $potential \leftarrow p_i - p_{dirichlet}$
else ▷ Neumann Randbedingung
  $potential \leftarrow p_{neumann}$
if $(potential) \geq 0$ **then**
  $\lambda_{upwind} \leftarrow \lambda(\overline{S}_i)$
else
  $\lambda_{upwind} \leftarrow \lambda(\overline{S}_{boundary})$
$\frac{dS_w}{dt}_i \leftarrow \frac{dS_w}{dt}_i + |surf|\, k\lambda_{upwind}\,(potential)\,/\,(\phi\,|c_i|\, CellDistance(i,\Gamma))$

`23-opennl.html`.

Sowohl für die Transport-Berechnung als auch für die Aufstellung des Druck-Gleichungssystems wird grundsätzlich ein ähnliches Programmgerüst verwendet. Der CUDA-Kernel Listing 3 muss dabei zunächst seine Position (Variablen `x,y,z`) beziehungsweise seinen Thread-Index (Variable `indexI`) für die Zelle im Berechnungsgitter, die er bearbeiten soll, ermitteln. Mit `indexI` greift die Kernel-Funktion auf die Variablenfelder Sättigung `Sw`, Druck `pn` usw. zu. Die Positionsvariablen `x,y,z` werden benötigt, um den Index `indexJ` der Nachbarzellen durch Hinzuaddieren eines Offsets zu bestimmen. Materialeigenschaften wie die effektive und relative Permeabilität, die Viskosität, die Dichte, die Residualsättigung und die Randbedingungen des Gitters werden in der Klasse `Cuda2pImpesParameter` verwaltet. Funktionen in dieser Klasse wie beispielsweise für die Bestimmung der Mobilität (`lambdaWetting()`) sind dabei sowohl von *host*- als auch von *device*-Seite aufrufbar. Dadurch kann auch für die CPU-Implementierung ein sehr ähnlicher Code mit den selben Materialparametern verwendet werden wie für den entsprechenden CUDA-Kernel. Der in Listing 3 verwendete Datentyp `scalar_` kann entweder als `float` oder als `double` definiert werden. Dies lässt sich in der Konfigurationsdatei des Programms zu dieser Arbeit umstellen.

Listing 3 auf Seite 33 zeigt einen Teil beziehungsweise die Grundstruktur des Sättigungs-Transport-Kernels mit Index-Logik und Schleife über die Grenzflächen. Hervorzuheben ist dabei die Logik für die Behandlung der Randbedingungen. Mit der *device*-Funktion `getBorderType(x,y,z)` wird für die Koordinaten bestimmt, ob und an welchem Rand des Berechnungsgitters sich die aktuelle Zelle befindet. Dies wird in einer Bitmaske gespeichert - für jede der sechs Seiten einer Zelle kann ein Bit als "Randbit" gesetzt werden. Dieses Randbit wird dann mit `cellBorder & (1<< neighbIndex)` für jede Seitenfläche abgefragt. Welche Bedingungen dann genau am Rand für Sättigung und Druck gelten, ist in der Klasse `Cuda2pImpesParameter` geregelt.

Für das *upwinding* entlang des Druckpotentials wird im CUDA-Kernel ein if-else Kontrollkonstrukt vermieden und statt dessen die Vergleichsvariable `upwindGreaterZero` verwendet. Diese nimmt je nachdem ob die Druckdifferenz zwischen den benachbarten Zellen positiv oder negativ ist den Wert 1 oder 0 an. Nun kann mit Ganzzahl-Multiplikation zwischen `lambdaI` und `lambdaJ` eine Auswahl getroffen werden. Auf diese Weise wird eine Divergenz des Kernel-Kontrollflusses vermieden.

Die Zell-Abmessungen wie die Seitenflächen oder Kantenlängen sind in regulären, kartesischen Gittern in jeder Raumachse konstant. Daher kann in einer globalen Variable für jede Raumachse die Seitenfläche (Variable `cellSurfaceArea`) und der Abstand zwischen zwei Zellen beziehungsweise Zellmittelpunkten (`cellMetrics`) gespeichert werden. Dies kann mit dem Oberflächen-Index abgefragt werden. Aufgrund der Reihenfolge der 6 Seitenflächen genügt es, den Oberflächen-Index `neighbIndex` durch 2 zu teilen, um den Raumachsen-Index zu erhalten. Dies wird mit einem Bit-Shift um eine Stelle nach rechts erreicht: `faceArea=cellSurfaceArea[neighbIndex>>1]`.

```
1 __global__ void CRSMatrixVectorMult (const scalar_ *values, const
    uint *colInd, const uint2 *rowPtr, const scalar_ *vector,
    scalar_ *result, uint numRows) {
2     // Thread-, beziehungsweise Zeilen-Index
3     const uint index = blockIdx.x * blockDim.x + threadIdx.x;
4
5     if (index < numRows) {
6         uint2 row_bounds = rowPtr[index];
7         scalar_ res = 0;
8         // Zeile multiplizieren
9         for (uint i = row_bounds.x; i < row_bounds.y; i++)
10            res += values[i] * vector[colInd[i]];
11        result[index] = res;
12    }
13 }
```

Listing 4: CRS Matrix-Vektor Multiplikations-Kernel.

4.2 Das Druckgleichungssystem

Um die Druckverteilung des Berechnungsgitters zu bestimmen, muss ein lineare Gleichungssystem aufgestellt und gelöst werden, so dass die Kontinuitätsgleichung $\nabla \cdot v = q$ erfüllt ist. Dazu sind zwei Schritte nötig: das Aufstellen des Gleichungssystems mit einer *dünnbesetzten Matrix* und der rechten Seite sowie das Lösen desselbigen, um als Ergebnis die Druckverteilung im Berechnungsgitter zu erhalten.

4.2.1 Dünnbesetzte Matrix

Die aufzustellende Matrix besitzt nur an wenigen Stellen Einträge ungleich Null, man spricht in diesem Fall von einer *dünnbesetzten Matrix*. In diesem Falle lohnt es sich eine Datenstruktur zu entwerfen, welche nur die Einträge speichert, welche auch tatsächlich Werte enthalten. Auf diese Weise lassen sich sehr große, dünnbesetzte Matrizen effizient im Simulationsprogramm speichern und bearbeiten.

Als Datenstruktur für die dünnbesetzte Matrix wird im Simulationsprogramm eine Variation des sogenannten *compressed row storage*-Formats (kurz *CRS*) verwendet. Dieses Format speichert die Einträge für Werte ungleich Null in einem Array ab ($values$) und in einem zweiten Array gleicher Größe die dazugehörigen Spalten-Indizes ($colInd$). Dadurch ist jeder Eintrag in $values$ eindeutig einer Spalte zuordenbar. In einem dritten Array werden mit Zweiertupeln der Anfangs- und Endindex des Werte-Arrays für jede Zeile gespeichert ($rowPtr$). Dieses Array indiziert die Einträge in $values$ und $colInd$ und ermöglicht die Zuordnung über die Zeilen. Die Indizes für Spalten und Zeilen beginnen immer mit 0. Zur Verdeutlichung ein kleines Beispiel:

$$\begin{pmatrix} 0 & 1 & 2 \\ 0 & 3 & 0 \\ 1 & 0 & 4 \end{pmatrix}, \qquad \begin{aligned} values &= \{1,2,3,1,4\}, \\ colInd &= \{1,2,1,0,2\}, \\ rowPtr &= \{\{0,2\},\{2,3\},\{3,5\}\}. \end{aligned}$$

Das Array *rowPtr* ist hier mit seinen Zweiertupeln redundant - es würde auch der jeweilige Endindex für jede Zeile der Matrix genügen. Diese Form der Datenstruktur, wie sie auch im *Concurrent Number Cruncher*[2] (kurz *CNC*) verwendet wird, ermöglicht es, Matrix-Operationen effizient in CUDA als *device*-Kernel zu implementieren. Die wichtigste Matrix-Operation ist dabei die Multiplikation mit einem Vektor. Dies spielt insbesondere für den Gleichungssystem-Löser eine wichtige Rolle.

Listing 4 zeigt den Kernel zum Multiplizieren einer Matrix im *CRS*-Format mit einem Vektor. Für jede Zeile der Matrix wird ein Kernel-Thread gestartet, der einen Eintrag im Ergebnisvektor `result` berechnet. Für ein Ganzzahl-Zweiertupel stellt CUDA den Datentyp `uint2` mit den Elementen `x` und `y` bereit.

4.2.2 Aufbau des Gleichungssystems

Ähnlich wie bei der Transport-Berechnung muss zum Aufstellen des Druck-Gleichungssystems über alle Oberflächen der Gitterzellen iteriert werden. Dabei wird wie bei der Transport-Berechnung ebenfalls upwinding auf Basis der Druckdifferenz beziehungsweise des Druckpotentials zwischen benachbarten Zellen angewendet.

Algorithmus 6 AssemblePressEqSystem.

```
procedure ASSEMBLEPRESSEQSYSTEM(S, p)                                    ▷
    A ← 0                                  ▷ Matrix initialisieren, alle Eintäge leer
    for all c_i ∈ GridCells do
        rhs_i ← 0                            ▷ keine Quellen/Senken im Normalfall
        for all surf ∈ Surfaces(c_i) do
            if surf is not a boundary surface then
                j ← NeighbIndex(surf)
                if (p_i − p_j) ≥ 0 then   ▷ upwinding auf Basis des Druck-Potentials
                    λ_upwind ← λ(S̄_i)
                else
                    λ_upwind ← λ(S̄_j)
                entry ← |surf| kλ_upwind/CellDistance(i, j)
                A_ii ← A_ii + entry                     ▷ Diagonal-Eintrag setzen
                A_ij ← −entry                       ▷ Nachbarschafts-Eintrag setzen
            else
                ...                          ▷ Randbedingung muss evaluiert werden
return (A, rhs)
```

Die Gleichung (2.36) auf Seite 17 beziehungsweise (2.37) bildet die Basis für die Matrix-Einträge und die rechte Seite des Gleichungssystems wie sie in Algorithmus 6 erzeugt werden. Da in der Simulation zu dieser Arbeit keine Quellen/Senken innerhalb des Gitters vorgesehen sind, ist hier rhs im Normalfall 0 - mit Ausnahme der Randbedingungen.

Algorithmus 7 AssemblePressEqSystem-Randbedingung.

```
if Dirichlet-condition set for surf then                ▷ Dirichlet Randbedingung
    if (p_i − p_dirichlet) ≥ 0 then
        λ_upwind ← λ(S̄_i)
    else
        λ_upwind ← λ(S̄_boundary)
    entry ← kλ_upwind |surf| /CellDistance(i, Γ)
    A_ii ← A_ii + entry
    rhs_i ← rhs_i + entry · p_dirichlet
else                                                     ▷ Neumann Randbedingung
    rhs_i ← rhs_i − p_neumann |surf| /ρ
```

Wie beim Transport-Problem müssen hier Zelloberflächen am Rand gesondert behandelt werden. In Algorithmus 7 ist zu sehen, wie für den Dirichlet-Fall der vordefinierte Dirichlet-Druck für die Bestimmung des Druckpotentials herangezogen wird. Auf Basis dieses Potentials wird λ mit *upwind-approximation* für die Randoberfläche bestimmt. $CellDistance(i, \Gamma)$ bezeichnet den Abstand vom Zellmittelpunkt zur Randoberfläche. Für den Neumann-Fall kann direkt die Druckänderung in das Gleichungssystem aufgenommen werden.

Für die Implementierung als CUDA-Kernel muss eine für CUDA geeignete Matrix-Datenstruktur mit entsprechenden Operationen verfügbar sein. Eine solche Datenstruktur für dünnbesetzte Matrizen wurde in Abschnitt 4.2.1 bereits vorgestellt. Um den CUDA-Kernel möglichst flexibel zu halten, wird die Matrix-Klasse als Template-Parameter übergeben - siehe Listing 5. Als Matrix-Klasse wird eine einfache Datenstruktur verwendet, welche Zeiger auf die CRS-Arrays enthält und direkt als Argument übergeben wird. Diese Klasse muss den elementweisen Zugriff auf *device*-Seite auf die CRS-Matrix-Struktur ermöglichen. Dies wird wie folgt in der Klasse `DeviceWriteMatrix` implementiert (siehe Listing 6).

Anmerkung: CUDA unterstützt auf *device*-Seite eingeschränkt C++ Konstrukte wie Templates und nicht-polymorphe Klassen. Weitergehende Features wie Funktions-Zeiger oder Rekursion sind derzeit noch nicht möglich. Das Schlüsselwort `__device__` definiert eine Funktion auf der *device*-Seite, sie ist von einem CUDA-Kernel aus aufrufbar. In diesem Fall muss auch Speicher, der in einer solchen Funktion verwendet wird, auf dem *device* liegen.

Die Grundstruktur des CUDA-Kernels für das Zusammenstellen der Gleichung in Listing 5 auf der nächsten Seite ist vom Aufbau sehr ähnlich zu dem bereits vorgestellten Kernel für das Transport-Problem (Listing 3). Das gilt insbesondere für die

```
template<typename Matrix>
__global__ void PressureMatrixAssembleKernel(Matrix A, scalar_ *rhs
    , const scalar_ *Sw, const scalar_ *pn) {
    uint x, y, z, indexI;
    indexI = getGridCoordinatesForThread(x, y, z);
    BorderType cellBorder = getBorderType(x, y, z);
    scalar_ A_diagonal_entry = 0; // A[indexI][indexI] sum
    scalar_ rightHandSide_entry = 0; // "Source" term is zero
    scalar_ SwI = Sw[indexI];
    scalar_ pnI = pn[indexI];
    scalar_ lambdaI = Cuda2pImpesParameter::lambda(SwI);

    for(int neighbIndex = 0; neighbIndex<gridNumNeighbours;
        neighbIndex++){
        BorderType faceBorder = (cellBorder & (1<<neighbIndex));
        scalar_ faceArea = cellSurfaceArea[neighbIndex>>1];/*
            Seitenflaeche*/
        scalar_ centerDist = cellMetrics[neighbIndex>>1];/*Abstand
            zw. 2 Zellen*/
        if (!faceBorder) { /* face between 2 cells */
            const int3& nOffs = cellNeighbOffset[neighbIndex];
            uint indexJ = getGridIndexForCoordinates(x+nOffs.x, y+
                nOffs.y, z+nOffs.z);
            scalar_ SwJ = Sw[indexJ];
            scalar_ pnJ = pn[indexJ];
            scalar_ lambdaJ = Cuda2pImpesParameter::lambda(SwJ);

            /* pressure potential ... */
            scalar_ potentialNonWetting = 0;
            potential = (pnI - pnJ) / centerDist;

            scalar_ upwindLambda;
            // variables to avoid branching (warp-friendly)
            int upwindGreaterZero = potential > 0;
            int upwindGreaterEqZero = potential >= 0;
            scalar_ upwindGreaterMean = upwindGreaterZero +
                upwindGreaterEqZero * (1-upwindGreaterZero) * 0.5;
            upwindLambda = upwindGreaterMean*lambdaI + (1-
                upwindGreaterMean)*lambdaJ;
            scalar_ entry = upwindLambda * Cuda2pImpesParameter::
                IntrinsicPermeability * faceArea / centerDist;
            A_diagonal_entry += entry;
            A.setEntryDevice(indexI, indexJ, -entry); // set off-
                diagonal entry
        } else {  /* border face */
            centerDist *= 0.5; //distance to border
            BoundaryCondition condPressure = Cuda2pImpesParameter::
                BoundaryConditionPressure(faceBorder);
            if(condPressure == BoundaryCondition_Dirichlet) {
                ...
            } else /*BoundaryCondition_Neumann*/ {
                ...
            }
        }
    }
    A.setEntryDevice(indexI, indexI, A_diagonal_entry);
    rightHandSide[indexI] = rightHandSide_entry;
}
```


Listing 5: PressureMatrixAssemble-Kernel.

```
1  class DeviceWriteMatrix {
2      scalar_ *values;
3      const uint *colInd;
4      const uint2 *redundantRowPtr;
5  public:
6      ...
7      __device__ inline void setEntryDevice(uint row, uint column,
           scalar_ value) {
8          uint2 rowptr_bounds = redundantRowPtr[row];
9          for (uint i = rowptr_bounds.x; i < rowptr_bounds.y; i++) {
10             uint eq = colInd[i] == column;
11             values[i] = (1-eq)*values[i] + eq*value;
12         }
13     }
14 };
```

Listing 6: Klasse `DeviceWriteMatrix`.

Index-Logik, die Iteration über die Zelloberflächen, die Handhabung mit *upwind-approximation* sowie die Behandlung der Randbedingungen.

Angefangen von der Index-Logik und die Schleife über die Zelloberflächen sowie die Handhabung der Rand-Oberflächen sind beide CUDA-Kernel nahezu identisch.

Um Zugriffe auf Elemente der Matrix-Datenstruktur zu minimieren, wird in Listing 5 für die Diagonal-Einträge der Matrix die Variable `A_diagonal_entry` eingeführt. Auf diese Weise muss das Diagonal-Element nur einmal für jeden Kernel-Aufruf mit `A.setEntryDevice()` für die Matrix `A` gesetzt werden.

4.2.3 Gleichungssystem-Löser in CUDA

Im Simulationsprogramm dieser Diplomarbeit sind zwei Ansätze für das Lösen des Druck-Gleichungssystems eingebaut. Einer setzt auf die externe Bibliothek OpenNL, die mit einem Plugin kompiliert werden kann, um die Berechnungen mit CUDA auf der GPU durchzuführen. Dieses Plugin wird *Concurrent Number Cruncher*[2] (kurz *CNC*) genannt.

Die OpenNL-API[13] bietet jedoch keine Möglichkeit, die Daten für das Gleichungssystem direkt als Speicherbereich auf die Grafikkarte zu übergeben. Das bedeutet, dass für das Lösen des Gleichungssystems mit OpenNL die Daten bei jedem Simulationsdurchlauf von der Grafikkarte auf die CPU kopiert und das Ergebnis von OpenNL wieder zurück auf die Grafikkarte kopiert werden muss. Außerdem bietet OpenNL bisher keine Möglichkeit, dünnbesetzte Matrizen direkt als Datenstruktur zu übergeben. Stattdessen müssen die Matrix-Elemente, welche nicht null sind, einzeln durch API-Aufrufe übergeben werden. Um diesen Mehraufwand zu vermeiden, ist im Simulationsprogramm auch eine eigene Version eines Gleichungssystem-Lösers

[13] *A*pplication *p*rogramming *i*nterface - Programmierschnittstelle einer Programm-Bibliothek.

eingebaut, welcher direkt mit der intern verwendeten Datenstruktur, der bereits vorgestellten dünnbesetzten CRS-Matrix, arbeiten kann. Die wesentlichen Operationen beim Lösen eines Gleichungssystems sind die bereits vorgestellte Matrix-Vektor-Multiplikation, das Skalarprodukt und die mit einem Faktor gewichtete Vektor-Addition.

Algorithmus 8 SolvePressEqSystem als PCG Verfahren Quelle: [2].

procedure SOLVEPRESSEQSYSTEM($\mathbf{A}, \vec{b}, \vec{x}_0$) ▷ Gl.-System $\mathbf{A}\vec{x} = \vec{b}$ lösen
    $\vec{r} \leftarrow \vec{b} - \mathbf{A}\vec{x}_0$
    $\mathbf{M} \leftarrow \text{diag}(\mathbf{A})$ ▷ Jacobi-Vorkonditionierung
    $\vec{d} \leftarrow \mathbf{M}^{-1}\vec{r}$
    $i \leftarrow 0$
    $\delta_i \leftarrow \left\langle \vec{r}, \vec{d} \right\rangle$
    while $i < maxIter$ and $\delta_i > \varepsilon^2 \delta_0$ **do**
        $\alpha \leftarrow \frac{\delta_i}{\left\langle \vec{d}, \mathbf{A}\vec{d} \right\rangle}$
        $\vec{x}_{i+1} \leftarrow \vec{x}_i + \alpha\vec{d}$ ▷ Iteration der Näherung $\vec{x}_i$
        $\vec{r} \leftarrow \vec{r} - \alpha\ \mathbf{A}\vec{d}$
        $\delta_{i+1} \leftarrow \langle \vec{r}, \mathbf{M}^{-1}\vec{r} \rangle$
        $\beta \leftarrow \frac{\delta_{i+1}}{\delta_i}$
        $\vec{d} \leftarrow \vec{r} + \beta\vec{d}$ ▷ **A**-konjugierte Abstiegsrichtung
        $i \leftarrow i + 1$
return x_i

Als Berechnungsverfahren für die Implementation des Lösers wird die sogenannte *konjugierte Gradienten*-Methode (kurz *CG*) verwendet ([1], ab Seite 85). Diese ist ein iteratives Verfahren und eignet sich besonders für dünnbesetzte, symmetrische Matrizen.

Das *konjugierte Gradienten*-Verfahren ist eine Verbesserung des herkömmlichen *Gradienten-Abstiegsverfahrens*. Für ein Gleichungssystem der Form $\mathbf{A}\vec{x} = \vec{b}$ berechnet das *Gradienten-Abstiegsverfahren* iterativ die Näherung in der i-ten Iterationsstufe durch Addition des Vektors in Richtung des steilsten Gradienten-Abstiegs: $\vec{x}_{i+1} = \vec{x}_i - \beta(\mathbf{A}\vec{x}_i - \vec{b})$. Es wird so lange iteriert, bis die Abweichung zwischen zwei Iterationsschritten $\|\vec{x}_{i+1} - \vec{x}_i\|$ einen gegebenen Schwellwert ε unterschreitet.

Beim *konjugierte Gradienten*-Verfahren wird die Näherung x_{i+1} nicht in Richtung des steilsten Abstiegs, sondern in die **A**-*konjugierte Richtung* entwickelt.

Der Unterschied zwischen *konjugierte Gradienten* Verfahren und dem *Gradient-Abstiegsverfahren* wird in Abbildung 13 veranschaulicht. Die Ellipsen um die Lösung $\vec{x}$ stellen das Gradientenfeld eines Gleichungssystem mit zwei Freiheitsgraden dar.

Algorithmus 8 zeigt den *konjugierte Gradienten* Ansatz, wie er im *Concurrent Number Cruncher*[2] verwendet wird. Es handelt sich dabei um einen erweiterten Algorithmus, der eine Vorkonditionierung verwendet (*Preconditioned Conjugate Gradi-*

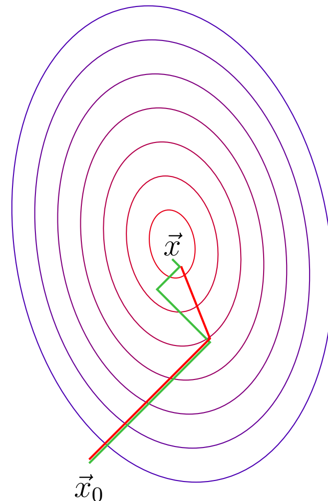


Abbildung 13: *Konjugierte Gradienten*(rot) im Vergleich zum *Gradient-Abstiegsverfahren*(grün).

ent, kurz *PCG*). Als Vorkonditionierer wird hier die Diagonalmatrix von **A** verwendet, man spricht in diesem Fall von einem *Jacobi Vorkonditionierer*. Diese Erweiterung verbessert die Konvergenzrate des Verfahrens deutlich.

In Algorithmus 8 ist es möglich, den Startwert x_0 anzugeben. Standardmäßig ist das der Nullvektor. In unserer Simulation kann jedoch für x_0 die Lösung einer vorangegangen Iteration eingesetzt werden. Dies führt wegen der zeitlichen Stetigkeit der gesuchten Lösung (Druckverteilung) ebenfalls zu einer Reduzierung der nötigen Iterationen beziehungsweise schnellerer Konvergenz des Algorithmus.

Die Iterations-Schleife für die Implementierung des Gleichungssystem-Lösers liegt auf der CPU/*host*-Seite. Dagegen werden die einzelnen Operationen wie die Matrix-Vektor-Multiplikation, Skalaprodukt etc. jedoch mit CUDA beschleunigt und liegen komplett mit allen Daten zum Lösen des Gleichungssystems auf der *device*-Site. In Listing 7 werden dazu auch Funktionen der CUBLAS-Bibliothek verwendet. Diese von Nvidia bereitgestellte Bibliothek implementiert einen Teil der populären Fortran-BLAS[14]-Funktionen in CUDA. Verwendet werden hier die Funktionen `cublasXaxpy` für die Vektor-Addition mit Skalarfaktor, `cublasXscal` für das einfache Skalieren eines Vektors sowie `cublasXdot` für das Skalarprodukt. Alle diese CUBLAS-API-Funktionen arbeiten direkt mit dem Speicher im *global memory* auf der *device*-Seite. Je nachdem, ob der Datentyp `scalar_` als `float` oder als `double` definiert ist, wird für `cublasXaxpy` entweder die Version für einfache Genauigkeit `cublasSaxpy` oder die Version für doppelte Genauigkeit `cublasDaxpy` verwendet. Das Gleiche geschieht bei den anderen verwendeten CUBLAS-Versionen.

Da die Vorkonditionierung mit der Diagonalmatrix von **A** erfolgt ($\mathbf{M} = \text{diag}(\mathbf{A})$), genügt es für $\mathbf{M}^{-1}$ einen Vektor mit den Kehrwerten der Diagonalmatrix von **A**

[14] *B*asic *L*inear *A*lgebra *S*ubprogramms

```
1  template<typename Matrix>
2  uint Cuda2pImpesRGrid::SolvePressureEqSystemDevice(const Matrix& A_
       , const scalar_ *rightHandSide_d, scalar_ *pressureNonWetting_d)
       {
3      ... /* Variablen initialisieren ... */
4      gpu_b = rightHandSide_d;/*"_d" steht fuer Device-Speicher*/
5      gpu_x = pressureNonWetting_d;/*frühere Lösung als Anfangswert*/
6      scalar_ alpha, beta;
7
8      /* Jacobi-Preconditioner: M^(-1) = 1 / diag(A) */
9      A_.getInvDiagonal(gpu_diag_inv);
10
11     A_.multMatrixVectorDevice(gpu_x, gpu_r); // r = A*x
12
13     cublasXaxpy(N, -1.0, gpu_b, 1, gpu_r, 1); // r = b - A*x
14     cublasXscal(N, -1.0, gpu_r, 1); // N ist die Vektor-Laenge /
          Anzahl Zellen
15
16     // d = M^(-1) * r (Hadamard: Elementwise Multiplikation)
17     HadamardVecMultKernel<<<numBlocks, threadsPerBlock>>>(N,
          gpu_diag_inv, gpu_r, gpu_d);
18
19     // cur_err = r*d err = epsilon^2 * <r,d>
20     scalar_ cur_err = cublasXdot(N, gpu_r, 1, gpu_d, 1);
21
22     uint iterations = 0;
23     scalar_ err = cur_err * epsilon * epsilon;
24     while (cur_err > err && its < Cuda2pImpesParameter::
          SolvePressureMaxIterations) {
25         A_.multMatrixVectorDevice(gpu_d, gpu_Ad);//Ad = A*d
26
27         // alpha = cur_err / <d,Ad>
28         alpha = cur_err / cublasXdot(N, gpu_Ad, 1, gpu_d, 1);
29
30         cublasXaxpy(N, alpha, gpu_d, 1, gpu_x, 1);//x = x + alpha*d
31
32         cublasXaxpy(N, -alpha, gpu_Ad, 1, gpu_r, 1);//r=r-alpha*Ad
33
34         // h = M^(-1) * r
35         HadamardVecMultKernel<<<numBlocks, threadsPerBlock>>>(N,
              gpu_diag_inv, gpu_r, gpu_h);
36
37         scalar_ old_err = cur_err ;
38         cur_err = cublasXdot(N, gpu_r, 1, gpu_h, 1);//cur_err=<r,h>
39         beta = cur_err / old_err ;
40
41         cublasXscal(N, beta, gpu_d, 1); // d = h + beta*d
42         cublasXaxpy(N, 1.0, gpu_h, 1, gpu_d, 1);
43         iterations++;
44     }
45     ... /* optional time and precision evaluation */
46     return iterations;
47 }
```

Listing 7: SolvePressureEqSystemDevice.

zu verwenden. Das Matrix-Vektor-Produkt $\mathbf{M}^{-1}\vec{r}$ wird in Listing 7 daher zur elementweisen Multiplikation der Vektoren `gpu_diag_inv` und `gpu_r`. Diese elementweise Multiplikation erledigt die Kernel-Funktion `HadamardVecMultKernel` und schreibt das Ergebnis in den letzten Parameter.

4.3 Die in-situ Visualisierung

Ziel dieser Arbeit ist es auch die Simulationsergebnisse grafisch darzustellen. Es soll dabei bereits während der Berechnung der einzelnen Simulationsschritte der Fortschritt der Gesamtsimulation in Echtzeit visualisiert werden. Dieser Ansatz wird auch *in-situ* Visualisierung genannt.

Für die Implementierung einer solchen in-situ Visualisierung wurde das plattformunabhägngige VTK[15]-Framework verwendet. Diese Bibliothek erlaubt es mit relativ wenig Aufwand ein Fenster für die Darstellung zu öffnen und Objekte mit Interaktionsmöglichkeit anzuzeigen.

Für die Daten einer Simulation bietet das VTK-Framework viele Möglichkeiten unstrukturierte und strukturierte Gitter in verschiedenen Modi und Visualisierungstechniken darzustellen.

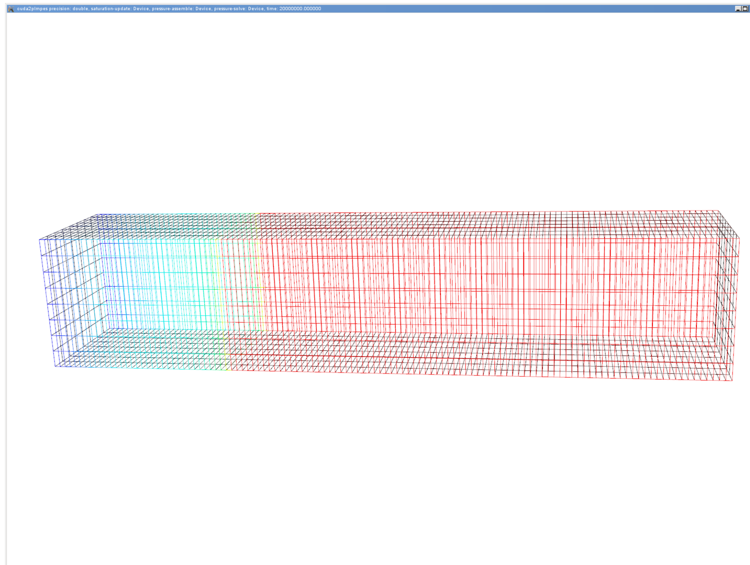


Abbildung 14: Gitter-Visualisierung von `porSimGPU`.

Im Simulationsprogramm wurde für die Visualisierung des Gitters die Klasse `vtkStructuredPoints` verwendet. Diese Klasse ist dafür vorgesehen, kartesische Gitter mit äquidistanten Zellbreiten zu verwalten und eignet sich damit für das Berechnungsgitter dieser Simulation.

[15] The *V*isualisation *T*ool*k*it, `http://www.vtk.org`.

Für die Variablen Druck und Sättigung wird in VTK jeweils ein Skalarfeld angelegt, das für jede Zelle einen Wert speichert. Man kann während der Darstellung die Daten der Skalarfelder im Hintergrund ändern und anschließend mit der `Modified()`-Methode die VTK-Pipeline aktualisieren. So entsteht während der Simulation eine Animation des zeitlichen Werteverlaufs.

Die Visualisierung der Sättigungsverteilung im Gitter-Modus ist in Abbildung 14 zu sehen. Der Wertverlauf wird mit einem Farbverlauf von Blau (für höhere Werte) nach Rot (für niedrigere Werte) verdeutlicht.

Neben der Darstellung als Drahtgittermodell unterstützt das Programm zu dieser Arbeit "`porSimGPU`" aber auch eine Volumenvisualisierung. Die Idee hinter dieser Art der Visualisierung besteht darin, die dreidimensionale Information auf der zweidimensionalen Projektionsfläche sichtbar zu machen. Dazu wird durch gewichtetes Aufsummieren der Werte in den Gitterzellen entlang von Sichtstrahlen der Farbwert eines Pixels bestimmt.

VTK bietet für die Volumenvisualisierung einen großen Funktionsumfang. Anders als bei einer Oberflächen-Visualisierung muss für den Volumen-Ansatz explizit eine *Transfer-Funktion* für die Farbdarstellung angegeben werden. *Transfer-Funktion* bedeutet hier, dass für jeden Wert im Skalarfeld eine Darstellungsfarbe definiert sein muss. Dies kann auch in Form einer linearen Funktion oder einer Farbtabelle erfolgen. Auch für die Opazität, also die Lichtundurchlässigkeit, muss eine Transfer-Funktion angegeben werden. Diese Transfer-Funktion steuert wie stark ein Raumpunkt des Gitters zum Gesamtbild beiträgt - abhängig von seinem Skalarwert.

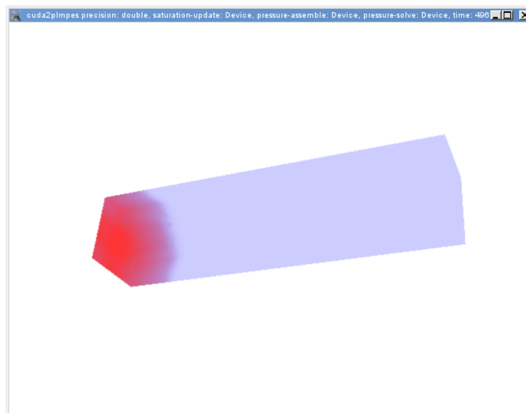
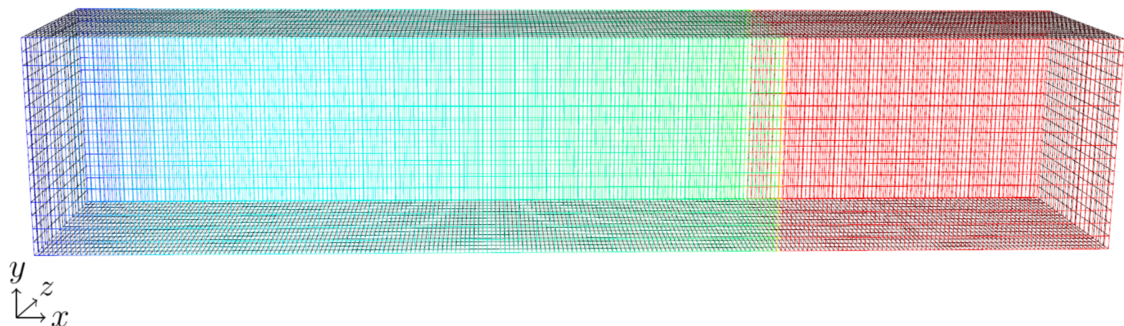


Abbildung 15: Volumen-Visualisierung.

Das gleiche Gitter, welches in Abbildung 14 als Drahtgittermodell zu sehen ist, wird in Abbildung 15 auf Basis seiner Sättigungsverteilung als Volumen dargestellt.

5 Ergebnisse

Dieses Kapitel wertet die Simulationsergebnisse aus, die das für diese Arbeit implementierte Simulationsprogramm “porSimGPU” liefert. Dabei werden hauptsächlich zwei Aspekte evaluiert. Das ist zum einen die Genauigkeit, gemessen am relativen Fehler der berechneten Simulationsdaten. Zum anderen der Geschwindigkeitsvorteil, welcher durch die Portierung der Berechnung auf die CUDA-Plattform gewonnen wird. Das Simulationsprogramm umfasst, wie Unterkapitel 4.3 beschrieben, eine eigene Visualisierung der Ergebnisse. Dazu werden ebenfalls Beispiele in verschiedenen Visualisierungs-Modi vorgestellt.



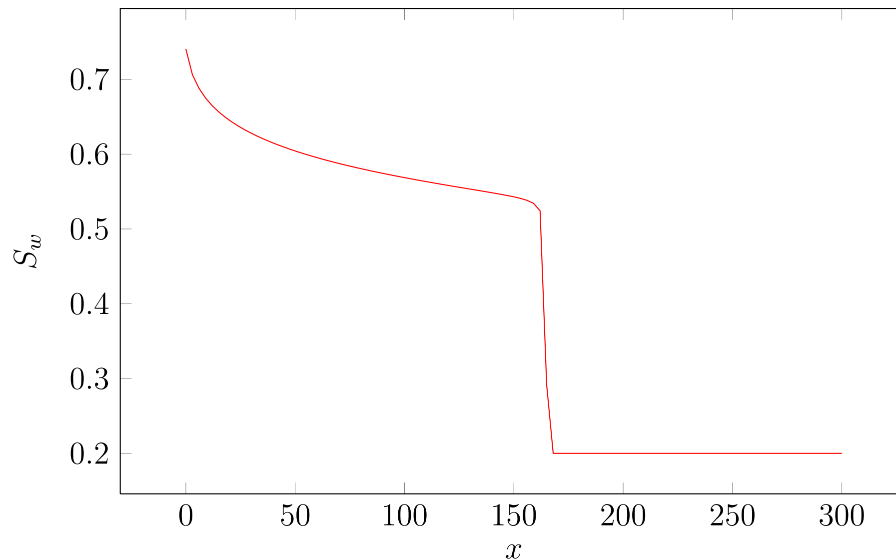


Abbildung 16: Sättigungsverteilung in einem 240×16×16 Gitter bei $t = 5 \cdot 10^7$.

Abbildung 16 zeigt die Visualisierung der Sättigungsverteilung von “porSimGPU” für ein einfaches Gitter mit 240×16×16 Zellen nach einer Simulationszeit von $t = 5 \cdot 10^7$. Die im Diagramm abgebildete Sättigungskurve ist ein Schnitt der Werte entlang der Hauptachse des 3D-Gitters (x-Achse). Die Randbedingung ist hier so gesetzt, dass die *wetting phase* konstant entlang der x-Achse in das Gitter hineinströmt. Dies

wird durch den farblichen Gradientverlauf entlang der Hauptachse deutlich. Auf dem dreidimensionalen Gitter wurde im Wesentlichen ein eindimensionales Problem berechnet. Das Diagramm in Abbildung 16 ist mit der in Kapitel 2.4 vorgestellten theoretischen Sättigungs-Schockwelle aus Abbildung 6 auf Seite 13 vergleichbar.

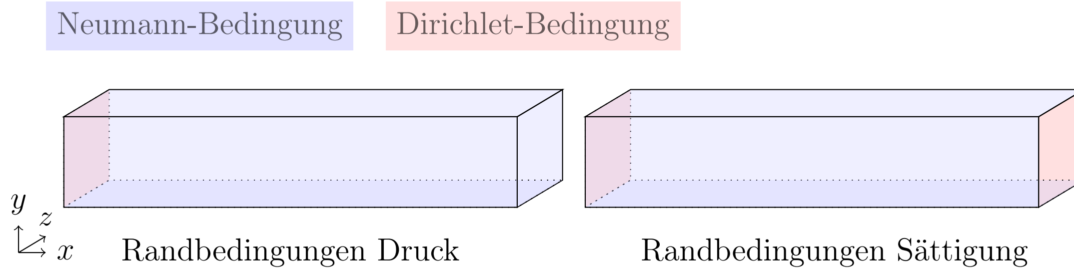


Abbildung 17: Randbedingungen des Berechnungsgitters.

Die Randbedingungen, wie sie für die Berechnung des Gitters in Abbildung 16 verwendet wurden, sind in Abbildung 17 dargestellt. Dabei wird für die Feldvariable der Sättigung eine andere Konfiguration verwendet, als für den Druck.

Die Dirichlet-Bedingung für den Druck an der yz-Oberfläche sorgt dafür, dass ein konstanter Transport in x-Richtung des Gitters stattfindet.

Das Gitter selbst ist zu Beginn der Simulation komplett mit der *non-wetting* Phase ausgefüllt - also für jede Zelle gilt zum Zeitpunkt $t = 0$: $\overline{S}_w = 0$ beziehungsweise $S_w = S_{wr}$. Die Dirichlet-Randbedingung der Sättigung ist so definiert, dass in x-Richtung die *wetting-phase* einströmt ($\overline{S}_w = 1$). Zusammen mit der Bedingung für den Druck wird nun ein fortlaufender Massenzufluss erzeugt.

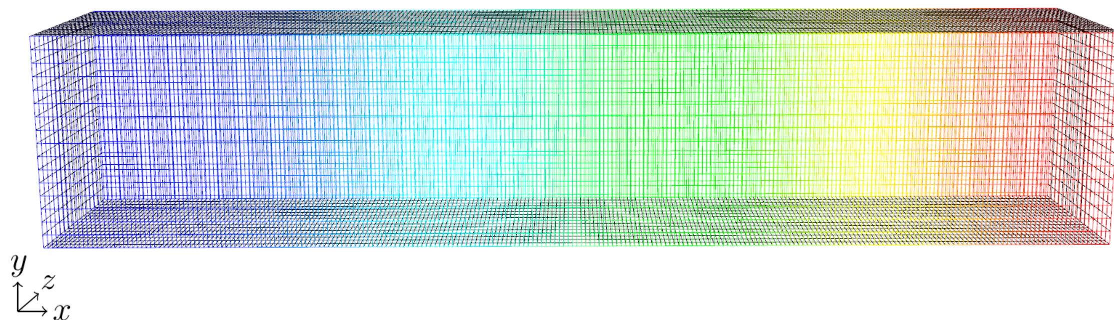
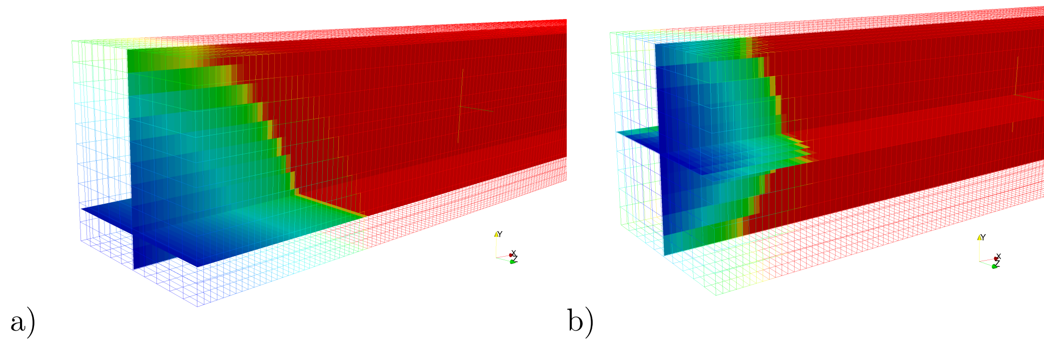


Abbildung 18: Druckverteilung nach einer Simulationszeit von $t = 5 \cdot 10^7$.

Abbildung 18 zeigt die zu Abbildung 16 gehörende Druckverteilung. Diese besteht aus einem entlang der x-Achse gleichmäßig abfallenden Gradienten, wie es für die Randbedingung des Druckes in Abbildung 17 zu erwarten ist.



a) b)

Abbildung 19: Veränderung der Randbedingung.

In Abbildung 16 werden die Randbedingungen der Simulation als konstant über die Fläche dargestellt. Das bedeutet, dass über die xy-Ebene konstant in x-Richtung Masse in das poröse Medium einfließt. Die Simulation im dreidimensionalen Gitter berechnet mit diesen Randbedingungen aus Sicht der Freiheitsgrade ein eindimensionales Problem. Wie jedoch bereits im Kapitel über die Randbedingungen 2.5.2 erwähnt, ist es auch möglich Randbedingungen als Funktion des Ortes zu definieren. Dadurch können die Einströmverhältnisse komplexer gestaltet werden. Abbildung 19 Teilbild a) zeigt das Einströmen der Sättigung mit einer Gradienten-Funktion entlang der y-Achse für die Dirichlet-Randbedingung des Druckes. Der Dirichlet-Druck nimmt hier mit zunehmender Gitterhöhe ab. Es entsteht ein Effekt wie man ihn beispielsweise bei Berücksichtigung der Gravitation vermuten würde. Abbildung 19 Teilbild b) zeigt eine Simulation mit konischem Verlauf des Dirichlet-Druckes an der xy-Ebene. Dieser Fall stellt das Einströmen von Flüssigkeit an einer bestimmten Stelle im Berechnungsgitter nach.

Der Verlauf der Sättigung durch die laufende Simulation wird in Abbildung 20 dargestellt. Die Sättigungsfront durchwandert das Gitter mit der deutlich sichtbaren Schockwelle.

5.1 Genauigkeit/Fehlertoleranz

Das Ziel dieser Arbeit lag zum Großteil darin, die Funktionalität von DuMu^x[6] auf die Grafikhardware zu portieren. Für DuMu^x ist bekannt, dass es adäquate Ergebnisse liefert und der Restfehler zu einer analytischen Lösung genügend klein ist. Daher sollte in dieser Arbeit sichergestellt werden, dass keine größeren Abweichungen der Simulation zum DuMu^x-Referenzprogramm entstehen.

Um eine Abweichung zu überprüfen, werden die Programme dieser Arbeit (`porSimGPU`) und DuMu^x das Zweiphasen-Problem in porösen Medien mit den gleichen Gitterabmessungen, Randbedingungen und Materialparametern simulieren. Innerhalb von `porSimGPU` ist es möglich zu bestimmen, welche Berechnungsschritte auf der

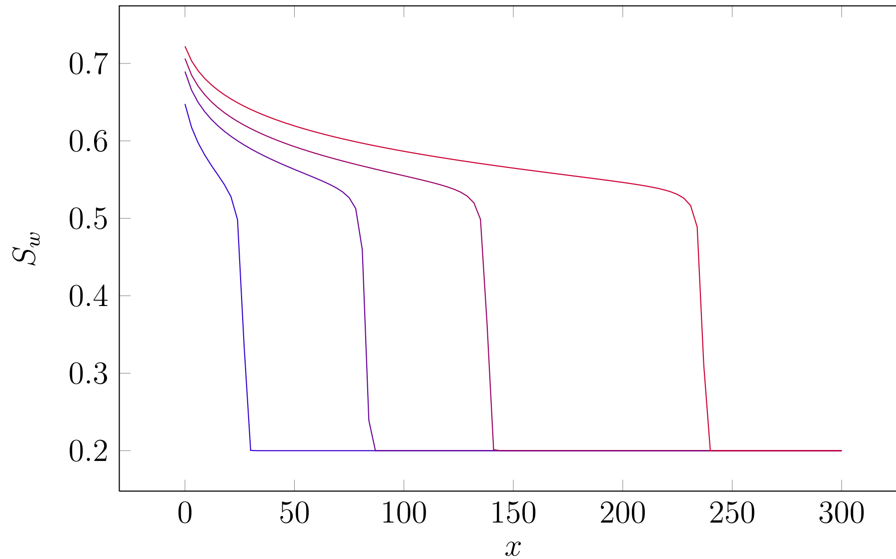


Abbildung 20: Zeitlicher Verlauf der Sättigungsfront.

GPU/*device* oder der CPU/*host* berechnet werden sollen. Da die GPU die arithmetischen Gleitkomma-Operationen nicht zu 100% gleich wie die CPU implementiert, sind auch hier kleinere Abweichungen zu erwarten.

Als Maß der Abweichung zwischen zwei Simulationsergebnissen wird die sogenannte L_2-Norm verwendet. Diese Norm ist als die kartesische Länge ($\|\|_2$) eines Vektors $\vec{x}$ wie folgt definiert:

$$\|\vec{x}\|_2 = \sqrt{\langle \vec{x}, \vec{x} \rangle}.$$

Um den *absoluten Fehler*, auch *Residuum* genannt, zu bestimmen, wird für einen festen Simulations-Zeitpunkt t eine Feldvariable (Sättigung S_w oder Druck p_n) zwischen den Sollwerten und den Werten aus `porSimGPU` verglichen. S_w und p_n sind hier Felder, die für jede Zelle des Berechnungsgitters einen Wert definieren - diese Felder werden als Vektoren mit der Anzahl der Zellen als Dimension interpretiert.

Das *Residuum* der Feldvariable X liefert den absoluten Fehler zwischen dem Simulationsergebnis A und den *Sollwerten* B an:

$$r_h = \|X_A - X_B\|_2 \,.$$

Die Angabe des *Residuums* r_h gibt zwar eine absolute Größe der Abweichung an, jedoch keine qualitative Angabe des Fehlers. Um eine Aussage über die Fehlertoleranz zu erhalten, wird der relative Fehler e_{rel} als Verhältnis von Residuum zur Norm der Sollwerte gebildet:

$$e_{rel} = \frac{r_h}{\|X_B\|_2} = \frac{\|X_A - X_B\|_2}{\|X_B\|_2}.$$

Durch Multiplikation von e_{rel} mit 100 erhält man eine prozentuale Abweichung der Ergebnisse zwischen der Berechnung (Quelle) X_A und dem Sollwert X_B. Die Feldvariable X kann für eine der beiden Hauptvariablen stehen: die Sättigungsverteilung S_w oder die Druckverteilung p_n. DuMu^x im Folgenden als Sollwert ($X_B = X_{Dumux}$) angenommen um den relativen Fehler der Simulation mit "`porSimGPU`" sowohl mit Berechnung auf *device*-Seite (X_{device}) als auch auf *host*-Seite (X_{host}) zu bestimmen.

Quelle A	**Sollwert** B	**Simulationszeit** t	**Fehler** $X = p_n$	**Fehler** $X = S_w$
X_{device}	X_{Dumux}	$2,981221 \cdot 10^6$	0%	0,532463%
X_{device}	X_{Dumux}	$5,368578 \cdot 10^6$	0%	0,543279%
X_{device}	X_{Dumux}	$1,01334 \cdot 10^7$	$6,4551 \cdot 10^{-5}$%	0,585823%
X_{host}	X_{Dumux}	$2,981221 \cdot 10^6$	0%	0,532752%
X_{host}	X_{Dumux}	$5,368578 \cdot 10^6$	0%	0,54365%
X_{host}	X_{Dumux}	$1,01334 \cdot 10^7$	$6,4551 \cdot 10^{-5}$%	0,586553%
X_{device}	X_{host}	$2,981221 \cdot 10^6$	0%	0,00030161%
X_{device}	X_{host}	$5,368578 \cdot 10^6$	0%	0,00040756%
X_{device}	X_{host}	$1,01334 \cdot 10^7$	0%	0,00078557%

Tabelle 1: Relativer Fehler der Simulation bei einem 60×4×4-Gitter.

Tabelle 1 zeigt die relativen Fehlerraten beziehungsweise Abweichungen zwischen X_{Dumux}, X_{device}und X_{host}. Der relative Fehler ist bei der Sättigung ($X = S_w$) im Vergleich zum Druck ($X = p_n$) deutlich größer, obwohl beide Feldvariablen gegenseitig gekoppelt sind. Dieser Unterschied liegt darin begründet, dass die absoluten Zahlenwerte für die Sättigung ($0 < S_w < 1$) deutlich niedriger liegen als für den Druck (~ 20000). Durch das hardwareseitige Format der Gleitkommazahlen mit Exponent und Mantisse hängen die möglichen Abstände zwischen zwei darstellbaren Zahlen vom Zahlenwert selbst ab. Für die Druckverteilung steht daher eine geringere numerische Darstellungsgenauigkeit zu Verfügung. Kombiniert mit Rundungsverhalten bei arithmetischen Operationen sinkt dadurch der relative Fehler.

Mit zunehmender Zahl an Iterationsschritten steigt der relative Fehler für beide Variablen an (vergl. Spalte "Simulationszeit" in Tabelle 1). Dies ist im expliziten Zeitschrittverfahren begründet, welches Teil der IMPES-Lösungsstrategie ist. Fehler summieren sich hier über den zeitlichen Verlauf der Simulation.

Da sich das `porSimGPU`-Programm stark an die Implementierung von DuMu^x anlehnt, ist der relative Fehler zwischen beiden Programmen bei gleicher Gleitkommagenauigkeit klein. Der entstehende Fehler ist hauptsächlich auf die verwendeten Gleichungssystem-Löser zurückzuführen. Bei X_{Dumux} wird ein DuMu^x/DUNE interner Löser verwendet, bei X_{host} die Bibliothek OpenNL und bei X_{device} eine Neuimplemetierung des CUDA-basierten Lösers aus OpenNL/CNC[16] wie in Abschnitt 4.2.3 beschrieben.

[16] *C*oncurrent *N*umber *C*runcher[2].

5.1.1 Probleme bei einfacher Gleitkomma-Genauigkeit

Für das Programm `porSimGPU` kann zur Übersetzungszeit zwischen einfacher und doppelter Gleitkomma-Genauigkeit (`float` statt `double` als Datentyp) umgestellt werden. Das ist insbesondere für die Simulation auf *device*-Seite relevant, da aufgrund der Bauart und der Historie der Grafikhardware die einzelnen arithmetischen Operationen bei einfacher Genauigkeit deutlich schneller als bei doppelter Genauigkeit ablaufen. Bei mancher Hardware unterscheiden sich die Berechnungszeiten zwischen `float` und `double` um den Faktor 8.

Die Verwendung von `float` als Datentyp birgt jedoch aus numerischer Sicht viele Herausforderungen. Da Gleitkomma-Zahlen eine endliche Darstellungsgenauigkeit haben, besteht die Gefahr der *numerischen Auslöschung*. Das bedeutet, dass bei einer arithmetischen Operation von einer sehr kleinen Zahl a mit einer sehr großen Zahl b die kleinere Zahl aufgrund der Darstellungsungenauigkeit verschwinden kann. Es gibt also Fälle, in denen $a + b = b$ für $|a| \lll |b|$ gilt. Dies wird bei Algorithmen insbesondere dann bemerkbar, wenn zu einer großen Zahl viele kleine Zahlen hinzuaddiert werden müssen.

Es gilt für Gleitkomma-Zahlen weder Assoziativität noch Kommutativität. Dies macht sich in numerisch anfälligen Algorithmen bei einfacher Genauigkeit deutlich früher bemerkbar als bei doppelter Genauigkeit.

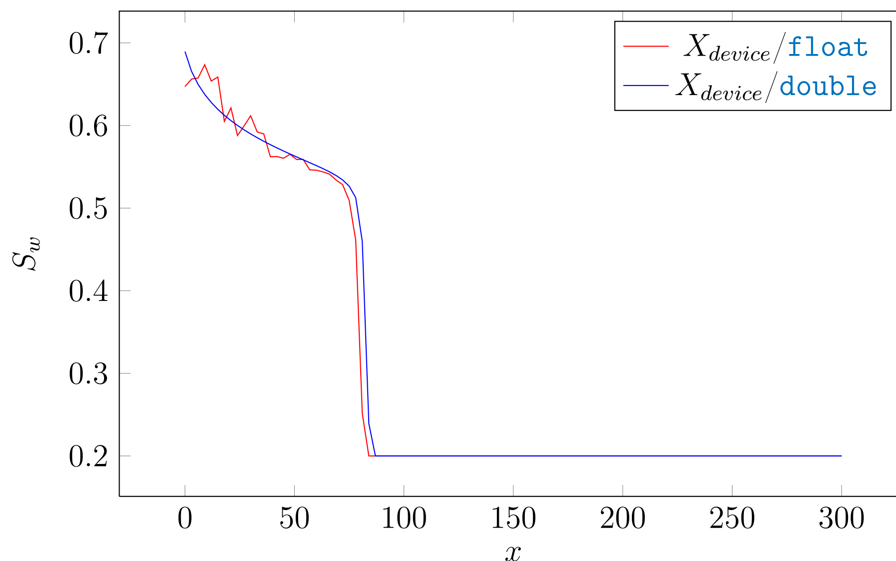


Abbildung 21: Oszillation der Sättigung bei einfacher und doppelter Gleitkomma-Genauigkeit.

Wird das Simulationsprogramm `porSimGPU` mit dem Datentyp `scalar_` als `float` kompiliert, so ergibt sich in der Simulation eine Sättigungskurve wie in Abbildung 21 rot dargestellt. Es ist hier sehr deutlich eine Oszillation der Sättigungskurve zu erkennen. Eine solche Oszillation führt nach kurzer Zeit zu instabilem Simulationsver-

halten. Zum Vergleich ist im Diagramm auch die Sättigungskurve bei gleichem Gitter und gleichen Randbedingungen mit doppelter Genauigkeit/`double` eingezeichnet (blau).

Beim verwendeten IMPES-Verfahren mit explizitem und implizitem Berechnungsschritt liegt die Vermutung nahe, dass der implizite Teil mit dem Gleichungssystem-Löser für numerische Ungenauigkeiten anfälliger ist als der explizite Zeitschritt. Um dies näher zu untersuchen, ist auch die Druckverteilung im Vergleich zwischen beiden Gleitkomma-Genauigkeiten in Abbildung 22 dargestellt. Hier ist zu erkennen, dass bei `float`-Genauigkeit ein sehr viel ausgeprägterer Druck-Gradient entsteht. Dies deutet darauf hin, dass das Druck-Gleichungssystem nicht exakt genug berechnet wird. Der steilere Druck-Gradient hat auch zur Folge, dass der Massentransport schneller stattfindet und damit auch die Zeitskalen zwischen einfacher und doppelter Genauigkeit nicht mehr vergleichbar sind.

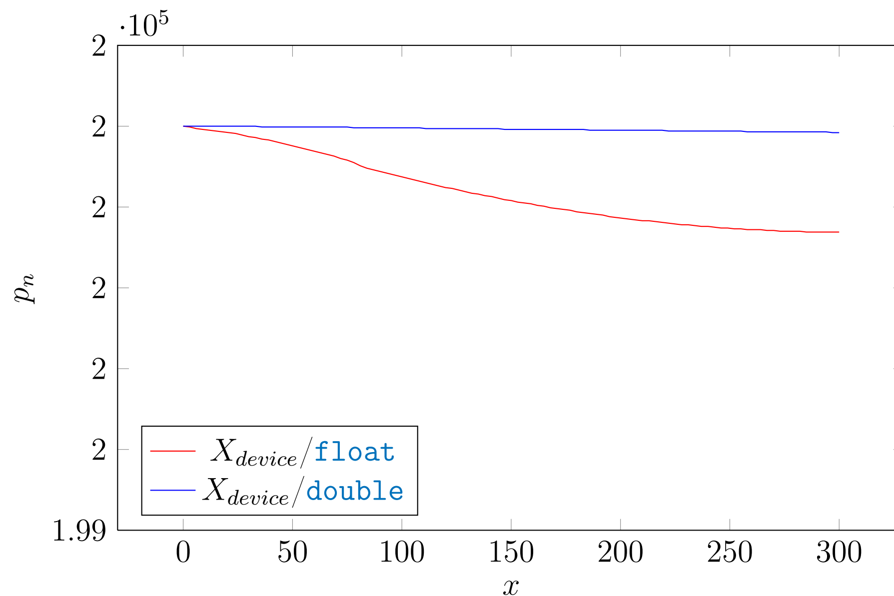


Abbildung 22: Druckverteilung bei einfacher und doppelter Gleitkomma-Genauigkeit.

Eine Verbesserung der Situation bei einfacher Genauigkeit kann durch ein spezielles Umgestaltung des Programms erreicht werden. Beispielweise ist es möglich durch Umsortieren der arithmetischen Operationen oder ganzer Berechnungsteile die numerische Stabilität des Programms zu erhöhen. Dies erfordert aber sehr spezialisierte Fachkenntnisse und übersteigt den Rahmen dieser Arbeit.

Den Herstellern von Grafikhardware ist das Problem bekannt und es ist damit zu rechnen, dass die `double`-Leistungsfähigkeit in Zukunft weiter ausgebaut wird.

5.2 Leistungsanalyse

Für die Geschwindigkeitsmessung ist es notwendig, die tatsächliche Berechnungszeit zu erfassen. Dazu müssen bei der Zeitmessung im Programm Operationen wie Speicherallokierung, Initialisierungen, und Ein/Ausgabe-Operationen ausgenommen werden.

Für diese Arbeit ist besonders relevant, inwieweit die Parallelisierung mit CUDA Fortschritte gegenüber einer CPU-basierten Implementierung bietet. In Tabelle 2 wird die Berechnungszeit für drei Gitter unterschiedlicher Größe jeweils auf CPU-Basis (X_{host}) und auf CUDA-Basis (X_{device}) für doppelte und einfache Gleitkomma-Genauigkeit gegenübergestellt. Es wurde mit jeweils gleich vielen Iterationen sowie gleichen Randbedingungen (wie in Abbildung 17) und Materialparametern gerechnet.

Gitter	**Domäne**	**Berechnungszeit [ms]**	benöt. Iterationen
60×4×4	X_{host}	568,671	37
60×4×4	X_{device}	301,28	37
120×8×8	X_{host}	3879,52	77
120×8×8	X_{device}	1397,47	77
240×16×16	X_{host}	58905,3	157
240×16×16	X_{device}	18213,7	157
300×20×20	X_{host}	151555,0	197
300×20×20	X_{device}	48625,6	197

Tabelle 2: Vergleich der Berechnungszeiten zwischen X_{host} und X_{device} bei $t = 1 \cdot 10^7$ und Datentyp `double`.

Als Hardware kam in Tabelle 2 ein Computer mit Intel Core i7 mit 8 logischen Prozessoren, 8GB RAM und Nvidia Grafikkarte GeForce GTX 480 als CUDA-*device* zum Einsatz. Der Unterschied in der Berechnungszeit im kleinen Gitter mit 60×4×4 Zellen ist noch nicht besonders groß - hier wird auf *device*-Seite kaum doppelt so schnell gerechnet wie auf der CPU/*host*-Seite. Deutlich größer wird der Leistungsunterschied in der Berechnungszeit bei einem Gitter mit 300×20×20 Zellen. Hier wird auf *device*-Seite mehr als dreimal so schnell gerechnet als auf *host*-Seite.

Bei größeren Gittern sinkt das Verhältnis von Gitteroberfläche zu Gittervolumen. Das hat zur Folge, dass das Verhältnis der Anzahl an Zellen, welche eine Randbedingung berücksichtigen müssen im Vergleich zu internen Zellen kleiner wird. Dadurch müssen auf *device*-Seite verhältnismäßig weniger Sonderfälle durch if-Abfragen behandelt werden. Dies führt dazu, dass in Tabelle 2 bei größeren Gittern ein größerer Leistungsvorsprung durch die *device*-seitige Berechnung entsteht.

Die in Tabelle 2 in zahlenmäßig dargestellte Beschleunigung der Berechnung durch CUDA wird in Abbildung 23 als Diagramm grafisch dargestellt. Mit *Anzahl der Freiheitsgrade* wird dabei das Produkt der Anzahl der Zellen mit der *Anzahl der*

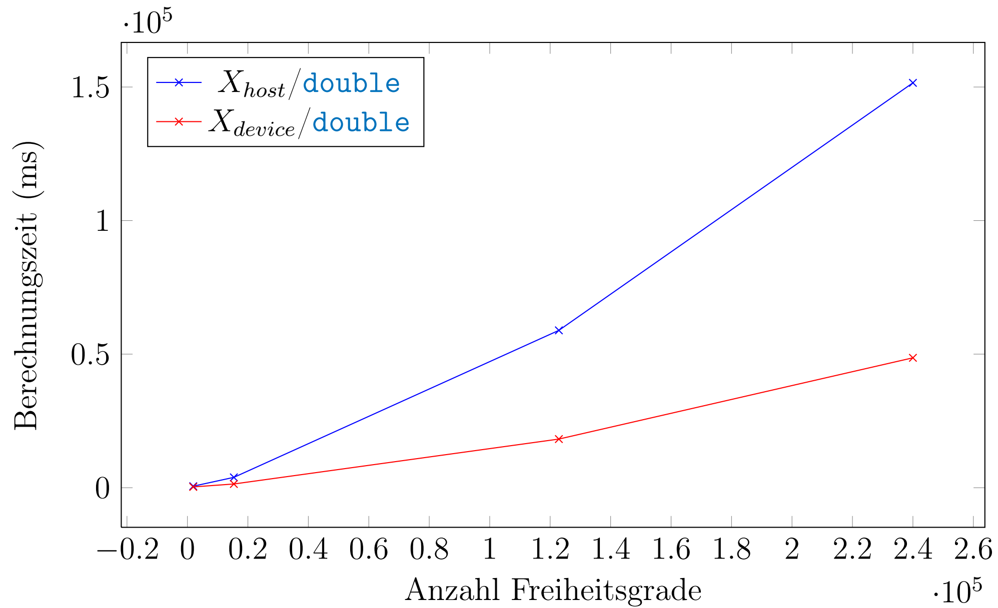


Abbildung 23: Leistungsverhalten bei zunehmender Gittergröße/Freiheitsgraden.

Variablen (=2) bezeichnet. Für ein $X \times Y \times Z$ Gitter errechnet sich diese Zahl mit $X \cdot Y \cdot Z \cdot 2$.

In Abbildung 23 ist deutlich zu erkennen, dass der Geschwindigkeitsvorteil der CUDA-Implementierung X_{device} bei zunehmender Gittergröße an Bedeutung gewinnt (rot), während die Berechnungszeit der CPU-Lösung X_{host} nahezu linear zunimmt (blau).

6 Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde die Aufgabe der Simulation einer Zweiphasen-Strömung auf massiv-parallel arbeitende Hardware portiert. Seriell arbeitende Programme für derartige Simulationen gibt es seit langem. Das Hauptproblem in dieser Arbeit lag darin, die Software auf Basis von CUDA zu implementieren. In der Masterarbeit von Asbjørn Bydal[3] wurde für eine Zweiphasen-Strömung bereits der explizite Teil des IMPES-Lösungsverfahrens, der Euler-Schritt der Sättigungsverteilung, auf die GPU mit CUDA portiert. Der implizite Teil, die Berechnung der Druckverteilung mit einem Gleichungssystem auf CUDA-Basis, wurde jedoch erst in der vorliegenden Arbeit realisiert. Auf diese Weise kann die Berechnung komplett auf *device*-Seite realisiert werden, ohne kostspielige Speichertransfers oder gar Kommunikation mit externen Programmen.

Das Ziel der Berechnungs-Beschleunigung durch die Portierung auf die CUDA-Plattform wurde erreicht. Insbesondere steigt die Beschleunigung mit zunehmender Gitterauflösung. Dieser Effekt hängt zum einen mit dem Verhältnis von Gitteroberfläche zu Gittervolumen zusammen und zum anderen mit einer generell besseren Parallelisierbarkeit größerer Probleme. Dadurch ist es möglich, größere Simulationen effizient zu berechnen.

Das Ziel dieser Arbeit bestand unter anderem darin, die Abweichung zu dem Referenzprogramm DuMu^x[6] gering zu halten. Dies wurde in der Fehleranalyse überprüft und nach diesem Maßstab arbeitet das Simulationsprogramm dieser Arbeit korrekt.

Die numerische Stabilität des Programms hat sich bei der Verwendung von einfacher Gleitkomma-Genauigkeit als problematisch herausgestellt. Das ist bei den verwendeten Verfahren nicht unüblich. Ob und wie sich dies durch Optimierung der Berechnungsabläufe in den Griff bekommen ließe, wäre eine mögliche Weiterentwicklung.

In dieser Arbeit wurden zur Vereinfachung mehrere Einschränkungen für das Problem der Zweiphasenströmung in porösen Medien angenommen. Im mathematischen Modell wurden der Kapillardruck sowie die Gravitation vernachlässigt. Bei der Diskretisierung wurden reguläre, kartesische beziehungsweise äquidistante Berechnungsgitter verwendet. Dies erleichterte insbesondere die Implementierung in CUDA zur Berechnung auf der Grafikkarte/dem *device*.

Zur Weiterentwicklung dieser Arbeit würde sich die Berücksichtigung der Gravitation und des Kapillardrucks im Modell anbieten. Denkbar wäre auch die Möglichkeit, andere Variablen-Kombinationen als S_w und p_n zu verwenden, beispielsweise S_n und p_w oder globale Verteilungen wie $p_{ges} = p_n + p_w$. Auch Modelle, die explizit ein Geschwindigkeitsfeld $\vec{v}_{n/w/ges}$ als Variable in der Simulation verwenden, wären als Erweiterung denkbar.

Eine Simulation räumlich komplexerer Gitterstrukturen würde es ermöglichen, lokal die Genauigkeit zu erhöhen oder kompliziertere Berechnungsgebiete zu erfassen. Eine solche Anpassung würde jedoch auch eine umfangreiche Erweiterung der

CUDA-basierten Teile des Simulationsprogramms nach sich ziehen. Insbesondere muss eine Divergenz des Kontrollflusses und der Speicherzugriffe gering gehalten werden, um Leistungseinbußen in CUDA zu vermeiden. Dies steht aber teilweise dem Ziel komplexer Gitterstrukturen und Randbedingungen entgegen. Optimierung zwischen diesen sich teilweise gegenseitig ausschließenden Zielen birgt großes Erweiterungspotential dieser Arbeit.

Die Randbedingungen bieten ebenfalls Ansätze für mögliche Optimierungen. Um die Kontrollfluss-Divergenz zu verringern, könnte man für die Randbedingungen separate Kernel-Funktionen entwerfen - anstatt mit if-else Anweisungen zu arbeiten.

A Bedienungsanleitung `porSimGPU`

Das Programm zu dieser Arbeit `porSimGPU` wird über die Kommandozeile gesteuert. Eine Liste der verfügbaren Befehle wird mit “`porSimGPU -help`” ausgegeben. Als Parameter muss zwingend die Simulationszeit angegeben werden. Um sichtbare Ergebnisse zu erhalten sollte die Simulationszeit größer als 10^6 sein. Ein einfaches Beispiel für den Programmaufruf: “`porSimGPU 1e7`” für eine Simulationsdauer von 10^7.

Weiter sind folgenden Optionen verfügbar:

`-time <number>`: Simulationszeit angeben - dies kann auch direkt mit einer Zahl als Parameter erfolgen wie in obigem Beispiel.

`-device <number>`: Das CUDA-device, auf welchem gerechnet werden soll, angeben. Diese Option ist nur sinnvoll, wenn mehrere CUDA-fähige Grafikkarten auf einem Computer verfügbar sind.

`-domain <host/device>`: Mit dieser Option kann festgelegt werden, ob die Berechnung auf der CPU (`host`) oder der Grafikkarte (`device`) stattfinden soll. Standardmäßig wird `device` verwendet.

`-maxIterations <number>`: Maximale Anzahl Iterationen, nach welcher die Berechnung abgebrochen werden soll. Standardmäßig nicht spezifiziert - es wird also bis zur angegebenen Simulationszeit gerechnet.

`-writeEvery <number>`: Diese Option spezifiziert nach wie vielen Iterationen eine Ausgabe (Visualisierung oder Datei) erfolgen soll. Standardmäßig erfolgt eine Ausgabe alle 10 Iterationen.

`-writeVTK <0/1>`: Diese Option gibt an, ob die Ausgabe in VTK-Dateien erfolgen soll. Dies ist standardmäßig aktiviert.

`-showVis <0/1>`: Mit dieser Option lässt sich die in-situ Visualisation ein oder ausschalten.

`-volumeVis <0/1>`: Hier lässt sich angeben, ob Volumen-Visualisierung verwendet werden soll - standardmäßig deaktiviert.

`-verbose <0/1>`: Mit dieser Option lassen sich zusätzliche Textausgaben des Programms aktivieren wie beispielweise die Konvergenzraten des Gleichungssystem-Lösers.

`-gridDim <x>:<y>:<z>`: Mit dieser Option lassen sich die Anzahl der Zellen in x, y und z-Richtung angeben. Standardmäßig ist dieser Wert auf `60:4:4` gesetzt.

`-coordinates <x>:<y>:<z>`: Mit dieser Option lassen sich Koordinaten für das Raumgitter angeben. Standardmäßig ist dieser Wert auf `300:60:60` gesetzt.

`-h` oder `-help`: Hilfe ausgeben.

Komplizierteres Aufruf-Beispiel:

```
> porSimGPU -writeEvery 1 -showVis 1 -writeVTK 0 -gridDim 240:16:16 1e7
```

Dieser Aufruf startet die Simulation eines $240 \times 16 \times 16$ Gitters mit einer Simulationszeit $t = 10^7$. Es wird jede Iteration ausgegeben und zwar mit der eingebauten Visualisierung. Dateien im VTK-Format werden nicht geschrieben.

A.1 Steuerung der in-situ Visualisierung

Nachdem die Simulation bis zur Endzeit gerechnet hat, lässt sich das Visualisierungsfenster mit der Maus steuern. Folgende Interatkionen sind dabei möglich:

Rechte Maustaste: Drehen des Berechnungsgitters um seine Achsen (Rotation).

Linke Maustaste: Vergrößern oder verkleinern des Gitters (Zoom).

Mittlere Maustaste: Das Berechnungsgitter bewegen (Translation).

Literatur

[1] Peter Bastian. *Vorlesungsskript: Numerische Lösung partieller Differentialgleichungen*. PhD thesis, Universität Stuttgart, Insitut für Parallele und Verteilte Systeme, 2008.

[2] L. Buatois, G. Caumon, and B. Lévy. *Concurrent Number Cruncher An Efficient Sparse Linear Solver on the GPU. High Performance Computing and Communications: Third International Conference, HPCC 2007*, 2007.

[3] Asbjorn Bydal. *GPU-accelerated simulation of flow through a porous medium*. Master's thesis, University of Adger, 2009.

[4] Zhangxin Chen, Guanren Huan, and Yuanle Ma. *Computational Methods for Multiphase Flows in Porous Media (Computational Science and Engineering 2)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2006.

[5] B. Flemisch, J. Fritz, R. Helmig, J. Niessner, and B. Wohlmuth. *DUMUX: A multi-scale multi-physics toolbox for flow and transport processes in porous media*. 2007.

[6] Institut für Wasserbau, Universität Stuttgart. The DuMu^x – Homepage. `http://www.dumux.uni-stuttgart.de`.

[7] Prof. Dr.-Ing. Rainer Helmig. *Vorlesungsskript: Model concepts and simulation methods for single-phase and multi-phase flow*. PhD thesis, Universität Stuttgart, Institut für Wasserbau, 2008.

[8] NVIDIA. *CUDA Programming Guide 3.0*. `http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf`, 2010.

[9] A. Dedner C. Engwer R. Klöfkorn M. Ohlberger O. Sander P. Bastian, M. Blatt. *A Generic Grid Interface for Parallel and Adaptive Scientific Computing. Part I: Abstract Framework. Computing, 82(2-3)*, pages 103–119, 2008.

[10] A. Dedner C. Engwer R. Klöfkorn R. Kornhuber M. Ohlberger O. Sander P. Bastian, M. Blatt. *A Generic Grid Interface for Parallel and Adaptive Scientific Computing. Part II: Implementation and Tests in DUNE. Computing, 82(2-3)*, pages 121–138, 2008.

Danksagung

Ich danke Herrn Prof. Dr. Thomas Ertl, dass er es mir ermöglichte, am Institut für Visualisierung und Interaktive Systeme diese Arbeit durchzuführen.

Außerdem danke ich meinen beiden Betreuern Dipl.-Inf. Markus Üffinger und Dipl.-Phys. oec. Steffen Müthing für die hilfreiche Unterstützung und das Beantworten meiner zahlreichen Verständnisfragen während dieser Diplomarbeit.

Für ihre Unterstützung während dieser anstrengenden Lebensphase und für das Korrekturlesen danke ich außerdem Frau Katja Förster. Weiter gilt meine Anerkennung den Korrekturlesern Alexey Abel, Felix Schiefer und Gerolf Thomaß.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

____

(Bertram Thomaß)