

Softwarepraktikum Parallele Numerik

Fabian Immanuel Miltenberger, Sébastien Thill, Thore Mehr

Betreuer: Markus Hoffmann, Thomas Becker
Lehrstuhl für Rechnerarchitektur und Parallelverarbeitung

Zusammenfassung Im Rahmen dieses Praktikums haben wir viel gelernt.

Projekt 1 OpenMP

In diesem Projekt lag der Schwerpunkt auf dem Kennenlernen der Bibliothek *OpenMP* sowie deren Handhabung. Weiter ging es um den *Intel Thread Checker*, ein Programm zum Analysieren von Programmcode auf potentielle Fehler in der Parallelisierung. Zu guter letzt haben wir uns mit der FEM-Methode beschäftigt, dabei im Speziellen mit dem Gauß-Seidel-Verfahren zum Lösen linearer Gleichungen wie sie bei der Differenzenmethode vorkommen. Zu guter letzt betrachteten wir einige andere Verfahren zum Lösen solcher Probleme und haben das CG-Verfahren implementiert.

Aufgabe 1.1 OpenMP

1.1.a) Wie in der Ausgabe 1.1 zu sehen, folgt die Reihenfolge der ausgeführten Fäden keinem bestimmten Muster. Auch die Reihenfolge zwischen verschiedenen Ausführungen ist in der Regel verschieden.

```
Hello World, this is Thread0
Hello World, this is Thread5
Hello World, this is Thread4
Hello World, this is Thread7
Hello World, this is Thread1
Hello World, this is Thread3
Hello World, this is Thread6
Hello World, this is Thread2
```

Listing 1.1. Beispielhafte Ausgabe des Programms bei Ausführung mit 8 Fäden.

1.1.b) Wie in Tabelle 1 zu sehen, nähert sich der *Speedup* mit zunehmenden N der Anzahl der Kerne an.

Die manuelle Variante hat eine schlechtere Performance, als die Variante ohne Parallelisierung. Dies lässt sich damit begründen, dass *OpenMP* bei der Variante mit *Reduction* weitere Optimierungen umsetzen kann, beispielsweise kann jeder Thread erst einmal all seine eigenen Beiträge aufsummieren, bevor am Ende nach einer einzigen Synchronisation (anstatt bei jeder Schleifenausführung) die Ergebnisse aller Threads aufsummiert werden.

Problemgröße N	Seq.	2 Threads		16 Threads		32 Threads		64 Threads	
	Laufzeit	Laufzeit	Speedup	Laufzeit	Speedup	Laufzeit	Speedup	Laufzeit	Speedup
10^6 critical	0,49	1,02	0,48	39,33	0,012	65,18	0,008	93,21	0,005
10^7 reduction	0,29	0,15	1,99	0,044	6,59	0,041	7,07	0,05	5,8
10^8 reduction	2,96	1,48	2	0,215	13,78	0,157	18,85	0,185	16
10^9 reduction	28,76	14,77	1,95	1,86	15,46	0,99	29,05	1,14	25,25

Tabelle 1. Laufzeiten der Berechnung von Pi unserer Parallelisierung gegenüber der sequentiellen Version. Getestet wurde auf dem Rechner *i82sn07*. Um Ungenauigkeiten auszugleichen wurden die Zeiten jeweils über 10 (für große Eingaben über 5) Durchläufe gemittelt.

Problemgröße N	Seq.	2 Threads		16 Threads		32 Threads	
	Laufzeit	Laufzeit	Speedup	Laufzeit	Speedup	Laufzeit	Speedup
1000	6,02	4,35	1,38	1,51	3,99	0,77	7,88
2000	24,65	17,41	1,44	5,92	4,16	3,11	7,92
4000	96,24	70,47	1,37	23,66	4,07	11,913	8,07
8000	394,05	278,34	1,44	94,60	4,16	47,85	8,24

Tabelle 2. Laufzeiten des Mandelbrot Programms unserer Parallelisierung gegenüber der sequentiellen Version. Getestet wurde auf dem Rechner *i82sn07*. Um Ungenauigkeiten auszugleichen wurden die Zeiten jeweils über 10 (für große Eingaben über 5) Durchläufe gemittelt.

1.1.c) Wie an der Tabelle 2 abgelesen werden kann, beträgt die Beschleunigung für 2 Threads etwa 1,4, für 16 Threads etwa 4 und für 32 Threads etwa 8. Dass die Beschleunigung schlechter ist, als bei der Berechnung von PI, liegt meiner Meinung nach daran, dass Speicherverwaltung und Caches einen wesentlichen Anteil der Laufzeit ausmachen.

Aufgabe 1.2 Testtools

1.2.a)

- Memory read at "`demo_with_bugs.c`":**27** conflicts with a prior memory write at "`demo_with_bugs.c`":**26** (flow dependence)
Kann behoben werden, indem die Schleife aufgeteilt wird in zwei Schleifen mit `#pragma omp prallel for simd`.
- Memory read at "`demo_with_bugs.c`":**50** conflicts with a prior memory write at "`demo_with_bugs.c`":**45** (flow dependence)
Kann behoben werden, indem das `nowait` statement weggelassen wird.
- Memory read at "`demo_with_bugs.c`":**66** conflicts with a prior memory write at "`demo_with_bugs.c`":**66** (flow dependence)
Memory write at "`demo_with_bugs.c`":**67** conflicts with a prior memory write at "`demo_with_bugs.c`":**66** (anti dependence)
Kann behoben werden, indem man das `x` als privat markiert, etwa durch Ersetzen des Pragma durch `#pragma omp parallel for private(x)`.

4. Vom Thread Checker nicht erkannt.
Kann behoben werden in dem das Pragma auf `#pragma omp parallel for lastprivate(x)` geändert wird.
5. Memory read at "demo_with_bugs.c":98 conflicts with a prior memory write at "demo_with_bugs.c":98 (flow dependence)
Memory write at "demo_with_bugs.c":98 conflicts with a prior memory write at "demo_with_bugs.c":98 (OUTPUT dependence)
Kann behoben werden in dem das Pragma auf `#pragma omp parallel for reduction(+:sum)` geändert wird.

1.2.b) Um die Ausnutzung des Caches zu gewährleisten, wird die eine Matrix als Zeilen-, die andere als Spaltenmatrix verarbeitet und nach dem *ikj*-Algorithmus multipliziert. Es wurden eine sequentielle, sowie drei parallelisierte Versionen untersucht. Die parallelisierten Versionen unterscheiden sich wie folgt:

1. Innerste Schleife mittels *Reduction* parallelisiert
2. Mittlere Schleife parallelisiert
3. Äußere Schleife parallelisiert

In den Tabellen 3 und 4 wird der Einfluss von Optimierungsoptionen vom GCC und ICC untersucht. Die Messungen wurden auf *i82pc31* ausgeführt, da dieser über eine Intel CPU verfügt. Wie aus den Tabellen abgelesen werden kann, sind die Unterschiede bei den Optimierungsstufen O0 bis O3 gering. GCC scheint etwas schnelleren sequentiellen, ICC hingegen schnelleren parallelen Code zu generieren. Wenn Vektorisierung benutzt werden kann – beim ICC mit der Option *fast* – wird der Code viel schneller, allerdings auf Kosten der Größe, das Binary hat 1,4 MB statt 30 kB. Im Allgemeinen ist das Binary, das vom ICC erzeugt wird, etwa doppelt so groß, wie das, das vom GCC erzeugt wird.

	ICC					GCC			
Optionen	O0	O1	O2	O3	fast	O0	O1	O2	O3
sequentiell	9,26	1,19	1,17	1,17	0,25	9,56	1,16	1,16	1,16
reduction	4,99	1,90	1,88	1,88	1,40	6,06	2,21	2,21	2,21
mittlere	5,88	1,02	1,02	1,02	0,96	6,95	2,55	1,12	1,18
äußere	5,67	0,59	0,59	0,59	0,014	6,02	1,81	0,82	0,82

Tabelle 3. Laufzeiten in s mit N=1000. Getestet wurde auf dem Rechner *i82pc31*. Um Ungenauigkeiten auszugleichen, wurden die Zeiten jeweils über 10 Durchläufe gemittelt.

1.2.c) Anhand der Ausgabe von *ompP* kann leicht gesehen werden, dass wenige der Threads einen Großteil der Arbeit leisten, was bei der Verwendung von *reduction* zu erwarten ist. Durch die explizite Angabe des Scheduling und der Chunksize kann die Arbeit besser verteilt werden. In Tabelle 5 kann der Einfluss der Optionen auf die Laufzeit abgelesen werden.

	ICC					GCC			
Optionen	O0	O1	O2	O3	fast	O0	O1	O2	O3
sequentiell	74,58	9,9	9,88	9,86	2,27	76,75	9,76	9,75	9,73
reduction	36,67	10,99	10,98	10,95	8,54	43,20	13,76	12,54	12,48
mittlere	47,04	9,01	8,66	8,37	7,95	54,19	30,05	9,60	9,50
äußere	46,50	5,19	4,95	4,92	1,15	44,83	15,07	6,93	6,84

Tabelle 4. Laufzeiten in s mit N=2000. Getestet wurde auf dem Rechner *i82pc31*. Um Ungenauigkeiten auszugleichen, wurden die Zeiten jeweils über 10 Durchläufe gemittelt.

	auto	Chunks.=1	Chunks.=10	Chunks.=100	Chunks.=1000
N=4000	Maxiter=500				
static	0,99	0,38	0,35	0,89	5,02
dynamic	0,37	0,31	0,30	0,84	5,02
N=8000	Maxiter=500				
static	3,91	1,16	1,15	1,79	13,74
dynamic	1,20	1,08	1,08	1,76	13,74
	auto	Chunks.=1	Chunks.=10	Chunks.=100	Chunks.=1000
N=4000	Maxiter=1000				
static	1,91	0,60	0,54	1,69	9,70
dynamic	0,54	0,51	0,52	1,64	9,70
N=8000	Maxiter=1000				
static	1,97	1,95	2,07	3,44	26,84
dynamic	1,99	1,99	2,00	3,52	26,85

Tabelle 5. Einfluss von Scheduling und Chunksize auf die Laufzeit von Mandelbrot. Getestet wurde auf dem Rechner *i82sn07* mit 32 Threads. Um Ungenauigkeiten auszugleichen wurden die Zeiten jeweils über 10 Durchläufe gemittelt.

Aufgabe 1.3 Parallelisierung

1.3.a) Im Folgenden einige bekannte Größen der Parallelisierung, wie sie etwa in der Vorlesung Rechnerstrukturen [9] gelehrt wurden.

Vorab sei $T(n)$ die Ausführungszeit auf n Prozessoren, $P(n)$ die Anzahl der auszuführenden Einheitsoperationen

Der Speedup/die Beschleunigung $S(n)$:

$$S(n) = \frac{T(1)}{T(n)} \quad (1)$$

Die Effizienz $E(n)$:

$$E(n) = \frac{S(n)}{n} = \frac{T(1)}{n \cdot T(n)} \quad (2)$$

Der Mehraufwand $R(n)$:

$$R(n) = \frac{P(n)}{P(1)} \quad (3)$$

Der Parallelindex $I(n)$:

$$I(n) = \frac{P(n)}{T(n)} \quad (4)$$

Die Auslastung $U(n)$:

$$U(n) = \frac{I(n)}{n} = R(n) \cdot E(n) = \frac{P(n)}{n \cdot T(n)} \quad (5)$$

1.3.b)

– Race Condition

Auch bekannt als Wettlaufsituation. Mehrere Fäden greifen lesend und schreibend auf die gleiche Variable zu. Durch den gleichzeitigen Zugriff auf die Variable durch die anderen Fäden hängt das Ergebnis von der konkreten Ausführungsreihenfolge ab. Das Ergebnis kann der Erwartung entsprechen, muss aber nicht. Die Lösung hierfür ist Synchronisation, beispielsweise mittels atomarer Variablenzugriffe oder Barriere. Damit kann sicher gestellt werden, dass kein anderer Faden die Variable manipuliert während ein Faden mit ihr arbeitet.

– Dead lock

Auch bekannt als Verklemmung. Erfordert mindestens zwei Fäden, die gegenseitig auf eine Resource warten, die gerade vom jeweils anderen Faden bereits allokiert wurde (formal: Zyklus im Allokationsgraphen). Damit tatsächlich alle Beteiligten Fäden nicht mehr weiter laufen können, müssen folgende Bedingungen erfüllt sein:

- Ressourcen können nicht von außen freigegeben werden

- Fäden können weitere Ressourcen anfordern, und gleichzeitig weiterhin andere halten
- Eine Resource kann immer nur von einem Faden gehalten betreten werden
- Es besteht eine zyklische Abhängigkeit im Allokationsgraphen

– **Bibliotheken**

Eine Fehlerquelle kann die Verwendung von Bibliotheken in einem parallelen Kontext darstellen, die nicht hierfür konstruiert wurden. Wenn eine Bibliothek beispielsweise von verschiedenen Fäden aus aufgerufen wird, aber intern keine Synchronisationsmechanismen bereithält, kann es auch hier zu Wettlaufsituationen kommen, die jedoch unter Umständen noch schwieriger zu lokalisieren sind. Auch kann es passieren, dass eine geplante Parallelisierung sich aufgrund Synchronisationsmechanismen innerhalb einer Bibliothek nicht wie erwartet umsetzen lässt. In diesem Fall wird die Implementierung der Bibliothek zum Flaschenhals der Parallelisierung. Um solche Fehler zu vermeiden, blockieren manche Bibliotheken Aufrufe, die nicht aus einem bestimmten Faden stammen. Verlassen sollte sich der Entwickler auf solche Mechanismen jedoch nicht. Viele Bibliotheken geben inzwischen in ihrer Dokumentation Hinweise auf den möglichen Einsatz im parallelen Kontext.

– **Messeffekt**

Wenn man das Debuggen zum Softwareentwurf zählt, so wären als weitere Fehlerquelle auch *Messeffekte* zu nennen. Verbreitete Methoden zum Debuggen, wie etwa die Ausgabe von Information auf der Konsole, können bereits selbst die Ausführung des parallelen Programms beeinflussen. So kann es passieren, dass die Mechanismen, die einen Fehler aufspüren sollen, dazu führen, dass dieser nicht weiter auftritt.

– **Cacheeffekte**

Spielt ebenfalls weniger im Entwurf eine Rolle, als in der Implementierung. Die meisten Prozessoren haben einen eigenen Cache für jeden Rechenkern. Wenn nun verschiedene Fäden auf verschiedenen Kernen gleichzeitig auf der gleichen Variablen arbeiten, so kann es passieren, dass mangels Synchronisation zwischen den Caches der eine Faden die vom anderen Faden verursachte Änderung nicht *sieht*. Lösung hierfür sind spezielle Mechanismen des Prozessors, um Caches bei Bedarf zu Synchronisieren. In C kann man hierfür eine Variable als `volatile` definieren.

1.3.c) Im Folgenden werden einige parallele Architekturen genauer erläutert. Ein grober Vergleich zur CPU kann Tabelle 6 entnommen werden.

– **CPU**

Die CPU stellt einen großen Instruktionssatz zur Verfügung und kann komplexe Sprünge im Code abarbeiten. Durch Techniken wie Sprungvorhersagen lässt sich in diesen Fällen die Performance steigern und mit Out-of-Order Architekturen erhält man einen hohen Grad an Instruktionslevel-Parallelismus. Im Allgemeinen finden sich selten mehr als 8 CPU Kerne in einem herkömmlichen Rechner. Es bietet sich die „gewöhnliche“ Programmierung an.

– **GPU**

Moderne Rechner haben fast alle eine GPU verbaut, welche anders als die CPU keine komplexen Befehle abarbeiten kann, sondern durch eine sehr hohe Anzahl an einfachen Rechenkernen massive Parallelität bereitstellen kann. Dieser Beschleuniger arbeitet auf einem Befehlsstrom und führt diesen mehrfach aus (*Single Instruction, Multiple Data* – kurz *SIMD*). Um GPUs für ein eigenes Programm zu nutzen helfen Bibliotheken wie *CUDA* (nVidia spezifisch), oder *OpenCL*.

– **FPGA**

Ein FPGA Board kann beliebige Hardware-Eigenschaften bereitstellen, sofern diese vorher auf das FPGA aufgespielt wurden. Im Allgemeinen werden FPGAs zum Testen von Chips, welche sich noch in der Entwicklung befinden, verwendet. Aber das FPGA kann auch als „multifunktionelle“ Beschleunigungskarte eingesetzt werden (lässt sich sehr gut an Anforderungen anpassen). Um ein FPGA zu programmieren bietet sich *OpenCL* an, oder eine der Bibliotheken welche die implementierte Schaltung nutzt.

– **MIC** (Intel *Many Integrated Core*)

Die Grundidee von MICs ist, die gewöhnliche x86 CPU Architektur zur Verfügung zu stellen, aber dabei die Parallelität verglichen mit normalen CPUs zu erhöhen. Dies wird durch ein Zusammenschalten von mehreren CPUs auf einem Chip erreicht, welche dann über PCIe mit dem Hostsystem kommunizieren. Es wurde bewusst die x86 Architektur gewählt, damit bekannte Parallelisierungstools wie *OpenMP* oder *OpenCL* genutzt werden können.

Beschleuniger	Parallelität	Anwenderfreundlichkeit
CPU	Niedrig	Gut
GPU	Sehr hoch	Mittel
FPGA	Hoch	Abhängig von Bibliotheken
MIC	(Sehr) hoch	Gut

Tabelle 6. Verschiedene Beschleuniger im Vergleich, die CPU wird als Referenz verwendet.

Aufgabe 1.4 Parallelisierung

1.4.a) Zu aller erst nehmen wir eine Differenzierung der in der Aufgabenstellung genannten Variablen vor. Unser n bezeichne dasjenige, welches in der Beschreibung des Gauß-Seidel-Verfahrens auftritt. Das in dem gegebenen Problem genannte n benennen wir zu d um. Die restlichen Variablen behalten ihren Namen bei. Sei ein l für das Verfahren vorgegeben, dann ergeben sich einige

andere Größen wie folgt:

$$\begin{aligned} d &= 2^l - 1 \\ h &= \frac{1}{2^l} \\ n &= d^2 \end{aligned} \tag{6}$$

Anschaulich sollen die Werte von $u_{x,y}$ auf einem $(d+2) \cdot (d+2)$ großen Gitter berechnet werden. Die Abstände zwischen den Gitterpunkten sei dabei Gitterkonstante h . Durch die Vorgabe $u_{x,y} = 0$, für x, y auf dem Rand, vereinfacht sich die Problemstellung auf ein Gitter der Größe $d \cdot d$, da der Rand implizit als 0 angenommen werden kann. Zum Lösen des Problems geben wir den Gitterpunkten $u_{x,y}$ eine Ordnung, sodass u sich als einfacher Spaltenvektor mit $n = d \cdot d$ Elementen auffassen lässt. Anschaulich sieht diese Ordnung wie folgt aus:

$$u = (u_{1,1} \dots u_{d,1} u_{1,2} \dots u_{d,d})^T \tag{7}$$

Der Definitionsbereich von u (als Funktion aufgefasst) sei \mathbb{R}^2 (ergibt sich aus den gegebenen Randbedingungen). Damit befindet sich ein Gitterpunkt $u_{x,y}$ an der Position $(x \cdot h, y \cdot h)$. Damit berechnen wir den Wert von f für alle unsere Gitterpunkte, also $f_{x,y} = f(x \cdot h, y \cdot h)$. Für diese berechneten Werte von f nehmen wir die gleiche Ordnung vor wie für u und können somit f ebenfalls als einen Spaltenvektor der Größe n ansehen.

Die Matrix $A \in \mathbb{R}^{n \times n}$ sei definiert wie vorgegeben und ihre Elemente seien $(a_{i,j})$. Wir werden sehen, dass wir zur Lösung A nicht explizit berechnen und speichern müssen. Nun sind alle Zutaten zur Berechnung von u in $Au = h^2 f$ mittels Gauß-Seidel-Verfahren gegeben.

Die Implementierung geht nun Schritt für Schritt vor wie im Algorithmus vorgegeben. u^k sei u in der k -ten Iterierten von u . Wir beginnen mit $u^0 = 0$, wählen also 0 als Startvektor. Dies geschieht der Einfachheit wegen, es hat sich herausgestellt, dass es hierdurch bei keiner der gestellten Aufgaben zu Problemen kommt.

In jeder Iterierten k soll nun u^{k+1} berechnet werden. dies geschieht nach der gegebenen Berechnungsvorschrift

$$u_j^{k+1} := \frac{1}{a_{j,j}} (h^2 f_j - \sum_{i=1}^{j-1} a_{j,i} u_i^{k+1} - \sum_{i=j+1}^n a_{j,i} u_i^k) \tag{8}$$

für $j = 1, \dots, n$. Es fällt auf, dass für jedes u_j nur solche u_i betrachtet werden, die bereits in der gleichen oder vorherigen Iterierten berechnet, und noch nicht überschrieben wurden. Somit müssen die einzelnen Iterierten u^k nicht explizit gespeichert werden, tatsächlich reicht hierfür ein Vektor u aus. Die Berechnungsvorschrift vereinfacht sich zu

$$u_j = \frac{1}{a_{j,j}} (h^2 f_j - \sum_{i=1, i \neq j}^n a_{j,i} u_i) \tag{9}$$

Nach der Definition von A ist klar, dass $a_{j,j} = 4$ gilt. Für $i \neq j$ gilt $a_{i,j} \in \{0, -1\}$. Um die dünne Struktur von A weiter auszunutzen, betrachten wir anschaulich, von welchen Gitterpunkten die Berechnung für einen Gitterpunkt (x, y) konkret abhängt (genau das wird von A beschrieben). Dargestellt wird dies in Abbildung 1, wobei A genau für die vier Nachbarknoten den Wert -1 annimmt, sonst 0 (außer für den Knoten selbst). Durch diese Betrachtung zerfällt die Summe der Zuweisung 9 in vier bedingte Additionen, wie in Code 1.2 illustriert.

```
double sum = h * h * f[j];
if (j % d > 0) sum += u[j - 1]; // Linker Nachbarknoten
if (j % d < d - 1) sum += u[j + 1]; // Rechter Nachbarknoten
if (j / d > 0) sum += u[j - d]; // Oberer Nachbarknoten
if (j / d < d - 1) sum += u[j + d]; // Unterer Nachbarknoten
u[j] = sum / 4;
```

Listing 1.2. Ausnutzung der dünnen Struktur von A zur Berechnung von u_j . Die Zeilen in u sind genau d Elemente lang, daher entspricht bspw. $j - d$ dem Zugriff auf den oberen Nachbar von j aus gesehen. Die Bedingungen Prüfen genau darauf, ob es sich bei einem Nachbarknoten um einen Randpunkt handelt. Falls ja, so ist keine weitere Betrachtung nötig, da dessen Wert 0 ist.

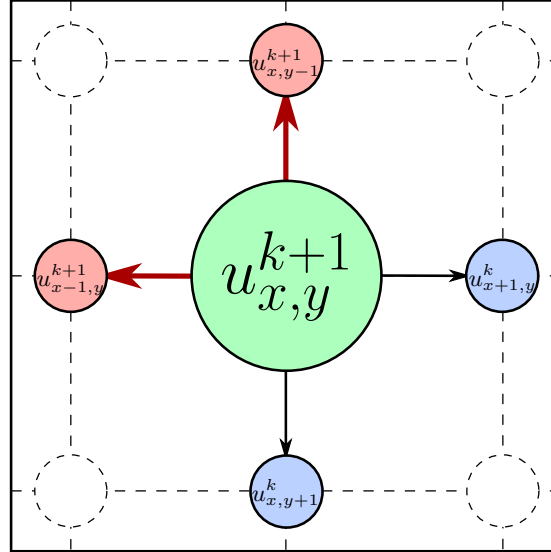


Abbildung 1. Abhängigkeiten zu Nachbarknoten, um Knoten $u_{x,y}^{k+1}$ zu berechnen. Es fällt auf, dass zwei Werte der gleichen Iterierten $k + 1$ benötigt werden (die roten Knoten). Durch diese Abhängigkeit können nicht alle Einträge einer Iterierten gleichzeitig, das heißt parallel, berechnet werden.

Es wurde gezeigt, wie eine einzelne Iterierte k von u sich berechnen lässt. Würde man dies nun immer wiederholen, so würde sich die Lösung u einem Optimum annähern. Nun haben wir aber eine begrenzte Rechenzeit und müssen daher die Berechnung irgendwann abbrechen, sobald uns das Ergebnis *gut genug* ist. In der Regel kennen wir jedoch die analytische Lösung nicht und wissen daher nicht, wie gut unsere bisherige Lösung ist (sie kann ohnehin durch das Verfahren selbst stark davon abweichen, wie wir in Aufgabe Aufgabe 1.5 sehen werden). Um das Problem zu lösen, brechen wir ab, sobald der *Fortschritt* zwischen zwei Iterationen klein genug ist. Die Annahme hierbei ist, dass sich die Lösung auch durch weitere Iterationen nur noch wenig ändert. Konkret betrachten wir die maximale Veränderung eines Eintrags zwischen den Iterierten k und $k + 1$ von u . Formal $maxError := \|u^{k+1} - u^k\|_{max}$. Unsere Lösung nehmen wir als gut genug an, sobald $maxError < \varepsilon_{Error}$ für eine Schranke ε_{Error} . Für unsere Implementierung verwenden wir $\varepsilon_{Error} = 1 \times 10^{-6}$, da es einerseits ausreichend klein ist, um in unseren Tests gute Ergebnisse zu liefern, andererseits groß genug, um in absehbarer Zeit berechnet werden zu können.

1.4.b) Eine naive Parallelisierung des in Aufgabe 1.4.a) implementierten Gauß-Seidel-Verfahrens ist nicht möglich, da innerhalb der Berechnung von u^{k+1} – eine naive Parallelisierung würde versuchen genau diese Berechnung zu parallelisieren, also eine einzelne Iterierte – Abhängigkeiten zwischen den Einträgen bestehen. Für die Berechnung von $u_{x,y}^{i+k}$ werden diese Abhängigkeiten in Abbildung 1 dargestellt. Um etwa $u_{x,y}^{i+k}$ zu berechnen, werden zu erst die Werte von $u_{x-1,y}^{i+k}$ und $u_{x,y-1}^{i+k}$ aus der gleichen Iteration benötigt. Diese wiederum benötigen in gleicher Weise aktuelle Werte von Nachbarknoten.

Eine Mögliche, nicht mehr ganz so naive Lösung, bestünde darin, nun über die Diagonalen zu parallelisieren. Innerhalb der Diagonalen (in $(1, -1)$ – *Richtung*) bestehen keine Abhängigkeiten zwischen den Knoten. Dennoch hat sich auch diese Herangehensweise nicht als Vorteilhaft erwiesen. Für jede Iterierte müssten d Thread-Pools mit maximal $2d - 1$ Fäden gestartet und synchronisiert werden. Der Synchronisierungsaufwand hierbei scheint um Größenordnungen größer, als die eigentliche Berechnung (in der Vorlesung *Software Engineering für moderne parallele Plattformen* [10] wird diese Herangehensweise auch als *Wavefront* bezeichnet).

1.4.c) In Aufgabe 1.4.b) haben wir erläutert, dass eine naive Herangehensweise für die Parallelisierung nicht zielführend ist. Insbesondere haben wir hierfür die zahlreichen Abhängigkeiten innerhalb der Berechnung einer Iterierten verantwortlich gemacht. Um also das Gauß-Seidel-Verfahren nun doch effizient zu parallelisieren, ist eine übergeordnete Betrachtung des Problems von Nöten.

Die Idee hinter unserer Lösung besteht darin, in einem parallelisierbaren Schritt Einträge von u für verschiedene Iterierte k der sequentiellen Variante zu berechnen. Wie in Abbildung 1 zu sehen, hängt die Berechnung eines Knotens von den *aktuellen* Werten des linken und des oberen Nachbarn ab. Diese

müssen also schon vorher berechnet worden sein. Hieraus ergibt sich eine Reihenfolge, in der die Knoten berechnet werden müssen, nämlich von links oben nach rechts unten (in unserer Lösung zu 1.4.b) haben wir die dabei auftretenden, parallelisierbaren Diagonalen bereits erwähnt). Diese Reihenfolge können wir beibehalten, wenn wir für jede Diagonale eine andere Iterierte berechnen.

Angenommen wir berechnen links oben $u_{1,1}^{k+1}$, dann können wir parallel dazu auch $u_{3,1}^k$, $u_{2,2}^k$ und $u_{1,3}^k$ berechnen (sowie auch alle weiteren Diagonalen mit je 2 Abstand). Im nächsten Schritt können wir dann alle nun noch nicht abgedeckten Diagonalen gleichzeitig berechnen. In Abbildung 2 sind all diese Diagonalen noch einmal aufgetragen. Die Felder, über die gleichzeitig berechnet werden kann, bilden ein Schachbrettmuster.

Der Grund, weshalb immer eine Diagonale ausgelassen werden muss, liegt in den vorhandenen Datenabhängigkeiten (dieses Mal in die andere Richtung, also zu den Nachbarn rechts und unten). Angenommen, man möchte $u_{1,1}^{k+1}$ und $u_{2,1}^k$ parallel berechnen, dann würde man bereits für ersteren den Wert von letzterem benötigen (gemäß Abbildung 1). Lässt man jedoch eine Diagonale frei, beispielhaft zwischen $u_{1,1}^{k+1}$ und $u_{3,1}^k$, so konnte man $u_{2,1}^k$ bereits vorher berechnen. Im nächsten Schritt (das heißt nach Synchronisierung) kann aus den berechneten $u_{1,1}^{k+1}$ und $u_{3,1}^k$ das dazwischen liegende $u_{2,1}^k$ berechnet werden usw.

$u_{1,1}^{k+1}$	$u_{2,1}^{k+1}$	$u_{3,1}^k$	$u_{4,1}^k$	$u_{5,1}^{k-1}$	$u_{6,1}^{k-1}$	$u_{7,1}^{k-2}$
$u_{1,2}^{k+1}$	$u_{2,2}^k$	$u_{3,2}^k$	$u_{4,2}^{k-1}$	$u_{5,2}^{k-1}$	$u_{6,2}^{k-2}$	$u_{7,2}^{k-2}$
$u_{1,3}^k$	$u_{2,3}^k$	$u_{3,3}^{k-1}$	$u_{4,3}^{k-1}$	$u_{5,3}^{k-2}$	$u_{6,3}^{k-2}$	$u_{7,3}^{k-3}$
$u_{1,4}^k$	$u_{2,4}^{k-1}$	$u_{3,4}^{k-1}$	$u_{4,4}^{k-2}$	$u_{5,4}^{k-2}$	$u_{6,4}^{k-3}$	$u_{7,4}^{k-3}$
$u_{1,5}^{k-1}$	$u_{2,5}^{k-1}$	$u_{3,5}^{k-2}$	$u_{4,5}^{k-2}$	$u_{5,5}^{k-3}$	$u_{6,5}^{k-3}$	$u_{7,5}^{k-4}$
$u_{1,6}^{k-1}$	$u_{2,6}^{k-2}$	$u_{3,6}^{k-2}$	$u_{4,6}^{k-3}$	$u_{5,6}^{k-3}$	$u_{6,6}^{k-4}$	$u_{7,6}^{k-4}$
$u_{1,7}^{k-2}$	$u_{2,7}^{k-2}$	$u_{3,7}^{k-3}$	$u_{4,7}^{k-3}$	$u_{5,7}^{k-4}$	$u_{6,7}^{k-4}$	$u_{7,7}^{k-5}$

Abbildung 2. Verdeutlichung der Vorgehensweise der Parallelisierung. Zuerst werden diejenigen Einträge von u parallel berechnet, die sich in grau Markierten Feldern befinden. Anschließend parallel die Einträge in den weißen Feldern. Es ist zu beachten, dass die Einträge $u_{x,y}$ für unterschiedliche Iterierte k berechnet werden.

Durch diese Schachbrett-artige Parallelisierung hat ein Berechnungsschritt die Gestalt:

- Berechne Parallel über alle schwarzen Felder
- Berechne Parallel über alle weißen Felder

Es sei nicht zu verschweigen, dass hier im Gegensatz zur seriellen Variante sehr wohl mehrere u gleichzeitig zu speichern sind. Sobald in einem Berechnungsschritt das Feld ganz rechts unten, also $u_{d,d}^{k+2-d}$, berechnet wurde, müssen im Falle des Abbruchs (weil das Abbruchkriterium aus 1.4.a) erfüllt ist), alle $u_{x,y}^{k+2-d}$ noch immer bekannt sein, um u^{k+2-d} als Ergebnis ausgeben zu können. Ebenso muss das Abbruchkriterium stets auf den Werten der gleichen Iterierten arbeiten, um exakt das gleiche Resultat wie die serielle Variante zu erhalten.

In jedem Berechnungsschritt wird genau eine Iterierte des Verfahrens berechnet. Allerdings gilt dies erst, sobald $d - 1$ Schritte ausgeführt wurden, denn erst dann ist schließlich der Eintrag rechts unten $u_{d,d}^{k+2-d} = u_{d,d}^1$ berechnet (die *Historie* muss erst gefüllt werden). So kommt es, dass die parallele Version bei gleichen Eingaben exakt die gleichen Ergebnisse wie die serielle Version ausgibt, jedoch dabei genau $d - 1$ Iterationen mehr benötigt. Wie der Tabelle 7 zu entnehmen, ist die parallele Version mit 32 Kernen bei großer Problemgröße ($l = 9$) mit einem *Speedup* von 7,57 aber immerhin noch deutlich schneller. Die eher geringe Effizienz schreiben wir dem Synchronisationsaufwand sowie den eher Cache-unfreundlichen Speicherzugriffen zu.

Problemgröße			Seq.	2 Threads		4 Threads		...	32 Threads	
l	d	n	Laufzeit	Laufzeit	Speedup	Laufzeit	Speedup	...	Laufzeit	Speedup
4	15	225	0,005	0,007	0,357	0,008	0,625	...	1,361	0,0389
5	31	961	0,053	0,056	0,473	0,049	1,08	...	4,041	0,178
8	255	65.025	115	81	1,42	45,3	2,54	...	25,33	4,54
9	511	261.121	1340	943	1,42	504	2,67	...	177	7,57

Tabelle 7. Laufzeiten unserer Parallelisierung gegenüber der sequentiellen Version. Getestet wurde auf dem Rechner *i82sn07*. Um Ungenauigkeiten auszugleichen wurden die Zeiten jeweils über 100 (für große Eingaben über 10) Durchläufe gemittelt.

Aufgabe 1.5 Partielle Differentialgleichungen

1.5.a) Die Bedingungen lauten nach den Folien der FEM-Einführung:

- Ω sei ein beschränktes Gebiet
- Γ sei hinreichend glatt
- $f : \Omega \rightarrow \mathbb{R}$ gegebene Funktion, wie es hier der Fall ist.

1.5.b) Gesucht ist f mit

$$u(x, y) = \sin(2M\pi x) \sin(2N\pi y) \quad (10)$$

Dies kann durch einfache Anwendung des *Laplace-Operators* Δ berechnet werden:

$$\begin{aligned}
 f(x, y) &= -\Delta u(x, y) \\
 &= -\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} \\
 &= -\frac{\partial}{\partial x}(2M\pi \cos(2M\pi x) \sin(2N\pi y)) - \frac{\partial}{\partial y}(2N\pi \sin(2M\pi x) \cos(2N\pi y)) \\
 &= 4M^2\pi^2 \sin(2M\pi x) \sin(2N\pi y) + 4N^2\pi^2 \sin(2M\pi x) \sin(2N\pi y) \\
 &= (M^2 + N^2)4\pi^2 \sin(2M\pi x) \sin(2N\pi y)
 \end{aligned} \tag{11}$$

1.5.c) Da in Gleichung 11 $M, N \in \mathbb{N}$ beliebig, wählen wir der Einfachheit halber $M = N = 1$ zur Lösung dieser Teilaufgabe. Damit ergibt sich

$$f(x, y) = 8\pi^2 \sin(2\pi x) \sin(2\pi y) \tag{12}$$

Die dazugehörige analytische Lösung ist dann nach Aufgabenstellung gegeben als

$$u(x, y) = \sin(2\pi x) \sin(2\pi y) \tag{13}$$

und wird verwendet um den maximalen Fehler der Näherung zu berechnen.

Das f auf Gleichung 12 kann einfach in den Code der vorangegangenen Aufgabe eingesetzt werden. Wie auch [7] entnommen werden kann, lässt sich das Problem mit dem in Aufgabe 1.4.a) gelösten Problem lösen. Es ergeben sich die in Abbildung 3 gezeigten Näherungen. Mit zunehmendem l bzw. abnehmender Gitterkonstante h werden die Lösungen nicht nur feiner, sondern der maximale Fehler zur analytischen Lösung von u auch kleiner wird. Ab $l = 8$ wird der Fehler allerdings wieder größer, wie Tabelle 8 entnommen werden kann. An dieser Stelle scheint das für das Abbruchkriterium gewählte ε_{Error} aus Aufgabe 1.4.a) zu grob gewählt zu sein, denn eine weitere Verfeinerung dessen reduzierte den Fehler der Näherung (nur) für große l deutlich (was die Frage aufwirft, ob man ε_{Error} abhängig von l wählen sollte).

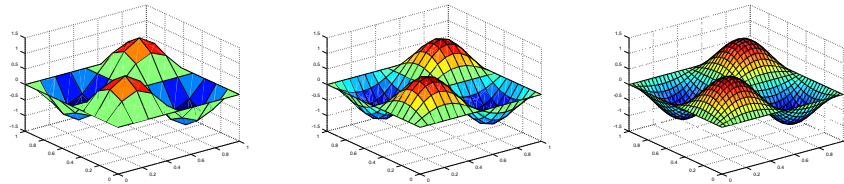


Abbildung 3. Näherungsweise Lösung für u , berechnet mittels Gauß-Seidel-Verfahren. Von links nach rechts: $l = 3, l = 4, l = 5$ bzw. $h = \frac{1}{8}, h = \frac{1}{16}, h = \frac{1}{32}$. Geplottet mittels GNU Octave.

l	2	3	4	5	6	7	8	9
d	3	7	15	31	63	127	255	511
h	0,25	0,125	0,0625	0,0313	0,0156	0,00781	0,00391	0,00195
Maximaler Fehler	0,234	0,053	0,0130	0,00326	0,000866	0,000326	0,00174	0,0266
Iterationen	21	64	175	409	1148	3515	11043	98558

Tabelle 8. Maximaler Fehler und Anzahl der durchgeführten Operationen in Abhängigkeit von l bzw. Gitterkonstante h .

Das verwendete Vorgehen wird als h -FEM bezeichnet, da wir lediglich die Gitterkonstante h anpassen. Eine p -FEM- oder gar hp -FEM-Methodik würde den Grad des zum Abtasten verwendeten Polynoms erhöhen.

Aufgabe 1.6 Partielle Differentialgleichungen

1.6.a) Alphabetische Liste gängiger Krylow-Unterraum-Verfahren (nach [1]):

- Arnoldi-Verfahren, zur Eigenwertapproximation
- BiCG, das CG-Verfahren für nicht SPD-Matrizen
- BiCGSTAB, Stabilisierung von CGS
- BiCGSTAB(ell), Stabilisierung von CGS
- BiCGSTABTFQMR, der Ansatz hinter TFQMR angewandt auf BiCGSTAB
- BiOres, eine Variante des BiCG-Verfahrens
- BiOmin, eine Variante des BiCG-Verfahrens
- BiOdir, eine Variante des BiCG-Verfahrens
- CG, zur approximativen Lösung linearer Gleichungssysteme
- CGNE, CG-Verfahren auf den Normalgleichungen, Variante 1
- CGNR, CG-Verfahren auf den Normalgleichungen, Variante 2
- CGS-Verfahren, quadrierte BiCG-Rekursion
- FOM, zur approximativen Lösung linearer Gleichungssysteme
- GMRES, zur approximativen Lösung linearer Gleichungssysteme
- Hessenberg-Verfahren, zur Eigenwertapproximation
- Lanczos-Verfahren, zur Eigenwertapproximation
- MinRes, zur approximativen Lösung linearer Gleichungssysteme
- Orthores, Orthomin und Orthodir, Verallgemeinerungen des CG-Verfahrens
- Ores, eine Variante des CG-Verfahrens
- Omin, eine Variante des CG-Verfahrens
- Odir, eine Variante des CG-Verfahrens
- Potenzmethode, älteste Methode zur Eigenwertapproximation
- QMR, zur approximativen Lösung linearer Gleichungssysteme
- Richardson-Iteration, bei geeigneter Interpretation
- SymmLQ, zur approximativen Lösung linearer Gleichungssysteme
- TFQMR, zur approximativen Lösung linearer Gleichungssysteme

Die häufig verwendeten Grundverfahren sind: CG-Verfahren, GMRES und Lanczos-Verfahren

CG-Verfahren: Hierbei handelt es sich um einen iterativen Algorithmus und ist gedacht für große lineare, symmetrische, positiv definite und dünn besetzte Gleichungssysteme. Das CG-Verfahren liefert nach spätestens n (Dimension der Matrix) Schritten die exakte Lösung.

GMRES: Ein iteratives Verfahren welches sich besonders für große, dünn besetzte lineare Gleichungssysteme eignet. GMRES kann und wird für nicht-symmetrische Matrizen eingesetzt, allerdings wird die exakte Lösung erst nach endlich vielen Schritten geliefert.

Lanczos-Verfahren: Ebenfalls ein iterativer Algorithmus, welcher Eigenwerte mit den entsprechenden Eigenvektoren bestimmen kann, als auch lineare Gleichungssysteme lösen. Die Konvergenz vom Algorithmus ist von den Eigenwerten abhängig.

Bei der Auswahl von diesen drei Verfahren für unser Problem, eignet sich das CG-Verfahren am besten, da die vom Problem gegebene Matrix alle Kriterien erfüllt. Des Weiteren ist die Konvergenz n (nach [11]) von Vorteil, da wir davon ausgehen können, dass wir schon viel früher eine Lösung erreicht haben werden welche unter einer gewissen Toleranz *gut genug* sein wird. Gegen GMRES spricht nur die möglicherweise langsamere Konvergenz und die hier nicht benötigte Flexibilität auch nicht-symmetrische Matrizen bearbeiten zu können. Das Lanczos-Verfahren wäre eine Alternative, vor allem da die Eigenwerte und somit die Konvergenz im Voraus bekannt sind.

Unsere Parallelisierung arbeitet auf der Ebene der Vektoreinträge, da diese zu großen Teilen unabhängig von einander berechnet werden können (ganz im Gegensatz zum Gauß-Seidel-Verfahren aus Aufgabe 1.4). Auch hier kann die dünne Struktur von A dahingehend ausgenutzt werden, dass Berechnungen der Art $A \cdot x$ im Algorithmus für jeden Eintrag im Ergebnisvektor zu vier bedingten Additionen zerfallen. Auch hier muss A daher nicht explizit gespeichert oder berechnet werden.

Der *Intel Thread Checker* gibt für unser Programm Folgendes aus:

- Thread termination at "**cg_parallel.c**":31 - includes stack allocation of 2,004 MB and use of 4,793 KB
- Thread termination at "**cg_parallel.c**":31 - includes stack allocation of 68 KB and use of 3,246 KB
- Thread termination at "**cg_parallel.c**":100 - includes stack allocation of 8 MB and use of 127,168 KB

1.6.b) Unsere parallele Version des CG-Verfahrens konnte gegenüber dem Gauß-Seidel-Verfahren das Problem aus Aufgabe 1.5 bis zu 40-mal so schnell lösen. Eine Übersicht über den *Speedup* sowie die *Efficiency* ist in Tabelle 9 zu finden.

Wie auch unsere Implementierung des Gauß-Seidel-Verfahrens aus Aufgabe 1.4, nutzen wir hier für unsere Implementierung die dünne Struktur der Matrix A aus. Eine Vorkonditionierung würde diese Struktur beeinträchtigen, wodurch ein sehr viel höherer Rechenaufwand zu erwarten wäre. Gesetzt den

Problemgröße			Seq.	2 Threads			...	16 Threads		32 Threads	
l	d	n	Speedup	Speedup	Efficiency	Speedup	Efficiency	Speedup	Efficiency
4	15	225	1,74	1,81	0,906	2,44	0,153	2,93	0,0917
5	31	961	4,01	3,71	1,86	1,44	0,0898	3,46	0,109
8	255	65.025	17,9	16,5	8,27	16,8	1,05	4,05	0,127
9	511	261.121	27,6	25,9	12,9	39,5	2,47	37,9	1,18

Tabelle 9. *Speedup* und *Efficiency* für das CG-Verfahren gegenüber dem Gauß-Seidel-Verfahren für das Problem aus Aufgabe 1.5. Die zugrunde liegenden Laufzeiten wurden anhand des Rechners *i82sn07* ermittelt.

Fall unsere Matrix A wäre nicht dünn besetzt, so könnte eine Vorkonditionierung Sinn machen, um die Stabilität des Algorithmus ggf. zu erhöhen.

1.6.c) In Tabelle 10 haben wir drei Bibliotheken zur Anwendung der Finiten Elemente Methode aufgelistet und verglichen. Weitere Bibliotheken, die es nicht in die Tabelle geschafft haben, wären beispielsweise Feel++[2] und Libmesh[3].

Im Rahmen dieses Praktikums würde sich, zumindest für Projekt 1, *HiFlow*³ gut eignen, da hier eine ausgereifte OpenMP Funktionalität gegeben zu sein scheint, die den Portierungsaufwand stark verringert.

Bibliothek	Unterschiede	Gemeinsamkeiten
HiFlow ³ [4]	MPI und OpenMP basiert, Hohe Parallelität (von Laptop bis zu Cluster), Keine externen Bibliotheken benötigt (optional Libraries, unter anderem OpenMP)	Für C++ gedacht, Implementieren Krylov-Unterraumverfahren, Diskretisierung wählbar
MFEM[5]	MPI basiert (und experimenteller OpenMP Support), Sehr hohe Parallelität (mehrere hunderttausend Kerne), Keine externen Bibliotheken benötigt	
Deal.II[6]	MPI basiert, Hohe Parallelität (16000 Kerne mindestens), Keine externen Bibliotheken benötigt (optional Libraries)	

Tabelle 10. Ausgewählte Bibliotheken zur Anwendung der Finiten Elemente Methode im Vergleich.

Projekt 2 CUDA

Aufgabe 2.1 Getting started


```

Found 2 CUDA devices
=====
Information for GeForce GTX 960 (device 0):
Total global memory: 2092957696
Total const memory: 65536
Shared memory per block: 49152
Warp size: 32
Max threads per block: 1024
Max threads dimension: [ 1024, 1024, 64 ]
Max grid size: [ 2147483647, 65535, 65535 ]
=====
Information for GeForce GTX 560 Ti (device 1):
Total global memory: 2080768000
Total const memory: 65536
Shared memory per block: 49152
Warp size: 32
Max threads per block: 1024
Max threads dimension: [ 1024, 1024, 64 ]
Max grid size: [ 65535, 65535, 65535 ]
=====

```

Listing 1.3. Ausgabe unseres Programms für die zwei CUDA-fähigen Grafikkarten in *i82sn02*.

In 1.3 ist die Ausgabe unseres Programms zu sehen. In den folgenden Aufgaben werden wir *device 0* (*GeForce GTX 960*) verwenden.

Aufgabe 2.2 Datentransferrate

Aufgabe 2.3 Gauß-Seidel-Verfahren

2.3.a) Implementierung in CUDA Die Portierung des parallelisierten Gauß-Seidel-Verfahrens aus Aufgabe Aufgabe 1.4 verlief sehr geradlinig. Die zuvor parallelisierten inneren Schleifen – die Berechnung über alle schwarzen/weißen Felder – werden nun mittels entsprechender Kernel auf der GPU ausgeführt. Die Synchronisation ist damit nun implizit durch die Kernelaufufe – die nacheinander ausgeführt werden – gegeben.

Das eigentlich zweidimensional gegebene Problem haben wir als eindimensionales Problem an die GPU weitergegeben. Das heißt wir haben als Größe des Problems den Vektor $(n, 1, 1)^T$ verwendet (mit n wie aus Aufgabe Aufgabe 1.4). Da wir bei der Berechnung die zweidimensionale Struktur nicht explizit benötigen, lässt sich so der Verschnitt bei der Aufteilung auf die *Warps* der GPU verkleinern.

Wie auch die CPU Variante haben wir zur Berechnung den Datentyp `double` verwendet. Damit der Compiler diesen auch tatsächlich verwendet, muss beim Kompilieren das Flag `-arch sm_13` mit angegeben werden (anderenfalls wird stattdessen mit `float` gearbeitet).

2.3.b) Asynchrone Parallelisierungsmethoden Bei *relaxierten Methoden* nimmt man Rechenungenauigkeiten für bessere Performanz in Kauf.

Die Verwendung des Compilerflags `-use-fast-math` – welches im Kontext von Relaxierung ebenfalls zu Gunsten der Berechnungszeit weniger Rechengenauigkeit garantiert – brachte in der Laufzeit keinen Unterschied. Wir führen das darauf zurück, dass wir keine aufwändigeren Mathematikoperationen wie Kosinus oder Sinus berechnen lassen.

Da die GPU deutlich weniger Recheneinheiten für *double precision* Fließkommazahlen hat als für *single precision*, haben wir uns dazu entschieden, für die GPU Variante den Datentyp *float* anstatt – wie bei der CPU – *double* zu verwenden. Die Ergebnisse sind unserer Auffassung nach dennoch vergleichbar, da Tests mit *float* anstatt *double* auf der CPU keinen nennenswerten Geschwindigkeitsvorteil ergaben.

Der Blogbeitrag [8] geht noch weiter und zeigt auf, wie sich mittels 16-Bit Fließkommazahlen (*half*) die Geschwindigkeit weiter steigern lässt. Es wird eine doppelt so große Geschwindigkeit gegenüber 32-Bit Fließkommazahlen angegeben. Für unsere Anwendung ist die damit verbundene Ungenauigkeit möglicherweise zu groß. Ferner wird *half* erst ab CUDA 7.5 unterstützt, wohingegen wir mit CUDA 5.5 arbeiten.

2.3.c) Eignung für andere Architekturen Zu Testzwecken haben wir die Implementierung aus Aufgabe 1.4.c) für den Datentyp *float* umgeschrieben. Die damit gemessenen Laufzeiten unterscheiden sich unwesentlich von denen bei Verwendung mit *double*. Für die CPU scheint diese Art der Relaxierung daher wenig gewinnbringend. Wir führen das darauf zurück, dass CPUs heutzutage darauf optimiert sind, mit *double* genau so schnell zu rechnen wie *float*.

Aufgabe 2.4 LU-Zerlegung

Aufgabe 2.5 Vorkonditionierung

Literaturverzeichnis

- [1] <https://de.wikipedia.org/wiki/Krylow-Unterraum-Verfahren>
- [2] <http://www.feelpp.org/>
- [3] <http://libmesh.github.io/>
- [4] <http://www.hiflow3.org/>
- [5] <http://mfem.org/>
- [6] <https://www.dealii.org/>
- [7] BRAESS, Dietrich: Finite Elemente : Theorie, schnelle Löser und Anwendungen in der Elastizitätstheorie. (2007)
- [8] HARRIS, Mark: *Mixed-Precision Programming with CUDA 8*. <https://devblogs.nvidia.com/parallelforall/mixed-precision-programming-cuda-8/>, 2016
- [9] KARL, Prof. Dr. W.: Vorlesung Rechnerstrukturen. (2015)
- [10] PFAFFE, Philip: Software Engineering für moderne parallele Plattformen. (2016)
- [11] WEISS, Daniel: Numerische Mathematik für die Fachrichtungen Informatik und Ingenieurwesen. (2015)