BERGISCHE
UNIVERSITÄT
WUPPERTAL

Fachbereich E
Elektrotechnik, Informationstechnik, Medientechnik

Lehrstuhl für Automatisierungstechnik / Informatik

Master-Thesis

Implementierung und Tests von Blockglättern auf Grafikkarten mittels OpenCL

Oliver Letterer
722211
Informationstechnologie
Information Science

Wuppertal, den 28. März 2013

Betreuer: Prof. Dr. Matthias Bolten

Erstgutachter: Prof. Dr. Matthias Bolten
Zweitgutachter: Dr. Karsten Kahl

Kurzfassung

Ziel dieser Master-Thesis ist es, eine effiziente Implementierung von Mehrgitterverfahren, die mit Blockglättern arbeiten, auf der GPU zu realisieren.

Dabei wird zunächst in einer Einleitung das Problem definiert, welches von dieser Implementierung gelöst werden soll und mit einigen Beispielen erklärt. Anschließend werden bestehende Mehrgitterverfahren als numerische Lösungsmethoden vorgestellt, die diese Klasse von Problemen lösen können. Es werden Blockglätter vorgestellt, die den bisherigen Glättungsverfahren wie beispielsweise der *Jacobi*- oder der *Red-Black Gauß-Seidel*-Iteration aufgrund von besseren Glättungseigenschaften bei hoher Lokalität überlegen sind. Weiterhin wird die Hardware der GPU vorgestellt, auf welcher die resultierende Implementierung ausgeführt wird. Dabei gibt es eine kurze Einführung in den offenen Standard OpenCL und es werden einige grundlegende architektonische Unterschiede zwischen CPU und GPU vorgestellt. Nachdem die Implementierung inklusive zweier unterschiedlicher Speicherhierarchien vorgestellt wird, folgen einige numerische Resultate, die zeigen, wie erfolgreich und effizient die entstandene Implementierung ist.

Abstract

The goal of this Master Thesis is to implement an efficient Multigrid algorithm based on Block-Smoothing on the GPU using OpenCL. First is a presentation of the Multigrid problem along with the discussion of additional examples. The existing Multigrid algorithms are explained, and an insight is provided into the motivation for choosing Block-Smoothing methods, comparing other iterative methods like Jacobi- or Red-Black Gauß-Seidel-Iteration that exhibit high locality with regard to smoothing properties. In order to facilitate a deeper understanding of the implementation, the GPU hardware is examined and it's inherent differences to the CPU are outlined, as well as a discussion about the usage of the open standard specification OpenCL. The final implementation using two different configurations of memory will be presented, along with numerical results to demonstrate the efficiency of the algorithms.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Problemklassifikation

Diese Master-Thesis beschäftigt sich mich mit numerischen Lösungsmethoden für elliptische partielle Differentialgleichungen. Darum sollen diese zuerst definiert und mit einem Beispiel motiviert werden. Diese Einführung basiert auf dem Vorlesungsskript „Numerik partieller Differentialgleichungen“[AER05], welches Anfang 2005 erstellt wurde.

Definition 1.1 (Partielle Differentialgleichung) *Eine Gleichung, die partielle Ableitungen einer Funktion $u : \Omega \to \mathbb{R}, n \in \mathbb{N}$ und $\Omega \subseteq \mathbb{R}^n$ ein Gebiet, enthält, heißt partielle Differentialgleichung (PDE):*

$$F\left(x_1, x_2, ..., x_n, u, \frac{\partial u}{\partial x_1}, \frac{\partial u}{\partial x_2}, ..., \frac{\partial u}{\partial x_n}, \frac{\partial^2 u}{\partial x_1 \partial x_2}, ...\right) = 0. \tag{1.1}$$

Es gibt verschiedene Formen von PDE's, die sich in ihren Eigenschaften grundlegend unterscheiden und unterschiedliche numerische Lösungsmethoden erfordern.

Definition 1.2 (Allgemeine lineare PDE 2. Ordnung) *Sei $n \in \mathbb{N}$*

$$x \in \mathbb{R}^n, u : \Omega \to \mathbb{R}, \Omega \subseteq \mathbb{R}^n \text{ ein Gebiet}$$

$$A(x) \in \mathbb{R}^{n \times n}, b(x) \in \mathbb{R}^n, c(x) \in \mathbb{R}, f(x) \in \mathbb{R}.$$

Dann lautet die allgemeine lineare partielle Differentialgleichung 2. Ordnung

$$\underbrace{< A(x)\nabla, \nabla u >}_{Hauptteil} + < b(x), \nabla u > + c(x)u = f(x), \tag{1.2}$$

mit $\nabla = \left(\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, ..., \frac{\partial}{\partial x_n}\right)^T$.

Definition 1.3 (Elliptische PDE 2. Ordnung) *Sei eine lineare PDE 2. Ordnung wie in Definition 1.2 gegeben und seien $\lambda_j(A(x))$ für $j = 1, 2, ..., n$ die Eigenwerte von $A(x)$.*
Die allgemeine lineare PDE 2. Ordnung heißt elliptisch, genau dann wenn alle Eigenwerte von $A(x)$ dasselbe Vorzeichen haben:

$$\lambda_j(A(x)) > 0 \ \forall j = 1, 2, ..., n \text{ oder } \lambda_j(A(x)) < 0 \ \forall j = 1, 2, ..., n.$$

$A(x)$ ist dann positiv oder negativ definit.

Um eine solche partielle Differentialgleichung eindeutig lösen zu können, benötigt es in der Regel zusätzliche Randbedingungen. In dieser Master-Thesis sind das ausschließlich Dirichlet Randbedingungen:
Zusätzlich zur partiellen Differentialgleichung wird der Wert der gesuchten Funktion u auf dem Rand des betrachteten Gebietes $\partial\Omega$ als analytische Funktion mit

$$u(x) = \gamma(x) \text{ für } x \in \partial\Omega$$

vorgegeben.

1.2 Beispiele

An dieser Stelle folgt ein Beispiel für eine elliptische PDE (die Poisson-Gleichung) mit der Herleitung einer analytischen Lösung einer konkreten Laplace-Gleichung.

Beispiel 1.1 (Poisson-Gleichung) *Die Poisson-Gleichung $\Delta u(x) = f(x)$, mit $u : \mathbb{R}^n \to \mathbb{R}, f(x) \in \mathbb{R}$ ist elliptisch, denn*

$$\Delta u(x) = \sum_{i=1}^{n} \frac{\partial^2}{\partial x_i^2} u(x) =< \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & \dots & 0 & 1 \end{pmatrix} \nabla, \nabla u(x) > \ =: \ < A(x)\nabla, \nabla u(x) > .$$

$A(x)$ besitzt ausschließlich den Eigenwerte $1 > 0$.

Die Poisson-Gleichung spielt beispielsweise bei der Lösung einiger wichtiger physikalischer Probleme eine große Rolle. Im Teilgebiet der Elektrostatik erfüllt der elektrische Fluss Φ, welcher über $E(x) = -\nabla\Phi(x)$ mit dem elektrischen Feld E einer elektrostatischen Anordnung zusammenhängt, die Poisson-Gleichung:

$$\Delta\Phi(x) = -\frac{\rho(x)}{\epsilon}.$$

Dabei ist $\rho(x)$ die ortsabhängige Ladungsdichte und ϵ die Permittivität.

Beispiel 1.2 (Laplace-Gleichung auf Einheitsquadrat in $\mathbb{R}^2$) *Bei der homogenen Poisson-Gleichung (rechte Seite = 0) spricht man von der Laplace-Gleichung. Gesucht ist die unbekannte Funktion $u : E_2 \to \mathbb{R}$ im Einheitsquadrat $E_2 = [0,1]^2 \subseteq \mathbb{R}^2$ mit $\Delta u = 0$ im inneren von E_2 und Dirichlet Randwertvorgabe*

$$u(x,0) = u(x,1) = 0 \text{ für } x \in [0,1]$$

und

$$u(0,y) = f(y), u(1,y) = 0 \text{ für } y \in [0,1].$$

Als Lösungsansatz wird hier Separation der Variablen gewählt. Annahme: $u(x,y) = v(x) \cdot w(y)$. Somit wird die PDE zu

$$v''(x)w(y) + v(x)w''(y) = 0$$

bzw.

$$\frac{v''(x)}{v(x)} = -\frac{w''(y)}{w(y)}$$

wenn angenommen werden kann, dass v und w im inneren von E_2 ungleich 0 sind. Da die linke Seite des Gleichheitszeichens nur von x und die rechte Seite nur von y abhängt, bedeutet das, dass beide Seiten gleich einer Konstanten seien müssen. Für $w(y)$ ergibt sich dann in Kombination mit der Randwertvorgabe

$$w(y) = w_k(y) = c_k sin(\alpha_k y), \text{ mit } \alpha_k = k\pi, k \in \mathbb{N}$$

und für $v(x)$

$$v(x) = v_k(x) = \tilde{c}_k sinh(\alpha_k(x-1)), \text{ mit } \alpha_k = k\pi, k \in \mathbb{N}.$$

Mittels Superposition erhält man eine allgemeine Lösung für die Laplace-Gleichung:

$$u(x,y) = \sum_{k=1}^{\infty} b_k sinh(\alpha_k(x-1))sin(\alpha_k y).$$

Mittels der Randwertvorgabe folgt eine weitere Bedingung an die unbekannten Koeffizienten

$$f(y) = u(0,y) = \sum_{k=1}^{\infty} \beta_k sin(\alpha_k y), \text{ mit } \beta_k = -b_k sinh(\alpha_k).$$

Die β_k sind gerade die Fourier-Koeffizienten der ungeraden und periodischen Fortführung von $f(y)$. Somit ist die Lösung des Dirichlet-Problems eindeutig beschrieben.

Das Aufstellen einer konkreten Lösung im obigen Beispiel fällt besonders einfach, wenn die Funktion $f(y)$ aus einer oder mehreren diskreten harmonischen Schwingungen $sin(\alpha_k y)$ besteht, wobei man hierbei die Koeffizienten einfach ablesen kann. Es wird komplizierter, wenn man die Fourier-Koeffizienten nicht mehr explizit ablesen kann sondern diese berechnen muss. Ebenfalls kann das Gebiet Ω beliebig komplizierte Formen annehmen oder es muss nicht mehr die Laplace-Gleichung sondern die Poisson-Gleichung mit beliebig komplizierter inhomogener rechter Seite gelöst werden. Aus diesem Grund werden im Folgenden Kapitel numerische Methoden vorgestellt werden, mit denen allgemeine elliptische partielle Differentialgleichungen, wie in Gleichung 1.2 beschrieben, gelöst werden können.

2 Numerische Methoden

Ziel dieses Kapitels ist es, eine grobe Einführung in die numerische Lösung von partiellen Differentialgleichungen mit Hilfe von sogenannten Mehrgitterfverfahren zu geben. Dabei basiert diese Einführung auf dem Buch „A Multigrid Tutorial"[BHM00]. Aus historischen Gründen wird hier zur Einführung dieser Methoden ein konkretes Dirichlet Randwertproblem in bis zu zwei Dimensionen als Modellproblem betrachtet. Zunächst das Dirichlet Randwertproblem für eine Dimension:

$$\begin{aligned} -u''(x) + \sigma u(x) &= f(x) \text{ mit } 0 < x < 1, \sigma \geq 0 \\ u(0) = u(1) &= 0. \end{aligned} \tag{2.1}$$

Numerische Methoden zur Lösung der allgemeinen elliptischen Gleichung 1.2 in höheren Dimensionen lassen sich aus der Theorie dieser hier betrachteten Randwertprobleme einfach erweitern und konstruieren. Zum Verständnis der Methoden soll diese vereinfachte Form einer elliptischen PDE genügen. Zur numerischen Lösung eines solchen Randwertproblems dient die *Finite-Differenzen-Methode*. Dabei wird das Gebiet Ω in $n \in \mathbb{N}$ diskrete Teilintervalle mit Gitterpunkten $x_j = j \cdot h$, wobei h die Schrittweite $h = \frac{1}{n}$ ist, zerlegt. Es resultiert das diskrete Gebiet Ω_h. Gesucht wird eine Approximation der Lösung u in den Gitterpunkten des diskreten Gebietes: $v_j \approx u(x_j)$. Bedingungen an die Approximationen v_j werden erhalten, indem die partielle Differentialgleichung in den Gitterpunkten diskretisiert wird. Bei diesem Diskretisierungsprozess werden alle partielle Ableitungen durch finite Differenzen [AER05] approximiert. Dabei sei $v := (v_1, v_2, ..., v_{n-1})^T$. Für das Modellproblem 2.1 ergibt sich dann:

$$\begin{aligned} \frac{-v_{j-1} + 2v_j - v_{j+1}}{h^2} + \sigma v_j &= f(x_j), \text{ für } j = 1, 2, ..., n-1 \\ v_0 = v_n &= 0. \end{aligned} \tag{2.2}$$

Hieraus resultiert mit $f = (f(x_1), f(x_2), ..., f(x_{n-1}))^T = (f_1, f_2, ..., f_{n-1})^T$ das lineares Gleichungssystem der Form

$$\underbrace{\frac{1}{h^2}\begin{pmatrix} 2+\sigma h^2 & -1 & & & & \\ -1 & 2+\sigma h^2 & -1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & -1 & \\ & & & -1 & 2+\sigma h^2 \end{pmatrix}}_{A} \cdot \underbrace{\begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ \vdots \\ v_{n-1} \end{pmatrix}}_{v} = \underbrace{\begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ \vdots \\ f_{n-1} \end{pmatrix}}_{f},$$

wobei $A \in \mathbb{R}^{(n-1)\times(n-1)}$ Tridiagonalgestalt hat.
Analog hierzu lässt sich das Modellproblem für den 2D-Fall im Einheitsquadrat formulieren:

$$\begin{aligned} -u_{xx} - u_{yy} + \sigma u &= f, \text{ für } (x,y) \in (0,1)^2 \\ u(x,y) &= 0, \text{ für } (x,y) \in \partial\Omega. \end{aligned} \tag{2.3}$$

Die Diskretisierung erfolgt mittels zwei unabhängigen Schrittweiten $m, n \in \mathbb{N}$ in einem diskreten Gebiet bestehend aus den Gitterpunkten $(x_j, y_i) = (jh_x, ih_y)$, $h_x = \frac{1}{m}, h_y = \frac{1}{n}$. Man erhält die folgenden Bedingungen an die Approximationen $v_{i,j} \approx u(x_i, x_j)$:

$$\frac{-v_{i-1,j} + 2v_{i,j} - v_{i+1,h}}{h_x^2} + \frac{-v_{i,j-1} + 2v_{i,j} - v_{i,j+1}}{h_y^2} + \sigma v_{i,j} = f_{i,j},$$
$$v_{i,0} = v_{i,n} = v_{0,j} = v_{m,j} = 0, \text{ für } 1 \le i \le m-1, 1 \le j \le n-1.$$

Auch diese Diskretisierung resultiert in einem linearen Gleichungssystem der Form $A \cdot v = f$:

$$\begin{pmatrix} B & -aI & & & \\ -aI & B & -aI & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & -aI \\ & & & -aI & B \end{pmatrix} \cdot \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ \vdots \\ v_{m-1} \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ \vdots \\ f_{m-1} \end{pmatrix},$$

wobei $a = \frac{1}{h_x^2}$, $B \in \mathbb{R}^{(n-1)\times(n-1)}$ eine Matrix mit Tridiagonalgestalt, die der tridiagonalen Matrix aus dem eindimensionalem Problem ähnelt und I die Einheitsmatrix ist. Die Matrix A ist dünnbesetzt, hat Block-Tridiagonalgestalt und die Blockgröße $(m-1) \times (m-1)$.
Eine solches lineares Gleichungssystem, welches aus einer Diskretisierung entsteht,

wird durch einen Differenzenstern charakterisiert. Dieser wäre für das eindimensionale Problem

$$A = \frac{1}{h^2}\begin{pmatrix} -1 & 2+\sigma h^2 & -1 \end{pmatrix}$$

und für das zweidimensionale Modellproblem

$$A = \frac{1}{h^2}\begin{pmatrix} & -1 & \\ -1 & 4+\sigma h^2 & -1 \\ & -1 & \end{pmatrix},$$

wenn in x- und y- Richtung äquidistante Schrittweiten vorausgesetzt werden: $h_x = h_y = h$.
Um die Approximationen v_i für die Lösung der PDE zu bestimmen, muss das entsprechende lineare Gleichungssystem gelöst werden. Dies könnte beispielsweise mittels Gaußelimination oder QR-Zerlegung [PG12] geschehen. Dafür müssten jedoch alle Einträge der Matrix, inklusive aller vorhandenen Nullen, explizit gespeichert werden. Dies ist bei einem Gleichungssystem mit einer dünnbesetzten Matrix wie sie hier vorliegt ineffizient, da diese fast ausschließlich mit Nullen gefüllt ist. Aus diesem Grund werden im nächsten Schritt iterative Lösungsmethoden vorgestellt, mit denen sich dünnbesetzte lineare Gleichungssysteme effizienter lösen lassen können.

2.1 Iterative Löser

Zur Vorstellung von iterativen Lösungsverfahren soll zunächst eine neue Notation eingeführt werden. Im Vergleich zu den oben beschriebenen partiellen Differentialgleichungen sei u nicht mehr die gesuchte exakte Lösung der partiellen Differentialgleichung, sondern die exakte Lösung des linearen Gleichungssystems

$$A \cdot u = f,$$

welches aus der Diskretisierung der partiellen Differentialgleichung kommt. v sei dabei eine Approximation der exakten Lösung u des linearen Gleichungssystems (welche zum Beispiel aus einer iterativen Methode stammt). v_j und u_j seien dabei die Komponenten der Vektoren v respektive u. Der exakte Fehler e ist damit gegeben durch $e = u - v$. Da dieser in der Praxis jedoch oft unbekannt ist (da die exakte Lösung unbekannt ist), dient das Residuum als Maß für die Qualität der Lösung: $r = f - A \cdot v$. Der Zusammenhang zwischen exaktem Fehler und Residuum in der *Residuums-Gleichung* ist schnell gezeigt:

$$A \cdot e = r. \tag{2.4}$$

Diese Gleichung wird im späteren Verlauf dieser Master-Thesis noch eine größere Rolle spielen.

Zur Motivation der iterativen Lösungsmethoden wird hier das Modellproblem 2.1 mit $\sigma = 0$ verwendet. Umgeschrieben ergibt sich dies zu

$$\begin{aligned} -u_{j-1} + 2u_j - u_{j+1} &= h^2 f_j, \text{ für } 1 \leq j \leq n-1 \\ u_0 = u_n &= 0. \end{aligned} \tag{2.5}$$

Die Idee hinter der *Jacobi-Iteration* – einem ersten iterativen Löser für dünnbesetzte lineare Gleichungssysteme – ist es, die exakte Lösung u_j zu approximieren, indem man die Gleichung für die j-te Unbekannte nach der j-ten Unbekannten auflöst. Als Approximationen für die Unbekannten u_{j-1} und u_{j+1} dienen dabei die aktuellen Approximationen v_{j-1} und v_{j+1}. Somit ergibt sich das iterative Schema:

$$v_j^{(1)} = \frac{1}{2}(v_{j-1}^{(0)} + v_{j+1}^{(0)} + h^2 f_j), \text{ für } 1 \leq j \leq n-1. \tag{2.6}$$

Der unbekannte Anfangswert $v^{(0)}$ muss häufig geraten werden. Oft wird hierfür 0 verwendet. Sobald alle $v_j^{(1)}$ berechnet worden sind, wird der Iterationsschritt wiederholt. Dies geschieht solange, bis eine hinreichend gute Näherung für die exakte Lösung u gefunden wurde.
Zur numerischen Analyse dieser und noch weiteren Methoden bedient man sich der *Splittingmethoden*. Dabei wird die Matrix A in unterschiedliche Matrizen zerlegt:

$$A = D - L - U. \tag{2.7}$$

Dabei ist D die Diagonale von A und $-L$ und $-U$ die untere respektive obere Dreiecksmatrix von A. Allgemein lassen sich Splittingmethoden über eine allgemeinere Zerlegung der Matrix A definieren. Für diese Master-Thesis soll die Zerlegung 2.7 jedoch genügen. Fasst man h^2 aus 2.5 mit f zusammen, so wird das System

$$(D - L - U)u = f$$

gelöst. Umformen

$$Du = (L + U)u + f$$

und anschließende Multiplikation mit D^{-1} liefert

$$u = D^{-1}(L + U)u + D^{-1}f.$$

Die Multiplikation mit D^{-1} entspricht der Umformung der j-ten Gleichung nach u_j. Somit erhält man ein iteratives Schema mit (in diesem Fall) der Jacobi-Iterationsmatrix

$$R_J = D^{-1}(L + U).$$

Eine Erweiterung zur Jacobi-Iteration ist die gedämpfte Jacobi-Iteration. Die bei der Jacobi-Iteration berechnete Iterierte $v^{(1)}$ wird genau wie bei der Jacobi-Iteration berechnet, ist hier jedoch nur ein Zwischenwert v_j^* der neuen Iterierten:

$$v_j^* = \frac{1}{2}(v_{j-1}^{(0)} + v_{j+1}^{(0)} + h^2 f_j), \text{ für } 1 \leq j \leq n-1.$$

Die neue Iterierte $v^{(1)}$ setzt sich aus dem gewichteten Durchschnitt

$$v^{(1)} = (1-\omega)v_j^{(0)} + \omega v_j^*, \text{ für } 1 \leq j \leq n-1$$

mit $\omega \in \mathbb{R}$ zusammen. Für $\omega = 1$ ergibt sich wieder die Jacobi-Iteration. Für die gedämpfte Jacobi-Iterations ergibt sich die Iterationsmatrix

$$R_\omega = (1-\omega)I + \omega R_J.$$

Ein weiteres wichtiges Iterationsverfahren ist das *Gauß-Seidel-Verfahren*. Dabei wird jede Approximation $v_j^{(1)}$, die errechnet wurde, direkt verwendet und nicht erst, sobald alle Unbekannten $v_j^{(0)}$ bekannt sind. Dabei wird der Wert von v_j mit jeder Iteration überschrieben durch $\frac{1}{2}(v_{j-1} + v_{j+1} + h^2 f)$. Die korrespondierende Splittingmethode erhält man, indem nicht wie bei der Jacobi-Iteration mit D^{-1} sondern mit $(D-L)^{-1}$ multipliziert wird:

$$(D-L)u = Uu + f \Rightarrow u = (D-L)^{-1}Uu + (D-L)^{-1}f.$$

Die entsprechende Iterationsmatrix ist

$$R_G = (D-L)^{-1}U.$$

Eine dem Gauß-Seidel-Verfahren ähnliches Verfahren ist das *Red-Black Gauß-Seidel-Verfahren*. Dabei wird das Gebiet Ω_h in gerade Punkte v_{2j} und ungerade Punkte v_{2j+1} unterteilt. Die geraden Punkte lassen sich ausschließlich mit den ungeraden Punkten und die ungeraden Punkte lassen sich ausschließlich mit den geraden Punkten berechnen. Dies hat für parallele Anwendungen eine große Bedeutung, da so alle geraden Punkte unabhängig voneinander berechnet werden können. Anschließend können alle ungeraden Punkte ebenfalls unabhängig voneinander berechnet werden. Man kann alle der oben beschriebenen Methoden durch ein Schema der Form

$$v^{(1)} = R \cdot v^{(0)} + g$$

beschreiben, wobei R die Iterationsmatrix und g eine entsprechende rechte Seite ist. Hieraus ergibt sich unmittelbar für den Fehler e nach einer Iteration

$$e^{(1)} = R \cdot e^{(0)}.$$

und nach m Iterationen:

$$e^{(m)} = R^m e^{(0)}.$$

Damit ist der Fehler eines solchen iterativen Verfahrens in einer Vektornorm mit Hilfe einer verträglichen Matrixnorm beschränkt durch

$$\left\|e^{(m)}\right\| \leq \|R\|^m \left\|e^{(0)}\right\|.$$

Damit konvergiert das iterative Verfahren genau dann, wenn $\|R\| < 1$ ist. Dies ist genau dann der Fall, wenn der Spektralradius $\rho(R) < 1$ ist [Fro05]. $\rho(R) < 1$ bedeutet, dass alle Eigenwerte von R im Betrag kleiner als 1 sein müssen. $\rho(R)$ heißt in diesem Kontext auch *asymptotischer Konvergenzfaktor*. Soll der Fehler um einen Faktor 10^{-d} gedämpft werden, dann gilt:

$$\frac{\left\|e^{(m)}\right\|}{\|e^{(0)}\|} \leq 10^{-d}.$$

Dies kann approximiert werden durch

$$\rho(R)^m \leq 10^{-d} \Leftrightarrow m \geq -\frac{d}{log_{10}|\rho(R)|}.$$

Geht $\rho(R)$ gegen 1, so nimmt die Konvergenzrate ab. Für die Jacobi-Iteration im Modellproblem mit

$$A = \begin{pmatrix} 2 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & \cdot & \cdot & \cdot & & \\ & & \cdot & \cdot & \cdot & \\ & & & \cdot & \cdot & -1 \\ & & & & -1 & 2 \end{pmatrix}$$

ergibt sich die Iterationsmatrix

$$R = \begin{pmatrix} 0 & 1 & & & & \\ 1 & 0 & 1 & & & \\ & \cdot & \cdot & \cdot & & \\ & & \cdot & \cdot & \cdot & \\ & & & \cdot & \cdot & 1 \\ & & & & 1 & 0 \end{pmatrix}.$$

mit den Eigenwerte $\lambda_k(R) = cos\left(\frac{k\pi}{N+1}\right)$ und entsprechenden Eigenvektoren $w_{k,j}$ [Str06]. Betrachtet man den Anfangsfehler $e^{(0)}$ in der Eigenwertzerlegung der Iterationsmatrix

$$e^{(0)} = \sum_{k=1}^{n-1} c_k w_k$$

mit entsprechenden Koeffizienten c_k, so ergibt sich unmittelbar für den Fehler $e^{(m)}$ nach m Iterationen

$$e^{(m)} = R^m e^{(0)} = \sum_{k=1}^{n-1} c_k R^m w_k = \sum_{k=1}^{n-1} c_k \lambda_k^m w_k.$$

Der Anteil des k-ten Eigenvektors im Fehler $e^{(0)}$ wird nach m Iterationen um den Faktor λ_k^m gedämpft. Je näher der Eigenwert bei 1 liegt, desto schlechter wird der Anteil des entsprechenden Eigenvektors gedämpft. Diese Anteile gehören beispielsweise bei der gedämpften Jacobi-Iteration zu den niederfrequenten Eigenvektoren. Je näher der Eigenwert bei 0 liegt, desto besser wird der Anteil des entsprechenden Eigenvektors gedämpft. Bei der gedämpften Jacobi-Iteration gehören diese Anteile zu den höherfrequenten Eigenvektoren. Je feiner das Gebiet diskretisiert wird, um eine möglichst gute Annäherung an die exakte Lösung der ursprünglichen partiellen Differentialgleichung zu erhalten, desto mehr Eigenwerte sind nahe bei 1 und werden schlecht gedämpft. Die Methode konvergiert langsam. Je gröber das Gebiet diskretisiert wird, desto weniger Eigenwerte liegen bei 1 und desto schneller kann der iterative Löser das Problem bis auf Diskretisierungsfehler hin lösen.
Zusammenfassend kann gesagt werden, das hochfrequente Komponenten des Fehlers gut gedämpft werden und niederfrequente Komponenten des Fehlers nicht sehr gut. Aus diesem Grund sagt man auch, dass der Fehler durch einen iterativen Löser geglättet wird. Dies wurde zwar nur für die gedämpfte Jacobi-Iteration eines speziellen Modellproblems verdeutlicht, jedoch weisen viele iterative Löser diese Glättungseigenschaft auf [BHM00, TOS01]. Ziel der Mehrgitterverfahren ist es, diesen Sachverhalt zu verbessern.

2.2 Mehrgitterverfahren

Zur Motivation der Mehrgitterverfahren betrachtet man an dieser Stelle das Modellproblem 2.5 auf dem Gebiet Ω^h, auf das ein iterativer Löser angewandt wurde, bis nur noch niederfrequente Komponenten des Fehlers vorhanden sind. Die zentrale Überlegung ist es, wie dieser Fehler auf einem gröberen Gitter Ω^{2h} aussieht, welches mit doppelter Schrittweite $2h$ diskretisiert wurde.
Der Fehler auf dem feinen Gitter gemäß Abbildung 2.1 sieht, projiziert auf ein gröberes Gitter wie in Abbildung 2.2, auf dem gröberen Gitter deutlich hochfrequenter aus. Diese Erkenntnis impliziert die grundlegende Idee hinter den Mehrgitterverfahren. Wenn ein iterativer Löser auf einem Gitter nicht mehr stark konvergiert, da alle hochfrequenten Fehlerkomponenten gedämpft wurden, so wechselt man auf ein gröberes Gitter, auf dem noch verbleibende niederfrequente Fehlerkomponenten hochfrequenter aussehen.

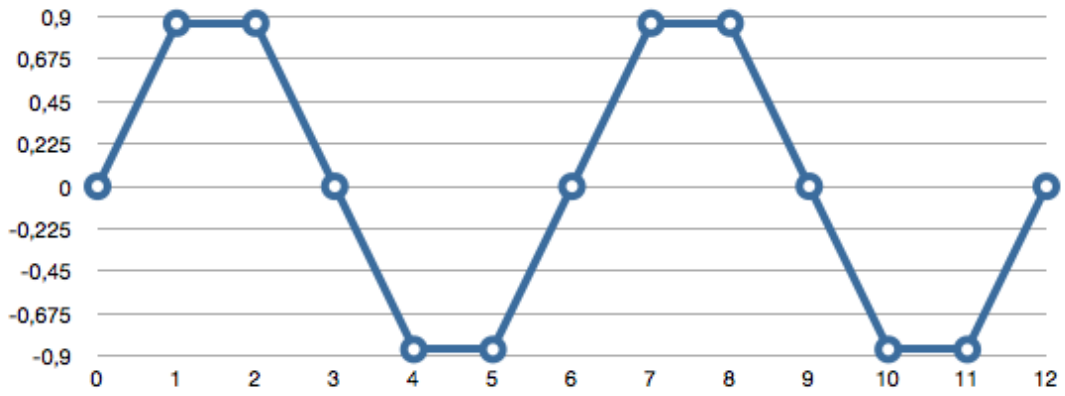

Abbildung 2.1: Fehler auf einem feinen Gitter Ω^h

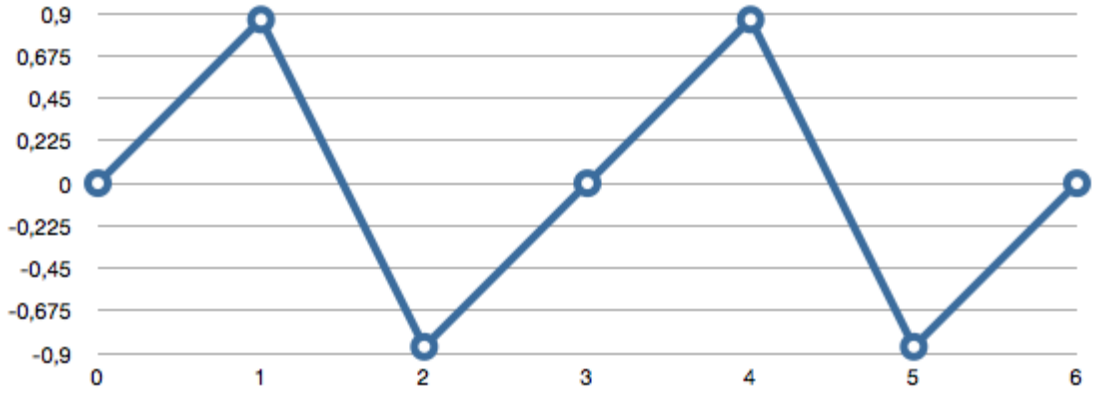

Abbildung 2.2: Fehler aus 2.1 projiziert auf ein gröberes Gitter Ω^{2h}

Auf dem gröberen Gitter verwendet man dann für den iterativen Löser anstelle der eigentlichen Gleichung $Av = f$ die Residuums-Gleichung $Ae = r$. Denn die Lösung der ursprünglichen Gleichung $Av = f$ mit einem willkürlich gewählten Anfangswert $v^{(0)}$ ist äquivalent dazu, die Residuums-Gleichung $Ae = r$ mit Anfangswert $e^{(0)} = 0$ zu lösen und anschließend den gefunden Fehler e zur bisherige Lösung v zu addieren: $v \leftarrow v + e$, wobei $\leftarrow$ andeuten soll, dass v mit dem Wert $v + e$ überschrieben wird. Dieser Ansatz führt zu einer allgemeinen Formulierung eines Korrekturschemas:

Definition 2.1 (Abstraktes Formulierung eines Korrekturschemas)

- *Iteriere $Au = f$ auf Ω^h um eine Approximation v^h zu erhalten.*
- *Berechne das Residuum $r = f - Av^h$.*
- *Iteriere die Residuums-Gleichung $Ae = r$ auf dem gröberen Gitter Ω^{2h} und erhalte eine Approximation für den Fehler e^{2h}.*
- *Die Approximation v^h wird mit der Approximation für den Fehler auf dem Gitter Ω^{2h} korrigiert: $v^h \leftarrow v^h + e^{2h}$.*

An dieser Stelle ist es passend zu untersuchen, wie genau der Transfer von einem feinen zu einem groben Gitter und umgekehrt von statten geht. Dabei soll sich an dieser Stelle nur auf Gitterpaare beschränkt werden, deren feines Gitter $2\times$ so viele Punkte wie das grobe Gitter hat.
Der Transfer von einem groben hin zu einem feinen Gitter ist ein gängiges Problem aus der Numerik: Die Interpolation bzw. Prolongation. Für die meisten Anwendungen in der Praxis genügt die lineare Interpolation [TOS01, BHM00] zwischen den Gitterpunkten:

$$\begin{aligned} v_{2j}^h &= v_j^{2h} \\ v_{2j+1}^h &= \frac{1}{2}\left(v_j^{2h} + v_{j+1}^{2h}\right), \text{ für } 0 \leq j \leq \frac{n}{2} - 1. \end{aligned} \tag{2.8}$$

Dabei wird der Wert für einen feinen Gitterpunkt mit geradem Index v_{2j}^h auf Ω^h übernommen vom Punkt v_j^{2h} auf dem groben Gitter Ω^{2h}. Die Punkte mit ungeradem Index v_{2j+1}^h werden linear zwischen ihren beiden Nachbarpunkten links und rechts interpoliert. Dieser Vorgang kann durch den Interpolationsoperator I_{2h}^h beschrieben werden, der die Vektoren v^{2h} und v^h über $I_{2h}^h \cdot v^{2h} = v^h$ miteinander verknüpft. Auf

einem eindimensionalen Gitter mit $n = 8$ ergibt sich dies wie folgt:

$$I_{2h}^{h} \cdot v^{2h} = \frac{1}{2} \begin{pmatrix} 1 & & & \\ 2 & & & \\ 1 & 1 & & \\ & 2 & & \\ & 1 & 1 & \\ & & 2 & \\ & & 1 & \end{pmatrix} \cdot \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix}_{2h} = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \end{pmatrix}_{h} = v^h. \tag{2.9}$$

Um zu verstehen, ob und wie gut eine solche Interpolation funktioniert, soll das folgende Gedankenexperiment genügen. Zunächst betrachtet man den Fehler auf dem feinen Gitter Ω^h unter der Annahme, dass dieser glatt ist (glatt bedeutet in diesem Kontext, dass keine oder sehr schwache/ wenige hochfrequente Komponenten im Fehler enthalten sind). Zusätzlich sei der exakte Fehler auf dem groben Gitter Ω^{2h} bereits berechnet worden. Dann ist die Interpolation des Fehlers e^{2h} auf das feine Gitter ebenfalls glatt und eine relativ gute Approximation für den exakten Fehler e^h auf dem feinen Gitter, da sich aufgrund der Glattheit des ursprünglichen Fehlers die Punkte mit ungeradem Index auf dem feinen Gitter nur wenig von den interpolierten Werten unterscheiden. In diesem Gedankenexperiment unterscheiden sich die Gitterpunkten mit geradem Index zwischen exaktem Fehler e^h und interpoliertem Fehler e^{2h} nicht. Ist der exakte Fehler auf e^h nicht mehr glatt (der Fehler ist hochfrequenter und hat mehr Oszillationen), so ist zu erwarten, dass die Interpolation in den meisten Fällen in den interpolierten Gitterpunkten mit ungeradem Index nicht exakt oder ausreichend gut ist. Somit ist die Interpolation für den Fehler am effizientesten, wenn der Fehler, der interpoliert wird, möglichst glatt ist, also keine hochfrequenten Komponenten und Oszillationen mehr enthält. Dieses steht im Kontrast dazu, dass die iterativen Lösungsmethoden am effizientesten sind, wenn der Fehler oszillatorisch mit vielen hochfrequenten Komponenten ist. Heuristisch gesehen ist zu erwarten, dass sich diese beiden Methoden gut ergänzen.
Der Vollständigkeit halber soll an dieser Stelle ebenfalls die lineare Interpolation auf einem zweidimensionalen Gitter angegeben werden:

$$\begin{aligned} v_{2i,2j}^{h} &= v_{i,j}^{2h} \\ v_{2i+1,2j}^{h} &= \frac{1}{2}\left(v_{i,j}^{2h} + v_{i+1,j}^{2h}\right) \\ v_{2i,2j+1}^{h} &= \frac{1}{2}\left(v_{i,j}^{2h} + v_{,j+1}^{2h}\right) \\ v_{2i+1,2j+1}^{h} &= \frac{1}{4}\left(v_{i,j}^{2h} + v_{i+1,j}^{2h} + v_{i,j+1}^{2h} + v_{i+1,j+1}^{2h}\right), \text{ für } 0 \leq i,j \leq \frac{n}{2} - 1. \end{aligned} \tag{2.10}$$

Der andere Fall von Gittertransferoperator, der noch zu wählen ist, kommt beim Transfer vom feinen zum groben Gitter zum Einsatz. Dieser soll durch den Operator

I_h^{2h} beschrieben werden. Der einfachste Fall für den Restriktionsoperator ist der der Injektion, wobei jeder Wert eines groben Gitterpunktes direkt vom darüber liegenden feinen Gitterpunkt übernommen wird:

$$v_j^{2h} = v_{2j}^h. \tag{2.11}$$

Eine Alternative zum Injektionsoperator stellt das *full-weighting* dar:

$$v_j^{2h} = \frac{1}{4}\left(v_{2j-1}^h + 2v_{2j}^h + v_{2j+1}^h\right), \text{ für } 0 \leq j \leq \frac{n}{2} - 1. \tag{2.12}$$

Jeder grobe Gitterpunkt errechnet sich dabei aus dem Mittelwert der ihn umliegenden feinen Gitterpunkte. Für den Operator I_h^{2h} ergibt sich dann:

$$I_h^{1h} \cdot v^h = \frac{1}{4}\begin{pmatrix} 1 & 2 & 1 & & & & \\ & & 1 & 2 & 1 & & \\ & & & & 1 & 2 & 1 \end{pmatrix} \cdot \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \end{pmatrix}_h = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix}_{2h} = v^{2h}. \tag{2.13}$$

Für den zweidimensionalen Fall ergibt sich der Restriktionsoperator mit full-weighting zu

$$\begin{aligned} v_{i,j}^{2h} = \frac{1}{16}\big[& v_{2i-1,2j-1}^h + v_{2i-1,2j+1}^h + v_{2i+1,2j-1}^h + v_{2i+1,2j+1}^h \\ & +2(v_{2i,2j-1}^h + v_{2i,2j+1}^h + v_{2i-1,2j}^h + v_{2i+1,2j}^h) \\ & +4v_{2i,2j}^h\big], \text{ für } 1 \leq i,j \leq \frac{n}{2} - 1. \end{aligned} \tag{2.14}$$

Mit diesem Wissen kann dass abstrakte Korrekturschema aus Definition 2.1 konkretisiert werden:

Definition 2.2 (Das Korrekturschema)

- *Iteriere ν_1 mal $A^h u^h = f^h$ auf Ω^h mit einem Anfangswert v^h.*
- *Berechne das Residuum auf dem feinen Gitter $r^h = f^h - A^h v^h$ und restringiere es mittels $r^{2h} = I_h^{2h} r^h$ auf das grobe Gitter.*
- *Löse $A^{2h} e^{2h} = r^{2h}$ auf dem groben Gitter Ω^{2h}.*
- *Der Fehler e^{2h} auf dem groben Gitter wird auf das feine Gitter mittels $e^h = I_{2h}^h e^{2h}$ interpoliert und v_h wird mit dieser Approximation für den Fehler auf dem feinen Gitter korrigiert: $v_h \leftarrow v_h + e_h$.*
- *Iteriere anschließend ν_2 mal auf dem feinen Gitter mit neuem Anfangswert v_h.*

A^{2h} stellt hierbei die Matrix dar, die man erhält, wenn man das ursprüngliche Problem auf dem groben Gitter diskretisiert.
An dieser Stelle fällt auf, dass die exakte Lösung des Problems $A^{2h}e^{2h} = r^{2h}$ für den Fehler auf dem groben Gitter aus Definition 2.2 das ursprüngliche Problem darstellt und sich ein rekursiver Lösungsansatz anbietet. Die Rekursion kann abgebrochen werden, wenn das gröbste Gitter eine so geringe Größe hat, dass es schnell direkt gelöst werden kann oder wenn man das gröbste Gitter mit einer diskreten und geringen Anzahl an Iterationen fast exakt lösen kann.
Zur Vereinfachung der Notation sollen ab jetzt alle rechten Seiten (insbesondere r^h der Residuums-Gleichung) eines zu lösenden linearen Gleichungssystems mit f^h auf dem Gitter Ω^h bezeichnet werden. Die Fehler e^h werden zu u^h umbenannt, da sie auch nur die Lösung eines linearen Gleichungssystems repräsentieren.
Der obige Ansatz mittels Rekursion führt unmittelbar auf einen ersten und für diese Master-Thesis wichtigen Mehrgitter-Algorithmus:

Definition 2.3 (V-Zyklus) *Der V-Zyklus sei mittels $v^h \leftarrow V^h(v^h, f^h)$ notiert und implementiert durch*

1. *Iteriere ν_1 mal $A^h u^h = f^h$ auf Ω^h mit einem Anfangswert v^h.*
2. *Ist Ω^h das gröbste Gitter, so springe zu Punkt 4, ansonsten berechne*

$$\begin{aligned} f^{2h} &\leftarrow I_h^{2h}(f^h - A^h v^h) \\ v^{2h} &\leftarrow 0 \\ 2^{2h} &\leftarrow V^{2h}(v^{2h}, f^{2h}). \end{aligned}$$

3. *Korrigiere die Approximation $v^h \leftarrow v^h + I_{2h}^h v^{2h}$.*
4. *Iteriere anschließend ν_2 mal $A^h u^h = f^h$ auf Ω^h mit Anfangswert v_h.*

Abbildung 2.3 demonstriert, welche Gitter in welcher Reihenfolge besucht werden, wenn der V-Zyklus aus Definition 2.3 auf einem Gitter Ω^h gestartet wird und Ω^{16h} das gröbste Gitter ist. Es gibt durchaus noch andere Mehrgitterverfahren, wie zum Beispiel den W-Zyklus, der entsprechend nach Abbildung 2.4 vorgeht oder den Full-Multigrid-Algorithmus (FMG), welcher nach Abbildung 2.5 vorgeht. Das besondere beim FMG-Algorithmus ist, dass bevor der V-Zyklus auf einem Gitter Ω^h gestartet wird, vorher ein V-Zyklus auf dem gröberen Gitter Ω^{2h} durchgeführt wird, um so schon einen möglichst guten Anfangswert für das Gitter Ω^h zu erhalten. Im weiteren Verlauf dieser Master-Thesis wird aber ausschließlich der V-Zyklus betrachtet.

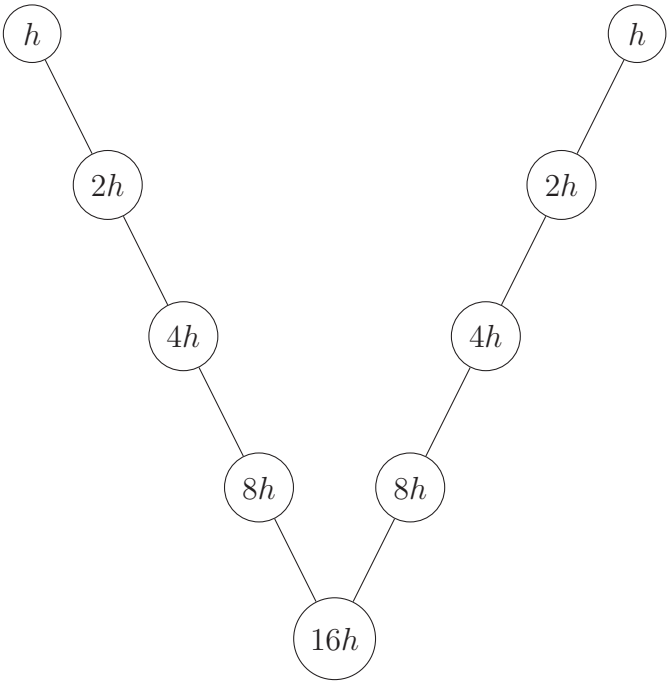

Abbildung 2.3: Stationen im V-Zyklus

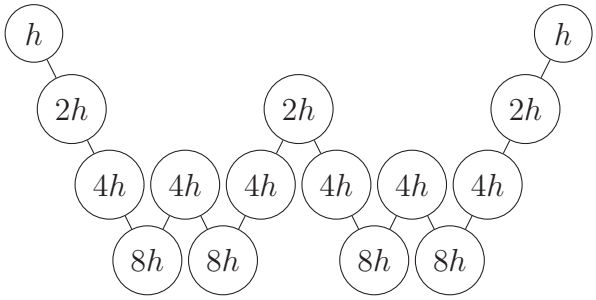

Abbildung 2.4: Stationen im W-Zyklus

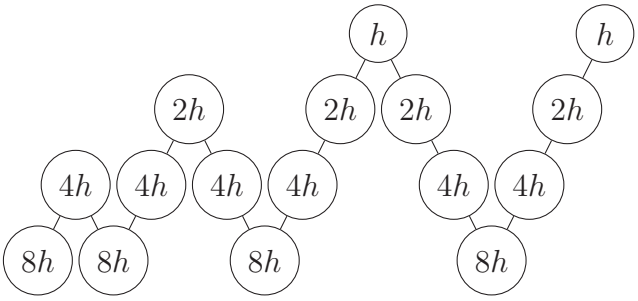

Abbildung 2.5: Stationen im FMG-Schema

3 Blockglätter

3.1 Mehrgitter im Parallelen

Im vorherigen Kapitel wurde erklärt, wie elliptische PDEs mittels eines Mehrgitterverfahrens effizient gelöst werden können. Bei der Diskretisierung des Problems wird ein Fehler gemacht, der Diskretisierungsfehler. Damit die numerische Lösung so exakt wie möglich ist, sollte das Gebiet sehr fein diskretisiert werden; die Schrittweite h muss sehr klein gewählt werden. Dadurch entstehen sehr viele Gitterpunkte, auf denen iteriert werden muss. Da dies die Leistung einer einzelnen Recheneinheit übersteigen kann, muss das Problem auf mehrere Recheneinheiten aufgeteilt und somit parallelisiert werden. Unter dem hier abstrakten Begriff einer Recheneinheit kann ein Thread einer CPU, eine GPU oder eine Workstation verstanden werden. Zur Parallelisierung der Iteration auf jeder Ebene eines Mehrgittercodes eignet sich besonders das in 2.1 vorgestellte *Red-Black Gauß-Seidel*-Verfahren. Um Problem der Parallelisierung eines Mehrgitterverfahren auf mehreren Recheneinheiten zu verdeutlichen, soll sich im Folgenden nur auf die Iterationsschritte der einzelnen Level im Mehrgitterverfahren konzentriert werden. Dies ist legitim, da die Iterationen auch einen signifikanten Teil des Aufwandes in einem Mehrgitterverfahren ausmachen.
Zur sehr abstrakten Einführung eines parallelen Mehrgitterverfahrens soll an dieser Stelle das mit $2^3 + 1 = 9$ Gitterpunkten in jeder Dimension diskretisierte Gebiet gemäß Abbildung 3.1 betrachtet werden.

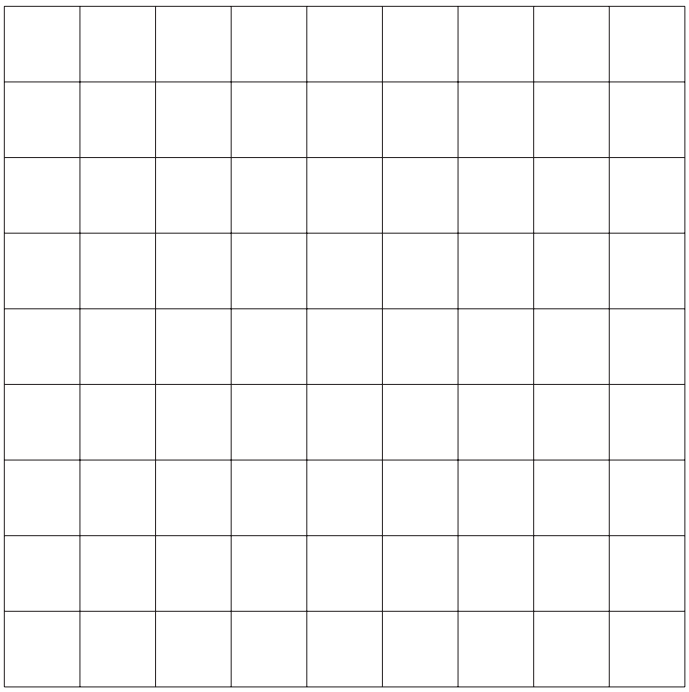

Abbildung 3.1: 2D Gebiet mit $2^3 + 1$ Punkten

Dieses Gebiet wird in Blöcke der Dimension 3×3 zerlegt und die neun so entstehenden Blöcke werden nach einem *Block-Torus-Wrapped*-Verfahren [Fro90] auf zwei Recheneinheiten nach Abbildung 3.2 verteilt. Recheneinheit 1 wird repräsentiert durch die Farbe Rot und Recheneinheit 2 durch die Farbe Orange.

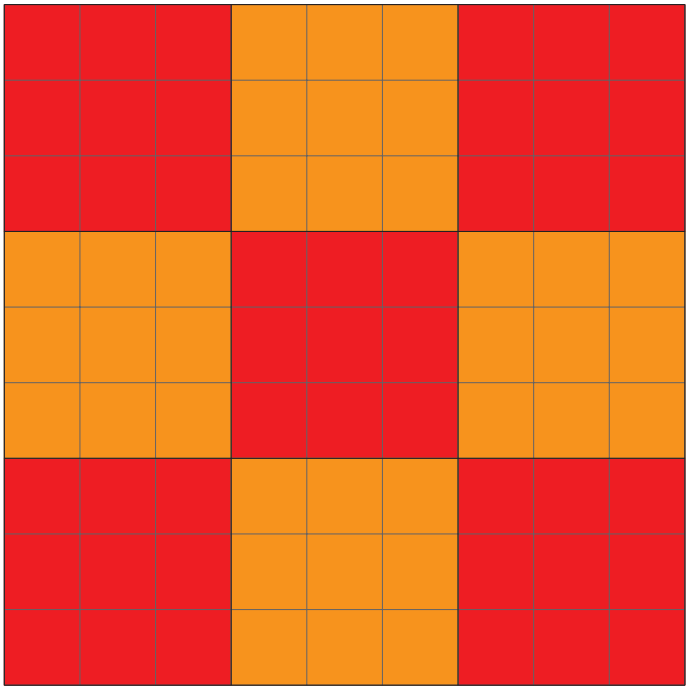

Abbildung 3.2: 2D Gebiet mit $2^3 + 1$ Punkten gleichmäßig verteilt auf 2 Recheneinheiten

Damit jedes der zwei Recheneinheiten seine neun Gitterpunkte pro Blockgebiet unabhängig von allen anderen Gitterpunkten bearbeiten kann, benötigt es den Wert der Randpunkte seiner Nachbargebiete. Diese müssen kommuniziert werden. Je nach Art der verwendeten Recheneinheiten kann diese Kommunikation lokal auf einer Workstation im Arbeitsspeicher oder über ein Netzwerk zwischen unterschiedlichen Workstations passieren. Zur Vereinfachung des folgenden Pseudocodes in Bezug auf Deadlocks wird davon ausgegangen, dass das Senden von irgendwelchen Daten zu einer anderen Recheneinheit asynchron passiert. Ein abstrakter Iterationscode könnte beispielsweise wie folgt aussehen:

for $i := 1, ..., \nu$ **do**
  for *subdomain* in *my_subdomains* **do**
    Sende Randpunkte von *subdomain* an anliegende Nachbargebiete
    Empfange Wert der Gitterpunkte von umliegenden Nachbargebiete
    Iteriere *subdomain*
  end for
end for

Jede Recheneinheit führt den obigen Pseudocode aus. Dabei wird jedes Teilgebiet ν mal iteriert und *my_subdomains* repräsentiert alle Teilgebiete, die eine Recheneinheit zugewiesen bekommen hat. Zur Verdeutlichung von Problemen, die hier auftre-

ten, soll im Folgenden ein konkretes Beispiel betrachtet werden:
Es wird das $2D$-Gebiet, welches für einen Mehrgittercode auf dem feinsten Level mit $2^{15} + 1 = 32.768 + 1$ Gitterpunkten in beiden Dimensionen diskretisiert wurde, betrachtet. Ein V-Zyklus würde dabei 15 verschiedene Gitter durchlaufen. Jede Recheneinheit soll zunächst ein Teilgebiet fester Größe mit 2^{10} Gitterpunkten bzw. am rechten und unteren Rand mit $2^{10} + 1$ Gitterpunkten erhalten. Es sollen 128 Recheneinheiten zur Verfügung stehen. Somit stehen auf dem feinsten Level 32×32 Teilgebiete zur Verfügung, die auf die einzelnen Recheneinheiten verteilt werden können. Tabelle 3.1 stellt einige Statistiken für unterschiedliche Level k im V-Zyklus dar. Dabei ist zu beachten, dass *Teilgebiete pro Recheneinheit* die durchschnittliche Anzahl an Teilgebieten pro Recheneinheit bei Teilgebieten fester Größe mit $2^{10}(+1)$ Gitterpunkten angibt.

Level k Level k	Teilgebiete insgesamt Teilgebiete insgesamt	Teilgebiete pro Recheneinheit	Ungenutzte Recheneinheiten
15	1024	8	0
14	256	2	0
13	64	$0,5$	64
10	1	$\frac{1}{128} \approx 0,0078$	127

Tabelle 3.1: Statistiken zu verschiedenen Leveln im V-Zyklus

Sobald der V-Zyklus auf Level $k = 10$ angekommen ist, gibt es nur noch eine einzige Recheneinheit, die Berechnungen durchführt. Dies ist durch die hier willkürlich gewählte Teilgebietgröße von $2^{10}(+1)$ zustande gekommen und man könnte die Teilgebietgröße entsprechend den zur Verfügung stehenden Gitterpunkten auf jedem Level anpassen. Würde man die zur Verfügung stehenden Gitterpunkte möglichst gleichmäßig auf alle Recheneinheiten verteilen, so hätte in diesem Beispiel (abgesehen vom Rand), jede Recheneinheit auf Level 10 nur 8 Gitterpunkte und auf Level 7 nur noch einen Gitterpunkt. Für diesen einen Gitterpunkt auf Level 7 müssten aber 8 angrenzende Punkte für einen Iterationsschritt kommuniziert werden. Abbildung 3.3 stellt diesen Sachverhalt dar.

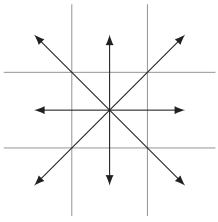

Abbildung 3.3: Kommunikation zwischen einer Recheneinheit und seinen 8 Nachbargebieten

Dies ist unpraktisch, da die Kommunikation zwischen einzelnen Recheneinheiten wesentlich zeitaufwendiger ist als die Iteration mit lokal vorliegenden Werten (gemäß dem Motto „FLOPS are for free, but bandwidth is very expensive“, [Feo12]). Im nächsten Kapitel werden Blockglätter mit der Intention vorgestellt, diese Problematik zu verbessern.

3.2 Einführung in Blockglätter

Ein Ziel von Mehrgitterverfahren mit Blockglättern ist es, die Anzahl der gröberen Gitter bzw. unteren Level, auf denen wenig Arbeit parallelisiert werden kann, möglichst stark zu reduzieren, um so zum Beispiel in einem V-Zyklus schnell wieder auf die feinen Gitter bzw. höheren Level zu gelangen, auf denen wieder mehr gerechnet werden kann. Dadurch soll ein gutes Verhältnis zwischen Kommunikation und Rechenarbeit gewährleistet werden. Dies soll hier anhand eines eindimensionalen Problems erläutert werden. Alle Überlegungen aus diesem Kapitel sind aber problemlos auf zwei- oder dreidimensionale Probleme erweiterbar. Es wird weiterhin das Gebiet $[0, 1]$ betrachtet. Dieses wird von nun an aber mit Hilfe einer *Basis* $b \in \mathbb{N}$ und einem *Diskretisierungslevel* $k \in \mathbb{N}$ diskretisiert. Das diskrete Gebiet Ω_h besteht aus

$$b^k + 1$$

äquidistanten Gitterpunkten. Für die Fälle $b = 2, k = 4$ und $b = 4, k = 2$ ergeben sich dann gemäß Abbildung 3.4 die exakt gleichen diskreten Gebiete:

Abbildung 3.4: Gebiet diskretisiert mit a) $2^4 + 1$ und b) $4^2 + 1$ Gitterpunkten

Um die Anzahl der Level im Mehrgitterverfahren zu reduzieren, wird stärker restringieren und interpolieren. Für die Restriktion des Residuums vom feinen zum groben Gitter wurde bis jetzt mittels full-weighting auf jeden Gitterpunkt mit geradem Index restringiert. Dabei sind die ungeraden Gitterpunkte heraus gefallen. Von nun an wird auf jeden Gitterpunkt mit Index $n \cdot b$, $n \in \mathbb{N}$ restringiert. Dabei werden $b - 1$ Punkte links von diesem und $b - 1$ Punkte rechts von diesem zu einem neuen Gitterpunkt auf dem groben Gitter verrechnet. Dieses Vorgehen ist konsistent zum vorherigen Verfahren für eine Diskretisierung mit Basis 2. Abbildung 3.5 demonstriert, aus welchen feinen Gitterpunkten auf dem oberen Gebiet sich der mittlere grobe Gitterpunkt auf dem unteren Gebiet errechnet.

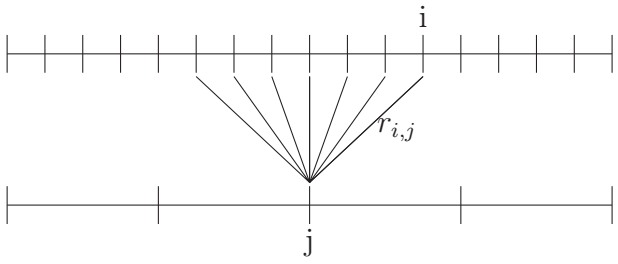

Abbildung 3.5: Restriktion des Residuums von $4^2 + 1$ auf $4^1 + 1$ Gitterpunkte

Die Koeffizienten für Abbildung 3.5 ergeben sich von links nach rechts in einem Vektor zusammengefasst zu

$$\frac{1}{4} \cdot \left(\frac{1}{4}, \frac{2}{4}, \frac{3}{4}, \frac{4}{4}, \frac{3}{4}, \frac{2}{4}, \frac{1}{4} \right)^T .$$

Die Gewichtungsfaktoren der Restriktion mittels full-weighting für eine allgemeine Basis b zwischen zwei Gitterpunkten i und j lassen sich wie folgt berechnen: Dabei sei i ein Gitterpunkt auf dem feinen Gitter und j ein Gitterpunkt auf dem groben Gitter. $d_{i,j}$ sei der Betrag des Abstand in Gitterpunkten auf dem feinen Gitter zwischen i und der Projektion von j auf das feine Gitter. Dann ergeben sich die Gewichtungsfaktoren zu

$$r_{i,j} = \frac{1}{b} \cdot \begin{cases} \frac{d_{i,j}-b}{b} & , \text{ falls } d_{i,j} < b \\ 0 & , \text{ sonst} \end{cases} .$$

Hierzu läuft analog die Interpolation des Fehlers von einem groben auf ein feines Gitter ab. Zuvor wurden die Gitterpunkte mit ungeradem Index auf dem feinen Gitter zwischen den Gitterpunkten mit geradem Index interpoliert. Nun werden alle Gitterpunkte auf dem feinen Gitter zwischen den Gitterpunkten mit Index $n \cdot b$, $n \in \mathbb{N}$ interpoliert. Dies läuft analog zur Restriktion in umgekehrter Richtung mit Koeffizienten $c_{i,j}$ nach Abbildung 3.6 ab.

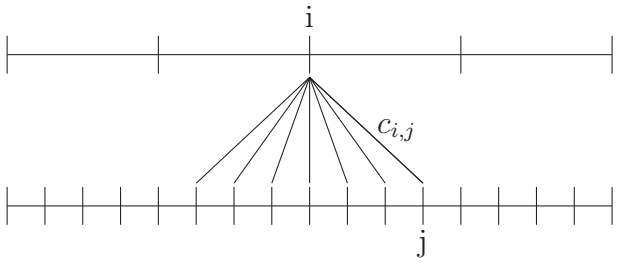

Abbildung 3.6: Interpolation des Fehlers von $4^1 + 1$ auf $4^2 + 1$ Gitterpunkte

Hier ist zu beachten, dass für die Berechnung der feinen Gitterpunkte mehrere grobe Gitterpunkte verwendet werden.
Da der Fehler von einem groben zu einem feinen Gitter in größeren Intervallen interpoliert wird, ist zu erwarten, dass die Interpolation schlechter wird und somit das Mehrgitterverfahren langsamer konvergiert. Um diesen zusätzlichen Fehler zu kompensieren, wird die Effektivität der Iteration auf einem Gitter verbessert. Dies geschieht mit Hilfe der *Blockglätter*.
Bei den Blockglättern wird das Gebiet im inneren in nicht überlappende Blöcke der Größe b aufgeteilt. Für das Gebiet mit 4^2+1 Gitterpunkten ergeben sich diese gemäß Abbildung 3.7, wobei vier aufeinander folgenden rote bzw. grüne Gitterpunkte einen Block bilden.

Abbildung 3.7: Zerlegung von $4^2 + 1$ Gitterpunkten in 4er Blöcke

Der letzte Block an rechten Rand besteht dabei nur aus $b - 1$ Gitterpunkten. Zur Notation sei in diesem Kapitel mit dem Index B immer eine lokale Größe in einem solchen Block gemeint. Für jeden Block wird pro Iterationsschritt einzeln die Residuums-Gleichung $A_B \cdot e_B = r_B$ lokal gelöst. u_B in diesem Block wird dann mit dem so gewonnen Fehler korrigiert: $u_B \leftarrow u_B + e_B$. Die Randwerte um jeden Block herum sind, da die Residuums-Gleichung für diesen Block verwendet wird, gleich 0. Bei nicht überlappenden Blöcken können alle Blöcke komplett unabhängig und parallel voneinander gelöst werden. Die einzelnen Blöcke können, so wie bei den Iterationsverfahren, beispielsweise in einer Jacobi-Manier bearbeitet werden. Dabei werden alle Blöcke nacheinander gelöst. Bei einer Red-Black Gauß-Seidel-Manier würde man die Blöcke in rote und schwarze Blöcke unterteilen und immer nur Blöcke mit einer Farbe gleichzeitig lösen. Weitere Variationen wären beispielsweise überlappende Blöcke [MO11], die in dieser Master-Thesis nicht weiter behandelt werden.
Ein Gebiet, das mit beispielsweise $2^9 + 1$ Gitterpunkten diskretisiert wird, müsste in einem klassischen V-Zyklus 9 Level durchlaufen. Dasselbe Gebiet kann mit Hilfe von Blockglättern mit $8^3 + 1$ Gitterpunkten diskretisiert werden, wodurch ein entsprechender V-Zyklus nur noch 3 Level durchlaufen muss. Dadurch gibt es weniger Level mit einem schlechten Verhältnis zwischen Kommunikation und Rechenarbeit in einem parallelen System.
Im Folgenden Kapitel wird die grundlegende Architektur der GPU vorgestellt, da die mit dieser Master-Thesis entstandene Implementierung auf der GPU ausgeführt werden wird.

4 OpenCL und GPU

Das in Kapitel 3.2 beschriebene Mehrgitterverfahren mit Blockglättern wurde in OpenCL implementiert und für die GPU optimiert. Darum soll zunächst eine kurze Einführung in die OpenCL und GPU Architektur folgen.

4.1 Einführung in OpenCL

OpenCL (Open Computing Language) ist eine Schnittstelle für heterogene Parallelrechner, die von der eigentlichen Hardware wie beispielsweise CPU oder GPU abstrahiert und die zugehörige Programmiersprache „OpenCL C“ bereitstellt. Sie ist ursprünglich von der Firma Apple entwickelt worden. In Zusammenarbeit mit den Firmen AMD, IBM, Intel und NVIDIA wurde ein Entwurf ausgearbeitet und schließlich von Apple bei der Khronos Group zur Standardisierung eingereicht. Die Spezifikation der ersten Version OpenCL 1.0 wurde am 8. Dezember 2008 veröffentlicht.
Die OpenCL Spezifikation beschreibt die OpenCL Architektur wie folgt [ME12]: Abstrakt gibt es einen *Host*, mit dem mehrere *OpenCL Devices* verbunden sind. Jedes dieser Devices enthält mehrere *Compute Units* (CUs), die jeweils eine diskrete Anzahl an *processing elements* (PEs) enthalten. Diese PEs sind die kleinsten und einzigen Elemente, die Berechnungen ausführen. Abbildung 4.1 veranschaulicht diesen Sachverhalt.

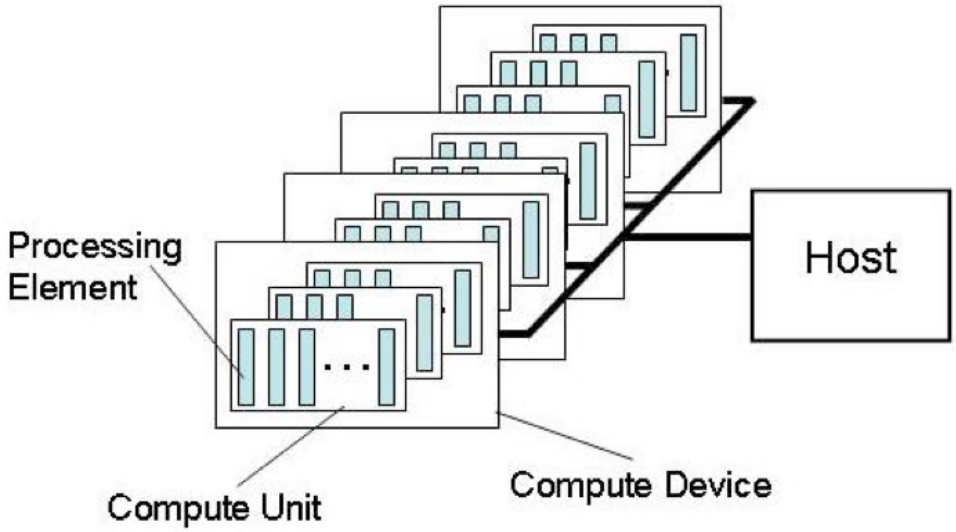

Abbildung 4.1: Schema der abstrakten OpenCL Plattform, aus [Goh09]

Ein allgemeines Programm besteht dabei aus zwei Teilen:

1. Einem Host-Programm, welches *Commands* an die PEs in einem Device sendet und die Koordination des Programmablaufs übernimmt.
2. Einem *Kernel* (Programm), welcher von den PEs auf einem OpenCL Device ausgeführt wird.

Jeder Kernel bekommt eine Dimension zugewiesen, die das zu berechnende Problem zerlegt. Ein Eintrag der Dimension wird als *Work-Item* bezeichnet und jede Instanz des Kernels kann auf diesen Eintrag als eindeutige globale ID zugreifen. Diese Work-Items sind in *Work-Groups* organisiert. Konkreter ist die Dimension eines Kernels ein *NDRange*, wobei N für 1-, 2- oder 3-dimensional steht. Dabei können die Work-Items nativ 1-, 2- oder 3-dimensional über die Kernel-Instanzen verteilt werden. Die globale ID eines jeden Work-Items ist dann entsprechend 1-, 2- respektive 3-dimensional.
Jedes Work-Item hat dabei Zugriff auf 4 unterschiedliche Kategorien von Speicher:

- **Global Memory**: Diese Speicherregion erlaubt Lese- und Schreibzugriffe von allen Work-Items in allen Work-Groups aus. Work-Items können von jedem Element lesen und in jedes Element eines entsprechenden Speicherobjektes schreiben. Ob solche Speicherzugriffe gecached werden, hängt von der entsprechenden Hardware ab.

- **Constant Memory**: Eine Speicherregion, die während der Laufzeit eines Kernels konstant bleibt. Das Host-System übernimmt das Allokieren, Initialisieren und Bereitstellen eines solchen Speichers.
- **Local Memory**: Eine lokale Speicherregion auf die nur eine Work-Group Zugriff hat.
- **Private Memory**: Eine Speicherregion, auf die nur ein Work-Item Zugriff hat.

Abbildung 4.2 demonstriert diese vier Speicherregionen.

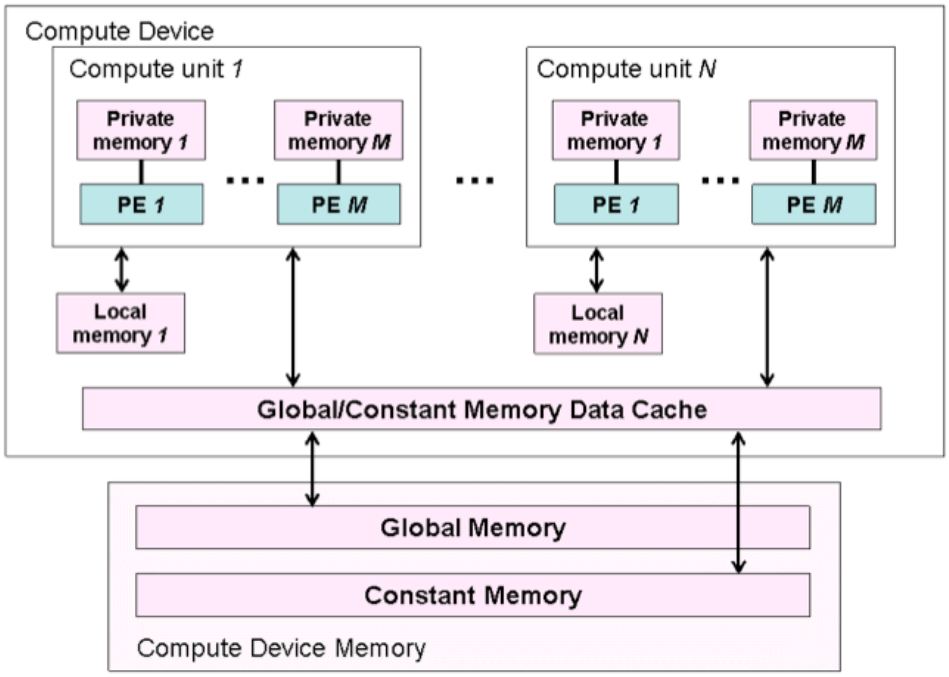

Abbildung 4.2: die vier OpenCL Speicherregionen, aus [Goh09]

Listing 4.1 enthält einen Kernel, der für zwei Vektoren a und b ihre Summe $c = a + b$ berechnet. Das Work-Item mit Index i des Kernels übernimmt dabei die Berechnung des Eintrags $c_i = a_i + b_i$ des Vektors c. Alle Variablen a, b und c werden dabei im globalen Speicher abgelegt.

```
1 __kernel void addition(__global float *a,
                         __global float *b,
3                        __global float *c)
```

```
{
    // hole globale ID fuer dieses Work-Item
    int i = get_global_id(0);

    // Work-Item i berechnet c_i = a_i + b_i
    c[i] = a[i] + b[i];
}
```

Listing 4.1: Beispiel eines OpenCL Kernels

Dies soll als kurze Einführung in OpenCL genügen. Damit ein entsprechendes Mehrgitterverfahren mit Blockglättern effizient auf der GPU implementiert werden kann, soll anschließend an dieses Kapitel eine kurze Einführung in die NVIDIA Architektur der GPU und einige OpenCL spezifische Hinweise zur effizienten Berechnung auf der NVIDIA CUDA Architektur folgen.

4.2 GPU Architektur

Das Zielsystem, auf dem die Implementierung des Mehrgitterverfahrens ausgeführt werden wird ist das 2011 in Betrieb genommene GPU-Computing-Cluster JUDGE (Jülich Dedicated GPU Environment) des Jülich Supercomputing Centres. Dabei stehen dem Anwender 108 Grafikprozessoren vom Typ NVIDIA Tesla M2050 und M2070 zur Verfügung, die mit jeweils 3 respektive 6 GB GDDR5 RAM ausgestattet sind und eine Peak-Performance von 1.03 TFLOPS Single Precision Floating Points bzw. 515 GFLOPS Double Precision Floating Points aufweisen.
Auf NVIDIA Grafikkarten mit CUDA Architektur [NVI12] gibt es, analog zur OpenCL Architektur, unterschiedliche Level von Gruppierungen. Die OpenCL Implementierung auf NVIDIA Grafikkarten ist über die CUDA Architektur realisiert, weswegen eine kurze Einführung an dieser Stelle folgt. Auf der obersten Ebene existieren die *Thread Processing Cluster* (TPC). Jeder dieser TPCs enthält drei *Streaming Multiprocessors* (SMs), einen Controller, der diese steuert und einen gemeinsamen Speicher, auf den die SMs Zugriff haben. Die Anzahl der SMs werden auch als *Cores* bezeichnet, welche aber nicht mit Cores im Sinne einer CPU zu verwechseln sind. Die Tesla M2050 und M2070 enthalten jeweils 448 dieser Cores. Abbildung 4.3 zeigt diesen Zusammenhang.

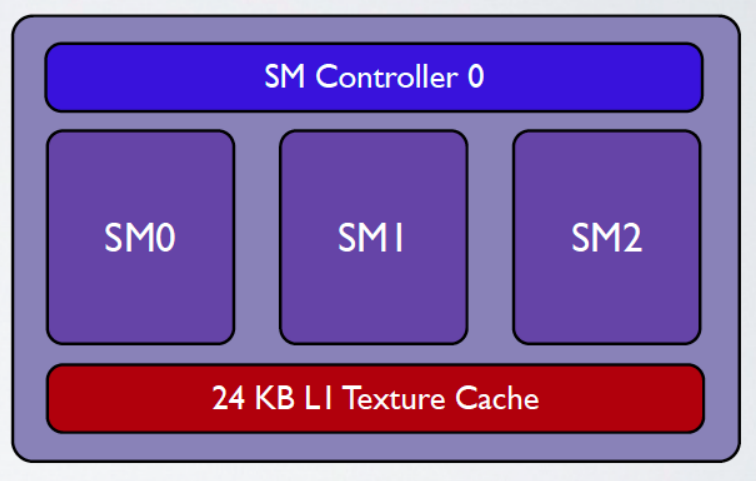

Abbildung 4.3: Darstellung eines TPCs, aus [Goh09]

Jeder dieser SM enthält mehrere *Streaming Processors* (SPs), die wiederum eine bestimmte Anzahl von Threads enthalten, wie in Abbildung 4.4 verdeutlicht. Dabei ist ein Thread in der CUDA Architektur gleichzusetzen mit einem Work-Item im OpenCL Kontext.

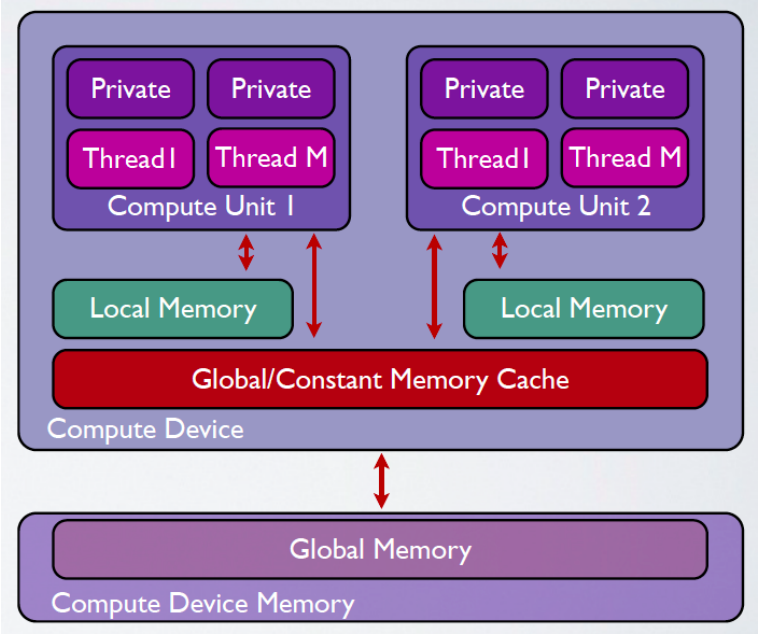

Abbildung 4.4: Darstellung eines SMs, aus [Goh09]

Diese Threads bzw. Work-Items sind gruppiert in die kleinste allokierbare Work-Item Größe, den *Warp*. Ein Warp ist eine Gruppe von genau 32 Threads. Auf der GPU kann nur eine Problemgröße, die ein Vielfaches eines Warps ist, bearbeitet werden. Die Tesla Architektur enthält pro SM genau 16 Warps. Damit stehen einem auf den NVIDIA Tesla M2050 und M2070 GPUs 229.376 Threads zur Verfügung, die parallel einen Kernel ausführen können. Im Vergleich dazu stehen einem auf einer modernen CPU 8 Cores und mit Hyper-Threading 16 Threads zur Verfügung, die gleichzeitig Code ausführen können. Zwar ist es auf der CPU möglich, mehr als 16 Threads zu verwenden, diese werden dann aber mittels Software und nicht mehr mittels Hardware realisiert und Code wird dann nicht mehr parallel sondern nacheinander ausgeführt. Die GPU ist besonders effizient darin, diese Threads zu managen, um so beispielsweise Wartezeiten auf angeforderten Speicher von Threads zu kompensieren. Die hohen Anzahl an Threads oder Work-Items auf der GPU gegenüber der Anzahl an Threads oder Work-Items auf der CPU bringt nicht nur Vorteile mit sich:
Die 32 Threads in einem Warp sind noch ein weiteres mal unterteilt in zwei gleich große *Half-Warps*, bestehend aus jeweils 16 Threads. Die Threads in einem Half-Warp arbeiten nach einer SIMD (Single Instruction, Multiple Data) Manier. Das bedeutet, dass diese Threads immer nur denselben Code gleichzeitig ausführen können. Wenn beispielsweise mittels *If*-Abfragen zwei Threads in einem Half-Warp in unterschiedliche Codezweige C1 und C2 springen würden, so arbeiten diese nicht

mehr parallel sondern die Ausführung des Codes wird serialisiert. Thread 1 würde Codezweig C1 abarbeiten und anschließend Thread 2 Codezweig C2. Pausiert ein Thread, so pausieren alle Threads in diesem Half-Warp. Ebenfalls ist der Speicher der GPU in 16 verschiedenen Bänken mit Wortlänge 8, 16, 32 oder 64 Bit organisiert und wenn ein Thread eine Speicheradresse anfordert, so werden automatische alle 15 folgenden Wörter mit übertragen, damit die anderen Threads in diesem Half-Warp mit den anderen Daten parallel arbeiten können.
NVIDIA hat 2009 einen Best Practice Guide [NVI09] für OpenCL auf ihrer GPU Hardware veröffentlicht, der die Implementierung des im vorherigen Kapitel beschriebenen Mehrgitterverfahrens maßgeblich beeinflusst hat. An dieser Stelle sollen lediglich die zwei wichtigsten dieser Faktoren angesprochen werden, die die Implementierung am meisten beeinflusst haben:

- **Speichertransfer zwischen Host und Device**: Der Datentransfer zwischen RAM des Host-Systems und RAM auf der Grafikkarte ist einer der größten Flaschenhälse. Darum sollten nach Möglichkeit so wenig Daten zwischen diesen beiden ausgetauscht werden, wie nur irgendwie möglich. Die Tesla M2050 und M2070 stellen beide eine Bandbreite von ca. 150 GByte/s auf der GPU zur Verfügung. Dem gegenüber stehen 8 GByte/s für PCI Express x16 Gen2. Dies entspricht einem Faktor von ungefähr 19.

- **Coalesced Access to Global Memory**: Aufgrund der oben beschriebenen Speicherorganisation in 16er-Bänken sollte Zugriff auf und in globalen Speicher von allen Thread in einem Half-Warp aus immer gemeinsam und einheitlich passieren. Abbildung 4.5 demonstriert, wie ein Half-Warp am besten auf globalen Speicher zugreifen sollte.

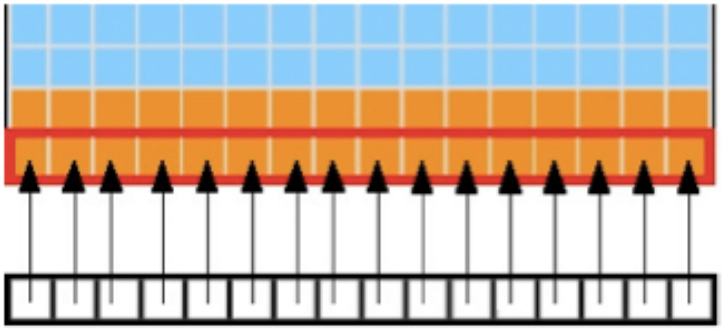

Abbildung 4.5: Optimaler Speicherzugriff eines Half-Warps auf globalen Speicher, aus [NVI09]

Jeder Thread (dargestellt durch die unteren Kästchen) greift auf genau ein Wort im globalen Speicher zu. Thread 0 auf ein Wort in Speicheradresse m_0,

mit m_0 mod 16 = 0 und Thread 15 auf ein Wort in Speicheradresse m_{15}, mit $m_{15} = m_0 + 15$. Anderer Zugriff wie beispielsweise in Abbildung 4.6 dargestellt,

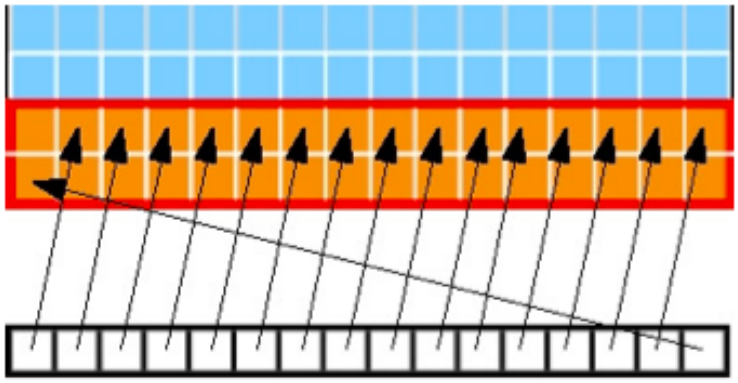

Abbildung 4.6: Nicht optimales Speicherlayout, aus [NVI09]

bei dem der Zugriff mit einem Speicher-Offset passiert sorgt dafür, dass unnötige Wörter für diesen Half-Warp mit angefordert werden. In Abbildung 4.6 würden für den entsprechenden Half-Warp 16 zusätzliche Wörter, 15 aus der unteren und ein Wort aus der oberen Reihe, angefordert werden, die nicht benötigt werden.

5 Implementierung

Dieses Kapitel beschäftigt sich mit der Implementierung des in Kapitel 3.2 vorgestellten Mehrgitterverfahrens in OpenCL auf der GPU. Aufgrund der Architektur der GPU wird zur Lösung eines einzelnen Blockes wieder ein iterativer Löser verwendet. Alle hier aufgeführten Beispiele und Erklärungen sind zur Einfachheit halber nur für Probleme in zwei Dimensionen. Die eigentliche Implementierung wurde für Probleme in drei Dimensionen vorgenommen. Algorithmus 5.1 beschreibt einen solchen Algorithmus als Pseudocode, wie er in der entstandenen Implementierung zu finden ist. Das Gebiet Ω_k sei dabei diskretisiert mit $b^k + 1$ Gitterpunkten. Eine Variable mit Index h gehöre immer zum aktuell betrachteten Gitter.

Algorithmus 5.1 *Pseudocode Mehrgitter nach Kapitel 3.2*

for Ω_k *...* Ω_2 ***do***
  loop *Anzahl Vorglättungsschritte*
    Berechne Residuum r_h
    loop *Anzahl Iterationen, um Block zu lösen*
      Iteriere $A_h e_h = r_h$ *lokal in den Blöcken*
    end loop
    $u_h += e_h$
  end loop
  Berechne Residuum r_h
  Restringiere r_h *auf* f_{h-1}
end for

Löse auf gröbstem Gitter

for Ω_2 *...* Ω_k ***do***
  Interpoliere u_{h-1} *auf* e_h
  $u_h += e_h$
  loop *Anzahl Nachglättungsschritte*
    Berechne Residuum r_h
    loop *Anzahl Iterationen, um Block zu lösen*
      Iteriere $A_h e_h = r_h$ *lokal in den Blöcken*
    end loop
    $u_h += e_h$
  end loop
end for

5.1 Advanced-Block-Layout

Algorithmus 5.1 benötigt Rechenleistung für die folgenden vier Aufgaben:

- Restriktion
- Interpolation
- Berechnung des Residuums
- Iteration der Residuums-Gleichung lokal in den Blöcken.

Die Iteration der Residuums-Gleichung lokal in den Blöcken stellt dabei den größten Aufwand in Bezug auf FLOP's dar. Das Ziel des *Advanced-Block-Layouts* ist es, diesen Teil für OpenCL auf der GPU zu optimieren. Die Blöcke werden dabei in einer

Red-Black Gauß-Seidel Manier bearbeitet. Das bedeutet, dass das Gebiet in rote und schwarze Blöcke zerlegt wird und zuerst die schwarzen Blöcke und anschließend die roten Blöcke gelöst werden. Eine solche Zerlegung für ein Gebiet mit $3^2 + 1$ Gitterpunkten findet sich in Abbildung 5.1. Dabei ist zu beachten, dass Abbildung 5.1 in jeder Dimension nur 8 statt 10 Gitterpunkte hat. Der Rand wird weggelassen, da die Residuums-Gleichung den Randwert 0 für jeden Block vorschreibt.

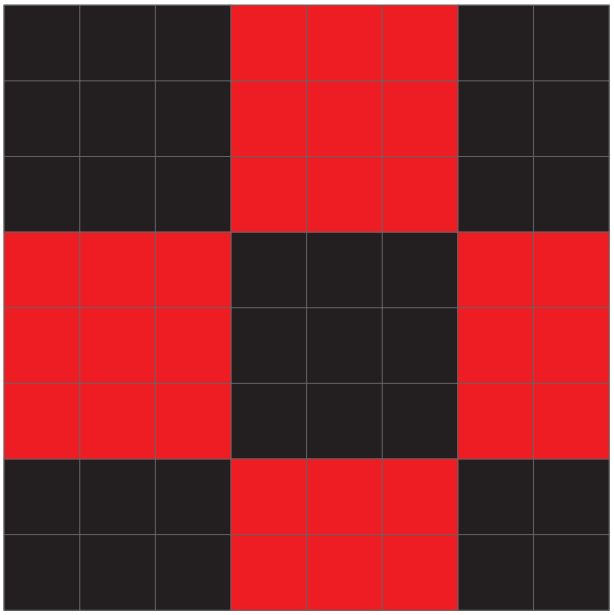

Abbildung 5.1: Zerlegung eines Gebiet mit 3^2+1 Gitterpunkten in rote und schwarze Blöcke

Aufgrund der in Kapitel 4.2 beschrieben GPU Architektur erhält jeder Thread einen Block. Jeder Thread arbeitet dabei den Block von links oben nach rechts unten ab. Jeder Block wird mittels einer Gauß-Seidel-Iteration gelöst. Damit jeder Thread in einem Half-Warp genau den gleichen Code ausführt und Speicher korrekt angefordert wird (der Rand kann bei einem hinreichend großen Gebiet vernachlässigt werden), wird im Folgenden ein entsprechendes Speicherlayout präsentiert. Dieses resultierende Speicherlayout wird in dieser Master-Thesis als *Advanced-Block-Layout* bezeichnet. Wie das normale Speicherlayout aussieht, indem das Residuum berechnet und die Restriktion und Interpolation durchgeführt wird, soll an dieser Stelle vernachlässigt werden und erst im nächsten Kapitel im Detail angesprochen werden. Für den Wechsel zwischen Speicherlayouts ist ein Umkopieren des Speichers nötig. Der Speicher für das Gitter im Advanced-Block-Layout wird dabei in zwei verschiedene Speicherregionen zerlegt: Eine für die schwarzen und eine für die roten Blöcke. Dabei werden die gleichfarbigen Blöcke von links nach rechts und von oben nach

unten gemäß Abbildung 5.2 durchnummeriert.

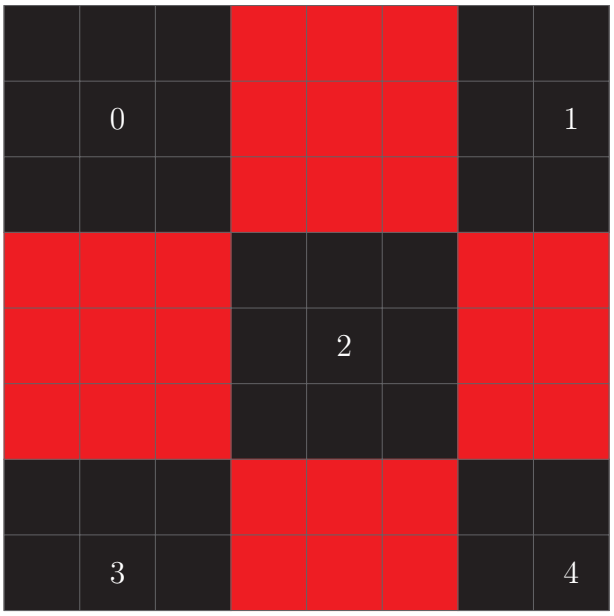

Abbildung 5.2: Nummerierung der schwarzen Blöcke für ein Gebiet der Größe 3^2+1

Eine entsprechende Nummerierung der Blöcke für den 3D-Fall in der Implementierung erhält man wie folgt: Die nächste freie Blocknummer erhält der Block unter allen noch nicht nummerierten Blöcken mit kleinstem z-Index. Gibt es hierfür mehrere, dann erhält der Block mit zusätzlich kleinstem x-Index die nächste Nummer. Ist auch diese Charakterisierung nicht eindeutig, erhält der Block mit kleinstem y-Index die nächste Nummer.
Die finale Speicherregion enthält in Reihenfolge der Nummerierung der Blöcke jeden ersten Eintrag eines Blockes, dann jeden zweiten Eintrag eines Blockes, dann jeden dritten Eintrag eines Blockes usw. Zusätzlich werden die Blöcke am Rand, die eine Dimension kleiner als b haben, vergrößert, damit alle Blöcke die gleiche Größe haben. Mit diesen eigentlich nicht existenten Werten am Rand wird nicht gerechnet. Abbildung 5.3 zeigt die Reihenfolge der Einträge, in welcher sie in der finalen Speicherregion der schwarzen Blöcke liegen.

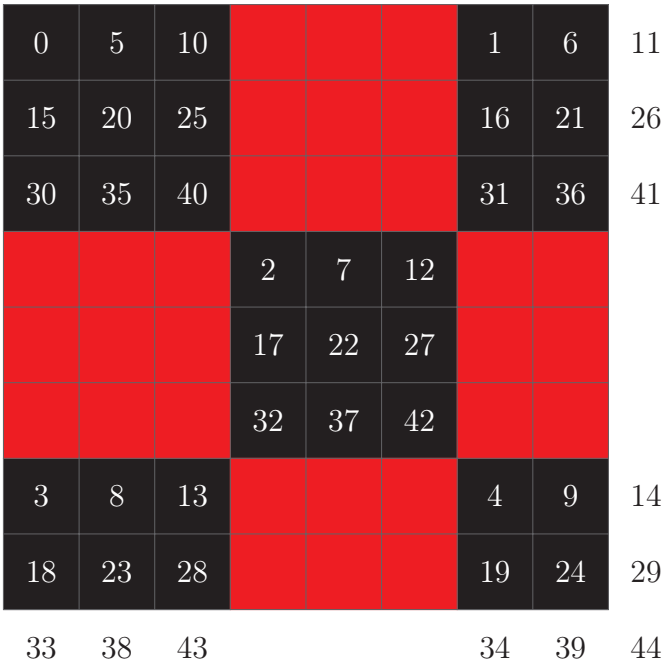

Abbildung 5.3: Reihenfolge der Elemente der schwarzen Blöcke, wie sie der finalen Speicherregion liegen

Zusätzlich wird die Anzahl der Blöcke auf ein Vielfaches von 16 Wörtern gepadded, damit jeder Thread den Speicher korrekt nach Kapitel 4.2 anfordert. Abbildung 5.4 zeigt die ersten zwei Einträge von jedem Block des resultierenden und endgültigen Speicherlayouts für das Gebiet mit der Größe $3^2 + 1$, wie es auf der GPU abgelegt ist.

Abbildung 5.4: Advanced-Block-Layout der ersten zwei Einträge für das Gebiet mit der Größe $3^2 + 1$

Um die Bijektion, die für das Umkopieren des Speichers verantwortlich ist, einfach zu halten, beschränkt sich diese Implementierung auf Gebiete, deren Basis b ein Vielfaches von 2 ist. Somit ergibt sich der neue Pseudocode für eine entsprechende Implementierung gemäß Algorithmus 5.2.

Algorithmus 5.2 *Pseudocode V-Zyklus mit Advanced-Block-Layout*

for Ω_k *...* Ω_2 ***do***
  loop *Anzahl Vorglättungsschritte*
    Berechne Residuum r_h
    Kopiere Residuum r_h *vom normalen Layout ins Advanced-Block-Layout*
    loop *Anzahl Iterationen, um Block zu lösen*
      Iteriere $A_h e_h = r_h$ *auf den schwarzen Blöcken*
    end loop
    Kopiere e_h *vom Advanced-Block-Layout zurück ins normale Layout*
    $u_h += e_h$

    Berechne Residuum r_h
    Kopiere Residuum r_h *vom normalen Layout ins Advanced-Block-Layout*
    loop *Anzahl Iterationen, um Block zu lösen*
      Iteriere $A_h e_h = r_h$ *auf den roten Blöcken*
    end loop
    Kopiere e_h *vom Advanced-Block-Layout zurück ins normale Layout*
    $u_h += e_h$
  end loop
  Berechne Residuum r_h
  Restringiere r_h *auf* f_{h-1}
end for

Löse gröbstes Gitter

for Ω_2 *...* Ω_k ***do***
  Interpoliere u_{h-1} *auf* e_h
  $u_h += e_h$
  loop *Anzahl Nachglättungsschritte*
    Berechne Residuum r_h
    Kopiere Residuum r_h *vom normalen Layout ins Advanced-Block-Layout*
    loop *Anzahl Iterationen, um Block zu lösen*
      Iteriere $A_h e_h = r_h$ *auf den schwarzen Blöcken*
    end loop
    Kopiere e_h *vom Advanced-Block-Layout zurück ins normale Layout*
    $u_h += e_h$

    Berechne Residuum r_h
    Kopiere Residuum r_h *vom normalen Layout ins Advanced-Block-Layout*
    loop *Anzahl Iterationen, um Block zu lösen*
      Iteriere $A_h e_h = r_h$ *auf den roten Blöcken*
    end loop
    Kopiere e_h *vom Advanced-Block-Layout zurück ins normale Layout*
    $u_h += e_h$
  end loop
end for

5.2 Simplified-Block-Layout

Das *Simplified-Block-Layout* beschreibt das Speicherlayout, in dem alle anderen Operationen wie Restriktion, Interpolation und Berechnung des Residuums durchgeführt werden. Die mit dieser Master-Thesis entstandene Implementierung kann auch ausschließlich mit Hilfe dieses Speicherlayouts einen V-Zyklus durchführen, ohne Speicher in das Advanced-Block-Layout umzukopieren. Dabei wird das diskrete 2D-Gitter in einem linearen Speicher von links nach rechts und von oben nach unten abgelegt. Zur Veranschaulichung soll wieder das Gitter mit 3^2+1 Gitterpunkten betrachtet werden, welches in Abbildung 5.5 zu sehen ist.

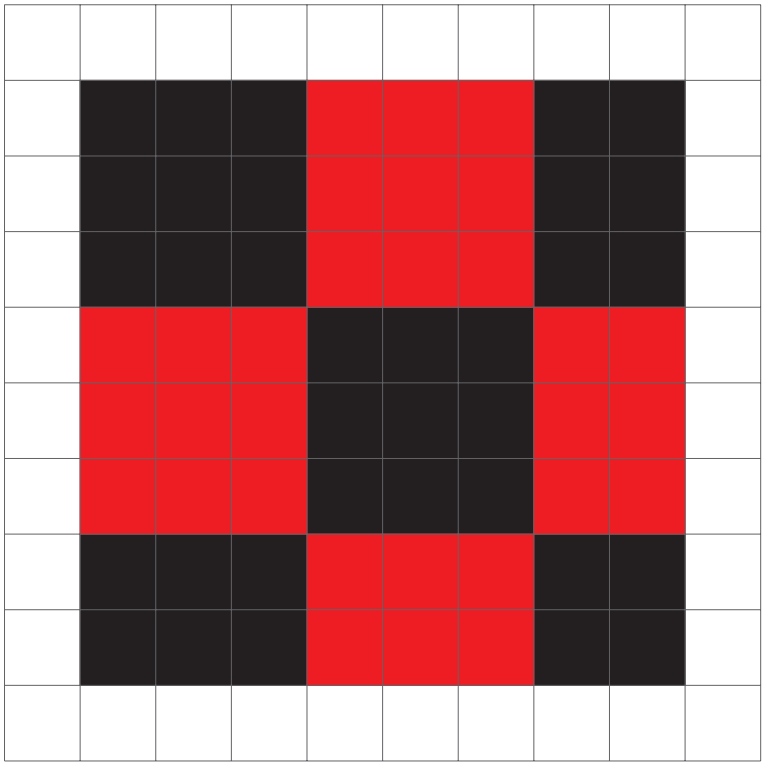

Abbildung 5.5: Gitter mit $3^2 + 1$ Gitterpunkten mit Rand

Zu beachten ist an dieser Stelle, dass der Rand des Gitter jetzt Bestandteil des resultierenden Speicherobjektes ist, da dieser zwar nicht für die Iteration aber für beispielsweise die Residuumsberechnung benötigt wird. Die Gitterpunkte werden wie oben beschrieben durchnummeriert. Abbildung 5.6 demonstriert diese Nummerierung für die ersten 40 Gitterpunkte $0, 1, \ldots 39$.

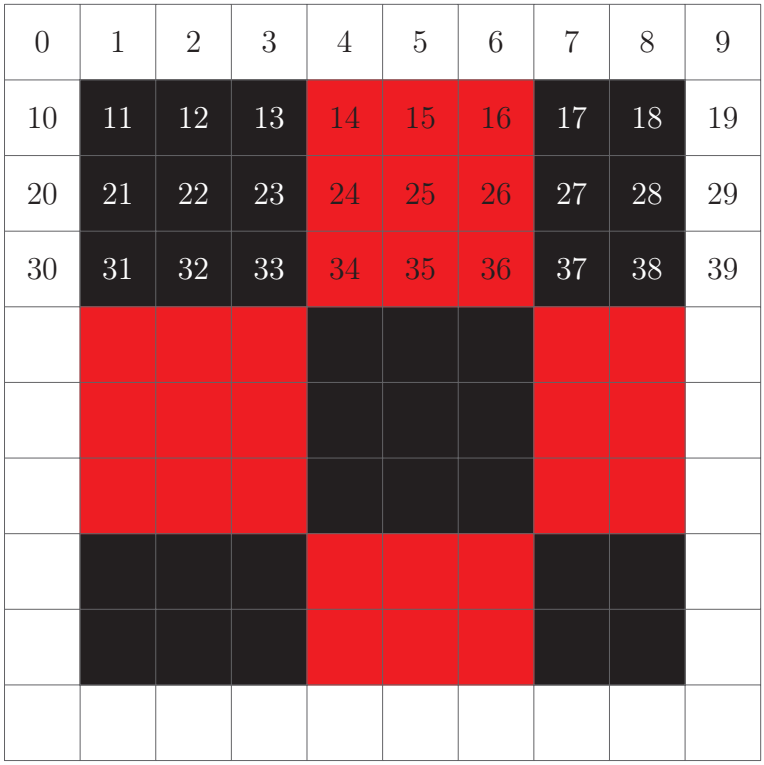

Abbildung 5.6: Nummerierung der Gitterpunkte

Jede Zeile wird dabei wieder auf ein Vielfaches von 16 Wörtern gepadded und jeder Thread erhält einen Gitterpunkt, den er iteriert. Dabei prüft jeder Thread, ob sein Gitterpunkt in einem der oben angezeigten Blöcke liegt. Wenn nicht rechnet der Thread nichts. Zusätzlich überprüft jeder Thread, an welcher Stelle sein Gitterpunkt lokal im Block liegt, um den Rand zu beachten. Um Speicher zu sparen wird in das selbe Speicherobjekt geschrieben, aus dem auch die aktuellen Werte für u_h gelesen werden. Eine direkte Konsequenz hieraus ist, dass der so implementierte V-Zyklus randomisiert arbeitet. Es kann passieren, dass ein Block auf zwei (oder Abhängig von der Blockgröße sogar auf mehrere) Half-Warps aufgeteilt wird. Je nachdem, wie die GPU die Half-Warps und Threads scheduled, kann es passieren, dass ein Thread in einem Gitterpunkt mit einer Jacobi-Iteration, einer Gauß-Seidel-Iteration oder einer Mischung aus beiden rechnet. Im schlimmsten Fall wird eine Jacobi-Iteration und im besten Fall wird eine Gauß-Seidel-Iteration durchgeführt.
Mit dem Advanced-Block-Layout wurden die einzelnen Blöcke nach einer Gauß-Seidel-Manier bearbeitet, indem sie in schwarz und rote Blöcke unterteilt wurden. Würde man dies hier tun, so gäbe es zwei Optionen, ohne den Speicher umzukopieren:

1. Die Dimension des Kernels ergibt sich, indem man eine der Dimensionen des

Gitters durch zwei teilt und alle Threads zunächst auf die schwarzen und danach auf die roten Blöcke verteilt.

2. Man verteilt die schwarzen und roten Blöcke gleichzeitig über alle verfügbaren Threads. Wenn die schwarzen Blöcke iteriert werden idlen alle Threads, die einen roten Gitterpunkt zugewiesen bekommen haben und wenn die roten Blöcke iteriert werden idlen alle Threads, die einen schwarzen Gitterpunkt zugewiesen bekommen haben.

Beide Optionen haben ihre Nachteile. Diese sollen zur Einfachheit an einem Gitter demonstriert werden, welches mit Basis 8 diskretisiert wurde. Somit besteht für ein 2D-Gitter jeder Block (abgesehen vom Rand) aus 8×8 Gitterpunkten. Es soll ebenfalls angenommen werden, dass der erste Block in einer Zeile bei einer Speicheradresse anfängt, die ein Vielfaches von 16 sein soll. Für Option 1 würde der Speicher nach Kapitel 4.2 nicht korrekt angefordert werden. Für den Zugriff auf das erste Element in einem schwarzen Block mit Dimension 8 in jeder Richtung würden automatisch die nächsten 16 Wörter mit angefordert werden. Da die Blockgröße aber nur 8 ist, werden 8 zusätzliche Wörter umsonst angefordert, mit denen nicht gerechnet wird. Dies trifft in leicht abgeänderter Weise auch auf andere Basen zu.
Option 2 hat unter denselben Rahmenbedingungen den Nachteil, dass die Hälfte aller Threads idlen und nichts zu rechnen haben. Dies würde die potentiell zu erreichende FLOP-Rate um einen Faktor 2 dämpfen. Daher kommt die Überlegung, dass wenn bei Option 2 die Hälfte aller Threads nichts zu rechnen haben, diese trotzdem benutzen werden. Als Resultat wird dann anstelle einer Red-Black Gauß-Seidel-Iteration eine Jacobi-Iteration mit den Blöcken angewandt. Heuristisch gesehen ist zu erwarten, dass die Red-Black Gauß-Seidel-Iteration die besseren Ergebnisse liefert.
Ein entsprechender Pseudocode für einen V-Zyklus mit dem Simplified-Block-Layout und Jacobi-Iteration für die Blöcke findet sich in Algorithmus 5.3.

Algorithmus 5.3 *Pseudocode V-Zyklus mit Simplified-Block-Layout*

for Ω_k ... Ω_2 ***do***
  loop *Anzahl Vorglättungsschritte*
    Berechne Residuum r_h
    loop *Anzahl Iterationen, um Block zu lösen*
      Iteriere $A_h e_h = r_h$ *auf allen Blöcken*
    end loop
    $u_h += e_h$

    Berechne Residuum r_h
    loop *Anzahl Iterationen, um Block zu lösen*
      Iteriere $A_h e_h = r_h$ *auf allen Blöcken*
    end loop
    $u_h += e_h$
  end loop
  Berechne Residuum r_h
  Restringiere r_h *auf* f_{h-1}
end for

Löse gröbstes Gitter

for Ω_2 ... Ω_k ***do***
  Interpoliere u_{h-1} *auf* e_h
  $u_h += e_h$
  loop *Anzahl Nachglättungsschritte*
    Berechne Residuum r_h
    loop *Anzahl Iterationen, um Block zu lösen*
      Iteriere $A_h e_h = r_h$ *auf allen Blöcken*
    end loop
    $u_h += e_h$

    Berechne Residuum r_h
    loop *Anzahl Iterationen, um Block zu lösen*
      Iteriere $A_h e_h = r_h$ *auf allen Blöcken*
    end loop
    $u_h += e_h$
  end loop
end for

Es ist anzumerken, dass mit dem Simplified-Block-Layout der Umkopierschritt des Speichers entfällt und mit der gleichen Effizienz 2 mal so viele Vor- und Nachglät-

tungsschritte durchgeführt werden können. Ebenfalls ist zu beachten, dass sowohl im V-Zyklus mit Advanced-Block-Layout und im V-Zyklus mit Simplified-Block-Layout das Lösen des gröbsten Gitters ebenfalls iterativ erfolgt. Dies erlaubt die Wiederverwendung von Code und ist gerechtfertigt, da das gröbste Gitter aus einen Block mit Dimension $b-1$ besteht und alle Blöcke auf den feineren Gittern ebenfalls iterativ gelöst werden.
Die Restriktion auf Diskretisierungen mit gerader Basis wie beim Advanced-Block-Layout gilt nicht für das Simplified-Block-Layout, da der Speicher nicht umkopiert werden muss.

5.3 Andere Operationen

An dieser Stelle soll noch kurz auf die anderen Operationen eingegangen werden. Diese sind hauptsächlich die Restriktion, die Interpolation und die Berechnung des Residuums. Die in den vorherigen Kapiteln vorgestellten Algorithmen sind so implementiert worden, dass die Gitter nur zu Beginn des V-Zyklus von der CPU auf die GPU kopiert werden müssen. Um sich die Resultate anzusehen, müssen diese Daten am Ende eines V-Zyklus wieder von der GPU herunter kopiert werden. Alle anderen Operationen erfolgen in dem Speicher, der auf der GPU liegt.

5.3.1 Restriktion

Bei der Restriktion des Residuums von einem feinen zu einem groben Gitter hin erhält jeder Thread einen Gitterpunkt auf dem groben Gitter und restringiert für diesen alle zu ihm gehörenden feinen Gitterpunkte. In Pseudocode erfolgt dies nach Algorithmus 5.4.

Algorithmus 5.4 *Restriktion von einem feinen zu einem groben Gitter hin*

```
if Mein Gitterpunkt liegt am Rand then
    return
end if
p_g := 0
for Alle meine feinen Gitterpunkt p_f do
    Berechne Gewichtungskoeffizieten k für p_f
    p_g += k · p_f
end for
Speichere p_g als Restriktion auf den groben Gitterpunkt
```

Dies ist zwar nicht optimal in Bezug auf Speicherzugriff für die GPU, jedoch war es zeitlichen für diese Master-Thesis nicht möglich, eine entsprechende Version zu

implementieren. Eine optimalere Implementierung würde jedem Thread einen feinen Gitterpunkt zuweisen. Dies ist jedoch schwierig zu synchronisieren, da die GPU Threads nur innerhalb eines Half-Warps synchronisieren kann. Die von einem Thread berechneten Koeffizienten $k \cdot p_f$ müssten dann mit einer Summenoperation von einem Thread auf den groben Gitterpunkt reduziert werden. Aus Effizienzgründen müsste man diese Werte im Shared-Memory (OpenCL Nomenklatur) ablegen. Das Problem hierbei ist, dass Shared-Memory nur für einen Half-Warp verfügbar ist. Trotz dieses nicht optimalen Vorgehens bietet die Implementierung eine angemessene Performance.

5.3.2 Interpolation

Da es keine Synchronisierungsprobleme für die Interpolation des Fehlers von einem groben zu einem feinen Gitter hin gibt, wie sie in Kapitel 5.3.1 angedeutet wurden, erhält für die Interpolation jeder Thread einen feinen Gitterpunkt. Zusätzlich ist zu beachten, dass sich jeder feine Gitterpunkt aus mehreren umliegenden groben Gitterpunkten errechnet. Für den 3D-Fall sind dies jeweils 8 grobe Gitterpunkte. Ein entsprechender Pseudocode ergibt sich analog zur Restriktion zu Algorithmus 5.5.

Algorithmus 5.5 *Interpolation von einem groben zu einem feinen Gitter hin*

```
if Mein Gitterpunkt liegt am Rand then
    return
end if
p_f := 0
for Alle involvierten groben Gitterpunkte p_g do
    Berechne Gewichtungskoeffizieten k für p_g
    p_f += k · p_g
end for
Speichere p_f als Interpolation auf den feinen Gitterpunkt
```

Trotz eines nicht optimalen Speicherzugriffs bietet die Implementierung eine angemessene Performance.

5.3.3 Berechnung des Residuums

Für die Berechnung des Residuums erhält jeder Thread einen Gitterpunkt und berechnet aus den umliegenden Gitterpunkten das Residuum. Ein entsprechender Pseudocode findet sich in Algorithmus 5.6.

Algorithmus 5.6 *Berechnung des Residuums*
if *Mein Gitterpunkt liegt am Rand* ***then***
return
end if
Berechne das Residuum für meinen Gitterpunkt
Speichere das Residuum in ein entsprechendes Speicherobjekt

Trotz eines ebenfalls nicht optimalen Speicherzugriffs bietet die Implementierung eine angemessene Performance.

5.3.4 Abbruchkriterium

Das mit dieser Master-Thesis entstandene Framework stellt eine Funktion bereit, die es dem Anwender des Frameworks ermöglicht, die diskrete l_2-Norm eines Gitters direkt auf der GPU zu berechnen. Mit dieser Funktion ist es dem Anwender dieses Frameworks möglich, ein Abbruchkriterium effizient zu implementieren. Das Abbruchkriterium entscheidet darüber, ob ein Algorithmus nach einem V-Zyklus terminieren soll, da das Residuum hinreichend klein ist. Dies kann absolut oder als Veränderung des Residuums zwischen zwei V-Zyklen gemessen werden.
Zur Berechnung der diskreten l_2-Norm wird das Gitter für u_h als linearer Speicher interpretiert. Dieser Speicher ist größer als der Speicher, den man erhalten würde, wenn man das 3D-Gitter, auf welchem die Werte gespeichert sind, Zeile für Zeile aneinander hängen würde. Dies liegt daran, dass es pro Zeile ein Padding auf 16 Wörter gibt. Zuerst wird von jedem Thread jeder Wert in diesem Gitter quadriert. Anschließend werden die Werte in jeweils disjunkte 2-er Gruppen aufgeteilt, wobei jede Gruppe aus zwei benachbarten Gitterpunkten besteht. Jeder Thread erhält eine solche Gruppe. Jeder Thread errechnet die Summe aus seinen beiden Werten und speichert diese in ein temporäres Speicherobjekt. Damit wurde ein Reduktionsschritt durchgeführt und die diskrete l_2-Norm muss nur noch auf einem Gitter berechnet werden, dass halb so groß wie das vorherige ist. Anschließend wiederholt sich dieser Reduktionsschritt, wobei die Rolle von Gitter und temporärem Gitter vertauscht werden. Nach dem letzten Reduktionsschritt ergibt sich die diskrete l_2-Norm zu $||r||_h = \sqrt{h^3 \cdot \text{ erster Wert des letzten Gitters}}$. Abbildung 5.7 demonstriert, wie die diskrete l_2-Norm berechnet wird. An jedem Pfeil steht jeweils, welche Operation auf den Wert in diesem Speicherelement angewendet wurde. Der Einfachheit halber wurde darauf verzichtet, temporäre Gitter zu benennen.

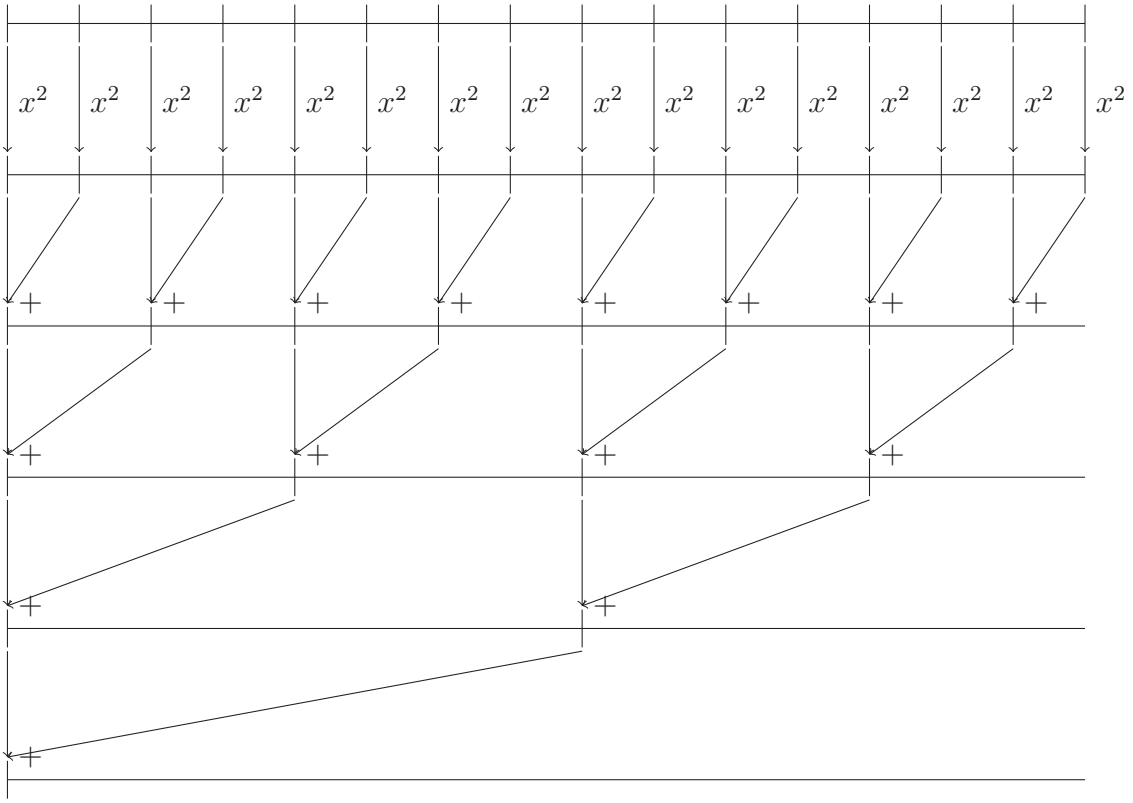

Abbildung 5.7: Berechnung der diskreten l_2-Norm eines linearen Speicherobjektes mit 16 Werten

6 Numerische Resultate

Dieses Kapitel dient dazu, die im vorherigen Kapitel vorgestellten V-Zyklen mit Advanced- und Simplified-Block-Layout miteinander zu vergleichen und mittels numerischer Resultate Aussagen über Konvergenz und Performance dieser beiden zu machen.

6.1 Rahmenbedingungen

Bei dem Testsystem, auf dem diese numerischen Resultate entstanden sind, handelt es sich um das in Kapitel 4.2 beschriebene GPU-Computing-Cluster JUDGE des Jülich Supercomputing Centres. Der Code wurde nicht parallel von mehreren GPU ausgeführt sonder ausschließlich von einer GPU die eine Peak-Performance von 1.03 TFLOPS Single Precision Floating Points aufweist. Es wurde ausschließlich mit *floats*, also in Single Precision auf der Tesla M2050 gerechnet. Als Testproblem wurde dabei die elliptische PDE

$$\begin{aligned} \Delta u(x,y,z) &= 3\pi^2 \cdot sin(\pi x) \cdot sin(\pi y) \cdot sin(\pi z), \text{ für } (x,y,z) \in (0,1)^3 \\ u(x,y,z) &= 0, \text{ für } (x,y,z) \in \partial(0,1)^3 \end{aligned}$$

ausgewählt. Zu diesem Problem ist die exakte Lösung mit

$$u(x,y,z) = sin(\pi x) \cdot sin(\pi y) \cdot sin(\pi z), \text{ für } (x,y,z) \in [0,1]^3$$

bekannt. Dadurch ist es möglich, sowohl das Residuum als auch den exakten Fehler auf dem Gitter in der diskreten l_2-Norm zu betrachten. In der Praxis kann der exakte Fehler oft nicht verwendet werden, da die Lösung nicht bekannt ist. Zur Zeitmessung nach jeweils einem V-Zyklus wurde die in *<sys/time.h>* definierte C-Funktion *gettimeofday* verwendet.
Weitere numerische Resultate liegen von einem Laptop vor, der mit einem Intel HD Graphics 4000 Chipsatz mit 512 MB RAM ausgerüstet ist. Jedoch unterstützt das verwendete Betriebssystem OpenCL auf dieser Grafikkarte nicht. Darum liegen für dieses System nur Resultate auf der verbauten CPU vor. Bei der CPU handelt es sich um einen Intel Core i5, der mit 1,8 GHz getaktet ist. Zusätzlich wurde die Implementierung auf einer ATI Radeon HD 4850 Grafikkarte mit 512 MB RAM und einer

Peak-Performance von ungefähr 1 TFLOPS Single Precision Floating Points getestet. Die hier erlangten Ergebnisse weichen jedoch stark von denen der Tesla M2050 ab. Eine mögliche Ursache hierfür könnte sein, dass es sich bei dem eingesetzten System, indem die ATI Radeon HD 4850 Grafikkarte verbaut ist, um ein Consumer-Produkt handelt und die Grafikkarte deshalb nicht für entsprechende numerische Verfahren ausgelegt ist. Diese zusätzlichen Resultate liegen der Implementierung bei und sollen in diesem Kapitel nicht weiter betrachtet werden.

6.2 Limitierungen

Für diese hier erwähnten Resultate gibt es Limitierungen, die an dieser Stelle kurz erwähnt werden sollen. Der V-Zyklus mit Advanced-Block-Layout unterstützt, wie oben beschrieben, keine Diskretisierungen mit ungeraden Basen. Zusätzlich weist die Architektur der GPU einige Limitierungen auf. Die Größe eines Speicherobjektes und der gesamte Speicher auf der GPU ist beschränkt. Die Tesla M2050 erlaubt es nur, Speicherobjekte zu allokieren, deren Größe 671 MB nicht überschreitet. Dadurch war es für das Advanced-Block-Layout nur möglich, Gebiete zu analysieren, die mit maximal $2^7 + 1$, $4^4 + 1$ oder $6^3 + 1$ Gitterpunkten diskretisiert wurden. Für größere Gebiete ist das Allokieren eines Speicherobjektes für das feinste Gitter fehlgeschlagen. Ebenfalls hat der V-Zyklus bei einer Diskretisierung mit $4^4 + 1$ Gitterpunkten mit der Fehlermeldung terminiert, dass kein weiterer Speicher allokiert werden kann. Eine genaue Erklärung hierfür ist nicht bekannt. Es könnte daran liegen, dass in einem V-Zyklus pro Diskretisierungslevel mehrfach Gitter allokiert und deallokiert werden, auf denen beispielsweise das Residuum und der Fehler gespeichert wird. Möglicherweise gibt die GPU den allokierten Speicher für diese Gitter nicht sofort wieder frei und stößt somit nach ein paar V-Zyklen an ihre Kapazitätsgrenze. Eine andere mögliche Ursache hierfür könnte sein, dass aufgrund einer falschen Implementierung an irgendeiner Stelle Speicher auf der GPU leaked und nicht wieder freigegeben wird. Die gleichen Limitierungen in Bezug auf verfügbaren Speicher traten auch für das Simplified-Block-Layout auf. Um die Ergebnisse der beiden Speicherlayouts besser vergleichen zu können, werden für das Simplified-Block-Layout in diesem Kapitel ebenfalls nur Diskretisierungen mit maximal $2^7 + 1$, $4^4 + 1$ oder $6^3 + 1$ Gitterpunkten betrachtet.

6.3 Konvergenzverhalten von Advanced- und Simplified-Block-Layout

Zum Vergleich des Konvergenzverhaltens zwischen Advanced- und Simplified-Block-Layout wurden jeweils 5 Vor- und Nachglättungsschritte, wie in Kapitel 5 definiert,

verwendet. Zusätzlich wurden 10 Iterationen verwendet, um einen Block zu lösen. Insgesamt wurden 10 V-Zyklen durchgeführt. Für dieses Kapitel wird sich auf das Residuum und den exakten Fehler in der diskreten l_2-Norm beschränkt. Tabelle 6.1 enthält die Resultate für das Advanced-Block-Layout und einer Diskretisierung mit $2^5 + 1$, $2^6 + 1$ und $2^7 + 1$ Gitterpunkten.

V-Zyklus	$2^5 + 1 = 33$		$2^6 + 1 = 65$		$2^7 + 1 = 129$	
	$\|\|r\|\|_h$	$\|\|e\|\|_h$	$\|\|r\|\|_h$	$\|\|e\|\|_h$	$\|\|r\|\|_h$	$\|\|e\|\|_h$
1	0.295447	0.006761	0.307201	0.007257	0.310012	0.007375
2	0.005873	0.000144	0.006389	0.000081	0.006857	0.000138
3	0.000182	0.000281	0.000389	0.000068	0.001464	0.000016
4	0.000092	0.000284	0.000366	0.000069	0.001456	0.000016
5	0.000092	0.000284	0.000365	0.000069	0.001455	0.000017
6	0.000091	0.000284	0.000364	0.000069	0.001455	0.000017
7	0.000091	0.000284	0.000361	0.000070	0.001459	0.000017
8	0.000091	0.000284	0.000362	0.000070	0.001455	0.000017
9	0.000092	0.000284	0.000363	0.000070	0.001457	0.000017
10	0.000093	0.000284	0.000363	0.000070	0.001456	0.000017

Tabelle 6.1: Konvergenz Advanced-Block-Layout für $2^5 + 1$, $2^6 + 1$ und $2^7 + 1$ Gitterpunkte

Tabelle 6.2 zeigt dieselben Resultate für das Simplified-Block-Layout.

V-Zyklus	$2^5 + 1 = 33$		$2^6 + 1 = 65$		$2^7 + 1 = 129$	
	$\|\|r\|\|_h$	$\|\|e\|\|_h$	$\|\|r\|\|_h$	$\|\|e\|\|_h$	$\|\|r\|\|_h$	$\|\|e\|\|_h$
1	0.325084	0.010683	0.338528	0.011355	0.341629	0.011511
2	0.010092	0.000056	0.010965	0.000298	0.011362	0.000359
3	0.000348	0.000273	0.000640	0.000059	0.002676	0.000004
4	0.000189	0.000284	0.000656	0.000070	0.002902	0.000017
5	0.000196	0.000284	0.000730	0.000071	0.002935	0.000017
6	0.000205	0.000284	0.000747	0.000071	0.002956	0.000017
7	0.000203	0.000284	0.000753	0.000071	0.002969	0.000017
8	0.000201	0.000284	0.000763	0.000071	0.002980	0.000017
9	0.000199	0.000284	0.000760	0.000071	0.002996	0.000017
10	0.000202	0.000284	0.000767	0.000071	0.003005	0.000017

Tabelle 6.2: Konvergenz Simplified-Block-Layout für $2^5 + 1$, $2^6 + 1$ und $2^7 + 1$ Gitterpunkte

Die Tabellen 6.3 und 6.4 zeigen die Ergebnisse für $4^3 + 1$, $4^4 + 1$ und $6^3 + 1$

Gitterpunkte. Ein Strich bedeutet, dass keine Ergebnisse vorliegen.

V-Zyklus	$4^3+1=65$		$4^4+1=257$		$6^3+1=217$	
	$\|\|r\|\|_h$	$\|\|e\|\|_h$	$\|\|r\|\|_h$	$\|\|e\|\|_h$	$\|\|r\|\|_h$	$\|\|e\|\|_h$
1	0.909092	0.018789	0.966699	0.018711	0.259558	0.023791
2	0.050275	0.000934	0.057140	0.001001	0.095446	0.001611
3	0.002797	0.000018	0.008019	0.000050	0.008984	0.000104
4	0.000472	0.000068	0.006620	0.000009	0.005160	0.000004
5	0.000417	0.000070	-	-	0.005110	0.000003
6	0.000416	0.000070	-	-	0.005108	0.000005
7	0.000419	0.000070	-	-	0.005109	0.000005
8	0.000418	0.000070	-	-	0.005108	0.000005
9	0.000421	0.000071	-	-	0.005106	0.000005
10	0.000422	0.000071	-	-	0.005104	0.000006

Tabelle 6.3: Konvergenz Advanced-Block-Layout für 4^3+1, 4^4+1 und 6^3+1 Gitterpunkte

V-Zyklus	$4^3+1=65$		$4^4+1=257$		$6^3+1=217$	
	$\|\|r\|\|_h$	$\|\|e\|\|_h$	$\|\|r\|\|_h$	$\|\|e\|\|_h$	$\|\|r\|\|_h$	$\|\|e\|\|_h$
1	0.681260	0.019999	0.696220	0.019965	0.879362	0.023551
2	0.039680	0.001072	0.042464	0.001140	0.065639	0.001583
3	0.002346	0.000006	-	-	0.007198	0.000101
4	0.000522	0.000067	-	-	0.005517	0.000006
5	0.000575	0.000071	-	-	0.005617	0.000003
6	0.000587	0.000071	-	-	0.005649	0.000005
7	0.000587	0.000071	-	-	0.005667	0.000005
8	0.000589	0.000071	-	-	0.005652	0.000006
9	0.000592	0.000071	-	-	0.005677	0.000006
10	0.000596	0.000071	-	-	0.005715	0.000006

Tabelle 6.4: Konvergenz Simplified-Block-Layout für 4^3+1, 4^4+1 und 6^3+1 Gitterpunkte

Abbildung 6.1 zeigt das Konvergenzverhalten des Residuums und des Fehlers für die beiden Speicherlayouts auf dem größten Gitter, welches mit 6^3+1 Gitterpunkten diskretisiert wurde. Die Speicherlayouts wurden mit ABL (Advanced-Block-Layout) und SBL (Simplified-Block-Layout) abgekürzt.

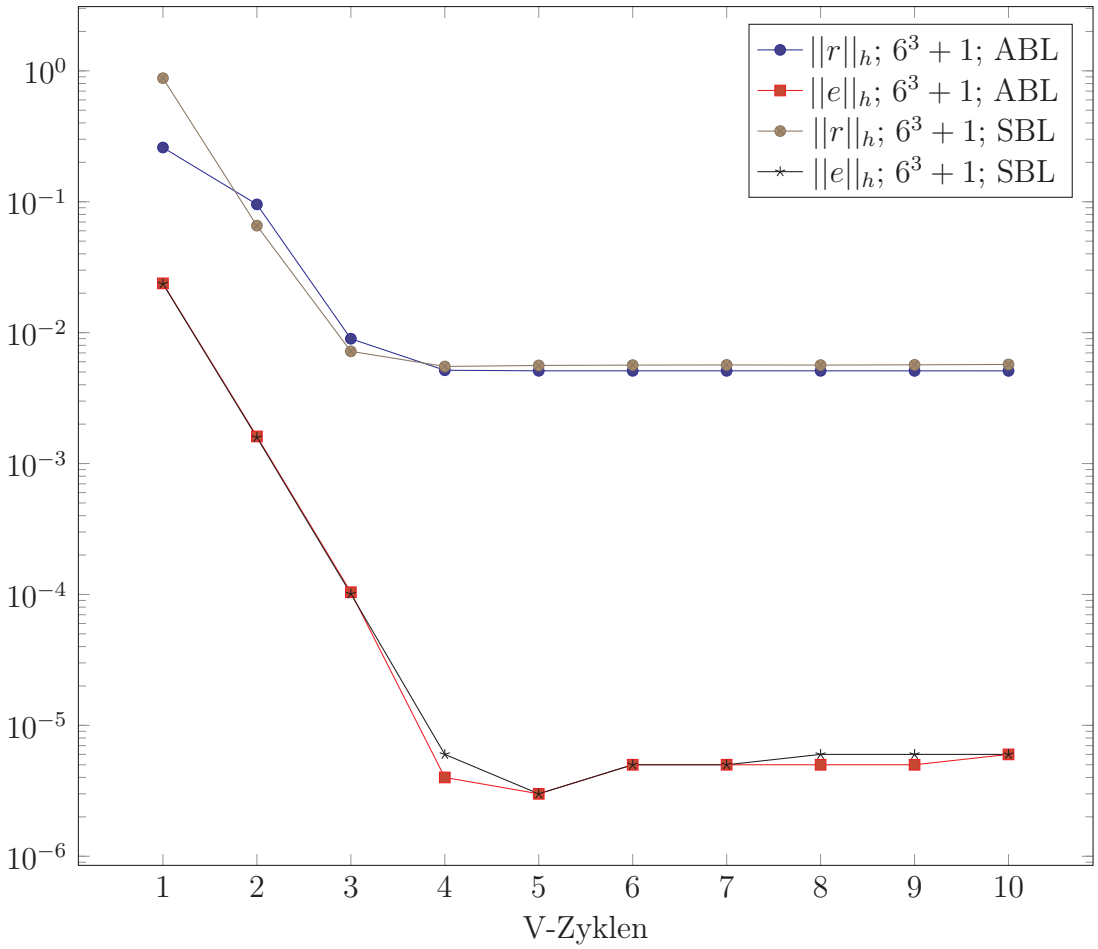

Abbildung 6.1: Konvergenz auf dem größten Gittern mit $6^3 + 1$ Gitterpunkten

Die Ergebnisse des Advanced-Block-Layouts sind in Bezug auf den exakten Fehler in den meisten Fällen leicht besser als die des Simplified-Block-Layouts, da in ersterem die Blöcke in einer Red-Black Gauß-Seidel-Manier bearbeitet werden. Dies ist vor allem der Fall, obwohl für das Simplified-Block-Layout problemlos doppelt so viele Vor- und Nachglättungsschritte verwendet werden konnten. In Bezug auf die Anzahl der benötigten V-Zyklen, um den Fehler bis auf Niveau des Diskretisierungsfehlers zu dämpfen, unterscheiden sich für das hier verwendetet Testproblem Advanced- und Simplified-Block-Layout nicht. Darum wird im nächsten Schritt betrachtet, wie sich Advanced- und Simplified-Block-Layout in ihrer Performance unterscheiden.

6.4 Performance von Advanced- und Simplified-Block-Layout

Zum Vergleich der Performance zwischen Advanced- und Simplified-Block-Layout wurden dieselben Randbedingungen wie in Kapitel 6.3 gewählt. Dies bedeutet, dass jeweils 5 Vor- und Nachglättungsschritte verwendet wurden. Zusätzlich wurden 10 Iterationen verwendet, um einen Block zu lösen. Insgesamt wurden 10 V-Zyklen durchgeführt. Diesmal wurde die Zeit in Sekunden pro V-Zyklus gemessen und die erreichten Floating Point Operations Per Second (FLOPS) berechnet. Tabellen 6.5 und 6.6 zeigen die jeweiligen Resultate für Gebiete, die mit 2^5+1, 2^6+1 und 2^7+1 Gitterpunkten diskretisiert wurden.

V-Zyklus	$2^5+1=33$		$2^6+1=65$		$2^7+1=129$	
	Zeit	GFLOPS	Zeit	GFLOPS	Zeit	GFLOPS
1	0.217761	1.897112	0.405095	8.085030	1.263347	20.717985
2	0.220698	1.871866	0.405618	8.074607	1.256099	20.837537
3	0.218811	1.888008	0.405286	8.081217	1.266431	20.667537
4	0.218063	1.894485	0.351479	9.318352	1.258561	20.796776
5	0.218133	1.893876	0.337884	9.693282	1.256661	20.828214
6	0.218147	1.893756	0.338021	9.689354	1.259962	20.773650
7	0.218382	1.891719	0.338249	9.682822	1.259029	20.789046
8	0.214308	1.927678	0.335298	9.768047	1.263895	20.709004
9	0.208869	1.977875	0.338726	9.669192	1.255149	20.853308
10	0.211816	1.950359	0.336810	9.724190	1.275359	20.522852

Tabelle 6.5: Performance Advanced-Block-Layout für 2^5+1, 2^6+1 und 2^7+1 Gitterpunkte

V-Zyklus	$2^5 + 1 = 33$		$2^6 + 1 = 65$		$2^7 + 1 = 129$	
	Zeit	GFLOPS	Zeit	GFLOPS	Zeit	GFLOPS
1	0.169350	4.406192	0.301999	19.760255	1.210847	39.427563
2	0.170043	4.388238	0.305373	19.541927	1.206556	39.567784
3	0.169791	4.394750	0.302413	19.733200	1.233328	38.708875
4	0.170028	4.388623	0.302427	19.732276	1.294369	36.883407
5	0.168541	4.427350	0.305216	19.551978	1.207919	39.523127
6	0.168313	4.433342	0.301295	19.806417	1.235948	39.626822
7	0.168554	4.427004	0.302572	19.722818	1.215245	39.284868
8	0.167801	4.446868	0.302587	19.721848	1.212941	39.359496
9	0.168594	4.425953	0.302749	19.711287	1.214567	39.306805
10	0.167920	4.443720	0.300570	19.854183	1.204142	39.647110

Tabelle 6.6: Performance Simplified-Block-Layout für $2^5 + 1$, $2^6 + 1$ und $2^7 + 1$ Gitterpunkte

Wie zu erwarten ist, unterscheiden sich die gemessenen Zeiten und die berechneten FLOPS pro V-Zyklus für ein Speicherlayout nur minimal. Darum sollen für die Gebiete, die mit 4^3+1, 4^4+1 und 6^3+1 Gitterpunkten diskretisiert wurden, nur die Werte mit kürzester gemessener Zeitspanne angegeben werden. Für das Advanced-Block-Layout wurden die folgenden Werte gemessen:

- $4^3 + 1$ Gitterpunkte: 0.238837 Sekunden mit 12.250450 GFLOPS
- $4^4 + 1$ Gitterpunkte: 5.091857 Sekunden mit 36.554632 GFLOPS
- $6^3 + 1$ Gitterpunkte: 3.353493 Sekunden mit 32.987157 GFLOPS.

Für das Simplified-Block-Layout entsprechend die nächsten Werte:

- $4^3 + 1$ Gitterpunkte: 0.215886 Sekunden mit 24.571963 GFLOPS
- $4^4 + 1$ Gitterpunkte: 4.960954 Sekunden mit 68.435386 GFLOPS
- $6^3 + 1$ Gitterpunkte: 2.774216 Sekunden mit 72.693519 GFLOPS.

Festzuhalten ist, dass ein V-Zyklus in den meisten Fällen mit Simplified-Block-Layout sowohl schneller ist, als auch eine annähernd doppelt so hohe FLOP-Rate im Vergleich zum Advanced-Block-Layout ausweist. Dies liegt zum einen am benötigten Umkopieren des Speichers im Advanced-Block-Layout, welches Zeit benötigt und nicht zur FLOP-Rate beiträgt. Zum anderen werden an der Stelle im Simplified-Block-Layout alle Blöcke iteriert, an der im Advanced-Block-Layout nur die roten bzw. schwarzen Blöcke iteriert werden. Dadurch kommt vermutlich annähernd der

Faktor 2 in der FLOP-Rate zustande.
Das Gitter, das im Simplified-Block-Layout mit $2^6 + 1 = 65$ Gitterpunkten diskretisiert wurde, hat drei V-Zyklen mit jeweils ungefähr $0,3s$ pro V-Zyklus benötigt, um das Testproblem bis auf Diskretisierungsfehler zu lösen. Dasselbe Gitter, welches mit $4^3 + 1 = 65$ Gitterpunkten diskretisiert wurde, hat vier V-Zyklen mit jeweils ungefähr $0,21s$ pro V-Zyklus benötigt, um das Testproblem bis auf Diskretisierungsfehler zu lösen. Damit ist die Diskretisierung mit $4^3 + 1$ Gitterpunkten um fast 7% schneller. Dies liegt daran, dass es für die Diskretisierung mit $4^3 + 1$ Gitterpunkten weniger Level pro V-Zyklus gibt, auf denen wenig gerechnet werden muss.
Betrachtet man im Simplified-Block-Layout weiterhin die Gitter, die mit $2^7+1 = 129$ Gitterpunkten und mit $6^3+1 = 217$ Gitterpunkten pro Dimension diskretisiert wurden, so stellt man fest, dass die Diskretisierung mit Basis 6 annähernd 3 mal soviel Zeit benötigt wie die Diskretisierung mit Basis 2, um das Testproblem bis auf Diskretisierungsfehler zu lösen. Jedoch schafft es die Diskretisierung mit Basis 6 ca. $4,7$ mal soviele Gitterpunkte zu lösen, wie die mit Basis 2. Dies spricht ebenfalls dafür, dass Diskretisierungen mit größerer Basis effizienter arbeiten.
In Kapitel 6.3 wurde in Bezug auf die Konvergenz von Advanced- und Simplified-Block-Layout festgestellt, dass sich diese beiden annähernd gleich für das Testproblem verhalten. Das Simplified-Block-Layout ist ein bisschen schneller als das Advanced-Block-Layout und weist eine annähernd doppelt so hohe FLOP-Rate auf. Aufgrund der FLOP-Rate ist zu erwarten, dass ein V-Zyklus mit Simplified-Block-Layout mehr Strom verbraucht als einer mit Advanced-Block-Layout. Nun liegt es am Anwender dieser Implementierung, einen Algorithmus zu verwenden, der langsamer ist und weniger Strom verbraucht, oder einen Algorithmus zu verwenden, der schneller ist und mehr Strom verbraucht. Dies ist für das Computing-Cluster JUDGE interessant, da in diesem 108 dieser Grafikkarten zur Verfügung stehen und ein Senkung des Stromverbrauchs zu Kosteneinsparungen führen kann.
Als nächstes soll analysiert werden, wie viele Iterationen pro Block nötig sind, um ein angemessenes Konvergenzverhalten zu erzeugen.

6.5 Angemessene Anzahl an Iterationen pro Block und Peak-Performance

In den vorherigen Kapiteln wurde stets 10 als Anzahl der Iterationen, um einen Block zu lösen, verwendet. Es ist zu erwarten, dass für größere Blöcke auch mehr Iterationen benötigt werden, um diese zu lösen. Um eine angemessene Schranke zu bestimmen, wurde zur Analyse des Testproblems das größtmögliche Gitter mit der größtmöglichen Blockgröße gewählt. Dies entspricht einer Diskretisierung mit $6^3 + 1$ Gitterpunkten. Im gleichen Kontext sollte die maximale Performance bestimmt wer-

den, die das hier implementierte Verfahren bieten kann. Aus diesem Grund wurde das V-Zyklus Mehrgitterverfahren mit Simplified-Block-Layout gewählt. In den folgenden Tabellen werden mit B in jeder Spalte die Anzahl der Iterationen angegeben, die verwendet wurden, um einen Block zu lösen. Die Tabellen enthalten zusätzlich alle V-Zyklen, die benötigt worden sind, um das Problem bis auf Diskretisierungsfehler zu lösen. In den letzten 2 Zeilen finden sich die jeweils besten Zeiten in Sekunden für einen V-Zyklus mit entsprechender Performance in GFLOPS. Tabelle 6.7 zeigt die Konvergenz in Bezug auf Residuum und exakten Fehler für 1, 2 und 3 Iterationen, um einen Block zu lösen.

V-Zyklus	$B = 1$		$B = 2$		$B = 3$	
	$\|\|r\|\|_h$	$\|\|e\|\|_h$	$\|\|r\|\|_h$	$\|\|e\|\|_h$	$\|\|r\|\|_h$	$\|\|e\|\|_h$
1	2.072854	0.051332	1.013558	0.032961	0.907918	0.029879
2	0.405183	0.007576	0.101133	0.003101	0.078784	0.002546
3	0.083167	0.001127	0.011362	0.000287	0.008644	0.000212
4	0.016660	0.000163	0.005043	0.000026	0.005161	0.000018
5	0.005974	0.000022	0.004869	0.000004	0.005097	0.000001
6	0.005299	0.000001	-	-	-	-
Zeit	1.278621		1.426805		1.585113	
Performance	29.445403		39.160033		46.746162	

Tabelle 6.7: Konvergenz mit 1, 2 und 3 Iterationen, um einen Block zu lösen

Auffällig bei diesen ersten drei Werten für B ist, dass schon zwei Iterationen pro Block die Anzahl der V-Zyklen, um das Testproblem bis auf Diskretisierungsfehler zu lösen, um eins verringert. Bis $B = 5$ wurden immer noch 5 Iterationen benötigt. Für $B = 10$ lediglich noch 4 mit einer Performance von 77.650853 GFLOPS. Selbst für $B = 50$ waren immer noch 4 V-Zyklen von Nöten. Die resultierende FLOP-Rate konvergierte für wachsendes B bis $B = 50$ gegen etwas über 100 GFLOPS, was 10% der eigentlichen Peak-Performance der Tesla M2050 ausmacht. Abbildung 6.2 stellt die gemessenen FLOP-Raten gegenüber den Verwendeten Iterationen B in einem Diagramm dar.

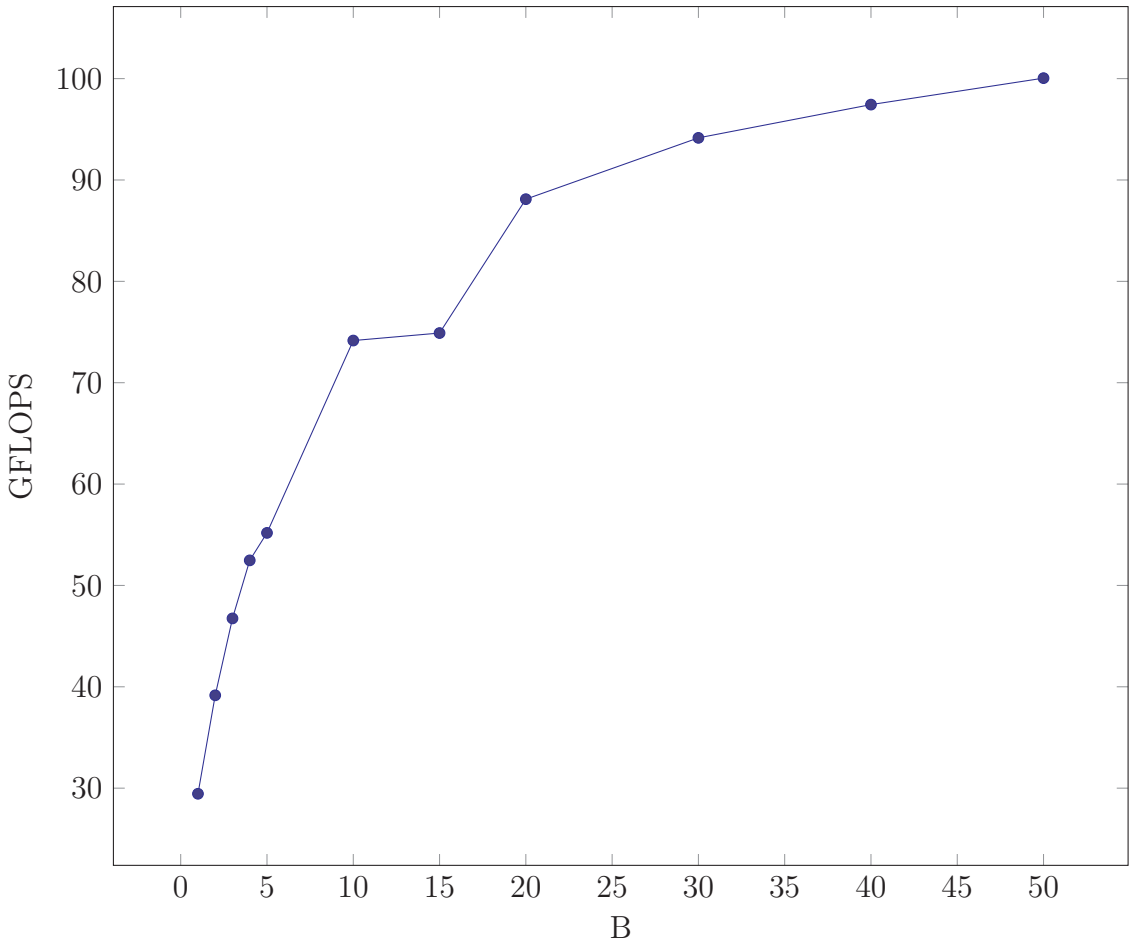

Abbildung 6.2: FLOP-Entwicklung für wachsendes B

Die minimal gemessene Zeit pro V-Zyklus für unterschiedliche B wird in Abbildung 6.3 dargestellt.

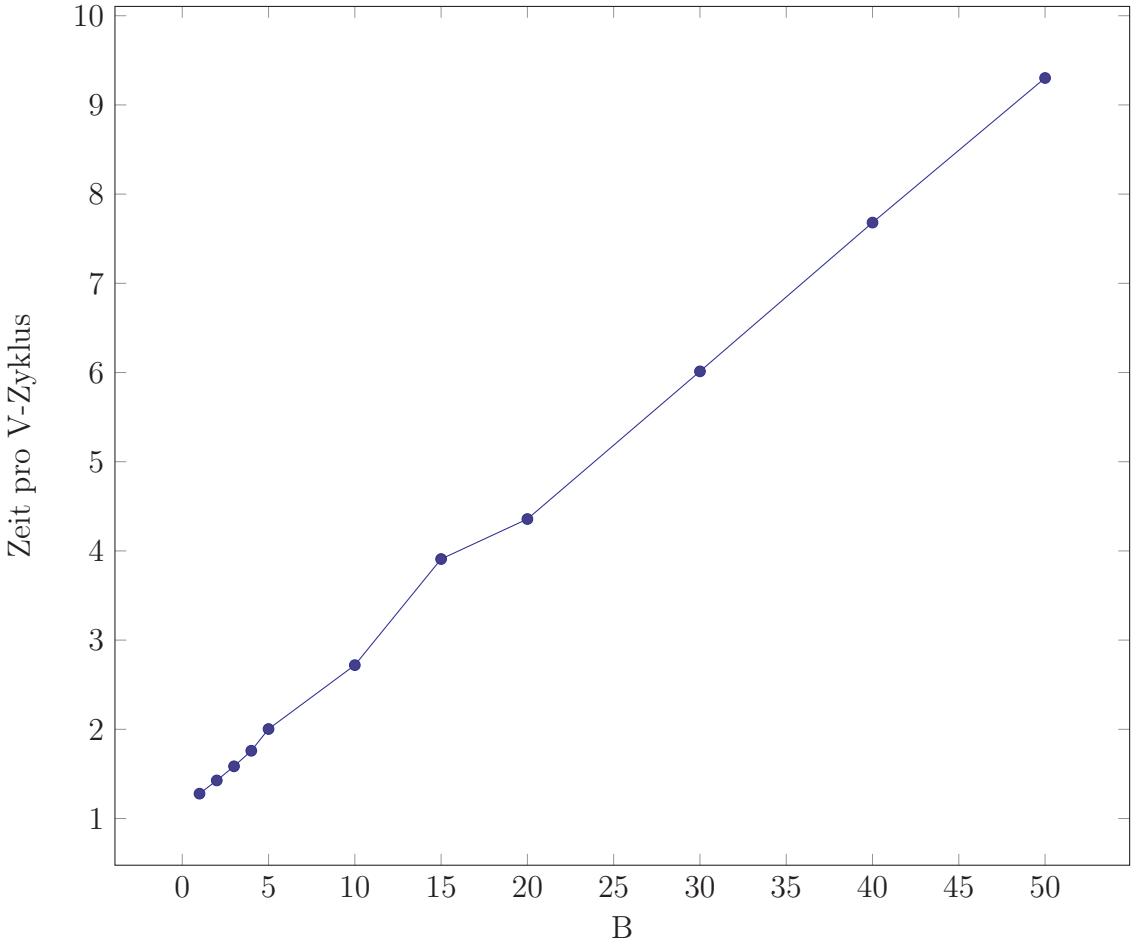

Abbildung 6.3: Zeit-Entwicklung für wachsendes B

Die benötigte Zeit scheint linear zu wachsen, wobei die erreichte FLOP-Rate gegen einen Wert von über 100 GFLOPS zu konvergieren scheint. Da aber für $B \geq 10$ immer 4 V-Zyklen von Nöten waren, ist 10 eine gute Anzahl an Iterationen, um einen Block zu lösen. Für kleinere Blöcke könnte dieser Wert unter 10 liegen, jedoch ist 10 eine angemessene Schranke für Performance und Zeitaufwand für Blöcke bis zu einer Größe von $6 \times 6 \times 6$.
Mit wachsendem B steigt auch die Zeit, die in einem V-Zyklus für die Iteration auf den Blöcke verwendet wird. Da die FLOP-Rate mit wachsendem B wächst, bestätigt dies noch mal, dass der Hauptaufwand in der Iteration auf den Blöcke liegt. Mit wachsendem B wird die Interpolation und die Restriktion zunehmend kompensiert, welche beide sehr wenig Rechenaufwand pro Gitterpunkt haben.

6.6 Vergleich mit einer CPU Implementierung

Zum Vergleich der gemessenen Ergebnisse auf der GPU liegt eine vergleichbare Implementierung eines Mehrgitterverfahrens auf der CPU vor. Diese wurde ebenfalls auf JUDGE getestet. Auf JUDGE ist jeder Knoten mit zwei Intel Xeon X5650 Prozessoren ausgestattet. Jede CPU verfügt über 6 Kerne, hat Zugriff auf insgesamt 12 Threads und ist mit 2.66 GHz getaktet. Zum Vergleich mit den gemessenen Werten auf der GPU wird auf der CPU das Gitter, welches mit $2^8 + 1$ Punkten diskretisiert wurde, betrachtet. Tabelle 6.8 zeigt das Residuum in der diskreten l_2-Norm und die gemessenen Zeiten pro V-Zyklus für 1, 2 und 4 verwendete CPU's.

V-Zyklus	$CPU's = 1$		$CPU's = 2$		$CPU's = 4$	
	$\|\|r\|\|_h$	Zeit [s]	$\|\|r\|\|_h$	Zeit [s]	$\|\|r\|\|_h$	Zeit [s]
1	$6.679 \cdot 10^{-2}$	11.4957	$6.679 \cdot 10^{-2}$	6.1502	$6.679 \cdot 10^{-2}$	3.0282
2	$3.881 \cdot 10^{-3}$	11.4993	$3.881 \cdot 10^{-3}$	6.1505	$3.881 \cdot 10^{-3}$	3.0292
3	$1.746 \cdot 10^{-4}$	11.5003	$1.746 \cdot 10^{-4}$	6.2584	$1.746 \cdot 10^{-4}$	3.0287
4	$7.077 \cdot 10^{-6}$	11.4951	$7.077 \cdot 10^{-6}$	6.1519	$7.077 \cdot 10^{-6}$	3.0293

Tabelle 6.8: Gemessene Zeiten pro V-Zyklus für 1, 2 und 4 CPU's

Eine GPU, welche einen vergleichbaren Mehrgittercode auf einem Gebiet, welches mit $4^4 + 1$ Gitterpunkten diskretisiert wurde, rechnet, benötigt hingegen nur 5.091857 Sekunden. Dies entspricht einer zeitlichen Verbesserung um einen Faktor von ungefähr 2.257. Zusätzlich kann man die Ergebnisse eines Gitters, welches auf der GPU mit $6^3+1 = 217$ Gitterpunkten diskretisiert wurde, betrachten. Das Gebiet auf der CPU hat ca. 1.66 mal so viele Gitterpunkte, benötigt jedoch ca. 3.42 mal so lang. Die mit dieser Master-Thesis entstandene Implementierung ist dem entsprechend auf JUDGE auf der GPU effizienter als ein vergleichbarer Mehrgittercode auf der CPU.

7 Schluss

Ziel dieser Master-Thesis war es, eine effiziente Implementierung von Mehrgitterverfahren, die mit Blockglättern arbeiten, auf der GPU zu realisieren. Wie Kapitel 6 anschaulich demonstriert hat, ist dies mit 10% der maximal erreichbaren Peak-Performance gelungen. Nach einer Vorstellung des zu lösenden Problems und einer Einführung in bisherige numerische Lösungsmethoden wurden die Blockglätter vorgestellt. Diese ermöglichen es, große Gebiete effizienter auf mehrere Teilgebiete zu verteilen und diese im Parallelen zu lösen. Nach einer Einführung in die Architektur der GPU und ihre Besonderheiten wurde die entstandene Implementierung von Mehrgitterverfahren mit Hilfe von Blockglättern vorgestellt. Dabei wurden zwei Speicherlayouts entwickelt, mit Hilfe derer eine effiziente Blockglättung durchgeführt werden konnte. Das Simplified-Block-Layout hat sich dabei als das überlegene Speicherlayout auf der GPU bewiesen. Ebenfalls hat sich herausgestellt, dass schon 10 Iterationen pro Block mit maximaler Größe von $6 \times 6 \times 6$ Werten ausreichen, um diesen zu lösen und ein effizientes Mehrgitterverfahren zu implementieren.
Aufgrund bestehender Limitierungen der GPU war es nicht möglich, Gebiete zu lösen, die mit beispielsweise $8^4 + 1$ Gitterpunkten diskretisiert wurden. Diese Limitierungen werden im Laufe der Zeit mit besserer Hardware aufgehoben, wodurch dann beispielsweise Gitter verglichen werden können, die mit $2^{12} + 1 = 4^6 + 1 = 8^4 + 1 = 16^3 + 1$ Gitterpunkten diskretisiert wurden. Die numerischen Resultate lassen darauf schließen, dass Blockglätter mit größerer Blockgröße effizienter arbeiten. Eine optimale Anzahl an Iterationen, um beispielsweise einen 16-er Block zu lösen, muss dann noch gefunden werden.
Anschließend an diese Master-Thesis könnte untersucht werden, ob es nicht noch bessere und effizientere Implementierungen auf der GPU gibt, um mit Blockglättern zu rechnen. Ebenfalls kann diese hier entstandene Implementierung so ausgebaut werden, dass ein Anwender des entstandenen Frameworks nicht nur auf eine GPU beschränkt ist, sondern ein sehr großes Gebiet auf mehrere GPU verteilen kann.

A Literaturverzeichnis

[AER05] Arnold, Anton ; Ehrhardt, Matthias ; Rjasanow, Sergej: *Numerik partieller Differentialgleichungen*. Version: 02 2005. http://www-amna.math.uni-wuppertal.de/%7Eehrhardt/NumPar/pdf/PDE-Script.pdf, Abruf: 14.01.2013. Vorlesungsskript

[BHM00] Briggs, William L. ; Henson, Van E. ; McCormick, Steve F.: *A Multigrid Tutorial*. Second Edition. Society for Industrial and Applied Mathematics, 2000

[Feo12] Feo, John: *Eldorado*. http://cseweb.ucsd.edu/classes/wi05/cse141/Eldorado.pdf. Version: 2012, Abruf: 14.01.2013

[Fro90] Frommer, Andreas: *Lösung linearer Gleichungssysteme auf Parallelrechnern*. Braunschweig : Vieweg, 1990

[Fro05] Frommer, Andreas: *Algorithmen und Datenstrukturen II - Parallele Algorithmen*. Version: 2005. http://www2.math.uni-wuppertal.de/~frommer/manuscripts/ParalleleAlg.ps.gz, Abruf: 14.01.2013. Vorlesungsskript

[Goh09] Gohara, David: *OpenCL Tutorials*. http://www.macresearch.org/opencl. Version: 08 2009, Abruf: 14.01.2013

[ME12] Munshi (Ed.), Aaftab: *The OpenCL Specification 1.2*. http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf. Version: 11 2012, Abruf: 14.01.2013

[MO11] MacLachlan, S. P. ; Oosterlee, C. W.: Local Fourier analysis for multigrid with overlapping smoothers applied to systems of PDEs. In: *Num. Lin. Alg. Appl.* 18 (2011), S. 751–774

[NVI09] NVIDIA: *NVIDIA OpenCL Best Practices Guide Version 1.0*. http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf. Version: 08 2009, Abruf: 14.01.2013

[NVI12] NVIDIA: *CUDA C Programming Guide*. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. Version: 10 2012, Abruf: 14.01.2013

[PG12] Pulch, Roland ; Günther, Michael: *Einführung in die Numerische Mathematik*. Version: 06 2012. http://www-num.math.uni-wuppertal.de/fileadmin/mathe/www-num/teaching/einf_num12/Skript_Numerik1.pdf, Abruf: 14.01.2013. Vorlesungsskript

[Str06] Strang, Gilbert: *Mathematical Methods for Engineers II*. Version: 2006. http://ocw.mit.edu/courses/mathematics/18-086-mathematical-methods-for-engineers-ii-spring-2006/index.htm, Abruf: 14.01.2013. Vorlesungsskript

[TOS01] Trottenberg, U. ; Oosterlee, C. ; Schüller, A.: *Multigrid*. San Diego : Academic Press, 2001