h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Hochschule Darmstadt

- Fachbereich Informatik -

Implementierung exemplarischer, paralleler, numerischer Verfahren aus dem CFD-Bereich auf Grafikkarten unter Verwendung von OpenCL

Abschlussarbeit zur Erlangung des akademischen Grades

Master of Science (M.Sc.)

vorgelegt von

Frank Kautz (B.Sc)

Referent:	Prof. Dr. Ronald Moore
Korreferent:	Dr.-Ing. Achim Basermann

Ausgabedatum:	30.09.2010
Abgabedatum:	13.04.2011

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, den

Zusammenfassung

Im Rahmen dieser Masterarbeit soll untersucht werden, ob das Lineare Modul des Strömungslösers TRACE (Turbo machinery Research Aerodynamic Computational Environment) durch die Verwendung von GPUs beschleunigt werden kann. In der aktuellen Implementierung wird der Strömungslöser auf CPU-Clustern mittels *Message Passing Interface* (MPI) ausgeführt. Es werden verschieden Datenformate für dünnbesetzte Matrizen untersucht. Die Grundlagen der *General-Purpose Computation on Graphics Processing Unit*-Programmierung (GPGPU) mit *Open Computing Language* (OpenCL) und *Compute Unified Device Architecture* (CUDA) werden beschrieben. Im ersten Schritt muss der Strömungslöser TRACE analysiert werden. Welcher Algorithmus wird verwendet? Welche Teile des Algorithmus können durch die GPU noch besser parallelisiert werden? Welches Datenformat wird verwendet?

In TRACE wird das *Blocked Compressed Sparse Row* Format (BCSR) mit einer Blockgröße von 5x5 Werten verwendet. Es muss untersucht werden, ob dieses Datenformat auch für die Verarbeitung durch die GPU geeignet ist oder ob es bessere Datenformate für die GPU gibt. Es werden verschiedene Datenkonverter implementiert, damit unterschiedliche Datenformate getestet werden können. Der Strömungslöser TRACE verwendet den *Generalized Minimum Residual*-Algorithmus (GMRES). GMRES enthält eine dünnbesetzte Matrix-Vektor-Multiplikation (SpMV), diese wird im Rahmen dieser Arbeit auf die GPU portiert. Die SpMV wird mit verschiedenen Datenformaten implementiert. Durch unterschiedliche Optimierungsansätze wird versucht, eine bessere Performanz zu erzielen. Die verschiedenen Implementierungen mit OpenCL und CUDA werden auf einem Testsystem mit der MPI-Variante von TRACE verglichen. Es werden Laufzeitmessungen durchgeführt und die erreicht Rechenleistung berechnet. Es hat sich gezeigt, dass die Implementierung mit dem *Blocked Ellpack* Datenformat die beste Performanz erzielt.

Abstract

This master thesis examines whether the linear module of the flow solver TRACE (Turbo machinery Research Aerodynamic Computational Environment) can be accelerated by the use of GPUs. The target platforms of the current implementation are CPU Clusters and employs MPI (Message Passing Interface) for parallelization. Different data formats are examined in this thesis. The basics of the *General-Purpose Computation on Graphics Processing Unit* programming (GPGPU) with the use of *Open Computing Language* (OpenCL) and *Compute Unified Device Architecture* CUDA are described. The first step is to analyze the flow solver TRACE. Which algorithm is used? Which parts of the algorithm are candidates to be executed by GPU? Which data format is used?

TRACE uses the *Blocked Compressed Sparse Row format* (BCSR) with a block size of 5x5 values. It is examined if this format is suitable for processing by the GPU or if another format works better on this architecture. For testing different formats several data converters have been implemented. TRACE uses the *Generalized Minimum Residual* algorithm (GMRES). GMRES contains a sparse matrix-vector multiplication (SpMV); this will be ported onto the GPU. The SpMV is implemented with the use of different data formats. By trying different optimization approaches it is tested if a better performance is possible. The various implementations with OpenCL and CUDA are compared on a test system with the MPI version of TRACE. The performance of the implemented SpMV kernels is measured and compared. It is found that block variants of the ELL format give the best results.

Danksagung

Allen voran möchte ich den Gutachtern der Master-Thesis Prof. Dr. Ronald Moore und Dr.-Ing. Achim Basermann für die Übernahme dieser Aufgabe danken. Ich möchte mich für die exzellente Betreuung der Master-Thesis bei Dr.-Ing. Achim Basermann und Dr. rer. nat. Hans-Peter Kersken bedanken. In vielen Gesprächen und Diskussionen erhielt ich interessante Hinweise, die mir sehr weiter geholfen haben. Desweitern möchte ich mich bei ihnen auch für das Korrekturlesen meiner Master-Thesis bedanken.

Mein besondere Dank gilt meinen Eltern Renate und Siegfried Kautz für ihre jahrelange Unterstützung und das entgegengebrachte Vertrauen. Ohne ihre Unterstützung wäre mir das Studium nicht möglich gewesen.

Diese Master Thesis entstand an der Einrichtung „Simulations- und Softwaretechnik“ des „Deutschen Zentrums für Luft- und Raumfahrt e.V.“ in der Abteilung „Verteilte Systeme und Komponentensoftware“. Die dortigen Arbeitsbedingungen für die Erstellung der Master Thesis waren ideal. Dafür möchte ich mich bei meinen Vorgesetzten und Kollegen bedanken.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation

Im Bereich der numerischen Strömungsdynamik (CFD: Computational Fluid Dynamics) wird immer wieder versucht, durch algorithmische Verbesserungen, z.B. der Methoden zur Lösung großer Gleichungssysteme, die Performanz zu steigern. Eine weitere Möglichkeit ist der Einsatz von neuen Rechnerarchitekturen. Eine geeignete aktuelle Technologie sind Computersysteme mit GPGPU-Beschleunigern (General-Purpose Computation on Graphics Processing Unit).

Am Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR) wird seit mehr als 10 Jahren der numerische Strömungslöser TRACE (Turbo machinery Research Aerodynamic Computational Environment) entwickelt. Der TRACE-Code wird zur Strömungssimulation in Flugzeugtriebwerken und stationären Gas-/Dampfturbinen zur Stromerzeugung eingesetzt. Die Parallelisierung des TRACE-Codes basiert auf der Kommunikationsbibliothek MPI (Message Passing Interface). Die Strömungsberechnungen werden im Regelfall auf großen Cluster-Systemen durchgeführt. In dieser Arbeit sollen verschiedene numerische Kernels des Strömungslösers hinsichtlich ihrer Eignung für die effiziente Ausführung auf GPGPU-Hardware untersucht werden. Das OpenCL-Framework (Open Computing Language) dient dabei als Basis-Technologie. Die Implementierung soll Hardware-herstellerübergreifend realisiert werden. Dies wird durch den Einsatz des OpenCL-Frameworks sichergestellt. Eine weitere GPGPU-Programmierumgebung ist CUDA (Compute Unified Device Architecture). Diese ist nur für NVIDIA-GPGPUs geeignet.

1.2 Problemstellung

Ein Schwerpunkt der Arbeit ist die Untersuchung von effizienten Operationen mit dünnbesetzten Matrizen auf der GPGPU. In den TRACE-Kernels verwendete Datenformate für Matrizenrechnungen und weitere Operationen müssen in geeignete Datenstrukturen für die GPGPU konvertiert werden. Ein Problem bei der Portierung sind die begrenzten Ressourcen auf der GPGPU. Die Daten der Berechnung müssen so aufgeteilt werden, dass sie möglichst lange im Speicher der GPGPU gehalten werden können, da der Datentransfer zwischen GPGPU und Hostrechner Performanz-kritisch ist. Während der Implementierung der TRACE-Kernels müssen die Hardware-Eigenschaften der GPGPU berücksichtigt werden. Eine Effizienz-Steigerung der TRACE-Kernels auf GPGPU-Basis gegenüber den TRACE-Kernels auf MPI-Basis kann am besten sichergestellt werden, wenn die Implementierung auf die Hardware abgestimmt wird.

1.3 Ziele

Die untersuchten TRACE-Kernels sollen durch Portierung auf GPGPU-Technologie eine Effizienz-Steigerung erfahren. Die Effizienz-Steigerung der GPGPU-Implementierung soll mittels Performanztests nachgewiesen werden. Hierzu wird die Performanz der MPI-Implementierung mit der Performanz der GPGPU-Implementierung verglichen. Die Masterarbeit soll zeigen, ob und wie eine deutliche Beschleunigung des gesamten TRACE-Codes durch den Einsatz von GPGPU-Technologie in Zukunft erreicht werden kann.

2 Datenformate für dünnbesetzte Matrizen

Dünnbesetzte Matrizen beinhalten hauptsächlich Nullen, diese sind in den Berechnungen nicht relevant. Bei großen Matrizen führen die nicht benötigten Nullen zu einem sehr hohen und unnötigen Speicherbedarf. Dieses kann sich bei Berechnungen sehr negativ auf die Performanz auswirken. Sollte die Matrix zu groß für den Arbeitsspeicher eines Computersystems sein, müsste die Matrix während der Berechnung mehrfach zwischen Arbeitsspeicher und Festplatte (Swap-Speicher bzw. Auslagerungsdatei) transferiert werden. Dieses Problem verschärft sich bei der GPGPU-Programmierung, da die Grafikkarten weniger Arbeitsspeicher als heutige Hostsysteme besitzen. Die dünnbesetzten Matrizen werden in speziellen Datenformaten gespeichert, damit nicht unnötig Speicherplatz verschwendet wird.

2.1 Coordinate Format

Das einfachste Datenformat für dünnbesetzte Matrizen ist das *Coordinate Format* (COO) (vgl. [SaadY-2003] Seite 89). Von einer Matrix werden nur die Elemente gespeichert, die ungleich Null sind. Das Datenformat besteht aus drei Arrays, zwei Integer Arrays und einem Array vom Datentyp der zu speichernden Daten. In einem Integer-Array wird der Zeilenindex und im anderen der Spaltenindex gespeichert. Im dritten Array wird der zu den Koordinaten passende Wert aus der Matrix gespeichert. Die Anzahl der Werte der Matrix, die ungleich Null sind, entsprechen der Länge der drei Arrays. Im *Coordinate Format* müssen die Daten in keiner bestimmten Reihenfolge abgespeichert werden. Ein Beispiel ist in Abbildung 2.1 dargestellt.

Matrix:

1	0	0	2	0	0
0	3	4	5	0	0
0	0	0	0	6	7
0	0	0	0	8	9

Werte:

1	2	3	4	5	6	7	8	9

Spaltenindex:

1	4	2	3	4	5	6	5	6

Zeilenindex:

1	1	2	2	2	3	3	4	4

Abbildung 2.1: Beispiel für das COO-Datenformat (vgl. [SaadY-2003] Seite 89).

2.2 Compressed Sparse Row

Das *Compressed Sparse Row* Datenformat (CSR) (vgl. [SaadY-2003] Seite 90) ist ein noch platzsparenderes Datenformat als das COO-Datenformat. Es werden eben-

falls nur die Werte der Matrix gespeichert, die ungleich Null sind. Dieses Datenformat besteht aus drei Arrays. Im ersten Array werden die Werte der Matrix gespeichert, deshalb muss dieses Array mit dem Datentyp der Matrixelemente definiert werden. In einem Integer-Array werden die Spaltenindizes der Werte gespeichert. Die Länge der beiden Arrays entspricht der Anzahl der Werte der Matrix, die ungleich Null sind. Im dritten Array werden Pointer auf Einträge im Spaltenindex-Array gespeichert. Die Pointer zeigen auf den ersten Wert einer Zeile. Wenn die Matrix N Zeilen hat, dann hat das dritte Array eine Länge von N + 1 Einträgen. Im letzten Wert wird ein Pointer außerhalb des Arrays für die Spaltenindizes gespeichert. Damit dieses Datenformat funktioniert, müssen die Zeilen aufsteigend von Zeile 1 bis Zeile n gespeichert werden. Abbildung 2.2 zeigt ein Beispiel für dieses Format.

Matrix:

1	0	0	2	0	0
0	3	4	5	0	0
0	0	0	0	6	7
0	0	0	0	8	9

Werte:

1	2	3	4	5	6	7	8	9

Spaltenindex:

1	4	2	3	4	5	6	5	6

Pointer für den Beginn einer Zeile im Spaltenindex:

1	3	6	8	10

Abbildung 2.2: Beispiel für das CSR-Datenformat (vgl. [SaadY-2003] Seite 90).

2.3 Block Compressed Row Storage

Das *Block Compressed Row Storage* Format (BCRS) (vgl. [ChoiJ-2010] Seite 117) oder auch *Blocked Compressed Sparse Row* Format (BCSR) genannt, ist in den Grundprinzipien dem CSR-Format sehr ähnlich. Die Matrix wird in Blöcke fester Größe unterteilt. Die Konzepte des CSR Formats werden auf die Blockspalten/Blockzeilen angewendet. Alle Blöcke, die nicht nur aus Nullen bestehen, werden vollständig gespeichert. Auch die Nullen der Blöcke werden gespeichert. Bei diesem Datenformat wird etwas Speicherplatz verschwendet, allerdings kann bei geschickter Größe der Blöcke eine Beschleunigung der Anwendung erreicht werden. Die Beschleunigung resultiert aus der größeren Datenmenge, die bei einem Lesezugriff aus dem Speicher geholt wird. Bei dem CSR-Format wird immer nur ein Wert aus dem Speicher gelesen. Die Blöcke werden aufsteigend, Zeile für Zeile, in einem Vektor gespeichert. Die Vektoren der Blöcke werden aufsteigen Blockzeile für Blockzeile in einem Array gespeichert. Dieses Array muss mit dem gleichen Datenformat, der Matrix in-

itialisiert werden. Die Blockgröße und die Anzahl der Blöcke, die gespeichert werden, bestimmten die Länge dieses Arrays (Blockgröße x Anzahl Blöcke). In einem Integer-Array werden die Blockspaltenindizes der Blöcke gespeichert. Die Größe dieses Arrays wird von der Anzahl der Blöcke bestimmt, die Werte ungleich Null enthalten. Im dritten Array werden Pointer auf Einträge im Blockspaltenindex-Array gespeichert. Die Pointer zeigen auf den ersten Block einer Blockzeile. Die Anzahl der Blockzeilen plus 1 bestimmt die Größe des dritten Arrays. Eine Variante des BCSR-Formats ist das *Variable Block Compressed Row Storage* Datenformat (VBCSR). Das VBCSR-Format verwendet Blöcke variabler Größe. Eine feste Blockgröße von 2 Zeilen mal 2 Spalten wird in dem Beispiel in Abbildung 2.3 verwendet.

Matrix (Blockgröße: 2x2):

1	0	0	2	0	0
0	3	4	5	0	0
0	0	0	0	6	7
0	0	0	0	8	9

Werte:

1	0	0	3	0	2	4	5	6	7	8	9

Blockspaltenindex:

1	2	3

Pointer für den Beginn einer Blockzeile im Blockspaltenindex:

1	3	4

Abbildung 2.3: Beispiel für das BCSR-Datenformat (vgl. [ChoiJ-2010] Seite 117).

2.4 Diagonal Sparse Matrix

Das Diagonal Sparse Matrix Format (DIA) (vgl. [SaadY-2003] Seite 91) eignet sich besonders gut zum Speichern von Bandmatrizen. Die besetzten Diagonalen der Matrix werden als Matrix gespeichert. Jede Diagonale wird zu einer Spalte in der neuen Matrix. In welcher Reihenfolge die Diagonalen gespeichert werden, spielt keine Rolle. Bedingt durch die unterschiedlichen Längen der Diagonalen werden in der Matrix auch Platzhalter für unbenutzte Einträge gespeichert. Nur mit Nullen belegte Diagonalen werden nicht gespeichert. In einem Integer-Array werden die Offsets der Diagonalen gespeichert. Die Hauptdiagonale hat den Offset 0. Die Offsets der anderen Diagonalen werden durch den Abstand zur Hauptdiagonalen berechnet. Die Länge des Arrays ist durch die Anzahl der gespeicherten Diagonalen gegeben. Die Anzahl der gespeicherten Diagonalen bestimmt ebenfalls die Anzahl der Spalten der Daten-Matrix. Die Anzahl der Zeilen ist identisch mit der Anzahl der Zeilen der Ursprungsmatrix. Ein Beispiel mit drei Diagonalen ist in der Abbildung 2.4 dargestellt.

Matrix:

1	2	0	0
3	4	5	0
0	6	7	8
0	0	9	10

Diagonalwerte:

x	1	2
3	4	5
6	7	8
9	10	x

Zeilenindex:

-1	0	1

Abbildung 2.4: Beispiel für das DIA-Datenformat (vgl. [SaadY-2003] Seite 91).

2.5 Ellpack-Itpack

Ein weiteres Datenformat für Matrizen mit ungefähr gleicher Anzahl Nichtnull-Einträge pro Zeile ist das *Ellpack-Itpack* Format (ELL) (vgl. [SaadY-2003] Seite 91). Alle Werte ungleich Null werden nach links geschoben und die Nullen entfernt. Die maximale Anzahl der Werte ungleich Null in einer Zeile bestimmt die Zeilenlänge der Wertenmatrix. Zeilen mit weniger Einträgen werden mit Nullen ergänzt. Die Anzahl der Zeilen ist identisch mit der Anzahl der Zeilen der Ursprungsmatrix. Im ELL-Datenformat wird für jeden Eintrag ein Spaltenindex gespeichert. Die Spaltenindizes werden in einer Integer-Matrix mit den gleichen Dimensionen der Wertematrix gespeichert. Für die hinzugefügten Nullen in der Wertematrix müssen Indizes in der Spaltenindexmatrix hinzugefügt werden. Diese müssen zwischen 1 und der maximalen Spaltenanzahl liegen. Saad verwendet in seinem Beispiel den Zeilenindex zum Auffüllen der Matrix mit den Spaltenindizes. Im ELL-Datenformat wird die Matrix spaltenweise (column-major) gespeichert und die Matrix-Operationen werden spaltenweise abgearbeitet, da dies eine bessere Performanz liefert. Das Umsortieren der Zeilen verspricht einen weiteren Performanzgewinn. Nach der Sortierung liegen alle Zeilen mit Nullen direkt hintereinander und die Bearbeitung einer Spalte mit führenden Nullen kann dann mit dem ersten Nichtnull-Element begonnen werden. Ein Beispiel ist in Abbildung 2.5 zu finden.

Matrix:

1	0	0	2
3	4	5	0
0	6	7	0
8	0	9	10

Werte:

1	2	0
6	7	0
3	4	5
8	9	10

Spaltenindex:

1	4	1
2	3	3
1	2	3
1	3	4

Abbildung 2.5: Beispiel für das ELL-Datenformat (vgl. [SaadY-2003] Seite 91).

2.6 Blocked Ellpack

Eine Kombination aus *Ellpack-Itpack*-Format und *Block-Compressed-Row-Storage*-Format ist das *Blocked-Ellpack*-Format (vgl. [ChoiJ-2010] Seite 4). Die Einträge der Matrix werden in Blöcke gruppiert. Alle Blöcke, die Werte ungleich Null enthalten, werden nach links geschoben. Dazwischenliegende Blöcke, die nur Nullen enthalten, werden entfernt. Die Blockzeile mit den meisten Blöcken bestimmt die Länge aller Blockzeilen. Die kürzeren Blockzeilen werden mit Blöcken aufgefüllt, die nur Nullen enthalten. Da alle Blockzeilen die gleiche Länge besitzen, kann eine Datenstruktur für den Anfang eine Blockzeile eingespart werden. Es wird eine Datenstruktur für das Speichern der Blockspaltenindizes benötigt. Die Blockspaltenindizes werden in einer Integer-Matrix gespeichert, die Länge entspricht der Anzahl der Blöcke. Die Größe der Werte-Matrix ist abhängig von der Anzahl der Blockzeilen und der Anzahl der Blöcke pro Zeile.

Matrix (Blockgröße: 2x2):

1	0	0	2	0	0
0	3	4	5	0	0
0	0	0	0	6	7
0	0	0	0	8	9

Werte:

1	0	0	3	0	2	4	5
6	7	8	9	0	0	0	0

Blockspaltenindex:

1	2
3	0

Abbildung 2.6: Beispiel für das BELL-Datenformat (vgl. [ChoiJ-2010] Seite 4).

3 GPGPU Programmierung

Das eigentliche Einsatzgebiet von GPUs ist die Berechnung der Bildschirmausgaben, diese können bei aufwendigen 3D-Szenen sehr rechenintensiv sein. Im Gegensatz zum herkömmlichen Einsatz der GPUs bezeichnet man mit dem Terminus *General-Purpose Computation on Graphics Processing Unit* (GPGPU) Berechnungen auf der GPU, die nicht für die Bildschirmausgabe gedacht sind, sondern Berechnungen für Anwendungen, die primäre nicht visualisiert werden. Falls notwendig werden die Ergebnisse der Berechnungen durch einen weiteren Prozess aufbereitet und anschließend visualisiert. Aktuell übernehmen hauptsächlich große CPU-Cluster die Berechnungen im *High Performance Computing* (HPC). Allerdings erobernd die ersten GPU-Cluster die TOP 500 der Supercomputer [TOP50-2011]. Platz 1 hat der chinesische GPU-Cluster Tianhe-1A mit 7.168 Nvidia-M2050 GPUs und 14.336 Intel-Xeon-CPUs erobert. Der Architekturvergleich aus Abbildung 3.1 unterstützt diese These. Die CPU ist für die Ausführung von sequenziellen Programmen und die GPU ist für hochparallele Programme entwickelt worden. Dies zeigt sich in der geringen Anzahl der Arithmetic Logic Units (ALU) einer CPU im Vergleich zur GPU. Die ALUs der CPU können in Verbindung mit der umfangreicheren Kontroll- und Steuereinheit komplexe Befehle handhaben. Die komplexe Logik der Kontroll-/Steuereinheit der CPU kann sequenziellen Code durch eine Neuanordnung bzw. Parallelisierung der Befehle beschleunigen. Die Logik der Kontroll-/Steuereinheit der GPU ist nicht so hochentwickelt. Die GPU verzichtet zugunsten der ALUs auf einen umfangreichen Cache.

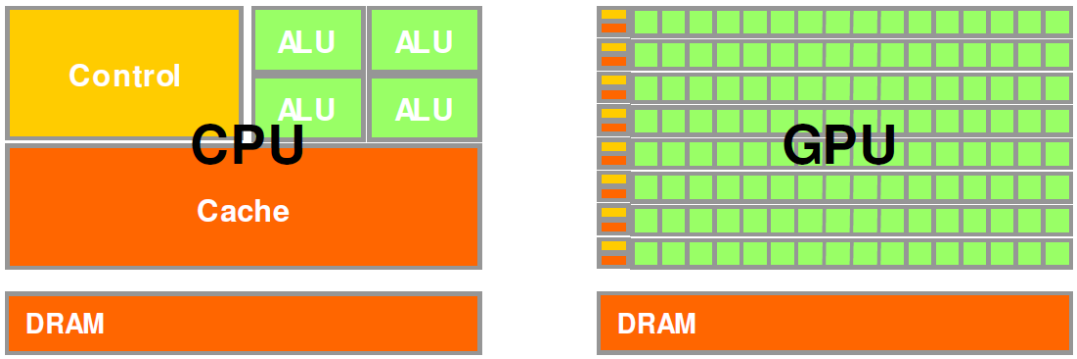

Abbildung 3.1: Vergleich der CPU und GPU Architektur ([KirkD-2010] Seite 4)

Nach Flynns Taxonomie sind die GPUs als *Single Instruction Multiple Data* (SIMD) Systeme zu klassifizieren. Die Vektorrechner gehören ebenfalls zur Klasse der *Single Instruction Multiple Data Systeme*, allerdings werden diese kaum noch eingesetzt. In der Vergangenheit ist im HPC-Bereich eine Reihe von Konzepten für den Einsatz von Vektorrechnern entwickelt worden. Diese Konzepte auf GPUs zu übertragen bzw. sie für die GPGPU-Programmierung zu nutzen, sind erfolgversprechend sein.

Die GPGPU-Programmierung bietet für den Einsatz im *High Performance Computing* großes Potenzial. GPUs sind hochparallele Prozessoren, die auf numerische Operationen spezialisiert wurden. Nicht alle Applikationen, die durch Erhöhung der CPU-Kerne beschleunigt werden, können vom Einsatz einer GPU profitieren. Vor allem rechenintensive Programme profitieren vom Einsatz einer GPU als Co-Prozessor. Geeignete Aufgabengebiete sind Simulationen in Biochemie, Physik oder auch bei der Aufbereitung von Bildern in der Medizin.

Die bisherigen Programmierschnittstellen der GPUs sind auf den Einsatz im Grafik- und Computerspieleumfeld zugeschnitten, z. B. OpenGL [Khron-2010a] oder Microsoft DirectX [Micro-2010]. Diese Programmierschnittstellen sind schlecht für den Einsatz im Bereich der Numerik geeignet. Erst durch die Entwicklung der geeigneteren Programmierschnittstellen *Compute Unified Device Architecture* (CUDA), *ATI Stream* und *Open Computing Language* (OpenCL) wurde der Einsatz von GPUs für numerische Probleme interessant. Im folgenden Teil werden die beiden Programmiertechniken CUDA und OpenCL beschrieben. Zusätzlich werden nützliche Bibliotheken für CUDA bzw. OpenCL vorgestellt.

3.1 CUDA

Die *Compute Unified Device Architecture* (CUDA) (vgl.[KirkD-2010], [NVIDI-2011e], [NVIDI-2011f]) wurde von NVIDIA entwickelt und ermöglicht neuere NVIDIA GPUs als Co-Prozessoren für nicht grafikspezifische Berechnungen einzusetzen. Die Programmentwicklung erfolgt in C mit der NVIDIA Erweiterung *C for CUDA*. Diese Erweiterung ergänzt die C-Sprachdefinition mit neuen Funktionen und einer Sprachsyntax, die für die GPGPU-Programmierung benötigt werden. Bevor mit der Entwicklung von CUDA-Programmen begonnen werden kann, müssen zuerst die Entwicklungstreiber für die GPU und die benötigten CUDA-Bibliotheken installiert werden. NVIDIA stellt mit dem *CUDA Toolkit 3.2* alle benötigten CUDA-Bibliotheken und Tools (z.B. Compiler NVCC) bereit. Zusätzlich werden auch die benötigten Bibliotheken für OpenCL 1.0 installiert, damit auf den NVIDIA GPUs OpenCL-Code ausgeführt werden kann. In einem zusätzlichen SDK sind CUDA und OpenCL Beispielprogramme enthalten. Der NVIDIA Compiler NVCC verarbeitet (kompilieren, linken) den CUDA-Code; Standard-C-Code gibt der NVCC an den Standard-Compiler des Systems weiter.

3.1.1 Parallelisierungsarchitektur

Ein CUDA-Programm besteht aus zwei Arten von Programmcode, der Host-Code und der Device-Code. Der Host-Code wird auf der CPU und im RAM des Computers ausgeführt. Der Device-Code wird auf der GPU und im RAM der GPU ausgeführt.

Ein CUDA-Programm besteht aus mindestens einem Kernel. Ein Kernel besteht aus C-Programmcode, der von beliebig vielen Threads auf einer GPU ausgeführt wird. Die Threads sind in Grids organisiert. Ein Grid kann nur einen Kernel abarbeiten, deshalb führen alle Threads eines Grids den gleichen Code aus. Die Threads eines Grids werden in Blöcke unterteilt, die Blöcke sind in einem 2D-Array organisiert. Das Array hat eine maximale Größe von 65.536 mal 65.536 Blöcken. Die Threads eines Blocks sind in einem 3D-Array angeordnet. Ein Block kann maximal 512 Threads verwalten. 32 Threads eines Blocks werden zu einem *Warp* zusammengefasst, deshalb sollte die Anzahl der Threads eines Blocks ein Vielfaches von 32 sein. Alle Threads eines *Warps* können gleichzeitig nur den gleichen Befehl abarbeiten. Sollte dies nicht möglich sein, werden die Threads eines *Warps* serialisiert. Die genannten Hardware Einschränkungen gelten für die Tesla- und die Fermi-Architektur. Die logische Organisation der Threads wird in Abbildung 3.2 verdeutlicht.

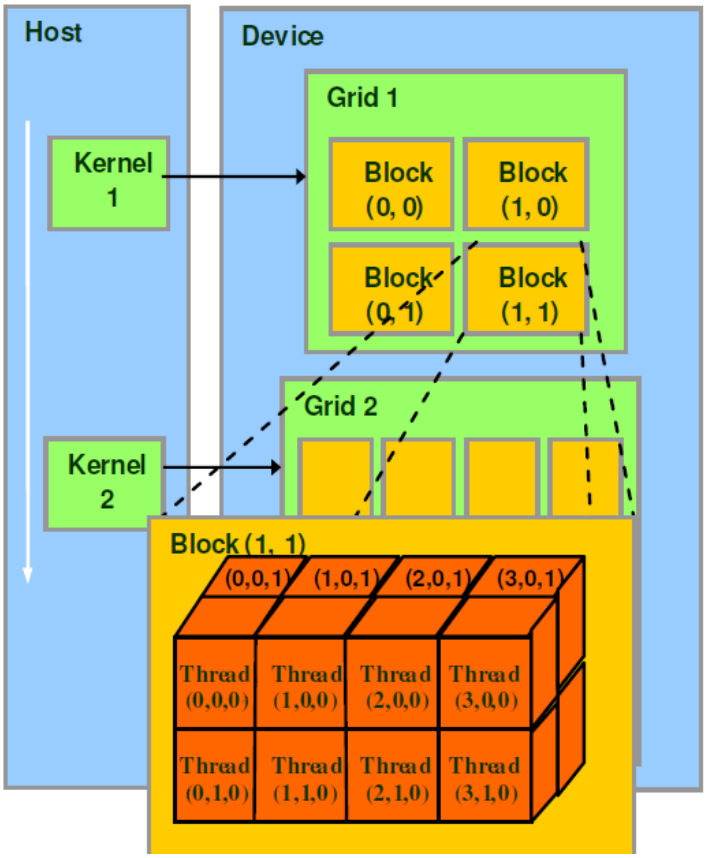

Abbildung 3.2: CUDA-Parallelisierungsmodell ([KirkD-2010] Seite 54).

Die logische Strukturierung der Threads ist für die Programmentwicklung essenziell. Die Adressierung der einzelnen Threads ist durch diese Struktur einfach möglich. Die Datenstrukturen vieler Probleme sind Vektoren und Matrizen, die in Arrays gespeichert werden. Während der Implementierung der Algorithmen werden die Thread-IDs zur Adressierung der Daten in den Arrays verwendet. Die Thread-ID lässt sich mittels

der X-,Y- und Z-Koordinate innerhalb eines Blocks ermitteln. Die Block-ID wird mittels X- und Y-Koordinate innerhalb des Grids berechnet. Laufzeitvariablen stellen die benötigten Informationen dem Programmierer zur Verfügung.

3.1.2 Speicherarchitektur

Die CUDA-Architektur kennt mehrere unterschiedliche Speicher, die unterschiedliche Transferraten und Größen besitzen. Die tatsächlichen Transferraten und Größen sind von der Hardware abhängig. Jeder Thread hat seinen eigenen *Local Memory* und Register. Der *Local Memory* wird verwendet, wenn keine freien Register übrig sind, dann werden diese Daten in den *Global Memory* ausgelagert. Alle Threads eines Blockes teilen sich den *Shared Memory*. Der *Shared Memory* wird für Daten verwendet, die von mehreren Threads eines Blocks benötigt werden. Die Register und der *Shared Memory* sind direkt auf dem GPU-Chip integriert. Alle anderen Speicher befinden sich auf dem RAM der GPU. Der *Global Memory* wird zum Datenaustausch zwischen dem Host und dem Device verwendet. Der *Global Memory* ist der langsamste Speicher. Einen speziellen Speicher stellt der *Constant Memory* da. Das Device kann auf den *Constant Memory* nur lesend zugreifen. Dieser ist schneller als der *Global Memory*, wenn mehrere Threads gleichzeitig auf die gleichen Daten im *Constant Memory* zugreifen. Abbildung 3.3 verdeutlicht die Speicherarchitektur.

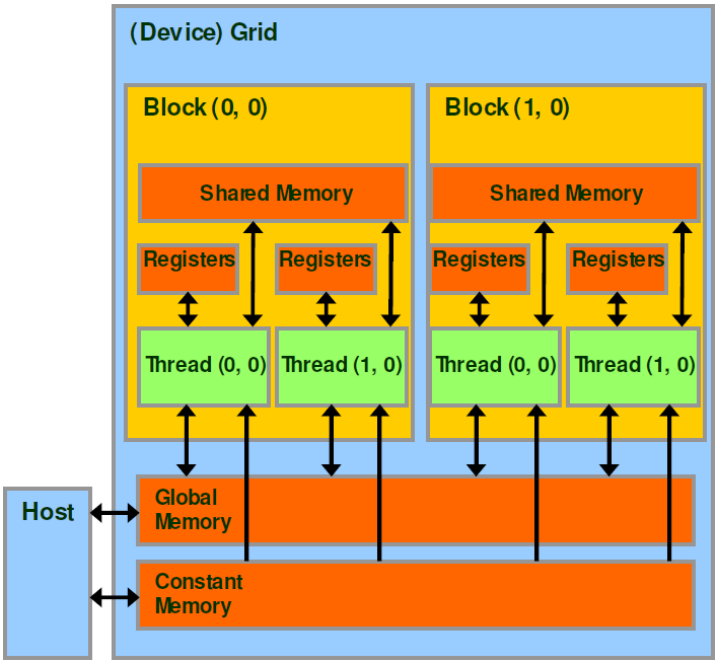

Abbildung 3.3: CUDA-Speicherarchitektur ([KirkD-2010] Seite 47).

Der Programmierer muss Speicher explizit im *Global Memory* des Devices allozieren und die Daten vom Host in den *Global Memory* des Device kopieren. Dafür

stellt CUDA eigene Befehle bereit. Die Register und der *Shared Memory* sind die schnellsten Speicher der GPU. Damit man diesen Vorteil nutzen kann, müssen die Algorithmen besonders sorgsam entwickelt werden, da diese Speicher im Vergleich zum *Global Memory* sehr klein sind.

3.2 OpenCL

Die *Open Computing Language* (OpenCL) (vgl.[Khron-2010], [NVIDI-2011b], [NVIDI-2011c]) ist eine Programmierplattform für die Entwicklung von parallelen Programmen, die auf GPUs oder/und CPUs ausführbar sein sollen. Aktuell ist die Spezifikation in der Version 1.1 erschienen, allerdings wird diese Version noch nicht von allen Hardwareherstellern unterstützt. Die Unterstützung von heterogenen Systemen macht OpenCL besonders interessant für das *High Performance Computing*. Die Simulationen könnten auf den unterschiedlichsten Clustern ausgeführt werden. CPU-Cluster oder GPU-Cluser von unterschiedlichen GPU-Herstellern werden von OpenCL unterstützt. Die Entwicklung der Programme ist nicht an eine bestimmte Hardware-Plattform gebunden. Ein Wechsle der Hardware-Plattform ist zu einem späteren Zeitpunkt immer noch möglich. Der Hardware Hersteller der GPU bzw. CPU muss einen OpenCL-Treiber und eine OpenCL-Bibliothek bereitstellen. OpenCL wurde ursprünglich von Apple entwickelt. Apple hat die Standardisierung an die Khronos Group abgegeben. An der Weiterentwicklung beteiligen sich AMD, Apple, NVIDIA und viele weitere Unternehmen der Hightech-Industrie. In Abbildung 3.4 ist das Plattformmodell von OpenCL dargestellt. Ein Host besitzt mindestens ein *Compute Device*, dies kann eine GPU oder eine CPU mit mehreren Kernen sein. Ein *Compute Device* besteht aus mehreren *Compute Units*, die jeweils aus mehreren *Processing Elements* bestehen.

In der Entwicklung von OpenCL-Programmen muss keine Rücksicht auf spezielle Zielplattformen genommen werden. OpenCL bietet eine generische API für die Entwicklung des Programms. Erst zur Laufzeit übersetzt der OpenCL-Compiler den OpenCL-Code passend zur konkreten Zielplattform und führt diesen nach der Übersetzung auf dem OpenCL-Gerät aus. Die Hardware-Hersteller müssen diesen Standard umsetzten. Geeignete Treiber und OpenCL-Bibliotheken müssen von den Hardware-Herstellern entwickelt werden. NVIDIA stellt im *CUDA Toolkit 3.2* alle benötigten Komponenten für OpenCL bereit. Im *ATI Stream SDK 2.2* stellt AMD Treiber und Bibliotheken für OpenCL bereit. Die Bibliotheken enthalten auch den OpenCL-Compiler.

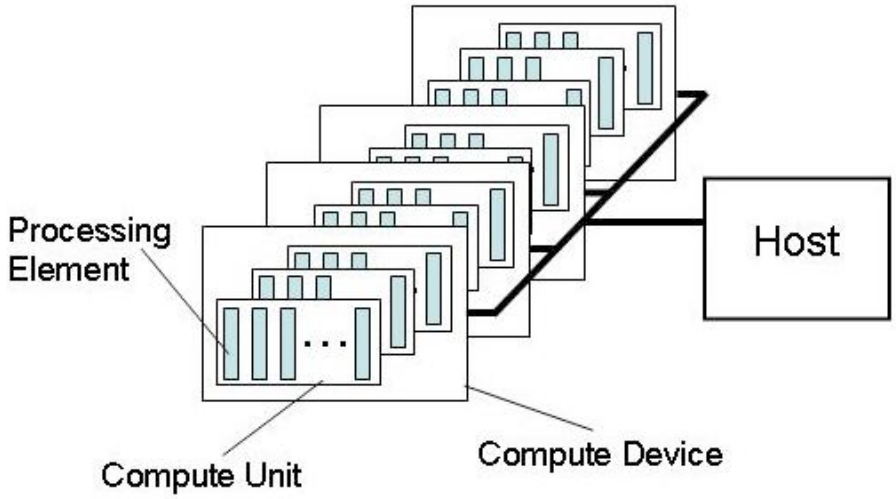

Abbildung 3.4: OpenCL-Platformmodell ([Khron-2010], Seite 22).

3.2.1 Parallelisierungsarchitektur

Ein OpenCL-Programm besteht aus zwei Arten von Programmcode, dem Host-Code und dem Kernel. Der Host-Code wird auf dem Host ausgeführt und steuert die Ausführung des Kernels. Der Kernel wird auf dem OpenCL-Device ausgeführt. Eine Instanz des Kernels wird *Work-Item* genannt und von einem *Processing Element* ausgeführt. Die *Work-Items* werden in einem Gitter organisiert, dass 1, 2 oder 3 Dimensionen besitzt. Diese Gruppierung wird *Work-Group* genannt. Ein *Work-Item* kann innerhalb der *Work-Group* mittels der N Indizes des N-dimensionalen Gitters adressiert werden, die *Locale ID*. Die *Work-Groups* sind in einer Matrix organisiert, die 1, 2 oder 3 Dimensionen besitzt. Diese Anordnung wird mit *NDRange* bezeichnet. Die Gitter des *NDRange* und der *Work-Group* müssen in der gleichen Dimension definiert werden. Die Adressierung der *Work-Groups* innerhalb des *NDRange* erfolgt mit dem gleichen Schema wie die Adressierung der *Work-Items* innerhalb der *Work-Group*. Ein *Work-Item* kann auch anhand der *Global ID* adressiert werden. Die *Global ID* bezieht sich auf die Position des *Work-Item* in Bezug auf den *NDRange*. Die N Indizes des N-dimensionalen Gitters bilden die *Global ID*. In Abbildung 3.6 sind die Details dargestellt.

In OpenCL ist eine logische Strukturierung der *Work-Items* für die Adressierung der Daten, die ein *Work-Item* für die Berechnungen benötigt, erforderlich. Die *Locale ID* und die *Global ID* können mittels Methoden, die OpenCL bereitstellt, abgefragt werden.

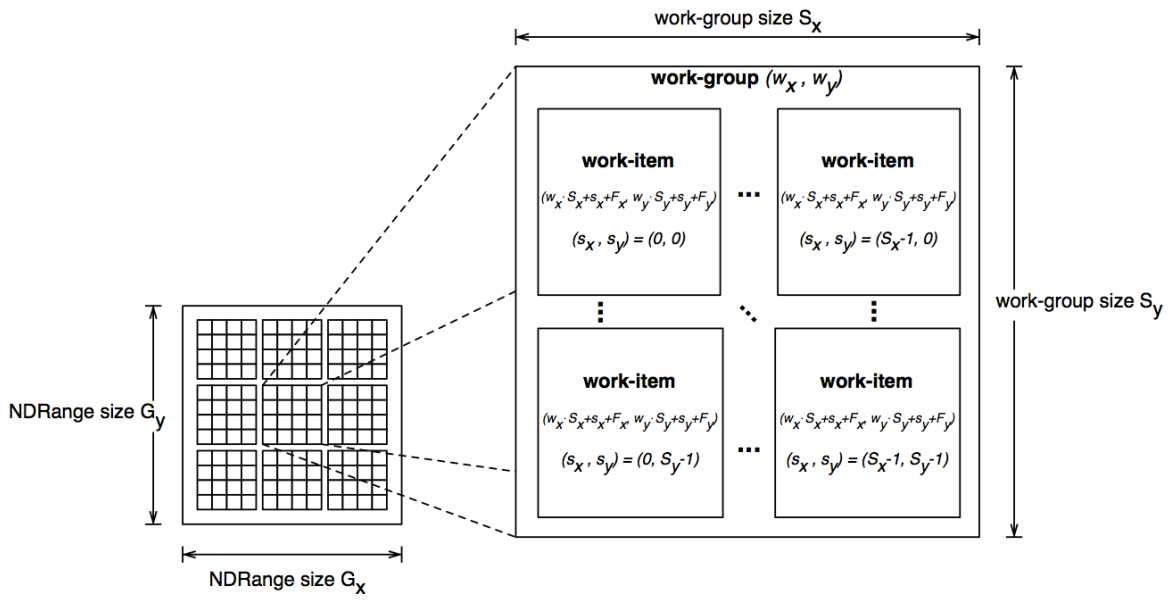

Abbildung 3.5: OpenCL Parallelisierungsmodell ([Khron-2010] Seite 24).

3.2.2 Speicherarchitektur

Die Speicherarchitektur von OpenCL ist eine logische Strukturierung des Speichers. Diese Strukturierung lässt keine Rückschlüsse auf die Hardware zu. Eine Aussage bezüglich der Geschwindigkeit und der Größe des Speichers ist nicht möglich. Diese Informationen sind für eine performante Implementierung sehr wichtig, deshalb ist ein genaues Studium der Dokumentation des Hardware-Herstellers notwendig. Auf den *Private Memory* kann nur ein *Processing Element* (*Work-Item*) zugreifen. Alle *Processing Element* (*Work-Item*) einer *Compute Unit* (*Work-Group*) können gemeinsam den *Local Memory* verwenden. Der *Local Memory* wird für den Austausch von Daten zwischen einzelnen *Work-Items* verwendet, die zur gleichen *Work-Group* gehören. Alle *Work-Items* können auf den *Global Memory* zugreifen. Der *Constant Memory* ist ein Teil des *Global Memory*. In den *Global Memory* werden Daten durch den Host geladen und die *Work-Items* können lesend auf diesen Speicher zugreifen.

3.3 GPGPU-Bibliotheken

Für CUDA und OpenCL wurden Bibliotheken entwickelt, die das Entwickeln von GPGPU-Programmen vereinfachen und beschleunigen können. Diese Bibliotheken ermöglichen es, ohne detaillierte Kenntnisse von CUDA oder OpenCL GPGPU-Programme zu entwickeln. Die beste Performanz für das eigene Programm kann allerdings nicht garantiert werden. Es kann sich lohnen auf den Einsatz der Bibliotheken zu verzichten und die Algorithmen für das eigene Programm selbst zu implementieren. Problematisch ist z.B., wenn die Datenstrukturen für den Einsatz der Bibliotheken

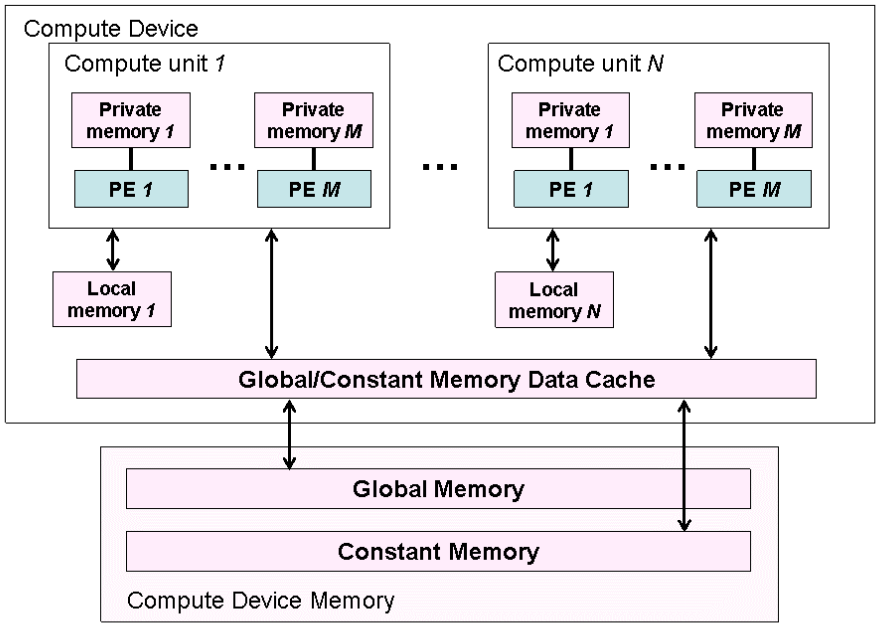

Abbildung 3.6: OpenCL-Speicherarchitektur ([Khron-2010], Seite 27).

konvertiert werden müssten.

3.3.1 Cusp

Die Cusp Bibliothek [Garla-2010] implementiert Algorithmen der linearen Algebra für Berechnungen mit dünnbesetzten Matrizen. Die Algorithmen führen ihre Berechnungen, mittels CUDA auf der GPU aus. Eine vereinfachte und flexiblere Handhabung wird durch den Einsatz der Thrust-Bibliothek erreicht. Cusp stellt zwei iterative Gleichungslöser zur Verfügung *Conjugate Gradient* (CG) und *Stabilized Bi-Conjugate Gradient* (BiCGStab). Eine Reihe von Datenformaten für dünnbesetzte Matrizen werden unterstützt (z.B. COO, CSR, DIA).

3.3.2 Thrust

Die Thrust Bibliothek [Hober-2010] implementiert eine Reihe von parallelen Algorithmen, zum Beispiel Sortier-, Transformations-, Reduktions- und Reordering-Algorithmen. Viele Funktionen der *C++ Standard Template Library* (STL) wurden implementiert und mit ähnlichen Interfaces versehen. Mithilfe dieser Bibliothek lassen sich schnell und ohne weitere Einarbeitung in CUDA GPGPU-Programme erstellen.

3.3.3 Vienna Computing Library

Die *Vienna Computing Library* (ViennaCL) [RuppK-2010] ist eine Header-only-Bibliothek, die in C++ implementiert wurde und OpenCL verwendet. Dem Einsatz von OpenCL ist es zu verdanken, dass mit ViennaCL die Berechnungen auf CPUs und GPUs (NVIDIA und AMD) ausgeführt werden können. In der aktuellen Version sind die *Basic Linear Algebra Subprograms* (BLAS) von Level 1 (Vektor-Vektor Routinen) und Level 2 (Matrix-Vektor-Routinen) implementiert. Die BLAS-Level-3-Routinen (Matrix-Matrix-Routinen) fehlen noch. Im Rahmen dieser Masterarbeit sind auch die Matrix-Matrix-Routinen notwendig, diese können implementiert werden und ggf. in das Projekt ViennaCL zurück fließen. ViennaCL stellt drei iterative Gleichungslöser zur Verfügung *Conjugate Gradient* (CG), *Stabilized Bi-Conjugate Gradient* (BiCG-Stab) und *Generalized Minimum Residual* (GMRES).

3.4 CUDA und OpenCL Nomenklatur

CUDA und OpenCL verwenden eine Reihe von unterschiedlichen Begriffen mit der gleichen Bedeutung. In den folgenden Tabellen 3.1 und 3.2 werden Begriffe der Parallelisierungsarchitektur und Speicherarchitektur in den korrekten Kontext gesetzt.

OpenCL-Parallelisierungsarchitektur	CUDA-Parallelisierungsarchitektur
Kernel	Kernel
Host Programm	Host Programm
NDRange	Grid
Work-Item	Thread
Work-Group	Block

Tabelle 3.1: Nomenklatur der Parallelisierungsarchitektur ([KirkD-2010] Seite 210).

OpenCL-Speicherarchitektur	CUDA-Speicherarchitektur
global memory	global memory
constant memory	constant memory
local memory	shared memory
private memory	local memory

Tabelle 3.2: Nomenklatur der Speicherarchitektur ([KirkD-2010] Seite 207).

3.5 Vergleich der Tesla und Fermi Architektur von NVIDIA

Nvidia beschreibt die Eigenschaften der CUDA-Grafikkarten mit den *Compute Capabilities* (siehe [NVIDI-2011c], Seite 50 - 60 und [NVIDI-2011d]). Die Eigenschaften

der Tesla-Architektur wird durch die *Compute Capabilities* 1.0 bis 1.3 beschrieben. Mit der *Compute Capability* 2.0 werden die Eigenschaften der Fermi-Architektur beschrieben. Die Fermi-Architektur stellt einige neue Leistungsmerkmale bereit, die eine zusätzlich Beschleunigung für GPGPU-Programme bewirken. Es wird eine Cache-Hierarchie eingeführt, die den Datendurchsatz erhöht. Mehr Recheneinheiten für die Berechnung von Gleitkommazahlen mit doppelter Genauigkeit werden bereitgestellt. Unterschiedliche Kernel eines Programms können konkurrierend auf der GPU ausgeführt werden. Damit kann eine bessere Auslastung der GPU erreicht werden. Es ist möglich, aus unterschiedlichen Threads GPGPU-Kernel auf der GPU zu starten. Im Gegensatz zur Tesla-Architektur ist die Fermi-Architektur *Thread-Safe*. In der Tabelle 3.3 sind die Hardwareeigenschaften der beiden Architekturen aufgelistet.

Eigenschaft	Tesla (1.3)	Fermi (2.0)
Anzahl Register pro Multiprozessor	16K	32K
Größe des Constant Memory	64 KB	64 KB
Maximale Anzahl der Warps pro Multiprozessor	32	48
Maximale Anzahl der Threads pro Multiprozessor	1024	1536
Maximale Größe des Shared Memory	16 KB	48 KB

Tabelle 3.3: Eigenschaften der Tesla und Fermi Architektur (vgl. [NVIDI-2011c], Seite 48).

3.6 Besonderheiten

Bei der Entwicklung von GPGPU-Programmen müssen einige Besonderheiten beachtet werden. Das Debugging des Kernel-Codes ist besonders problematisch. Einige Hilfestellungen werden im folgenden Kapitel behandelt.

3.6.1 Debuggen von CUDA-Code

NVIDIA stellt den Debugger *CUDA-GDB* bereit, dieser ist eine Erweiterung des *GNU-Debugger* (GDB). Der *CUDA-GDB* bietet einige zusätzliche Befehle an, die es ermöglichen CUDA-Code zu debuggen. Es können einzelne Blöcke und Threads untersucht werden. Die Inhalte der Variablen können angezeigt werden. Es gibt keine grafische Benutzeroberfläche für den *CUDA-GDB*. Die Bedienung ist Konsolen-basiert. Im Entwicklungssystem sind zwei Grafikkarten notwendig. Eine Grafikkarte wird für die Ausführung des CUDA-Codes benötigt und die zweite Grafikkarte muss die Bildschirmausgabe erzeugen. Wenn das System nur eine GPU hätte, müsste diese die Bildschirmausgabe und die Ausführung des GPGPU-Codes bewältigen. Allerdings wird während des Debugging die Codeausführung gestoppt und damit auch die GPU

angehalten, dann könnten keine Ausgaben mehr auf dem Bildschirm erzeugt werden. Dies hätte zur Folge, dass man keine Debuginformationen sehen könnte. NVIDIA bietet die Erweiterung *Parallel Nsight* für *Microsoft Visual Studio* an. *Parallel Nsight* ist ein grafisches Debugging und Profiling Werkzeug für NVIDIA-Grafikkarten. Da *Visual Studio* benötigt wird, kann *Parallel Nsight* nur auf Windows Systemen eingesetzt werden.

3.6.2 Debuggen von OpenCL-Code

Das Debugging eines OpenCL-Kernels ist unter Linux nur auf der CPU möglich. Es gibt keine Möglichkeit, den OpenCL-Kernel unter Linux auf einer GPU zu debuggen. Es kann nur der Host-Code mit einem C/C++-Debugger untersucht werden. Zwischen den OpenCL-Befehlen im Host-Code kann man mittels OpenCL-Befehlen die Speicherbereiche der GPU auslesen. Die Ausgabe der Daten muss zusätzlich implementiert werden. Diese Variante ist sehr umständlich und ermöglicht nur sehr eingeschränktes Untersuchen des Kernel-Codes. *Graphic Remedy* hat den Debugger *gDEBugger* [Graph-2010] entwickelt, der diese Variante implementiert. Es können Haltepunkte an den OpenCL-Befehlen im Host-Code definiert werden, und die grafische Oberfläche des Debuggers zeigt alle allozierten Speicherbereiche der GPU an. NVIDIA bietet das grafische Debugging und Profiling Werkzeug *Parallel Nsight* an, dass eine Erweiterung für *Microsoft Visual Studio* ist. *Parallel Nsight* kann OpenCL-Code nur auch NVIDIA-Grafikkarten debuggen und profilen. Da *Parallel Nsight* eine Erweiterung für *Visual Studio* ist, kann es nur auf Windows Systemen eingesetzt werden.

3.6.3 Timeout Problem unter Windows

Ab *Microsoft Windows Vista SP1* können Programme die GPU maximal 2 Sekunden lang belegen, wenn diese GPU auch zur Anzeige benötigt wird. Nach 2 Sekunden bricht das *Timeout Detection and Recovery* (TDR) [Micro-2011] von Windows das Programm ab. TDR wurde zur Stabilitätsverbesserung von Windows eingeführt. Das Einfrieren des Desktops soll durch TDR verhindert werden. Für ein GPGPU-Programm ist eine maximale Laufzeit von 2 Sekunden viel zu kurz, deshalb muss diese Funktion abgeschaltet werden. Mit folgendem Registry-Schlüssel kann TDR abgeschaltet werden:

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\GraphicsDrivers]
    "TdrLevel"=dword:00000000
```

4 Performanzanalyse

Die Performanzanalyse ist ein zentraler Bestandteil dieser Arbeit, damit eine qualitative Aussage über die Performanz der GPGPU-Variante des TRACE-Codes möglich ist. Deshalb ist es notwendig, geeignete Metriken für GPGPU-Programme zu identifizieren. Dazu müssen geeignete Testfälle erstellt werden und ein faires Testsystem und Testverfahren für beide Varianten des TRACE-Codes bereitgestellt werden.

4.1 Theoretische maximale Beschleunigung

Gene Amdahl (vgl. [Amdah-1967] und [Moore-2011], Seite 23 - 25) beschrieb eine Formel zur Berechnung der maximalen Beschleunigung eines Programms durch parallele Ausführung. Dieses Gesetz wurde unter dem Namen *Amdahlsches Gesetz* bekannt. Ein Programm wird für die Abschätzung der Beschleunigung in sequenzielle und parallele Abschnitte unterteilt. Die theoretische maximale Beschleunigung ist abhängig von dem parallelisierbaren Anteil des Programms und der Anzahl der verwendeten Prozessoren (p). Der sequenzielle Anteil des Programms (f) ist der begrenzende Faktor für die Beschleunigung. Der Anteil des parallel ausführbaren Programms sei $(1 - f)$. Durch Einsatz von p Prozessoren lässt sich dieser auf $(\frac{(1-f)}{p})$ reduzieren. Die Beschleunigung des gesamten Programms ergibt sich aus dem Kehrwert der Summe des sequenziellen Programmanteils und des reduzierten parallelen Programmanteils. Die Effizienz der Beschleunigung berechnet sich aus dem Verhältnis der Beschleunigung zu der Anzahl der verwendeten Prozessoren.

$S_t(p) = \frac{1}{f + \frac{(1-f)}{p}}$	Amdahlsches Gesetz: Theoretische maximale Beschleunigung
$S_r(p) = T(1)/T(p)$	Beschleunigung
$E(p) = S(p)/p$	Effizienz

Beispiel: Die Ausführung eines Programms auf einer CPU mit einem Kern benötigt 10 Stunden. Von den 10 Stunden können 2 Stunden nur sequenziell abgearbeitet werden. Nur 80% des Programms können parellel ausgeführt werden. Diese 80% könnten von unendlich vielen Prozessoren ausgeführt werden, damit würde die Ausführung des parallelen Programmteils theoretisch keine Zeit benötigen. Es bleibt eine Gesamtlaufzeit von 2 Stunden (sequenzieller Programmteil) übrig. Es kann eine maximale Beschleunigung um den Faktor 5 erreicht werden.

4.2 Metriken für GPUs

Im folgend Teil werden einige Metriken für GPUs erläutert und die geeigneten Metriken ausgewählt. Im *NVIDIA OpenCL Best Practices Guide* [NVIDI-2011b] sind einige Performanzmetriken erläutert.

4.2.1 Laufzeit

Es gibt zwei Arten von Laufzeit-Messungen, die CPU-Zeit und die Laufzeit (Wall-Clock-Time). Die CPU-Zeit ist die benötigte Zeit für das Abarbeiten der Befehle eines Programms auf der CPU bzw. den CPUs. Die Zeiten der einzelnen CPUs werden bei der CPU-Zeit addiert, deshalb kann die CPU-Zeit bei parallelisierten Programmen größer als die Laufzeit sein. Die Zeitspanne zwischen Start und Ende eines Programms wird als Laufzeit bezeichnet. Bei GPGPU-Programmen muss zusätzlich die GPU-Zeit gemessen werden, damit auf die GPU ausgelagerte Berechnungen erfasst werden können. Mit dem Befehl `clGetEventProfilingInfo(...)` können in OpenCL die GPU-Timer abgefragt werden.

Für Zeitmessungen müssen mehrere Timer in den Programmcode integriert werden. Für die GPU-Zeit, die CPU-Zeit und die Laufzeit ist jeweils ein Timer notwendig.

4.2.2 Speicherbandbreite

Eine der wichtigsten Metriken ist die erreichte Speicherbandbreite. Für eine Beurteilung der erreichten Bandbreite ist es notwendig, die theoretische Bandbreite zu berechnen und die tatsächlich erreichte Bandbreite zu messen. Wenn die erreichte Bandbreite wesentlich geringer als die theoretische Bandbreite ist, sollten einige Optimierungen beim Speicherzugriff vorgenommen werden. Eventuell können die Speicherzugriffe auf schnellere Speicherbereiche verlagert werden. Negative Auswirkungen auf die Bandbreite kann der Zugriff auf zu kleine Speicherbereiche im *Global Memory* haben. Der *Global Memory* ist in Speicherbereiche von 64 Byte bzw. 128 Byte ausgerichtet. Die Zugriffe auf den *Global Memory* erfolgen immer für *Half-Warps* (16 *Threads* bzw. 16 *work-items*). Damit eine hohe Bandbreite erreicht werden kann, sollte pro *Half-Warp* nur auf einen der Speicherbereiche von 64 Bytes bzw. 128 Bytes zugegriffen werden. Die zu lesenden Speichersegmente pro *Half-Warp* sollten eine Länge von 32, 64 oder 128 Bytes besitzen. Bei kleineren Segmentgrößen geht Bandbreite verloren. Falls abweichende bzw. größere Segmentgrößen verwendet werden, müssen mehrere Transaktionen für das Lesen der Daten angestoßen werden. Sind in den gelesenen Speichersegmenten Daten enthalten, die nicht benötigt werden, sinkt die effektive Bandbreite. NVIDIA bezeichnet dies mit *Coalesced Access to Global*

Memory. Weitere Details zu diesem Thema sind im *NVIDIA OpenCL Best Practices Guide* [NVIDI-2011b], Seite 13 bis Seite 18, zu finden.

Die theoretische Bandbreite des Speichers kann mit den Hardwareeigenschaften der GPU errechnet werden. Im folgenden Beispiel werden die Hardwareeigenschaften der *NVIDIA Tesla C1060* GPU verwendet. Die GPU hat eine Speicher-Taktrate von 800 MHz ($800 \cdot 10^6$). Das Speicherinterface hat eine Breite von 512 Bit, diese müssen in Byte umgerechnet werden ($512/8$). Die GPU verwendet Double Data Rate (DDR3) Speicher, deshalb muss das bisherige Ergebnis verdoppelt werden. Die effektive Bandbreite wird mittels Addition der gelesenen (B_r) und der geschriebenen Bytes (B_w) berechnet. Diese müssen in GB umgerechnet werden ($/10^9$). Die gesamte Menge der transferierten Daten wird durch die benötigte Zeit dividiert.

$B_t = ((800 \cdot 10^6)/\text{s} \cdot (512/8)\ \text{Bytes} \cdot 2)/10^9 = 102,4\ \text{GB/s}$	Theoretische Bandbreite B_t
$B_e = ((B_r + B_w)/10^9)/t$	Effektive Bandbreite B_e in GB/s t in Sekunden B_r, B_w in Bytes

4.2.3 GPU Auslastung

Diese Metrik zeigt ob die GPGPU-Kernel effektiv auf der GPU ausgeführt werden können. Die tatsächliche Auslastung der GPU lässt sich mit Profiling-Programmen ermitteln. Eine schlechte Auslastung der GPU kann viele Gründe haben:

- Die implementierten Kernel sind zu komplex. Es werden dadurch zu viele Register benötigt.
- Es werden *Branches* (If-Else-Anweisungen) im Programmablauf verwendet. Die *Branches* können zur Serialisierung des Programms führen und beeinflussen deshalb die GPU-Auslastung negativ.
- Es werden nicht genügend *Threads* erzeugt, damit die GPU ausgelastet ist. Möglicherweise könnte das Problem zu klein für eine Berechnung auf der GPU sein. Eine weitere Grund könnte sein, dass zu große Arbeitspakete definiert wurden und deshalb zu wenige *Threads* erzeugt werden.
- Zu viele Synchronisierungen der *Threads* können die GPU-Auslastung erheblich verringern.

4.2.4 Gleitkommaoperationen pro Sekunde

Die Gleitkommaoperationen pro Sekunde (Flop/s) ist eine Metrik für die Leistungsfähigkeit von Computersystemen. Diese Metrik beschreibt die Anzahl der Gleitkommaoperationen pro Sekunde, die ein Computersystem für bestimmte Operationen bzw. Algorithmen berechnen kann. Die FLOPS werden mittels einer Laufzeitmessung und der Anzahl der benötigten Gleitkommaoperationen für die Nutzdaten ermittelt. Zusätzliche Rechenoperationen für die Berechnung der Indizes oder der Schleifenkörper werden nicht mitgezählt. Die Anzahl der Gleitkommaoperationen wird durch die benötigte Zeit in Sekunden geteilt.

4.2.5 Verwendete Metriken

Zum Vergleich der beiden Varianten des TRACE-Codes werden Laufzeit-Messungen und die Berechnung der Gleitkommazahl-Operationen (FLOPS) durchgeführt. Die Qualität der implementierten Kernel wird mittels Bandbreiten-Berechnungen und GPU-Auslastung überprüft. Damit können eventuelle Schwächen in der Implementierung gefunden und die Kernel optimiert werden.

4.3 Analyse Werkzeuge

Der *Compute Visual Profiler* [NVIDI-2010] ist ein grafisches Profiling Werkzeug von NVIDIA. Im *CUDA Toolkit* von NVIDIA ist der *Compute Visual Profiler* enthalten. Es können eine Reihe von Leistungsindikatoren von GPGPU-Programmen gemessen werden, die auf einer NVIDIA GPU ausgeführt werden. Im Benutzerhandbuch des *Compute Visual Profiler* [NVIDI-2010] sind die Leistungsindikatoren beschrieben. Diese Leistungsindikatoren können Optimierungspotenzial im GPGPU-Programm aufzeigen. Zusammen mit dem *CUDA GPU Occupancy Calculator* [NVIDI-2011] lässt sich eine theoretische Abschätzung der Auslastung der GPU durch den GPGPU-Kernel errechnen. Dazu werden einige Parameter des GPGPU-Programms benötigt, die der *Compute Visual Profiler* liefert. Welcher Punkt die GPU-Ausnutzung begrenzt, wird in der Excel-Tabelle des *CUDA GPU Occupancy Calculator* angezeigt. Damit erhält man einige Hinweise für die Performanzoptimierung des GPGPU-Programms.

NVIDIA Parallel Nsight [NVIDI-2011a] ist ein Debugging- und Profiling-Programm von NVIDIA. Es ist eine Erweiterung von *Microsoft Visual Studio* und ist nur unter *Microsoft Windows* lauffähig. Deshalb ist es im Rahmen dieser Masterarbeit nicht einsetzbar.

gDEBugger ist ein Programm mit grafischer Benutzeroberfläche von *Graphic Remedy* zum Debuggen, Profilen und zur Speicheranalyse von OpenGL bzw. OpenCL Programmen. Dieses Programm ist unter Windows und Linux verfügbar. In dieser

Arbeit wird es zur Speicheranalyse und zum Profiling eingesetzt. Die verwendete Speicherbandbreite und die GPU-Auslastung kann mit diesem Programm ermittelt werden.

4.4 Beschreibung des ersten Testsystems

Das erste Testsystem ist ein Celsius R570 von Fujitsu Siemens Computer. Folgende Hardwareausstattung enthält das System:

- **Prozessoren:** 2x Intel XEON E5520 2.26GHz 8MB mit Turbo Boost mit je 4 Kernen
- **Arbeitsspeicher:** 16GB DDR3-1333 registered ECC
- **Chipsatz:** Intel 5520
- **1. Grafikkarte:** NVIDIA Quadro NVS 290 256MB GDDR3 (Bildschirmausgabe)
- **2. Grafikkarte:** NVIDIA Tesla C1060 4GB GDDR3 (Computing-Processor)
- **Festplatten:** 2TB (2x 500GB, 1x 1TB), SATA II, 7200 U/min

Das Betriebssystem ist Ubuntu 10.04.1 LTS 64 Bit mit der Kernel Version 2.6.32.27. Das spätere Zielsystem ist ebenfalls ein Linux/Unix System, deshalb erfolgt die Entwicklung ebenfalls auf einem Linux/Unix System. Der Compiler für den C/C++-Code ist die *GNU Compiler Collection* (GCC) in der Version 4.4. Der GPGPU-Code wird mit den Compilern übersetzt, die das *CUDA Toolkit 3.2.16* bereitstellt. Für OpenCL-Code wird der OpenCL-Compiler und für CUDA-Code der *NVIDIA CUDA Compiler Driver* (NVCC) verwendet. Die 64-Bit-Variante des Entwickler Grafikkartentreibers (Developer Driver) in der Version 260.19.26 wird verwendet. Die GPGPU-Variante und die MPI-Variante des TRACE-Codes wird auf dem Testsystem ausgeführt. Durch 2 Intel XEON Prozessoren mit jeweils 4 Kernen, bzw. durch Hyper-Threading sogar 8 virtuellen Kernen, kann die MPI-Variante gute Laufzeiten erreichen. Die zwei Grafikkarten des Systems ermöglichen eine einfachere Entwicklung der GPGPU-Variante des TRACE-Codes. Ein Debugging des GPGPU-Codes ist nur möglich, wenn das Entwicklungssystem zwei GPUs besitzt. Eine GPU kann für die Bildschirmausgaben verwendet werden und die andere GPU für das Ausführen des GPGPU-Codes. Der GPGPU-Code wird auf der NVIDIA Tesla C1060 ausgeführt. Das Hardwaredesign der Grafikkarte entspricht der NVIDIA Tesla Architektur. Folgende Hardwareeigenschaften hat die Grafikkarte:

- **Bezeichnung:** NVIDIA Tesla C1060

- **Architektur:** Tesla
- **NVIDIA compute capability:** 1.3
- **Multiprozessoren:** 30
- **Streaming-Prozessorkerne:** 240 (8 pro Multiprozessor)
- **Prozessor-Taktrate:** 1,3 GHz
- **Arbeitsspeicher:** 4 GB GDDR3
- **Speicher-Taktrate:** 800 MHz
- **Speicherschnittstelle:** 512 Bits
- **Constant Memory:** 64 KB
- **Shared Memory pro Multiprozessor:** 16 KB
- **Registers pro Multiprozessor:** 16384 je 32 Bit

Dieses Testsystem ist für beide Varianten des TRACE-Codes gut ausgestattet, deshalb ist eine faire Vergleichsmessung möglich.

4.5 Beschreibung des zweiten Testsystems

Das zweite Testsystem ist ein Precision T5500 von DELL. Folgende Hardwareausstattung enthält das System:

- **Prozessoren:** 2x Intel XEON E5630 2.53GHz 12MB mit Turbo Boost mit je 4 Kernen
- **Arbeitsspeicher:** 24GB DDR3-1333 registered ECC
- **Chipsatz:** Intel 5520
- **Grafikkarte:** NVIDIA Quadro 5000 2,5 GB GDDR5
- **Festplatten:** 500GB, SATA II, 7200 U/min

Das Betriebssystem ist SUSE Linux Enterprise Desktop 11.1 64 Bit mit der Kernel Version 2.6.32.24. Der Compiler für den C/C++-Code ist die *GNU Compiler Collection* (GCC) in der Version 4.3.4. Der GPGPU-Code wird mit den Compilern übersetzt, die das *CUDA Toolkit 3.2.16* bereitstellt. Für OpenCL-Code wird der OpenCL-Compiler

und für CUDA-Code der *NVIDIA CUDA Compiler Driver* (NVCC) verwendet. Die 64-Bit-Variante des Entwickler Grafikkartentreibers (Developer Driver) in der Version 260.19.26 wird verwendet. Dieses Testsystem besitzt nur eine Grafikkarte, deshalb muss das System für die Tests in den Runlevel 3 versetzt werden. Im Runlevel 3 steht kein X-Server und damit keine Grafische Benutzeroberfläche zur Verfügung, sondern nur die Konsole bzw. Terminalsitzungen. Dies entlastet die Grafikkarte und stellt mehr Rechenleistung für die GPGPU-Variante des TRACE-Codes bereit. CUDA Kernel lassen sich nur starten, wenn das X-System nicht aktiv ist. Das Hardwaredesign der Grafikkarte entspricht der NVIDIA Fermi Architektur. Folgende Hardwareeigenschaften hat die Grafikkarte:

- **Bezeichnung:** NVIDIA Quadro 5000
- **Architektur:** Fermi
- **NVIDIA compute capability:** 2.0
- **Multiprozessoren:** 11
- **Streaming-Prozessorkerne:** 352 (32 pro Multiprozessor)
- **Prozessor-Taktrate:** 1 GHz
- **Arbeitsspeicher:** 2,5 GB GDDR5 mit ECC
- **Speicher-Taktrate:** 1,5 GHz
- **Speicherschnittstelle:** 320 Bits
- **Constant Memory:** 64 KB
- **Shared Memory pro Multiprozessor:** 48 KB
- **Registers pro Multiprozessor:** 32768 je 32 Bit

4.6 Beschreibung der Testfälle

Die Testfälle sind Strömungsberechnungen im 3D-Raum. Die Lösung wird mit den Euler-Gleichungen (siehe Kapitel 5.1) berechnet. Es werden 6 Testfälle für die Performanzanalyse verwendet. Die Testfälle 2 bis 5 sind vom ersten Testfall abgeleitet. Die Anzahl der Zellen auf den 3 Achsen des Raums pro Block variieren. Alle Testfälle bestehen aus 8 Blöcken. Es wurden 8 Blöcke ausgewählt, damit die MPI-Variante des TRACE-Codes alle 8 Kerne des Testsystems ausnutzen kann. Details über die Testfälle sind in der Tabelle 4.1 festgehalten. Es wird die Strömung in einem Rohr mit ringförmigen Querschnitt simuliert. Ein Viertel des Rohres ist in Abbildung 4.1 dargestellt. In

Abbildung 4.2 ist das Rechengitter innerhalb des viertel Rohres zu erkennen. An den Rändern zu den weiteren Vierteln des Rohres sind periodische Randbedingungen definiert. Die Begrenzungen der Blöcke sind mit roten Linien dargestellt. Die weißen Linien begrenzen die Zellen. Der Körper wird mit dem Zylinder-Koordinatensystem (x Koordinate der Zylinderachse, θ Winkelkoordinate, r Radialkoordinate) beschrieben.

ID	Anzahl Blöcke	Zellen in x, θ, r-Richtung	Zellen pro Block	Gesamtanzahl Zellen
1	8	10 x 40 x 20	8.000	64.000
2	8	10 x 40 x 40	16.000	128.000
3	8	20 x 40 x 20	16.000	128.000
4	8	10 x 40 x 80	32.000	256.000
5	8	10 x 80 x 40	32.000	256.000
6	8	40 x 80 x 40	128.000	1.024.000

Tabelle 4.1: Eigenschaften der Testfälle.

Die Testfälle 2 und 5 sowie die Testfälle 3 und 4 haben jeweils die gleiche Anzahl von Zellen. Allerdings sind die Zellen in den Dimensionen der Blöcke unterschiedlich verteilt. Die unterschiedlichen Seitenverhältnisse führen zu einer anderen Reihenfolge der Rechenoperationen. Deshalb variiert auch die Anzahl der Iterationen des GMRES-Algorithmus (siehe 5.3). In Tabelle 4.2 sind die Anzahl der Fließkommazahlen pro Zelle, pro Block und der Speicherbedarf angegeben. Die Anzahl der Fließkommazahlen ergibt sich aus der Anzahl der Zellen, die zur Berechnung einer Zelle benötigt werden (siehe Kapitel 5.1). Im *2D Euler* Fall sind es 9 Zellen (siehe Abbildung 5.1). Im *3D Euler* Fall sind es 13 Zellen (siehe Abbildung 5.1). Im *2D Navier-Stokes* Fall sind es 13 Zellen (siehe Abbildung 5.3). Im *3D Navier-Stokes* Fall sind es 25 Zellen (siehe Abbildung 5.4). Die Testfälle sind *3D Euler* Fälle, deshalb müssen 13 Zellen in die Berechnung einbezogen werden. Es werden 5 konservative Variablen (Freiheitsgrade) verwendet (siehe Kapitel 5.1). Für die Berechnung dieser Variablen muss ein Gleichungssystem mit 5 Formeln aufgestellt werden, die jeweils 5 Variablen enthalten. Deshalb sind 25 Fließkommazahlen notwendig. Es wird mit komplexen Zahlen gerechnet, deshalb sind zwei Fließkommazahlen (Real- und Imaginärteil) notwendig. Dies ergibt 650 Fließkommazahlen pro Zelle. Die Anzahl der Fließkommazahlen pro Block bzw. für alle Blöcke können der Tabelle 4.2 entnommen werden.

Die Testfälle sind in *CFD General Notation System*-Dateien (CGNS) gespeichert. Das CGNS-Datenformat ist ein portables und erweiterbares Datenformat für CFD-Daten. Es beinhaltet die Beschreibung einer Strömungssimulation und deren Ergebnisse. Das CGNS-Projekt wurde 1994 von Boeing und der NASA gestartet. 1999 wurde die Kontrolle an das *CGNS Steering Committee* abgegeben. Heute ist es ein weit verbreitetes Format für den Datenaustausch im CFD-Bereich. Weitere Informationen

ID	Fließkommazahlen pro Zelle	Fließkommazahlen pro Block	Gesamtanzahl der Fließkomma-zahlen	Speicherbedarf
1	13 x 25 x 2 = 650	5,2 Mio	41,6 Mio	166,4 MB
2	13 x 25 x 2 = 650	10,4 Mio	83,2 Mio	332,8 MB
3	13 x 25 x 2 = 650	10,4 Mio	83,2 Mio	332,8 MB
4	13 x 25 x 2 = 650	20,8 Mio	166,4 Mio	1,3312 GB
5	13 x 25 x 2 = 650	20,8 Mio	166,4 Mio	1,3312 GB
6	13 x 25 x 2 = 650	83,2 Mio	665,6 Mio	2,6624 GB

Tabelle 4.2: Anzahl der Fließkommazahlen pro Testfall.

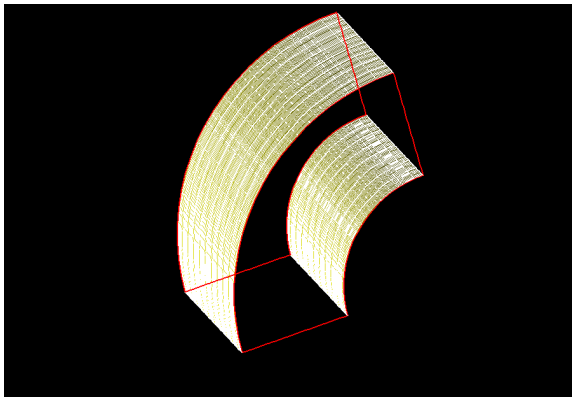

Abbildung 4.1: Darstellung des Testfalls.

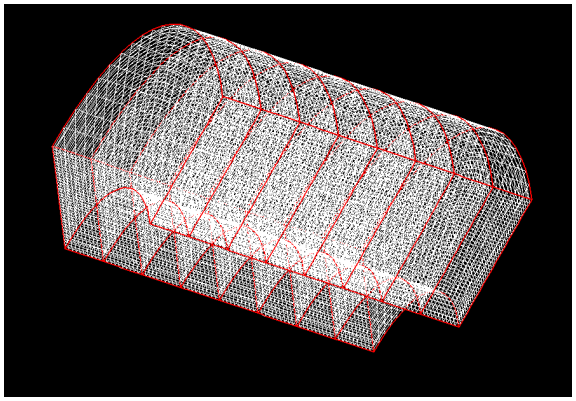

Abbildung 4.2: Das Rechengitter des Testfalls.

sind auf der Projekt-Homepage [CGNSS-2011] verfügbar.

4.7 Beschreibung des Testablaufs

Bevor die Tests gestartet werden, müssen Voraussetzungen erfüllt werden. Es muss gewährleistet werden, dass keine andere CPU und GPU intensive Prozesse zur Ausführung erhalten. Eine große I/O-Last durch andere Prozesse muss ausgeschlossen werden. Damit diese Punkte sichergestellt werden können, müssen Vorkehrungen getroffen werden. Zunächst werden die *Cronjobs* überprüft, ob performanzkritische Prozesse während der Tests gestartet werden. Die Netzwerkverbindung wird getrennt, damit keine Terminal-Verbindungen zum System durch andere Personen aufgebaut werden und diese performanzkritische Prozesse starten. Die Kompilierung der MPI-Variante und der GPGPU-Variante des TRACE-Codes erfolgt als Release-Version mit Optimierungsstufe 3. Die beiden TRACE-Varianten werden nacheinander jeweils mit den unterschiedlichen Testfällen gestartet. Die MPI-Variante wird mit dem folgenden Befehl gestartet: `mpirun -np 8 TRACE -cgns <CGNS-Datei> -cntl TRACE_control.input`. Mit `mpirun -np 8` werden 8 Instanzen von TRACE mittels MPI gestartet. Mit dem Parameter `-cgns <CGNS-Datei>` wird die Datei mit dem Testfall an TRACE übergeben. Der letzte Parameter `-cntl TRACE_control.input` übergibt eine Datei mit den Kontrollinformationen an TRACE. In den Kontrollinformationen ist die Abbruchbedingung, die maximale Anzahl der Iterationen und die Anzahl der Iterationen bis zum Neustart angegeben. Es werden 100 Iterationen ohne Neustarts durchgeführt. Die GPGPU-Variante wird mit dem folgenden Befehl gestartet: `TRACE -cgns <CGNS-Datei> -cntl TRACE_control.input`. Die GPGPU-Variante wird direkt gestartet, und es werden die gleichen Parameter übergeben. Die Tests werden 5-mal durchgeführt, damit eventuelle Ausreißer erkannt werden können. Der schnellste Durchlauf der 5 Messungen wird in den Ergebnissen der Performanzmessung festgehalten. Bei der MPI-Variante müsste der langsamste Block einer Messung verwendet werden, da nach der Berechnung eines Block eine Synchronisation zwischen den Knoten stattfindet. Allerdings werden bei 3 der 8 Blöcken abnormalen Laufzeiten gemessen (siehe Kapitel 6.11.2), deshalb muss der schnellste Block pro Messung in die Auswertung einfließen. Aus den 5 Messungen wird wieder der schnellste Durchlauf verwendet.

5 Analyse des TRACE-Codes

Der numerische Strömungslöser *Turbo machinery Research Aerodynamic Computational Environment* (TRACE) [Kersk-2010] wird zur Strömungssimulation in Flugzeugtriebwerken und stationären Gasturbinen zur Stromerzeugung eingesetzt. Im folgenden Teil werden einige Details zu den mathematischen Grundlagen, den Datenstrukturen und den verwendeten Algorithmus erläutert.

5.1 Grundlagen

Die mathematischen Grundlagen des TRACE-Codes basieren auf den *Navier-Stokes*-Gleichungen. Die *Navier-Stokes*-Gleichungen beschreiben die Eigenschaften eines strömenden Mediums (z.B. Wasser, Gas). Sie bilden ein System von nichtlinearen partiellen Differenzialgleichungen zweiter Ordnung. Der TRACE-Code verwendet die kompressible, instationäre *Reynolds-Averaged-Navier-Stokes*-Gleichungen (URANS).

$$\frac{\partial \mathbf{q}}{\partial t} + \nabla \cdot \mathbf{F}(\mathbf{q}) + \mathbf{S}(\mathbf{q}) = 0$$ URANS-Gleichung

$$\mathbf{q} = \begin{pmatrix} \rho \\ v_x \\ v_y \\ v_z \\ e \end{pmatrix}$$ Erhaltungsvektor, konservative Variablen (Dichte, Geschwindigkeit$_x$, Geschwindigkeit$_y$, Geschwindigkeit$_z$, innere Energie)

$\mathbf{F}$ Euler-Flüße und viskose Flüsse

$\mathbf{S}$ Quell-Terme, Coriolis- und Zentrifugalkraft

Die Navier-Stokes-Gleichungen können Strömungen mit den folgenden Eigenschaften berechnen (siehe [Haken-2011], Seite 21):

- reibungsbehaftete Strömungen,
- thermische Energieübertragung,
- instationäre Strömung (zeitlich Variabel),
- kompressible Strömung (Die Dichte des Mediums verändert sich durch äußere Einwirkungen z.B. Druck.),
- dreidimensionale Strömung.

Wird die Reibung und die thermische Energieübertragung bei den Navier-Stokes-Gleichungen vernachlässigt, erhält man die Euler-Gleichungen. Mit der

Finite-Volumen-Diskretisierung wird aus den Navier-Stokes-Gleichungen bzw. Euler-Gleichungen ein Raster mit einer endlichen Anzahl von Zellen (finite Volumen) erzeugt. Das Raster deckt das gesamte zu simulierende Gebiet ab. Bei der Berechnung einer Zelle müssen die Nachbarzellen berücksichtigt werden. Welche Nachbarzellen berücksichtigt werden müssen, hängt von den verwendeten Gleichungen ab (z.B. Navier-Stokes-Gleichungen oder Euler-Gleichungen). In Abbildung 5.1 sind die Zellen dargestellt, die bei der Berechnung der Zelle im Zentrum des Körpers unter Verwendung der Euler-Gleichungen im 2D-Fall benötigt werden. Bei den Euler-Gleichungen wird die N4-Nachbarschaft verwendet. Zusätzlich werden noch die Nachbarzellen dieser Nachbarn benötigt, die sich auf der Verlängerung der Achsen befinden. Im 3D Fall werden zusätzlich die 2 Nachbarn in jeder Richtung auf der zusätzlichen Achse benötigt. Der 3D-Fall unter Verwendung der Euler-Gleichungen ist in Abbildung 5.2 dargestellt.

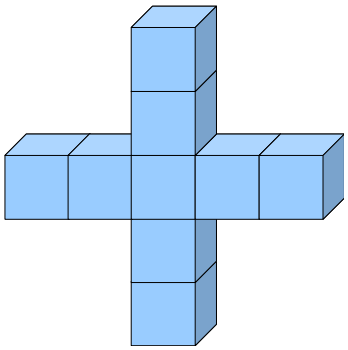

Abbildung 5.1: Euler 2D

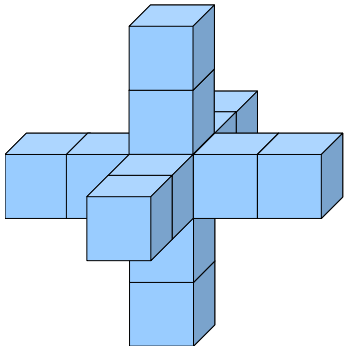

Abbildung 5.2: Euler 3D

In Abbildung 5.3 sind die Zellen dargestellt, die bei der Berechnung der Zelle im Zentrum des Körpers unter Verwendung der Navier-Stokes-Gleichungen im 2D-Fall benötigt werden. Zusätzlich zu den Zellen aus dem Euler-Fall in 2D werden die bisher noch nicht verwendeten Zellen aus der N8-Nachbarschaft benötigt. Diese Nachbarschaft wird auch *Manhattan-Nachbarschaft* genannt unter der Randbedingung, dass der maximale Abstand zur berechnenden Zelle auf 2 festgelegt ist. In Abbildung 5.4 ist der 3D-Fall unter Verwendung der Navier-Stokes-Gleichungen dargestellt. Im 3D-Fall wird die *Manhattan-Nachbarschaft* auf alle drei Achsen angewendet. Diese Zellen werden alle für die Berechnung der Zelle im Zentrum benötigt. Die zusätzlichen Zellen bei den Navier-Stokes-Gleichungen sind notwendig, damit die Reibung berücksichtigt werden kann.

In dieser Arbeit wird das lineare Modul des TRACE-Codes untersucht. Das zugrunde liegende Problem basiert auf einem nicht-linearen Gleichungssystem. Allerdings kann zur Approximierung der Lösung des nicht-linearen Gleichungssystems ein lineares Gleichungssystem verwendet werden. Die Abweichung zwischen den Ergebnissen der nicht-linearen Berechnung und der linearen Berechnung sind minimal, wenn

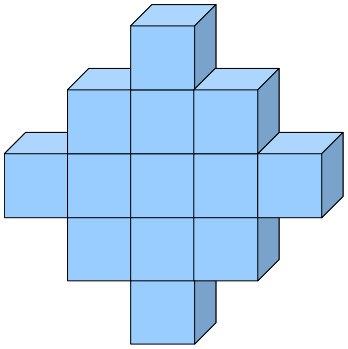

Abbildung 5.3: Navier-Stokes 2D

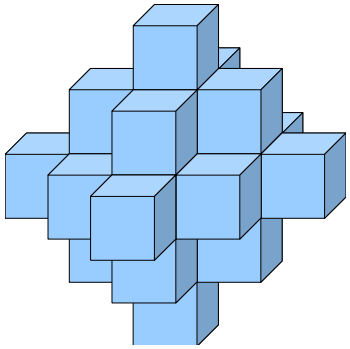

Abbildung 5.4: Navier-Stokes 3D

die Störungen hinreichend klein sind. In Abbildung 5.5 sind die Ergebnisse der Berechnung der Wirkung einer harmonischen Störung der Grundlösung am Eintritt des Rechengebiets dargestellt. Die Daten der rechten Abbildung wurde mit dem linearen Modul und die Daten der linken Abbildung wurde mit einem anderen Modul des TRACE-Codes berechnet. Unterschiede zwischen den beiden Lösungen sind kaum zu erkennen.

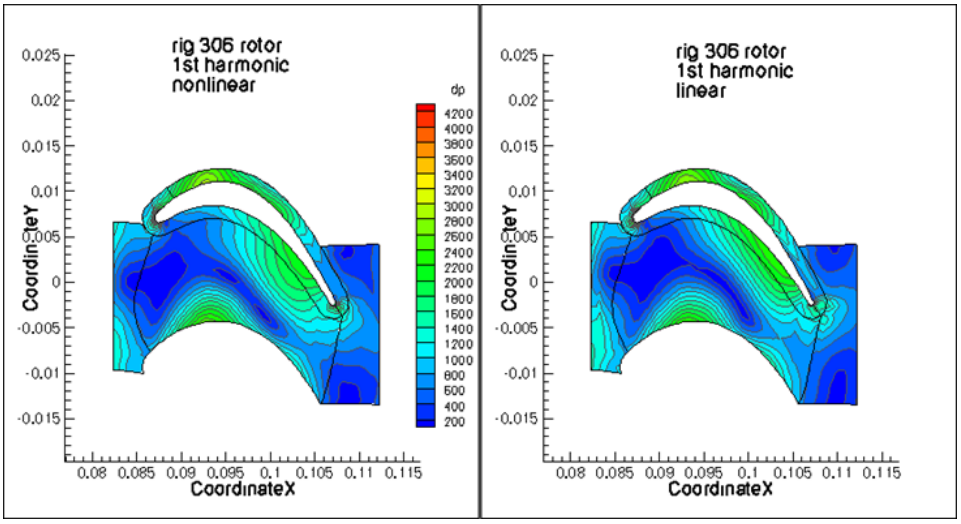

Abbildung 5.5: Vergleich einer nicht-linearen und einer linearen Lösung [Kersk-2011].

5.2 Datenstruktur

Die Strömungssimulation soll für ein bestimmtes Rechengebiet das Strömungsfeld berechnen. Dieses Rechengebiet enthält meist einen Körper, der umströmt werden muss. Dieser Körper beeinflusst das Strömungsfeld. Die Eigenschaften des Strömungsfeldes sollen simuliert werden. Für die Berechnung muss das Rechengebiet in nicht überlappende Zellen aufgeteilt werden, dies wird auch Gittergenerierung genannt. Bei der Gittergenerierung können zwei Varianten von Gittern erzeugt werden,

die strukturierten Gitter und die unstrukturierten Gitter. In Abbildung 5.6 ist ein strukturiertes Gitter und in Abbildung 5.7 ist ein unstrukturiertes Gitter entlang eines Flügelprofiles abgebildet (vgl. [Hicke-2011]).

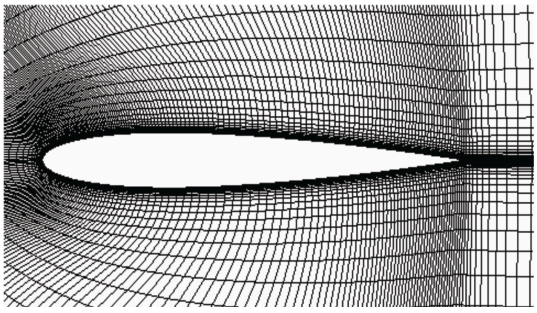

Abbildung 5.6: Strukturiertes Gitter (siehe [Hicke-2011], Seite 12)

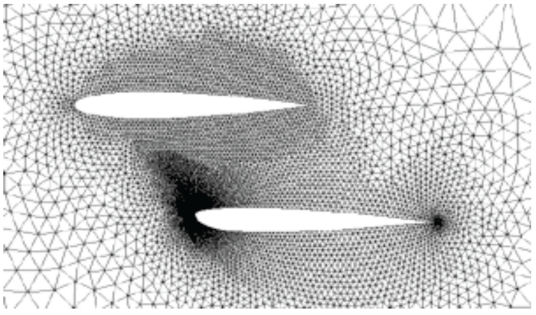

Abbildung 5.7: Unstrukturiertes Gitter (siehe [Hicke-2011], Seite 12)

Die wesentlichen Eigenschaften von strukturierten und unstrukturierten Gittern sind in Tabelle 5.1 festgehalten. Der lineare Gleichungslöser des TRACE-Codes unterstützt ausschließlich strukturierte Gitter.

	strukturierte Gitter	unstrukturierte Gitter
Erzeugung 2D	meist Vierecke	meist Dreiecke
Erzeugung 3D	meist Quader	Tetraeder, Hexaeder, Prismen
Datenstruktur	einfache Datenstruktur (z.B. Matrizen)	komplexe Datenstruktur; zusätzlich zu den Daten müssen die Nachbarschaftsbeziehungen gespeichert werden.
Vorteile	- einfache Topologie und schnelle Implementierung - Gebiete werden auf kartesische Gitter abgebildet - effiziente Algorithmen anwendbar	- komplexe Geometrien können sehr leicht modelliert werden - einfache Gitterverfeinerung durch lokales Hinzufügen von Punkten im Bereich starker Gradienten
Nachteile	- Gitterverfeinerung aufwendig	- ständiges Zugreifen auf die Nachbarschaftsbeziehungen - höhere Rechenzeiten

Tabelle 5.1: Vergleich strukturierter und unstrukturierter Gitter (vgl. [Hicke-2011]).

Durch die Diskretisierung des linearisierten Gleichungssystems entsteht eine dünnbesetzte Matrix. Die Matrix muss in einer geeigneten Form gespeichert und möglichst effizient für den Algorithmus bereitgestellt werden. Im TRACE-Code werden die Daten im *Block Compressed Row Storage*-Datenformat (BCSR) gespeichert. Es werden 5 x 5 Blöcke verwendet. Der Aufbau des BCSR-Datenformats wurde in Kapitel 2.3 erläutert. Das Rechengitter wird für die Berechnung in mehrere Blöcke unterteilt. Für

jeden Block wird eine Matrix im BCSR-Datenformat gespeichert. Die Blöcke werden während der Berechnung auf die Knoten des Clusters aufgeteilt und von den Knoten berechnet.

Das Gitter jeden Blocks wird zusätzlich um Geisterzellen erweitert. Die äußeren Geisterzellen umranden das gesamte Rechengitter und werden verwendet, um die Randbedingungen der Strömungsgleichung einzuhalten. Randbedingungen können z.B. Wände oder einfließende Strömungen sein. Die inneren Geisterzellen werden für die Kommunikation zwischen den Blöcken benötigt und werden an den Rändern zu anderen Blöcken erzeugt. In Abbildung 5.8 werden die Geisterzellen verdeutlicht. Die linke Grafik zeigt das Rechengitter (Blau), dass von den äußeren Geisterzellen (Grau) umrahmt wird. Das Gitter ist in vierundzwanzig 3x3 Blöcke unterteilt. Die rechte Grafik zeigt einen Block, der von den inneren Geisterzellen umrahmt ist. An allen Seiten des Blocks befinden sich innere Geisterzellen, da der Block im Inneren des Gitters liegt. Die Geisterzellen umranden das Gitter bzw. den Block immer mit einer Breite von zwei Zellen, damit alle Zellen für die in den Abbildungen 5.1 bis 5.4 beschriebenen Nachbarzellen zur Verfügung stehen.

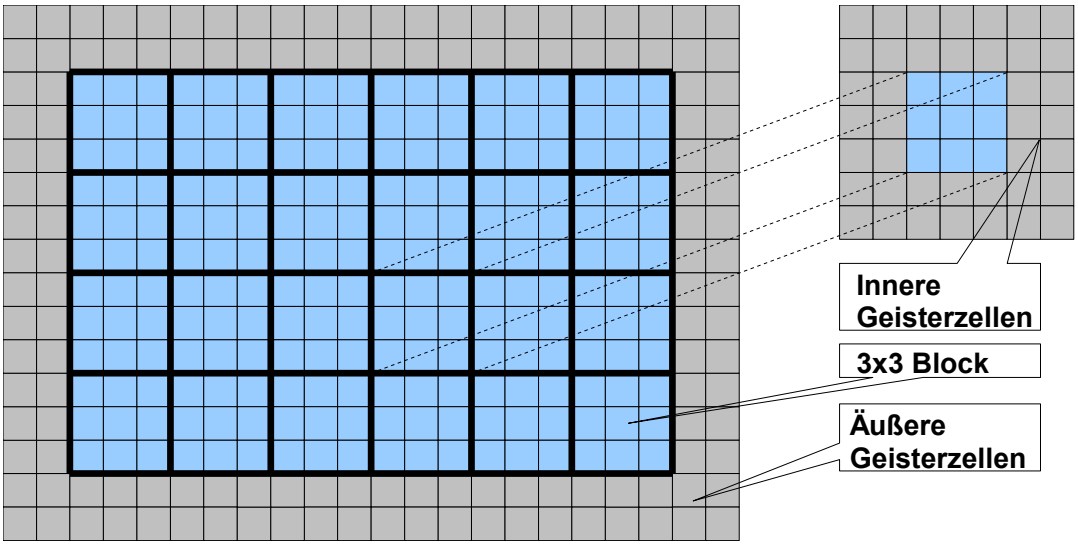

Abbildung 5.8: Schaubild zu den Geisterzellen.

5.3 Algorithmus

Der lineare-Gleichungslöser des TRACE-Codes arbeitet mit der *Generalized Minimum Residual*-Methode (GMRES). GMRES gehört zu den *Krylow-Unterraum-Verfahren*. Dies sind iterative Algorithmen zum Lösen von großen linearen Gleichungssystemen. Ein geeigneter Vorkonditionierer beschleunigt den Algorithmus erheblich. Der Vorkonditionierer formt das zugrunde liegende mathematische Problem um. Da-

bei bleibt die Lösung erhalten aber der Algorithmus kann schneller die Lösung berechnen. Der TRACE-Code verwendet folgende Vorkonditionierungsverfahren:

- Symmetric Successive Over-Relaxation (SSOR) (siehe [SaadY-2003], Seite 284 - 287),
- Incomplete LU-Decomposition (ILU) (siehe [SaadY-2003], Seite 288 - 293),
- Modified ILU (MILU) (siehe [SaadY-2003], Seite 305 - 306),
- ILU with Level of Fill (ILU(p)) (siehe [SaadY-2003], Seite 296 - 300),
- ILU with Threshold (ILUT) (siehe [SaadY-2003], Seite 306 - 312),
- Jacobi (siehe [SaadY-2003], Seite 284 - 287).

In Algorithmus 1 ist der GMRES-Algorithmus in Pseudocode dargestellt. In Zeile 1 wird die Startnäherung und der dazugehörige Vektor berechnet. Von Zeile 2 bis Zeile 13 werden die Iterationen zur Lösung des Gleichungssystems durchgeführt. In Zeile 3 wird eine Matrix-Vektor-Multiplikation mit der Matrix A und dem Basisvektor v berechnet. Von Zeile 4 bis Zeile 7 wird eine Orthogonalisierung nach dem modifizierten *Gram-Schmidt-Verfahren* durchgeführt. Für diese Methode werden Skalarprodukte berechnet. Die Anzahl der Skalarprodukte erhöht sich pro Iterartionsschritt um ein Skalarprodukt. Dieser Abschnitt wird als *Krylov Unterraum* bezeichnet. In Zeile 8 wird die zweite euklidische Norm des Residuums berechnet. Von Zeile 9 bis Zeile 11 wird die Abbruchbedingung überprüft und ggf. die Iteration abgebrochen. Falls das Residuum kleiner als die gewünschte Genauigkeit ist, wird er Algorithmus abgebrochen. Im Pseudocode wird der Algorithmus erst bei der exakten Lösung abgebrochen. Bei realen Problemen wird die Iteration bei einer Näherungslösung mit ausreichender Genauigkeit beendet. In Zeile 12 wird der nächste Basisvektor für die Matrix-Vektor-Multiplikation des nächsten Iterationsschritts berechnet. Die Basisvektoren müssen während des gesamten Algorithmus gespeichert werden, da diese in jedem Iterationsschritt wieder benötigt werden. In Zeile 14 wird die *Methode der kleinsten Quadrate* über die *Hessenberg Matrix* definiert. In Zeile 15 wird die Lösung bzw. die Näherungslösung des gesuchten Gleichungssystems berechnet.

Durch die ansteigende zahl der Operationen pro Iteration wird die Berechnung immer aufwendiger. Der Speicherbedarf steigt ebenfalls mit jeder Iteration. Deshalb gibt es den Ansatz des *Neustarts* beim GMRES-Algorithmus. Beim *Neustart* werden die bisher berechneten Basisvektoren verworfen, und der GMRES-Algorithmus wird mit der letzten Näherungslösung neu gestartet.

Algorithmus 1 Pseudocode des GMRES-Algorithmus (siehe [SaadY-2003], Seite 172).

1: Berechne $r_0 = b - Ax_0$, $\beta := ||r_0||_2$, und $v_1 := r_0/\beta$
2: **for** j = 1, 2, ..., m **do**
3: Berechne $w_j := Av_j$
4: **for** i = 1, 2, ..., j **do**
5: $h_{ij} := (w_j, v_i)$
6: $w_j := w_j - h_{ij}v_i$
7: **end for**
8: $h_{j+1,j} = ||w_j||_2$.
9: **if** $h_{j+1,j} = 0$ **then**
10: setze $m := j$ und weiter mit Zeile 14
11: **end if**
12: $v_{j+1} = w_j/h_{j+1,j}$
13: **end for**
14: Definiere $(m+1) \times m$ für die Hessenberg Matrix $\bar{H}_m = h_{ij1<i<m+1,1<j<m}$.
15: Berechne y_m durch Minimierung von $||\beta e_1 - \bar{H}_m y||_2$ und $x_m = x_0 + V_m y_m$.

6 Implementierung

Der lineare Gleichungslöser des TRACE-Codes arbeitet mit dem GMRES-Algorithmus. Eine Beschleunigung des TRACE-Codes kann durch die Parallelisierung der Rechenoperationen des GMRES-Algorithmus erreicht werden. Die interessanten Rechenoperationen sind die dünnbesetzte Matrix-Vektor-Multiplikation (SpMV) und die Skalarmultiplikationen des Unterraums. Alle Kernel für die SpMV müssen in drei Varianten implementiert werden: eine Variante für eine SpMV mit einer komplexen Matrix und einen komplexen Vektor, eine weitere Variante für eine SpMV mit einer reellen Matrix und einem komplexen Vektor. Eine dritte Variante für eine SpMV mit einer reellen Matrix und einem reellen Vektor. Die Variante einer SpMV mit einer komplexen Matrix und einem reellen Vektor wird von TRACE nicht benötigt.

6.1 Vorbereitung

Zur Vorbereitung müssen verschieden Bibliotheken installiert und Umgebungsvariablen eingerichtet werden. Es muss eine MPI-Bibliothek installiert werden, entweder MPICH2 oder OpenMPI. Die Umgebungsvariable `$MPI_HOME` muss auf das Basisverzeichnis der MPI Installation verweisen. Dieses Verzeichnis muss den *lib* und *include* Ordner von MPI enthalten. Die CGNS-Bibliothek muss in der Version 2.5.x installiert werden. Die Umgebungsvariable `$CGNS_HOME` muss auf das Basisverzeichnis der CGNS Installation verweisen. Dieses Verzeichnis muss den *lib* und *include* Ordner von CGNS enthalten. Die OpenCL-Bibliothek muss installiert werden. Diese ist im *NVIDIA CUDA Toolkit* oder im *AMD Stream SDK* enthalten. Die Umgebungsvariable `$OCL_INCLUDE` muss auf den *include* Ordner der OpenCL-Bibliothek verweisen. Diese 3 Umgebungsvariablen sind für das Makefile notwendig, damit die benötigten *includes* und Bibliotheken während des Kompilierens bzw. Linkens gefunden werden. Die Bibliotheken von MPI, OpenCL und CGNS müssen vom Betriebssystem geladen werden, damit TRACE ausgeführt werden kann. Zusätzlich muss die Umgebungsvariable `$TRACE_HOME` eingerichtet werden. Diese muss auf das Basisverzeichnis des TRACE-Quellcodes verweisen. Die OpenCL-Kernel werden erst zur Laufzeit kompiliert und benötigen einige *Header* Dateien von TRACE. Falls nur ein Binary von TRACE ausgeliefert wird, müssen folgende Dateien in den angegebenen Ordnern vorhanden sein.

- *$TRACE_HOME*/INCLUDE/opencl.h
- *$TRACE_HOME*/traceControl.h
- *$TRACE_HOME*/OPENCL/lalgebra.cl

Damit die unterschiedlichen Kernel verwendet werden können, müssen in der *Header-Datei ocltype.h* die benötigten *Defines* aktiviert bzw. deaktiviert werden. Der *Define* `#define USE_OPENCL ON` muss bei allen OpenCL-Kernel aktiviert werden. Andernfalls wird die CPU-Implementierung der SpMV verwendet. Die *Defines* steuern die Präprozessor-Anweisungen, die den benötigten Quellcode für einen Kernel aktivieren. Welche *Defines* aktiviert werden ist in den Implementierungsdetails des jeweiligen Kernels beschrieben. Alle nicht genannten *Defines* müssen deaktiviert werden.

6.2 OpenCL Umgebung

Die Ausführung von OpenCL-Kerneln benötigt eine OpenCL-Umgebung. Diese Umgebung besteht aus einem *cl_context*, einem *cl_program*, einer *cl_command_queue* und mehreren *cl_kernel*. Dem *cl_context* muss mindestens ein *OpenCL-Device* zugeordnet werden. Einem *OpenCL-Device* wird eine *cl_command_queue* zugeordnet. Alle Kernel, die von einem *OpenCL-Device* ausgeführt werden sollen, werden an die *cl_command_queue* des *OpenCL-Device* übergeben. Die *cl_command_queue* sorgt für eine optimale Ausführung der Kernel. Das *cl_program* besteht aus den kompilierten Kerneln. Die OpenCL-Kernel werden erst nach dem Programmstart kompiliert, dazu wird die Text-Datei mit den Kerneln eingelesen und auf die GPU kopiert. Auf der GPU werden die Kernel kompiliert und gespeichert. Alle Speicheradressen der *cl_kernel* müssen aus dem *cl_program* ausgelesen werden, damit diese vom *Host-Code* aufgerufen werden kann. Die Speicheradressen der Kernel werden in einem Array gespeichert das im *Struct OpenCLEnvironment* gespeichert ist. Für die Adressierung der Kernel wurde ein *enum* mit dem Namen *OpenCLKernel* angelegt. Dieses *enum* mapped den Kernelnamen auf einen Integer-Wert. Dieser Wert kann als Index für das Array mit den Kerneln verwendet werden. Der *cl_context*, das *cl_program* und die *cl_command_queue* werden auch im dem *Struct OpenCLEnvironment* gespeichert. Das *Struct OpenCLEnvironment* wird in dem zentralen *Struct SolverCntl* gespeichert, dies ist im gesamten TRACE-Code verfügbar.

6.3 Dünnbesetzte Matrix-Vektor-Multiplikation mit BCSR-Datensatz

Im ersten Ansatz wird in OpenCL die dünnbesetzte Matrix-Vektor-Multiplikation mit dem BCSR-Datenformat implementiert. Das BCSR-Datenformat ist das Standard-Datenformat für dünnbesetzte Matrizen im TRACE-Code. Mit den Daten, die auf die GPU transferiert wurden, kann eine große Anzahl von Rechenoperationen auf der GPU ausgeführt werden. Dies sollte zu einer guten Auslastung der GPU führen.

Es wird eine Matrix-Vektor-Multiplikation implementiert, die generisch im gesamten TRACE-Code eingesetzt werden kann.

6.3.1 Aufbereitung der BCSR Daten

Im TRACE-Code muss das BCSR-Datenformat in einer modifizierten Variante verwendet werden. Im TRACE-Code werden Geisterzellen für die Randbedingungen und zur Kommunikation benötigt (siehe 5.2). Diese Geisterzellen werden in der Matrix nicht gespeichert, allerdings beeinflussen diese die Indizes der Matrix. Eine Menge Blockzeilen und Blockspalten sind nicht in der Matrix gespeichert. Die fehlenden Spalten sind nicht problematisch, da es den Blockspaltenindex gibt, der die Blockspalte für einen konkreten Block speichert. Für die Zeilen ist dies ein Problem. Im BSCR-Format wird ein Index erzeugt, der für jede Zeile festhält, an welcher Stelle im Blockspaltenindex eine Blockzeile beginnt. Da einige Blockzeilen fehlen müssen Platzhalter im Index gespeichert werden, da sonst die richtige Zeilennummer verloren geht. In der MPI-Variante werden die Platzhalter mittels *while-Schleife* übersprungen. In der OpenCL-Variante sollte aus Performanzgründen keine *while-Schleife* eingesetzt werden. Die *while-Schleife* erzeugt unterschiedliche Ausführungspfade innerhalb eines *Warps*. Diese müssten auf der GPU serialisiert werden und der Performanzvorteil der GPU würde verloren gehen. Deshalb ist es sinnvoller, die Daten vorher aufzubereiten.

Bei der Aufbereitung der Daten werden die Platzhalter aus dem Index für den Anfang einer Blockzeile entfernt. Allerdings gehen die Zeilennummern der Blockzeilen verloren. Die Berechnung und Speicherung der Zeilennummern in einem zusätzlichen Index wird notwendig. Diese Implementierung erspart die Verwendung der *while-Schleife* zum Überspringen der Platzhalter und verhindert die Serialisierung der Kernel. Der zusätzliche Index erzeugt zusätzlichen Datentransfer zur GPU und zusätzlichen Platzbedarf auf der GPU. Während der Ausführung des Kernels muss zusätzlich ein weiterer Index aus dem Speicher der GPU geladen werden. Dies beeinflusst die Performanz negativ. Die Aufbereitung der BCSR-Daten muss für alle OpenCL-Kernel durchgeführt werden, die das BSCR-Datenformat verwenden.

6.3.2 Implementierungsdetails

Die folgende Implementierung wird im weiteren Text und den Diagrammen mit *BCSR V1a* bezeichnet. Es werden zwei Kernel für diese Implementierung benötigt, ein Multiplikationskernel und ein Reduktionskernel. Für die Implementierung muss der *Define* `#define USE_BCSR_2a ON` aktiviert werden.

Die *work-items* für den Multiplikationskernel werden in *work-groups* zu 256 *work-items* mit einer Ausdehnung von 16 mal 16 *work-items* zusammengefasst. Die ge-

samte Anzahl der *work-items* wird durch die Anzahl der BCSR-Blöcke bestimmt und bis zur Größe der nächsten *work-group* erhöht. Die Anzahl der *work-items* einer *work-group* sollte immer ein Vielfaches von 32 sein, mindestens aber 64 *work-items*; besser sind 128 oder 256 *work-items* (siehe [NVIDI-2011b], Seite 36). Eine Optimale Aufteilung der *work-items* für die konkrete Matrix zu finden ist nur mit sehr hohem Aufwand möglich. Diese Problem gehört zum allgemeinen Partitionierungsproblem. Das allgemeinen Partitionierungsproblem gehört zu der Klasse der NP-vollständigen Probleme. Die Partitionierung der Daten für den Multiplikationskernel der Implementierung *BCSR V1a* ist in Abbildung 6.1 dargestellt. Im oberen Teil der Grafik ist die Partitionierung der Daten dargestellt. Im unteren Teil der Grafik ist die Repräsentation der Daten im Speicher abgebildet. Ein BCSR-Block wird von einem *work-item* abgearbeitet.

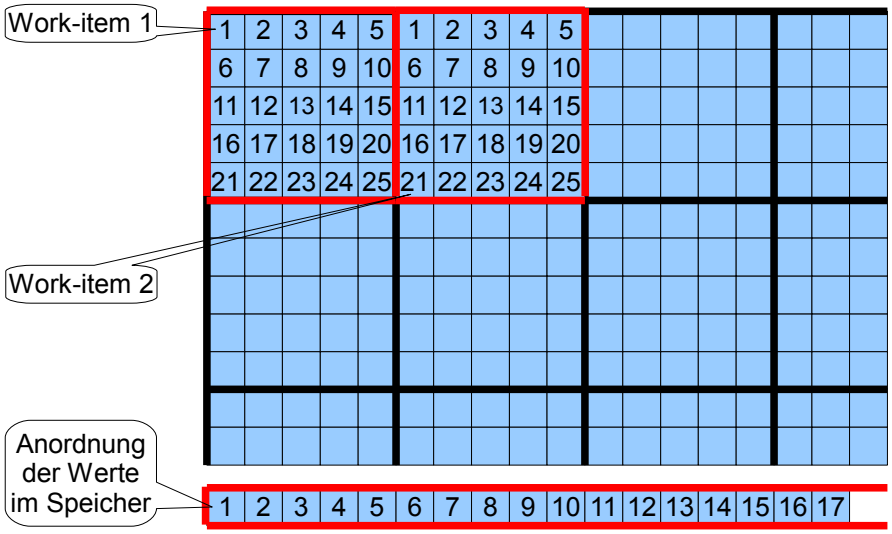

Abbildung 6.1: Partitionierung der Daten für die Multiplikation der Kernel BCSR V1a und V1b.

Der Multiplikationskernel benötigt die Einträge der Matrix, die Einträge des Vektors, den Blockspaltenindex, den temporären Zwischenspeicher und die Anzahl der BCSR-Blöcke als Übergabeparameter. Die Einträge der Matrix, die Einträge des Vektors, der Blockspaltenindex und der temporäre Zwischenspeicher muss im Host-Code des Kernels in den *global memory* der GPU geladen werden. Der Ablauf der Multiplikation ist in Abbildung 6.2 dargestellt. Die *work-item ID* bestimmt den zu bearbeitenden BCSR-Block. Der Index für den Eintrag des Vektors für die Multiplikation muss aus dem Blockspaltenindex mit der *work-item ID* ausgelesen werden. Eine *For-Schleife* iteriert über die Spalten eins bis vier des BCSR-Blocks. Im Schleifen-Körper ist die Multiplikation für die fünf Zeilen des BCSR-Blocks implementiert. Vor der *For-Schleife* ist die Multiplikation der Spalte Null für alle fünf Zeilen implementiert. Dies ist notwendig, damit der Temporäre Zwischenspeicher im *global memory* der GPU nicht vor dem Start

mit Nullen initialisiert werden muss. Die Aufsummierung der Ergebnisse der Multiplikationen einer Zeile des BCSR-Blocks erfolgt im temporäre Zwischenspeicher. Der Eintrag im temporären Zwischenspeicher wird mit der *work-item ID* und der Zeilennummer der Zeile des BCSR-Blocks adressiert. Pro *work-item* werden fünf Ergebnisse im temporären Zwischenspeicher gespeichert. Die Anzahl der Blöcke dienen als Begrenzung für die Ausführung des Kernel, damit nicht mehr Instanzen des Kernels ausgeführt werden als benötigt. Dies würde zu Speicherzugriffsfehlern führen, da Daten aus nicht alloziertem Speicher gelesen würden. In einer *If-Abfrage* wird überprüft, ob die *work-item ID* kleiner der Anzahl der benötigten *work-items* ist. Diese *If-Abfrage* führt in der letzten *work-group* zur serialisierten Ausführung von *work-items*. Alle benötigten *work-items* können nicht gleichzeitig mit den nicht benötigten *work-items* ausgeführt werden, da diese unterschiedliche Operationen ausführen. Dies beiden Gruppen von *work-items* werden nacheinander ausgeführt. Die *work-items* innerhalb einer Gruppe können parallel ausgeführt werden. Allerdings sollte die Serialisierung nicht viel Performanz kosten, da die nicht benötigten Kernel keine Berechnungen ausführen. Die Werte einer Zeile der Matrix im temporären Zwischenspeicher werden durch einen Reduktionskernel aufsummiert und im Ergebnisvektor gespeichert.

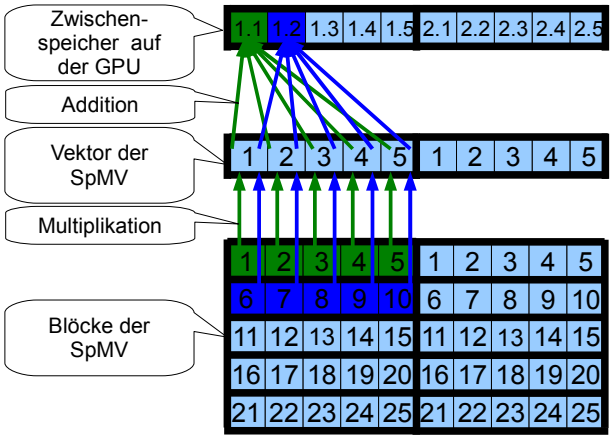

Abbildung 6.2: Ablauf der Multiplikation für die Kernel BCSR V1a bis V2b.

Für den Reduktionskernel werden 256 *work-items* zu einer *work-group* mit einer Ausdehnung von 1 mal 256 *work-items* zusammengefasst. Die Anzahl der *work-items* wird durch die Anzahl der Blockzeilen mal die Anzahl der Zeilen pro BCSR-Block ermittelt. In Abbildung 6.3 ist die Partitionierung der Daten für den Reduktionskernel dargestellt. Die grünen Felder sollen die Zwischensummen im temporären Zwischenspeicher darstellen. Pro Zeile der Matrix wird ein *work-item* benötigt.

Der Reduktionskernel benötigt die Einträge des temporären Zwischenspeichers, die Einträge des Ergebnisvektors, den Index für den Anfang einer Zeile im Blockspaltenindex, den Blockzeilenindex und die Größe des Index für den Anfang einer Zeile im Blockspaltenindex als Übergabeparameter. Die Einträge des temporären Zwi-

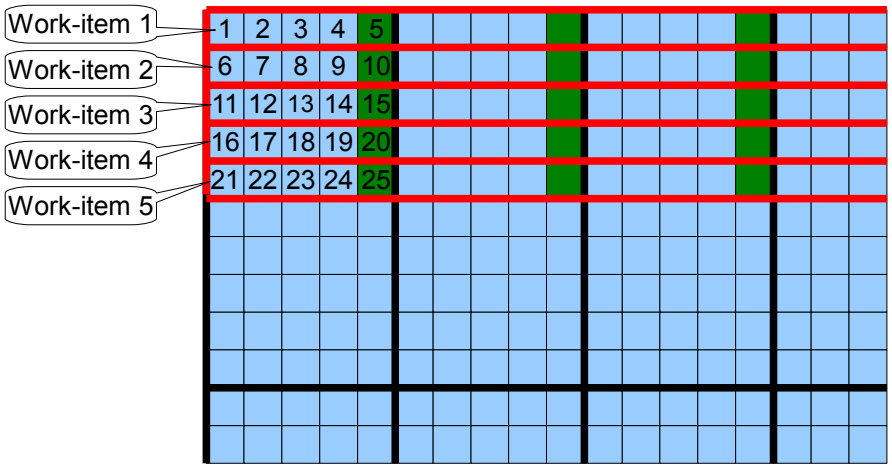

Abbildung 6.3: Partitionierung der Daten für den Reduktionskernel der Kernel BCSR V1a bis V3.

schenspeichers, der Index für den Anfang einer Zeile im Blockspaltenindex und der Blockzeilenindex muss im Host-Code des Kernels in den *global memory* der GPU geladen werden. Der benötigt Speicher für die Einträge des Ergebnisvektors werden im Host-Code im *global memory* der GPU alloziert und mit einem Kernel werden alle Einträge auf Null gesetzt. Der Ablauf des Reduktion ist in Abbildung 6.4 dargestellt. Die *work-item ID* bestimmt die zu bearbeitende Zeile der Matrix. Mit einer *For-Schleife* werden die Zwischensummen einer Zeile, aus dem temporären Zwischenspeicher zum Ergebnis aufsummiert. Welche Werte für eine Zeile aufsummiert werden müssen, kann mit dem Index für den Anfang einer Zeile im Blockspaltenindex bestimmt werden. Mit diesem Index kann der erste Block und der letzte Block einer Blockzeile bestimmt werden. Diese Information kann auch zur Adressierung der Daten im temporären Zwischenspeicher genutzt werden. Der Index zum Speichern des Ergebnisses im Ergebnisvektor wird aus dem Blockzeilenindex ausgelesen. Die Größe des Index für den Anfang einer Zeile im Blockspaltenindex minus 1 und mal der Anzahl der Zeilen pro BCSR-Block dienen als Begrenzung für die Ausführung des Kernel, damit nicht mehr Instanzen des Kernels ausgeführt werden als benötigt. Dies würde zu Speicherzugriffsfehlern führen, da Daten aus nicht alloziertem Speicher gelesen werden würde. Der Ergbnisvektor wird nach der Berechnung der SpMV vom Host-Code aus dem *global memory* GPU ausgelesen und im Speicher des Hosts gespeichert.

6.3.3 Resultat

Die Ergebnisse in Tabelle 6.1 zeigen, dass diese einfache Implementierung der SpMV nicht sehr performant ist. Diese Implementierung ist bis zu 36 mal langsamer als die MPI-Variante. Die Fermi-Hardware kann ihre Vorteile bei dieser Implementierung nicht umsetzen. Der Cache und die größere Anzahl an Streaming-Prozessorkernen

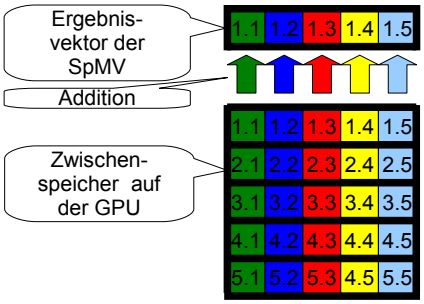

Abbildung 6.4: Ablauf des Reduktion für die Kernel BCSR V1a bis V3.

können bei diesem Kernel die Performanz nicht steigern. Die großen Datenmengen, die auf die GPU transferiert werden, verhindern eine Beschleunigung durch die bessere Hardware.

	10x40x20	10x40x40	20x40x20	10x40x80	10x80x40	40x80x40
Tesla	21,45	60,07	60,01	110,59	110,62	439,49
Fermi	24,1	59,77	59,67	118,28	119,07	470,95
MPI	0,85	1,67	1,67	3,31	3,31	13,23

Tabelle 6.1: Die Laufzeiten der Testfälle mit dem BCSR-Kernel V1a in Sekunden.

Die Untersuchung dieses Kernel mit dem *Compute Visual Profiler* zeigt, dass der Multiplikationskernel 18 Register benötigt. Allerdings stehen auf der Tesla-Architektur pro *work-item* nur 16 Register zur Verfügung. Die Analyse mit dem *CUDA GPU Occupancy Calculator* zeigt, dass die GPU bei der Verwendung von 18 Registern theoretisch nur zu 75% ausgelastet werden kann. Auf der Fermi-Architektur stehen pro *work-item* 20 Register zur Verfügung; deshalb kann die GPU theoretisch zu 100% ausgelastet werden. Der Reduktionskernel benötigt 7 Register und kann die GPU theoretisch zu 100% auslasten. Die Anzahl der verwendeten Register im Multiplikationskernel sollte in einem nächsten Schritt reduziert werden, damit eine bessere Auslastung der GPU möglich wird.

6.4 Verbesserte SpMV mit BCSR-Datensatz

Für die Abarbeitung eines Blocks sollen mehrere *work-items* verwendet werden. Die Verkleinerung der Arbeitspakete wird eine Beschleunigung der SpMV und eine bessere Auslastung der SpMV bewirken. Die Komplexität der Kernel soll verringert werden und damit auch die Anzahl der verwendeten Register.

6.4.1 Implementierungsdetails

Die folgende Implementierung wird im weiteren Text und den Diagrammen mit *BCSR V2a* bezeichnet. Es werden zwei Kernel für diese Implementierung benötigt, ein Multiplikationskernel und ein Reduktionskernel. Für die Implementierung muss der *Define* `#define USE_BCSR_2a ON` aktiviert werden.

Die *work-items* für den Multiplikationskernel werden in *work-groups* zu 256 *work-items* mit einer Ausdehnung von 16 mal 16 *work-items* zusammengefasst. Die gesamte Anzahl der *work-items* wird durch die Anzahl der BCSR-Blöcke multipliziert mit der Anzahl der Zeilen eines BCSR-Block bestimmt. Die Partitionierung der Daten für den Multiplikationskernel der Implementierung *BCSR V2a* ist in Abbildung 6.5 dargestellt. Im oberen Teil der Grafik ist die Partitionierung der Daten dargestellt. Im unteren Teil der Grafik ist die Repräsentation der Daten im Speicher abgebildet. Der Multiplikationskernel benötigt die Einträge der Matrix, die Einträge des Vektors, den Blockspaltenindex, den temporären Zwischenspeicher und die Anzahl der BCSR-Blöcke als Übergabeparameter. Die Einträge der Matrix, die Einträge des Vektors, der Blockspaltenindex und der temporäre Zwischenspeicher müssen im Host-Code des Kernels in den *global memory* der GPU geladen werden. In dieser Implementierung wird ein BCSR-Block durch fünf *work-items* bearbeitet. In der Implementierung *BCSR V1a* war es nur ein *work-item* pro Block. Jede Zeile eines BCSR-Blocks wird durch ein *work-item* verarbeitet. Deshalb muss die Verarbeitung von fünf Zeilen auf eine Zeile im Kernel-Code reduziert werden. Die Adressierung der Werte der Matrix muss ebenfalls modifiziert werden. Der Startpunkt für die Multiplikation muss vom Anfang eines Blocks auf den Anfang einer Zeile innerhalb eines Blocks umgestellt werden. Die Aufsummierung der Ergebnisse der Multiplikationen wird in einem Register gespeichert. Nach dem eine ganze BCSR-Blockzeile berechnet ist, wird der Inhalt des Registers in den temporären Zwischenspeicher im *global memory* der GPU geschrieben. Die Anzahl der Blöcke mal die Anzahl der Zeilen eines BCSR-Blocks dienen als Begrenzung für die Ausführung des Kernels, damit nicht mehr Instanzen des Kernels ausgeführt werden als benötigt. Dies würde zu Speicherzugriffsfehlern führen, da Daten aus nicht alloziertem Speicher gelesen werden würden. Der Ablauf der Multiplikation ist identisch mit dem Ablauf der Multiplikation des Kernels *BCSR 1a*. Der Ablauf ist in Abbildung 6.2 dargestellt.

Für den Reduktionskernel werden 256 *work-items* zu einer *work-group* mit einer Ausdehnung von 1 mal 256 *work-items* zusammengefasst. Die Anzahl der *work-items* wird durch die Anzahl der Blockzeilen mal die Anzahl der Zeilen pro BCSR-Block ermittelt. Die Partitionierung der Daten für den Reduktionskernel der Implementierung *BCSR V2a* ist identisch zur Aufteilung der Daten für die Implementierung des Reduktionskernels in Kapitel 6.3.2. Die Partitionierung ist in Abbildung 6.3 dargestellt. Der Ab-

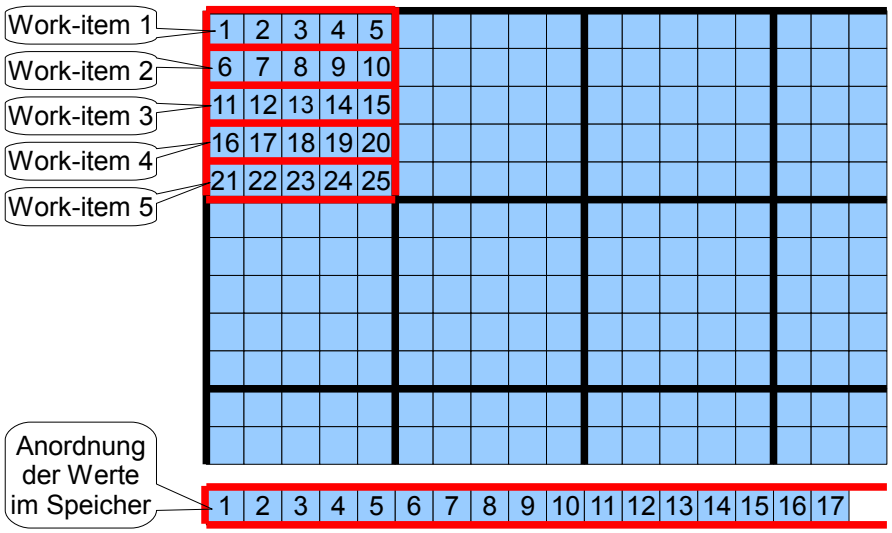

Abbildung 6.5: Partitionierung der Daten für die Multiplikation der Kernel BCSR V2a und V2b.

lauf des Reduktionskernels ist ebenfalls identisch zum Ablauf des Reduktionskernels in Kapitel 6.3.2. In Abbildung 6.4 ist der Ablauf des Reduktionskernels dargestellt.

6.4.2 Resultat

Die Verkleinerung der Arbeitspakete und zugleich die Erhöhung der Anzahl der verwendeten *work-items* für die Berechnung der SpMV bewirkt eine deutliche Performanzsteigerung. Die Verkleinerung der Arbeitspakete bewirkt zusätzlich eine Verringerung der Komplexität des Kernels. Dies konnte mit dem *Compute Visual Profiler* nachgewiesen werden. Die Implementierung des Multiplikationskernels benötigt nur noch 12 Register und kann deshalb beide Grafikkarten theoretisch zu 100% auslasten. Die Laufzeiten dieses Kernel sind in Tabelle 6.2 aufgelistet. Im nächste Schritt muss die Menge der transferierten Daten pro Iteration verringert werden.

	10x40x20	10x40x40	20x40x20	10x40x80	10x80x40	40x80x40
Tesla	14,97	43,89	43,83	80,56	80,7	305,34
Fermi	14,77	40,77	40,61	79,84	79,09	311,19
MPI	0,85	1,67	1,67	3,31	3,31	13,23

Tabelle 6.2: Die Laufzeiten der Testfälle mit dem BCSR-Kernel V2a in Sekunden.

6.5 Optimierung der SpMV mit BCSR-Datensatz für den GMRES-Algorithmus

Die Matrix-Vektor-Multiplikation wird für den GMRES-Algorithmus optimiert. Ein möglicher Optimierungsansatz ist der Transfer der Matrix auf die GPU. Die Daten der Matrix bleiben während der Iteration unverändert, deshalb bietet es sich an, die Matrizen der Blöcke vor dem Start des GMRES-Algorithmus auf die GPU zu kopieren. Während der Iteration müssen nur die Vektoren zwischen *Device* und *Host* ausgetauscht werden.

6.5.1 Implementierungsdetails

Während der Initialisierung der Daten werden für jeden Block die BCSR-Daten aufbereitet und anschließend in den *global memory* der GPU geladen. Die Struktur der Blöcke ist in dem *Struct LblockMatrix* implementiert. Das *Struct LblockMatrix* für die Blöcke muss um einige Attribute erweitert werden. Für die Einträge der Matrix, den Blockzeilenindex, den Blockspaltenindex, den temporären Zwischenspeicher und den überarbeiteten Index für den Beginn einer Zeile im Blockspaltenindex müssen Pointervariablen eingerichtet werden, die auf den *global memory* der GPU zeigen. Die neue Größe des Index für den Beginn einer Zeile im Blockspaltenindex wird in einer zusätzlichen Integer-Variable gespeichert. Der temporäre Zwischenspeicher wird aus Platzgründen nur einmal im *global memory* der GPU alloziert und nicht für jeden Block einen eigenen temporären Zwischenspeicher. Deshalb muss für alle Blöcke die benötigte Größe ermittelt werden und der größte Platzbedarf wird auf der GPU alloziert. Ein Pointer zu dem temporären Zwischenspeicher wird in jedem Block gespeichert. Mittels OpenCL-Methoden wird der benötigte Speicher auf der GPU alloziert, und die Daten werden in den *global memory* der GPU geladen. Die Daten der Matrix dürfen während der gesamten Laufzeit des Programms nicht mehr geändert werden und verbleiben bis zum Programmende auf der GPU. Damit kann die Menge der zu transferierenden Daten während der Iteration stark reduziert werden. Der temporäre Zwischenspeicher wird von jedem Kernel modifiziert, allerdings wird dieser auch nur beim Programmstart angelegt. Deshalb muss jeder Kernel dafür sorgen, dass die benötigten Werte des temporären Zwischenspeichers auf Null gesetzt sind oder der bisherige Wert das Ergebnis nicht beeinflusst. Die Verwendung des Datenspeichers auf der GPU muss in der Implementierung mittels des *Define* `#define USE_CENTRAL_DATASTORE ON` aktiviert werden.

Die Kernel aus den Kapiteln 6.3 und 6.4 müssen nicht neu implementiert werden. Es wird der bisherige Kernel-Code verwendet. Der Host-Code muss angepasst werden. An den Kernelaufruf müssen die Pointervariablen aus der Datenstruktur des

Blocks übergeben werden. Die neue Variante des Kernels aus Kapitel 6.3 wird im weiteren Text und den Diagrammen mit *BCSR V1b* bezeichnet. Für die Implementierung müssen die *Defines* `#define USE_BCSR_1b ON`, `USE_CENTRAL_DATASTORE ON` und `USE_BCSR_DATAFORMAT ON` aktiviert werden. Die neue Variante des Kernels aus Kapitel 6.4 wird im weiteren Text und den Diagrammen mit *BCSR V2b* bezeichnet. Für die Implementierung müssen die *Defines* `#define USE_BCSR_2b ON`, `USE_CENTRAL_DATASTORE ON` und `USE_BCSR_DATAFORMAT ON` aktiviert werden.

6.5.2 Resultat

Die Kernel wurden nicht verändert, deshalb wird auch die gleiche Anzahl an Registern benötigt, die auch schon ohne die zentrale Speicherung der Daten auf der GPU nötig war. Die Tabellen 6.3, 6.4 und das Diagramm 6.6 zeigen, dass die Verminderung der zu transferierenden Daten eine erhebliche Beschleunigung bewirken. Mit den wesentlich geringeren Datenmengen kann die Fermi-Grafikkarte in der Implementierung *BCSR V2b* ihre Hardwarevorteile umsetzen. Im Vergleich zur MPI-Variante ist die Implementierung *BCSR V1b* 15-mal langsamer. Die Implementierung *BCSR V2b* ist nur noch um den Faktor 3 langsamer als die MPI-Variante. Bisher wurden nur die Aufteilung der Arbeitspakete und die Datenmengen optimiert. In den nächsten Schritten müssen die Speicherung der Daten (Datenstruktur) und das verwendete Datenformat optimiert werden.

	10x40x20	10x40x40	20x40x20	10x40x80	10x80x40	40x80x40
Tesla	10,75	24,28	24,48	44,96	45,69	179,15
Fermi	12,13	24,07	24,08	47,33	48,89	-[1]
MPI	0,85	1,67	1,67	3,31	3,31	13,23

Tabelle 6.3: Die Laufzeiten der Testfälle mit dem BCSR-Kernel V1b in Sekunden.

[1] Konnte nicht berechnet werden, da nicht genügend Arbeitsspeicher auf der Grafikkarte vorhanden ist.

	10x40x20	10x40x40	20x40x20	10x40x80	10x80x40	40x80x40
Tesla	4,24	8,19	8,23	15,35	15,49	58,83
Fermi	2,77	5,1	5,07	9,03	9,03	-[1]
MPI	0,85	1,67	1,67	3,31	3,31	13,23

Tabelle 6.4: Die Laufzeiten der Testfälle mit dem BCSR-Kernel V2b in Sekunden.

[1] Konnte nicht berechnet werden, da nicht genügend Arbeitsspeicher auf der Grafikkarte vorhanden ist.

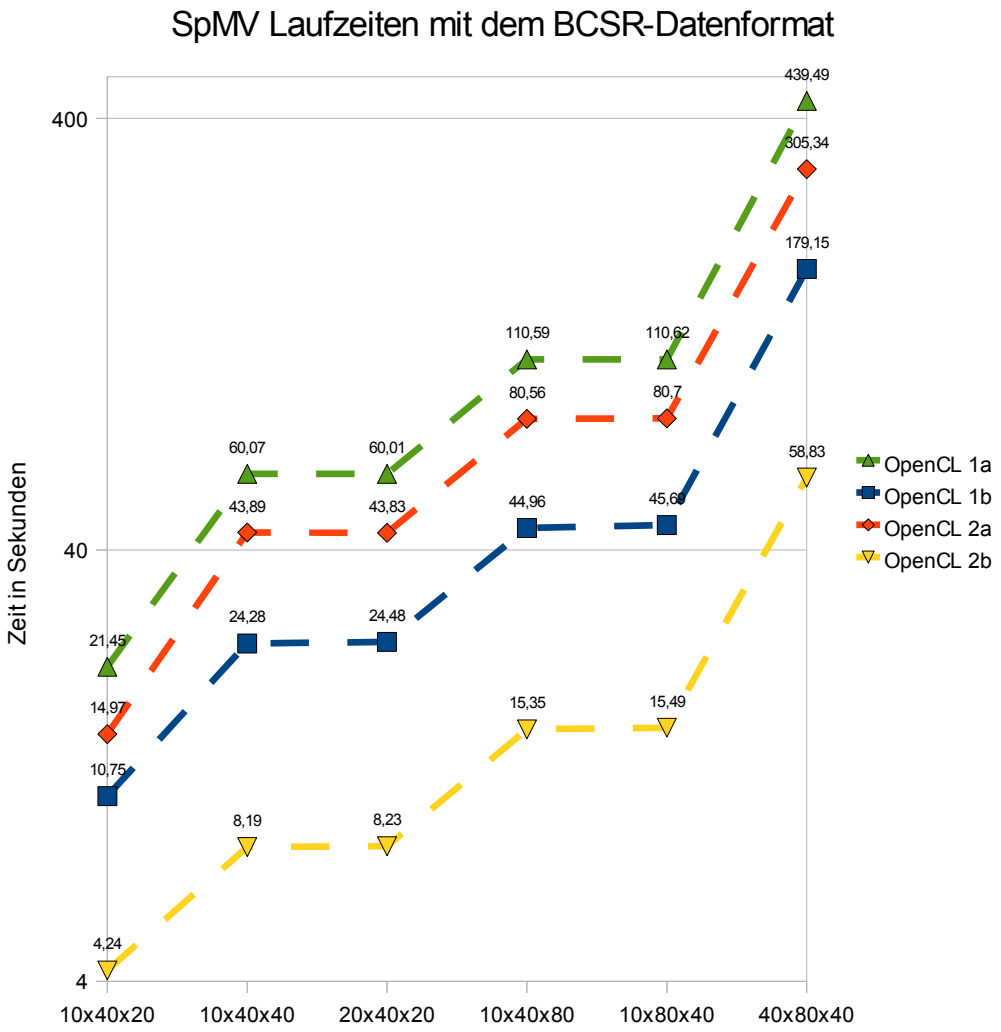

Abbildung 6.6: Vergleich der Laufzeiten der Implementierungen aus Kapitel 6.3 bis Kapitel 6.5.

6.6 SpMV mit permutierten BCSR-Blöcken

Die Werte eines 5x5 BCSR Blocks werden nicht mehr zeilenweise sondern spaltenweise gespeichert. Durch die Permutation der Werte soll ein Teil der Daten in einer besseren Reihenfolge für die Verarbeitung durch die GPU gespeichert werden.

6.6.1 Implementierungsdetails

Im *NVIDIA OpenCL Best Practices Guide* [NVIDI-2011b] sind einige Hinweise zur Optimierung von OpenCL-Kerneln enthalten. Die optimale Anordnung der Daten im Speicher der GPU ist in Kapitel 3.2.1 des *NVIDIA OpenCL Best Practices Guide* beschrieben. Die Daten, die bei einem Speicherzugriff für einen *half warp* gelesen werden, sollten im Speicher direkt hintereinander und in der passenden Reihenfolge der *work-items* liegen. In den bisherigen Implementierungen war dies nicht der Fall. Im Kernel V2b müssen nach jedem benötigten Wert 4 weitere Werte eingelesen werden, bevor der nächste benötigte Wert im Speicher liegt. Dies führt zu einer Verschlechterung der Performanz. Die *work-items* verarbeiten gleichzeitig aus unterschiedlichen Zeilen eines BCSR-Blocks bzw. mehrere BCSR-Blöcke die Werte in der gleichen Spalte. Deshalb sollte eine Umstellung von zeilenweiser Speicherung der Blöcke auf spaltenweiser Speicherung der Blöcke eine Performanzsteigerung bewirken. In Abbildung 6.8 kann man erkennen, dass pro BCSR-Block die benötigten Daten direkt hintereinander im Speicher liegen. In Abbildung 6.7 ist die Permutation der Werte eines BCSR-Blocks dargestellt.

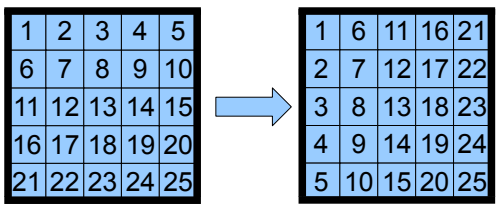

Abbildung 6.7: Permutation der BCSR Blöcke.

Die folgende Implementierung wird im weiteren Text und den Diagrammen mit *BCSR V3* bezeichnet. Das *Struct LblockMatrix* für die Blöcke muss um ein Array für die Speicherung der Matrix mit den permutierten BCSR-Blöcken erweitert werden. Ein Pointer auf den Speicherplatz der GPU an dem die Matrix mit den permutierten BCSR-Blöcken gespeichert wird, muss auch zum *Struct LblockMatrix* hinzugefügt werden. Die neu erzeugte Matrix wird während der Initialisierung auf die GPU geladen und verbleibt bis zum Programmende auf der GPU. Es werden zwei Kernel für diese Implementierung benötigt, ein Multiplikationskernel und ein Reduktionskernel.

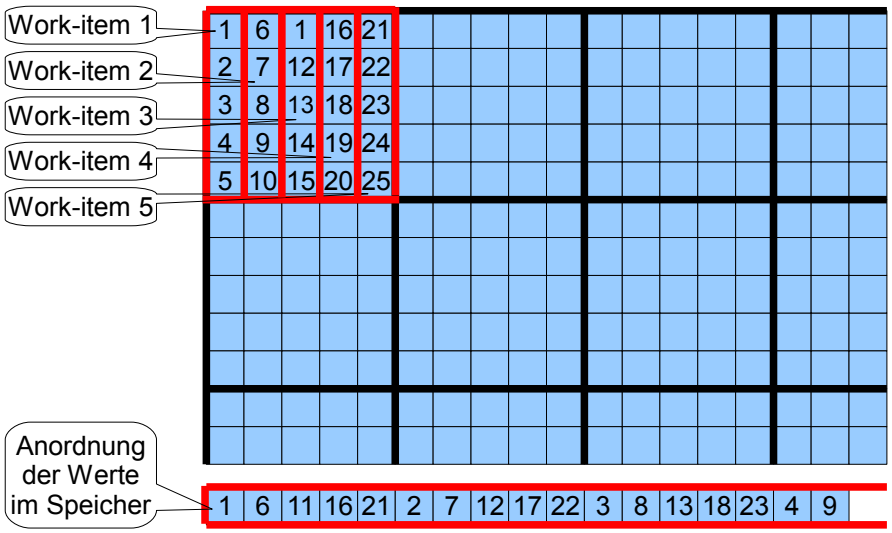

Abbildung 6.8: Partitionierung der Daten für die Multiplikation des BCSR Kernel V3.

Die *work-items* für den Multiplikationskernel werden in *work-groups* zu 256 *work-items* mit einer Ausdehnung von 16 mal 16 *work-items* zusammengefasst. Die gesamt Anzahl der *work-items* wird durch die Anzahl der BCSR-Blöcke multipliziert mit der Anzahl der Zeilen eines BCSR-Block bestimmt. Die Partitionierung der Daten für den Multiplikationskernel der Implementierung *BCSR V3* ist in Abbildung 6.8 dargestellt. Im oberen Teil der Grafik ist die Partitionierung der Daten dargestellt. Im unteren Teil der Grafik ist die Repräsentation der Daten im Speicher abgebildet. Der Multiplikationskernel benötigt die Einträge der Matrix, die Einträge des Vektors, den Blockspaltenindex, den temporären Zwischenspeicher und die Anzahl der BCSR-Blöcke als Übergabeparameter. Die Einträge des Vektors müssen im Host-Code des Kernels in den *global memory* der GPU geladen werden. Die anderen Übergabeparameter wurden während der Inizialisierung in den *global memory* der GPU geladen. Jeder BCSR-Block wird durch fünf *work-items* abgearbeitet. Jedes *work-item* bearbeitet eine Spalte des permutierten BCSR-Blocks. Die Implementierung des Multiplikationskernels basiert auf der Implementierung *BCSR 2b*. Die Indizierung der Werte der Matrix muss auf die permutierte Datenstruktur angepasst werden. Eine Instanz des Kernel muss die Werte einer Spalte eines BCSR-Blocks mit den passenden Werten aus dem Vektor multiplizieren. Dies ist in Abbildung 6.9 dargestellt. Der Index für den Wert des Vektors muss aus dem Blockspaltenindex ausgelesen werden. Zur Adressierung des richtigen Index im Blockspaltenindex muss der Kernel aus der *work-item ID* die BCSR-Block ID bestimmen. Diese wird als Index für den Blockspaltenindex verwendet. Die Summe der fünf Multiplikationen wird im temporären Zwischenspeicher gespeichert. Die Adressierung im temporären Zwischenspeicher erfolgt mit der *work-item ID*. Die Anzahl der Blöcke mal die Anzahl der Zeilen eines BCSR-Blocks dienen als Begrenzung für die Ausführung des Kernels, damit nicht mehr Instanzen

des Kernels ausgeführt werden als benötigt. Dies würde zu Speicherzugriffsfehlern führen, da Daten aus nicht alloziertem Speicher gelesen würden. Die Zwischensummen einer Zeile der Matrix werden von einem Reduktionskernel aufsummiert und im Ergebnisvektor gespeichert.

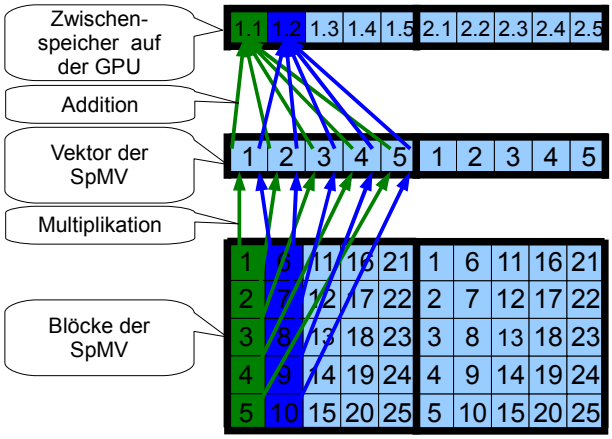

Abbildung 6.9: Ablauf der Multiplikation für die Kernel BCSR V3.

Für den Reduktionskernel werden 256 *work-items* zu einer *work-group* mit einer Ausdehnung von 1 mal 256 *work-items* zusammengefasst. Die Anzahl der *work-items* wird durch die Anzahl der Blockzeilen mal die Anzahl der Zeilen pro BCSR-Block ermittelt. Die Partitionierung der Daten für den Reduktionskernel der Implementierung *BCSR V3* ist identisch zur Aufteilung der Daten für die Implementierung des Reduktionskernels in Kapitel 6.3.2. Die Partitionierung ist in Abbildung 6.3 dargestellt. Der Ablauf des Reduktionskernels ist ebenfalls identisch zum Ablauf des Reduktionskernels in Kapitel 6.3.2. In Abbildung 6.4 ist der Ablauf des Reduktionskernels dargestellt. Für die Implementierung müssen die *Defines* `#define USE_BCSR_V3 ON` und `USE_CENTRAL_DATASTORE ON` aktiviert werden.

6.6.2 Resultat

Die Permutation der BCSR-Blöcke konnte auf der Tesla-Hardware eine erhebliche Beschleunigung im Vergleich zum Kernel *BCSR V2b* bewirken. In der Tabelle 6.5 ist zuerkennen, dass auf der Fermi-Hardware im Vergleich zur Tabelle 6.4 nur eine geringe Beschleunigung erreicht werden konnte. Der Vergleich der Ergebnisse des Kernels *BCSR V2b* und des Kernels *BCSR V3* zeigen, dass der Cache die schlechte Struktur der Daten in Kernel *BCSR V2b* ausgleichen konnte. Deshalb hat die Permutation der BCSR-Blöcke nur zu einer geringen Performanzsteigerung geführt. Die Analyse mit dem *Compute Visual Profiler* zeigt, dass der Multiplikationskernel dieser Implementierung 11 Register benötigt. Damit kann die GPU theoretisch zu 100% ausgelastet werden. Auf der Fermi GPU ist die OpenCL Implementierung unverändert um den

den Faktor 3 langsamer. Auf der Tesla GPU konnte eine Verbesserung auf den Faktor 4 vonm ursprünglichen Faktor 5 erreicht werden. Im nächsten Schritt sollten die Werte der gesamten Matrix neu angeordnet werden und nicht nur die BCSR-Blöcke. Damit sollte eine weitere Performanzsteigerung möglich sein.

	10x40x20	10x40x40	20x40x20	10x40x80	10x80x40	40x80x40
Tesla	3,3	6,58	6,59	12,45	12,74	48,87
Fermi	2,74	5,06	5,02	8,96	8,95	-[1]
MPI	0,85	1,67	1,67	3,31	3,31	13,23

Tabelle 6.5: Die Laufzeiten der Testfälle mit dem BCSR-Kernel V3 in Sekunden.

[1] Konnte nicht berechnet werden, da nicht genügend Arbeitsspeicher auf der Grafikkarte vorhanden ist.

6.7 Dünnbesetzte Matrix-Vektor-Multiplikation mit ELL-Datensatz

Der BCSR-Datensatz soll in das ELL-Datenformat (siehe Kapitel 2.5) konvertiert werden. Die Daten werden spaltenweise gespeichert. Damit sollen die gesamten Daten der Matrix besonders effizient für die Verarbeitung durch die GPU gespeichert werden.

6.7.1 Aufbereitung der Daten der Matrix

Zur Vorbereitung müssen die Daten der Matrix vom *BCSR*-Datenformat in das *ELL*-Datenformat konvertiert werden. Während der Konvertierung werden die Blockstrukturen aufgelöst und die gesamte Matrix von einer zeilenweisen Speicherung auf eine spaltenweisen Speicherung umgestellt. Die Konvertierung ist in Abbildung 6.10 dargestellt. Im *ELL*-Datenformat muss für jeden Wert zusätzlich der Spaltenindex gespeichert werden. Im *BCSR*-Datenformat war dies nur pro Block notwendig. Daraus folgt ein größerer Speicherbedarf für die Speicherung der Matrix. Allerdings wird bei den Daten der Matrix auch Speicher eingespart, da Einträge mit Nullen entfernt werden und nur bis zur Länge der längsten Zeile wieder mit Nullen aufgefüllt werden. Beim Kernel *ELL* wird die Zeilenlänge bis zum nächsten Vielfachen der Länge der Unterabschnitte aufgefüllt. Die Unterabschnitte teilen eine Zeile in gleich lange Unterabschnitte auf, die von unterschiedlichen *work-items* berechnet werden.

Für die Daten der Matrix im *ELL*-Format muss das *Struct LblockMatrix* um einige Variablen erweitert werden. Für die konvertierte Matrix muss ein neues Array angelegt werden und der dazugehörige Pointer auf den *global memory* der GPU, in dem die Matrix gespeichert wird. Für die Spaltenindizes muss ein Integer Array angelegt werden, und ein Pointer auf den *global memory* der GPU ist notwendig. Ein weiteres Integer-Array für den Zeilenindex und der dazugehörige Pointer auf den *global*

memory der GPU muss zum *Struct LblockMatrix* hinzugefügt werden. Diese Daten der konvertierten Matrix werden während der Initialisierung der Daten in den *global memory* der GPU kopiert. Für die Anzahl der Zeilen und der Spalten wird jeweils eine Integer-Variable zum *Struct LblockMatrix* hinzugefügt. Der Kernel *ELL* benötigt zusätzlich noch einen temporären Zwischenspeicher im *global memory* der GPU. Dieser wird während der Initialisierung der Daten im *global memory* der GPU angelegt und wird für alle Blöcke nur einmal angelegt. Deshalb muss vorher die maximal notwendige Größe bestimmt werden.

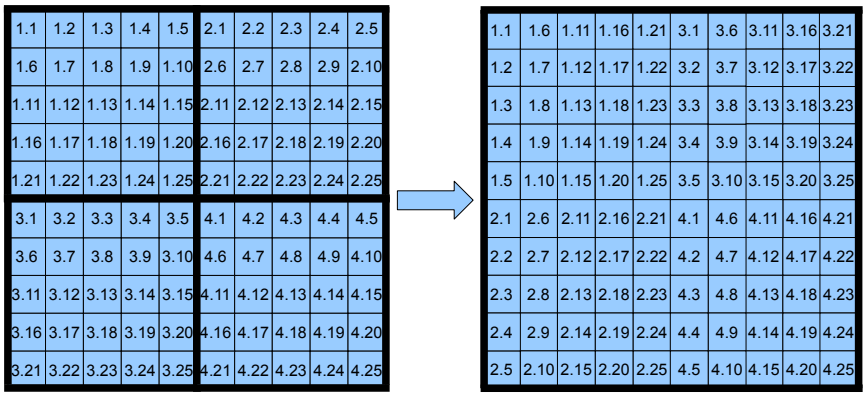

Abbildung 6.10: Konvertierung der BCSR-Daten in das ELL-Datenformat.

6.7.2 Implementierungsdetails

Diese Implementierung wird im weiteren Text und den Diagrammen mit *ELL* bezeichnet. Ein Multiplikationskernel und ein Reduktionskernel sind für diese Implementierung notwendig. Für die Implementierung müssen die *Defines* `#define USE_ELL_DATAFORMAT ON` und `USE_CENTRAL_DATASTORE ON` aktiviert werden.

Die *work-items* für den Multiplikationskernel werden in *work-groups* zu 256 *work-items* mit einer Ausdehnung von 16 mal 16 *work-items* zusammengefasst. Die gesamte Anzahl der *work-items* wird bestimmt durch die Anzahl der Spalten dividiert durch die Länge der Unterabschnitte multipliziert mit der Anzahl der Zeilen der Matrix. Die Partitionierung der Daten für den Multiplikationskernel der Implementierung *ELL* ist in Abbildung 6.11 dargestellt. Im oberen Teil der Grafik ist die Partitionierung der Daten dargestellt. Im unteren Teil der Grafik ist die Repräsentation der Daten im Speicher abgebildet. Dieser Multiplikationskernel benötigt als Übergabeparameter die Einträge der Matrix, die Einträge des Vektors, den Spaltenindex, den Zeilenindex, den temporären Zwischenspeicher, die Anzahl der Spalten und die Anzahl der Zeilen. Die Einträge des Vektors müssen vom Host-Code in den *global memory* der GPU kopiert werden. Die Zeilen der Matrix werden für die Multiplikation in gleichlange Unterabschnitte aufgeteilt. Die Länge der Unterabschnitte wird durch das *Define*

`#define ELL_TASK_SIZE 30` in der Datei *opencl.h* bestimmt. Mit einer *For-Schleife* wird über die Spalten eines Unterabschnitts iteriert. Mit der Anzahl der Spalten, der *work-item ID* und der Größe des Unterabschnitts wird der erste Index für den Matrix Eintrag nach folgender Formel $((workItemID/rowCount) * ELL\_TASK\_SIZE) * rowCount + (workItemID\%rowCount)$ bestimmt. Dieser Index wird auch für das Auslesen des Spaltenindex benötigt, mit dem der Wert aus dem Vektor gelesen wird. Das Ergebnis eines Unterabschnitts wird im temporären Zwischenspeicher gespeichert, der mit der *work-item ID* adressiert wird. Die Zwischensummen einer Zeile aus dem temporären Zwischenspeicher werden mit einem Reduktionskernel aufsummiert.

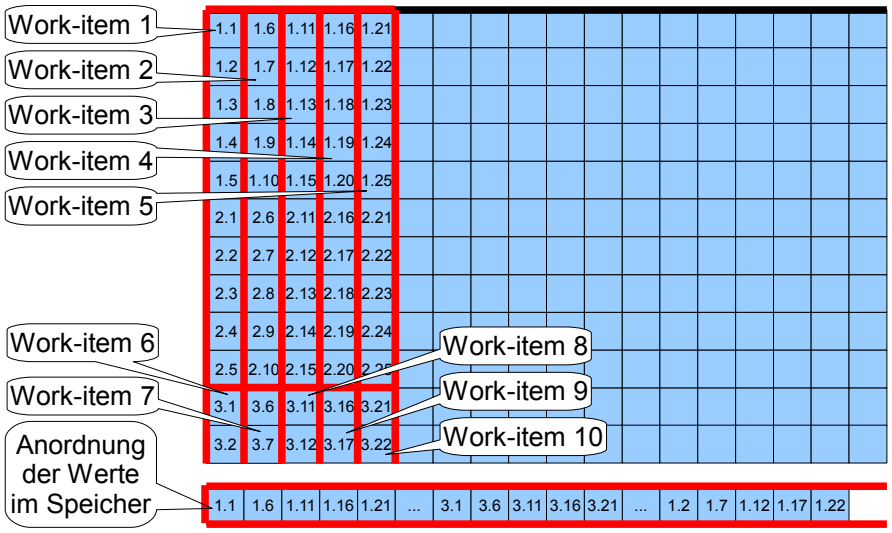

Abbildung 6.11: Partitionierung der Daten für die Multiplikation des Kernels ELL.

Für den Reduktionskernel werden 256 *work-items* zu einer *work-group* mit einer Ausdehnung von 1 mal 256 *work-items* zusammengefasst. Die Anzahl der *work-items* wird durch die Anzahl der Zeilen der Matrix ermittelt. In Abbildung 6.12 ist die Partitionierung der Daten für den Reduktionskernel dargestellt. Die grünen Felder sollen die Zwischensummen im temporären Zwischenspeicher darstellen. Pro Zeile der Matrix wird ein *work-item* benötigt.

Der Reduktionskernel benötigt als Übergabeparameter den temporären Zwischenspeicher, die Einträge des Ergebnisvektors, den Zeilenindex, die maximale Größe des temporären Zwischenspeichers und die Anzahl der Zeilen. Der Ergebnis-Vektor muss im Host-Code auf der GPU im *global memory* alloziert werden und mittels eines Kernelaufrufs müssen alle Einträge auf Null gesetzt werden. Im Kernel werden mit einer *For-Schleife* die Zwischensummen aus dem *global memory* der GPU ausgelesen und aufsummiert. Der Index der *For-Schleife* wird mit der Anzahl der Zeilen pro Iteration inkrementiert. Als Abbruchkriterium für die *For-Schleife* ist die maximale Größe des temporären Zwischenspeichers definiert. Das Ergebnis einer Zeile wird im Ergebnisvektor gespeichert. Der Index für den Eintrag des Ergebnisvektors wird mit

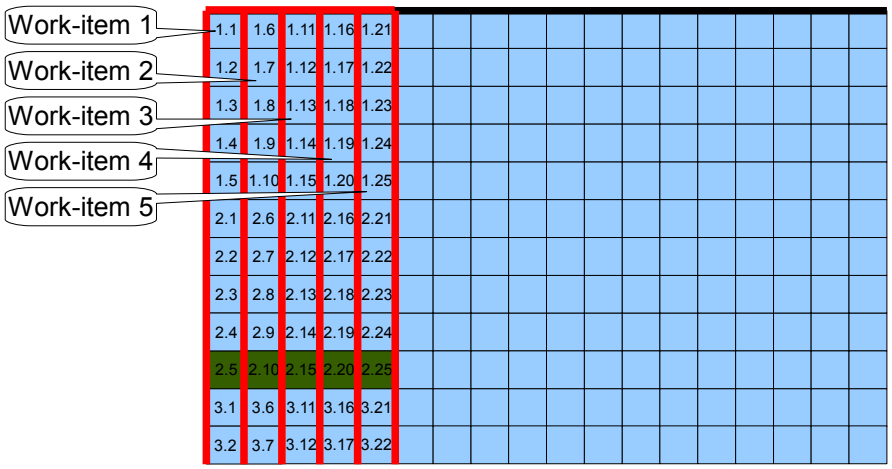

Abbildung 6.12: Partitionierung der Daten für die Reduktion des Kernels ELL.

der *work-item ID* aus dem Zeilenindex ausgelesen. Die Anzahl der Zeilen der Matrix dienen als Begrenzung für die Ausführung des Kernels, damit nicht mehr Instanzen des Kernels ausgeführt werden als benötigt. Dies würde zu Speicherzugriffsfehlern führen, da Daten aus nicht alloziertem Speicher gelesen würden. Der Ergebnisvektor wird nach der Berechnung der SpMV vom Host-Code aus dem *global memory* GPU ausgelesen und im Speicher des Hosts gespeichert.

Die folgende Implementierung wird im weiteren Text und in den Diagrammen mit *ELL V2* bezeichnet. In dieser Implementierung soll untersucht werden, ob der zusätzliche Aufwand durch das Aufrufen des Reduktionskernel größer ist als der Nutzen durch die Aufteilung der Zeile in kleiner Arbeitspakete. Diese Variante benötigt nur einen Multiplikationskernel. Für die Implementierung müssen die *Defines* `#define USE_ELLV2_DATAFORMAT ON` und `USE_CENTRAL_DATASTORE ON` aktiviert werden.

Die *work-items* für den Multiplikationskernel werden in *work-groups* zu 256 *work-items* mit einer Ausdehnung von 16 mal 16 *work-items* zusammengefasst. Die gesamte Anzahl der *work-items* wird durch die Anzahl der Zeilen der Matrix bestimmt. Die Partitionierung der Daten ist in Abbildung 6.13 dargestellt. Im oberen Teil der Grafik ist die Partitionierung der Daten dargestellt. Im unteren Teil der Grafik ist die Repräsentation der Daten im Speicher abgebildet. Dieser Multiplikationskernel benötigt als Übergabeparameter die Einträge der Matrix, die Einträge des Vektors, den Spaltenindex, den Zeilenindex, die Einträge des Ergebnisvektors, die Anzahl der Spalten und die Anzahl der Zeilen. Die Einträge des Vektors und die Einträge des Ergebnisvektors müssen vom Host-Code in den *global memory* der GPU kopiert werden. Der Kernel *ELL V2* basiert im wesentlichen auf dem Kernel *ELL*. Der temporäre Zwischenspeicher wird in dieser Implementierung nicht mehr benötigt, da ein *work-item* eine gesamte Zeile der Matrix berechnet und nicht nur einen Unterabschnitt. Deshalb ite-

riert die *For-Schleife* in dieser Implementierung über eine ganze Spalte und nicht nur über einen Unterabschnitt. Das Ergebnis wird im Ergebnisvektor gespeichert. Der Index für den Wert im Ergebnisvektor wird mit der *work-item ID* aus dem Zeilenindex gelesen. Die Anzahl der Zeilen der Matrix dient als Begrenzung für die Ausführung des Kernels, damit nicht mehr Instanzen des Kernels ausgeführt werden als benötigt. Dies würde zu Speicherzugriffsfehlern führen, da Daten aus nicht alloziertem Speicher gelesen würden. Nach der SpMV wird der Ergebnisvektor aus dem *global memory* in den Speicher des Host-Systems geladen.

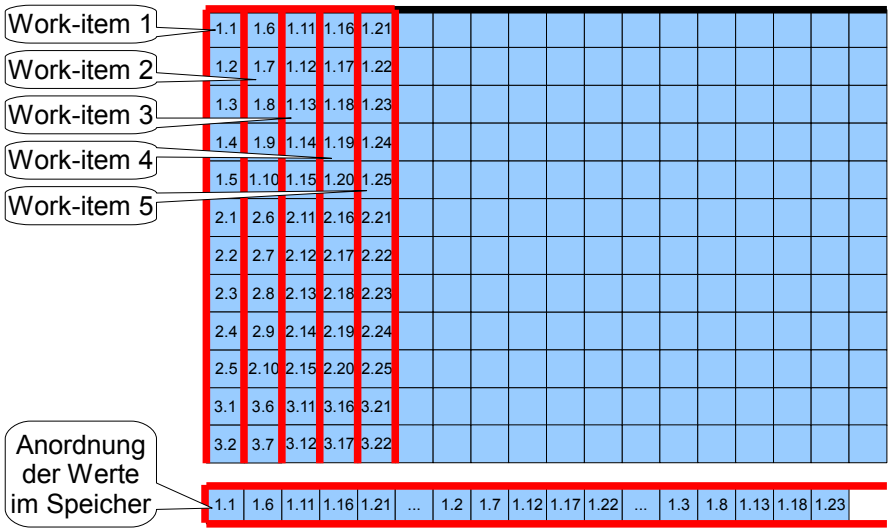

Abbildung 6.13: Partitionierung der Daten für die Multiplikation des Kernel ELL V2.

6.7.3 Resultat

In Diagramm 6.14 und Diagramm 6.15 ist zu erkennen, dass der Kernel *ELL* mit einer Unterabschnittslänge von 20 Werten die beste Performanz erreicht. In Tabelle 6.6 sind die Laufzeiten des Kernel *ELL* mit einer Unterabschnittslänge von 20 Werten aufgelistet. Im Vergleich zum Kernel *BCSR V3* konnte auf der Tesla- und der Fermi-Hardware fast eine Halbierung der Laufzeiten erreicht werden. Damit ist diese Implementierung auf der Tesla-Hardware im Vergleich zur MPI-Implementierung nur noch um den Faktor 2 langsamer und auf der Fermi-Hardware nur noch um den Faktor 1,4 langsamer. Die Analyse mit dem *Compute Visual Profiler* zeigt, dass der Multiplikationskernel dieser Implementierung 20 Register benötigt. Nach dem *CUDA Occupancy Calculator* kann die GPU auf der Tesla-Hardware theoretisch nur zu 75% und auf der Fermi-Hardware zu 100% ausgelastet werden. Der Reduktionskernel benötigt 8 Register und kann damit die GPU theoretisch zu 100% auslasten.

In Tabelle 6.7 ist zu erkennen, dass der Kernel *ELL V2* im Vergleich zum Kernel *BCSR V3* eine Beschleunigung erreichen konnte. Allerdings gegenüber dem Kernel

	10x40x20	10x40x40	20x40x20	10x40x80	10x80x40	40x80x40
Tesla [1]	2,18	4,11	3,87	6,7	7,12	21,11
Fermi [1]	1,61	2,86	2,76	4,73	4,7	-[2]
MPI	0,85	1,67	1,67	3,31	3,31	13,23

Tabelle 6.6: Die Laufzeiten der Testfälle mit dem ELL-Kernel in Sekunden.

[1] Die Länge der Unterabschnitte beträgt 20 Werte.
[2] Konnte nicht berechnet werden, da nicht genügend Arbeitsspeicher auf der Grafikkarte vorhanden ist.

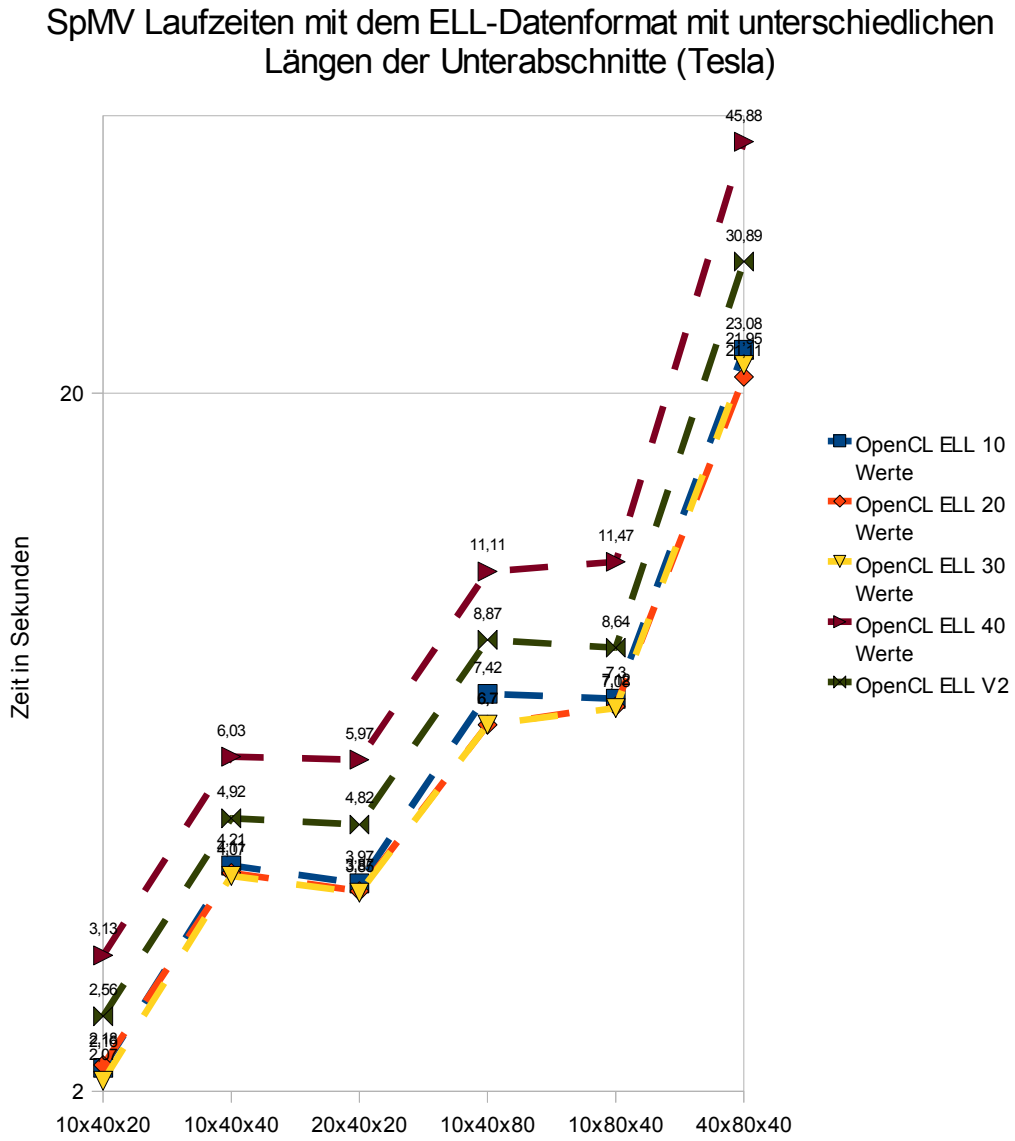

Abbildung 6.14: Die Laufzeiten der Testfälle mit dem ELL-Kernel auf der Tesla GPU in Sekunden.

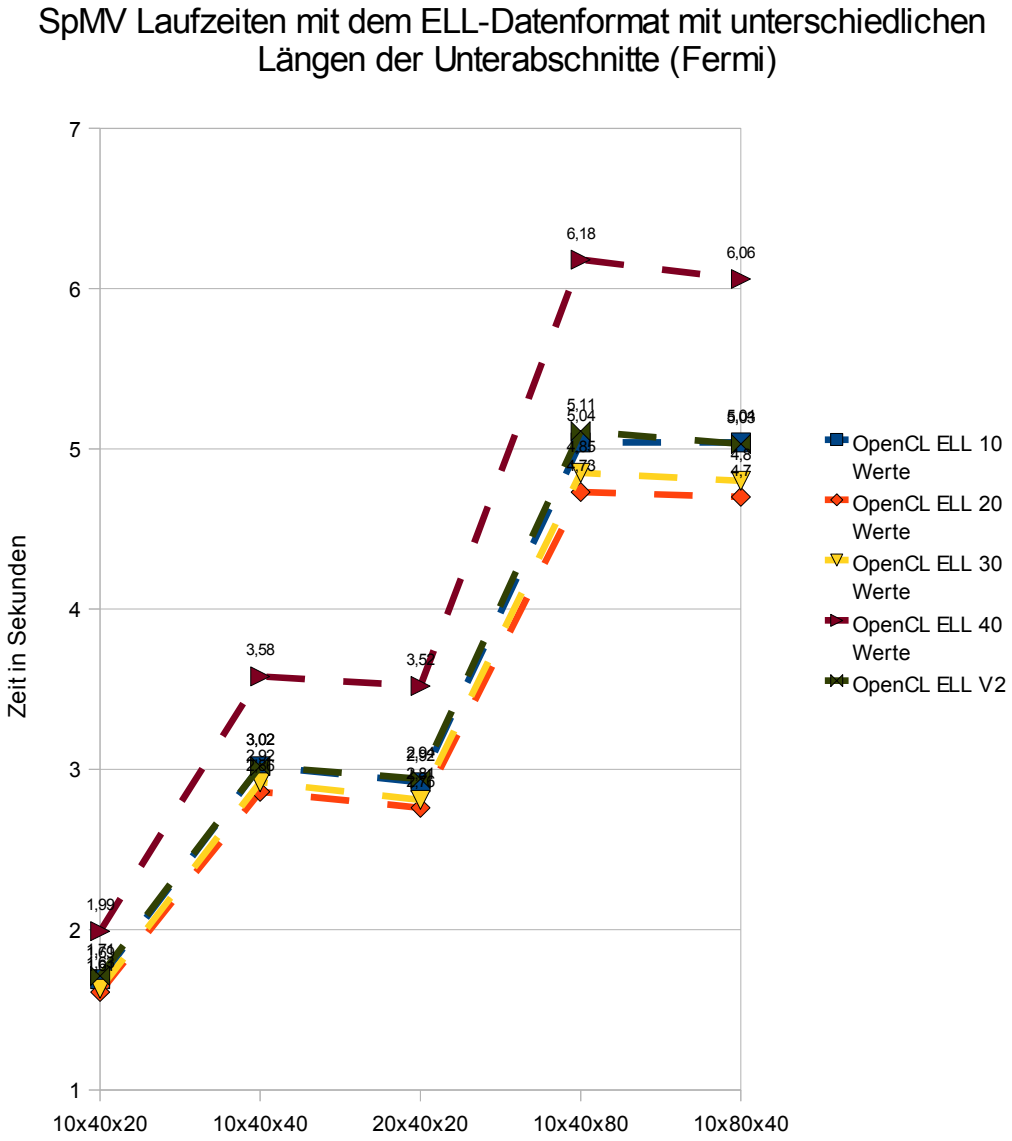

Abbildung 6.15: Die Laufzeiten der Testfälle mit dem ELL-Kernel auf der Fermi GPU in Sekunden.

ELL musste eine Verschlechterung der Performanz registriert werden. Die Analyse mit dem *Compute Visual Profiler* zeigt, dass diese Implementierung 13 Register benötigt, damit kann die GPU theoretisch zu 100% ausgelastet werden. Damit ist diese Implementierung auf der Tesla-Hardware im Vergleich zur MPI-Implementierung um den Faktor 2,7 langsamer und auf der Fermi-Hardware nur noch um den Faktor 1,5 langsamer. Obwohl der Kernel der Implementierung *ELL* die GPU nicht vollständig auslasten kann und die Implementierung *ELL V2* die GPU vollständig auslasten kann, ist die Implementierung *ELL* schneller. Im *ELL*-Datenformat werden weniger Einträge der Matrix gespeichert. Diesem Umstand kann die bessere Laufzeit geschuldet sein. Die tatsächliche Performanz der Kernel mit dem ELL-Datenformat kann nur mit den erreichten GFLOP/s mit den anderen Implementierungen verglichen werden. Diese Vergleich folgt in Kapitel 7. Im nächsten Schritt muss der Datentransfer während der Ausführung des Kernels verringert werden. Im *ELL*-Datenformat muss für jeden Wert der Matrix auch ein Zugriff auf den Spaltenindex erfolgen, da dieser im *global memory* liegt, wirkt sich dies negativ auf die Performanz aus. Diese Zugriffe sind bei der Verwendung des *BELL*-Datenformats geringer, da nur für jeden Block ein Wert im Blockspaltenindex gespeichert wird.

	10x40x20	10x40x40	20x40x20	10x40x80	10x80x40	40x80x40
Tesla	2,56	4,92	4,82	8,87	8,64	30,89
Fermi	1,71	3,02	2,94	5,11	5,03	-[1]
MPI	0,85	1,67	1,67	3,31	3,31	13,23

Tabelle 6.7: Die Laufzeiten der Testfälle mit dem ELL-Kernel V2 in Sekunden.

[1] Konnte nicht berechnet werden, da nicht genügend Arbeitsspeicher auf der Grafikkarte vorhanden ist.

6.8 Dünnbesetzte Matrix-Vektor-Multiplikation mit Blocked ELL-Datensatz

In dem Paper *Model-driven Autotuning of Sparse Matrix-Vector Multiply on GPUs* [ChoiJ-2010] konnte gezeigt werden, dass das *Blocked ELLPACK*-Datenformat gut für dünnbesetzte Matrix-Vektor-Multiplikationen geeignet ist. Das *Blocked ELLPACK*-Datenformat ist eine Modifizierung des ELL-Datenformats. In dieser Implementierung wird die Matrix-Vektor-Multiplikation für das *Blocked ELLPACK*-Datenformat implementiert. Vor dem Start des Algorithmus müssen die Matrizen der TRACE-Blöcke vom BCSR-Datenformat in das *Blocked ELLPACK*-Datenformat konvertiert werden. Diese Implementierung soll zeigen, ob ein Wechsel des Datenformats im TRACE-Code erfolgversprechend ist.

6.8.1 Aufbereitung der Daten der Matrix

Die Matrix im *BCSR*-Format muss in das *BELL*-Format konvertiert werden. Zuerst muss die maximale Blockanzahl pro Blockzeile bestimmt werden, die Werte ungleich Null enthält. Auf diese Blockanzahl werden alle Blockzeilen mit Blöcken aufgefüllt, diese Blöcke enthalten nur Nullen. Danach wird die Matrix von blockzeilenweiser Speicherung auf spaltenweise Speicherung geändert. Der Blockspaltenindex muss entsprechend angepasst werden. Für die Implementierung *BELL* wird ein Spaltenindex erstellt für die Implementierung *BELL V2* wird ein Blockspaltenindex erstellt.

6.8.2 Implementierungsdetails

Die folgende Implementierung wird im weiteren Text und in den Diagrammen mit *BELL* bezeichnet. Eine Beschleunigung soll durch die Verringerung der Zugriffe auf den *global memory* der GPU erreicht werden. Das *ELL*-Datenformat wird mit Blockstrukturen versehen. Pro Zeile eines Blocks muss aus dem Blockspaltenindex nur ein Wert anstatt fünf Werten gelesen werden. Diese Variante benötigt nur einen Multiplikationskernel. Für die Implementierung müssen die *Defines* `#define USE_BELL_DATAFORMAT ON` und `USE_CENTRAL_DATASTORE ON` aktiviert werden.

Die *work-items* für den Multiplikationskernel werden in *work-groups* zu 256 *work-items* mit einer Ausdehnung von 16 mal 16 *work-items* zusammengefasst. Die gesamte Anzahl der *work-items* wird durch die Anzahl der Zeilen der Matrix bestimmt. Die Partitionierung der Daten ist in Abbildung 6.16 dargestellt. Im oberen Teil der Grafik ist die Partitionierung der Daten dargestellt. Im unteren Teil der Grafik ist die Repräsentation der Daten im Speicher abgebildet. Dieser Multiplikationskernel benötigt als Übergabeparameter die Einträge der Matrix, die Einträge des Vektors, den Blockspaltenindex, den Blockzeilenindex, die Einträge des Ergebnisvektors, die Anzahl der Blockspalten und die Anzahl der Blockzeilen. Die Einträge des Vektors und die Einträge des Ergebnisvektors müssen vom Host-Code in den *global memory* der GPU kopiert werden. Diese Implementierung benötigt zwei *For-Schleifen*. Die äußere *For-Schleife* iteriert über die Blockspalten. Die innere *For-Schleife* iteriert über die die Spalten eines *BELL*-Blocks. In der äußeren *For-Schleife*, bevor die innere *For-Schleife* beginnt, wird der Blockspaltenindex für den Block bestimmt. In der inneren *For-Schleife* werden die Ergebnisse der Multiplikationen einer Zeile der Matrix aufsummiert. Am Ende wird das Ergebnis der Zeile in den Ergebnisvektor geschrieben. Die Anzahl der Zeilen der Matrix dienen als Begrenzung für die Ausführung des Kernels, damit nicht mehr Instanzen des Kernels ausgeführt werden als benötigt. Dies würde zu Speicherzugriffsfehlern führen, da Daten aus nicht alloziertem Speicher gelesen würden. Nach der SpMV wird der Ergebnisvektor aus dem *global memory* in

den Speicher des Host-Systems geladen.

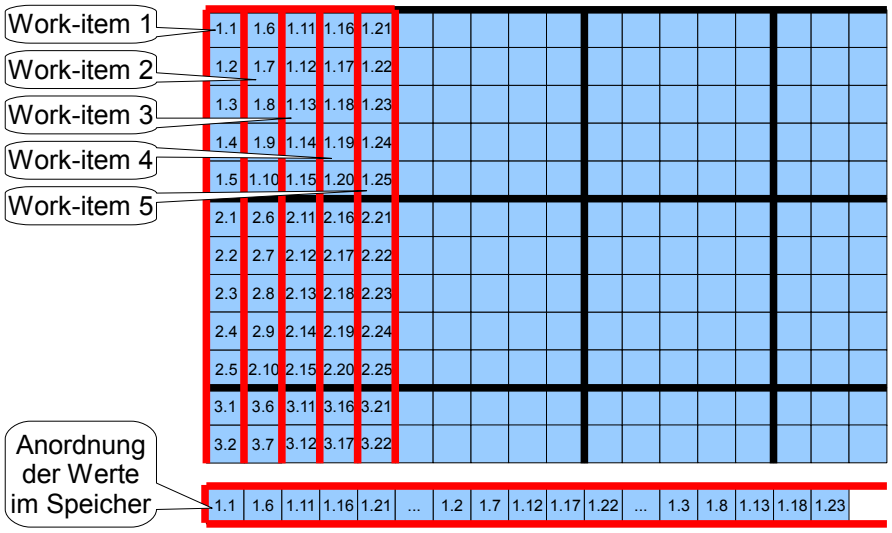

Abbildung 6.16: Partitionierung der Daten für die Multiplikation des Kernels BELL.

Die folgende Implementierung wird im weiteren Text und in den Diagrammen mit *BELL V2* bezeichnet. In dieser Variante sollen noch mehr Zugriffe auf den *global memory* der GPU eingespart werden. Die Abarbeitung einer ganzen Blockzeile durch ein *work-item* erspart weitere vier Zugriffe auf den *global memory* der GPU. Pro Block muss nur noch ein Wert aus dem Blockspaltenindex gelesen werden. Im Vergleich zum *ELL*-Datenformat können 24 Zugriffe auf den *global memory* der GPU pro Block eingespart werden. Diese Variante benötigt nur einen Multiplikationskernel. Für die Implementierung müssen die *Defines* `#define USE_BELLV2_DATAFORMAT ON` und `USE_CENTRAL_DATASTORE ON` aktiviert werden.

Die *work-items* für den Multiplikationskernel werden in *work-groups* zu 256 *work-items* mit einer Ausdehnung von 16 mal 16 *work-items* zusammengefasst. Die gesamte Anzahl der *work-items* wird durch die Anzahl der Blockzeilen der Matrix bestimmt. Die Partitionierung der Daten ist in Abbildung 6.17 dargestellt. Im oberen Teil der Grafik ist die Partitionierung der Daten dargestellt. Im unteren Teil der Grafik ist die Repräsentation der Daten im Speicher abgebildet. Dieser Multiplikationskernel benötigt als Übergabeparameter die Einträge der Matrix, die Einträge des Vektors, den Blockspaltenindex, den Blockzeilenindex, die Einträge des Ergebnisvektors, die Anzahl der Blockspalten und die Anzahl der Blockzeilen. Die Einträge des Vektors und die Einträge des Ergebnisvektors müssen vom Host-Code in den *global memory* der GPU kopiert werden. Es müssen nur wenige Modifikationen am Kernel *BELL* für diese Implementierung umgesetzt werden. Die innere *For-Schleife* wird um die Berechnung 4 weiterer Zeilen der Matrix erweitert. Zum Aufsummieren der Ergebnisse werden 4 zusätzliche Variablen benötigt. Die Anzahl der Blockzeilen der Matrix dienen als Begrenzung für die Ausführung des Kernels, damit nicht mehr Instanzen

des Kernels ausgeführt werden als benötigt. Dies würde zu Speicherzugriffsfehlern führen, da Daten aus nicht alloziertem Speicher gelesen würden. Nach der SpMV wird der Ergebnisvektor aus dem *global memory* in den Speicher des Host-Systems geladen.

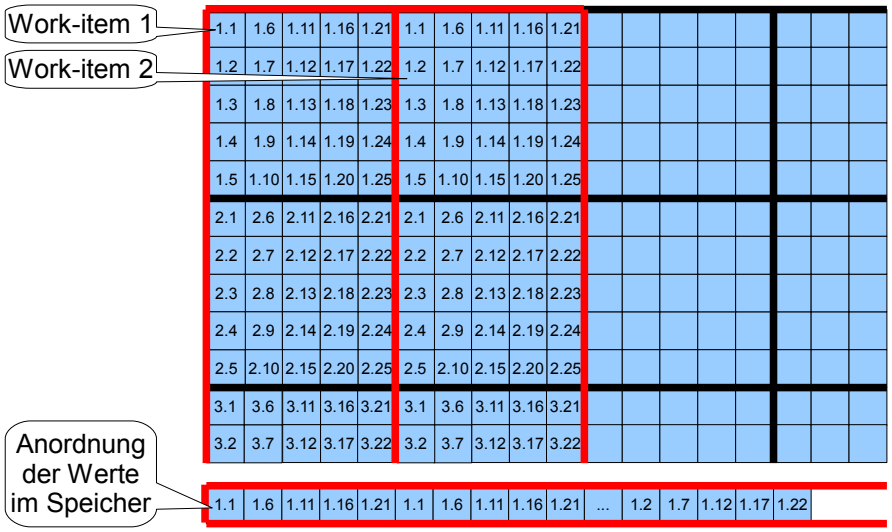

Abbildung 6.17: Partitionierung der Daten für die Multiplikation des Kernels BELL V2.

6.8.3 Resultat

In Tabelle 6.8 sind die Laufzeiten des Kernels *BELL* aufgelistet. Im Vergleich zum Kernel *ELL* konnte die Fermi-Hardware eine geringe Beschleunigung erzielen. Die Laufzeit auf der Tesla-Hardware hat sich verschlechtert. Damit ist diese Implementierung auf der Tesla-Hardware im Vergleich zur MPI-Implementierung um den Faktor 2,2 langsamer und auf der Fermi-Hardware um den Faktor 1,3 langsamer. Die Analyse mit dem *Compute Visual Profiler* zeigt, dass der Multiplikationskernel dieser Implementierung 19 Register benötigt. Nach dem *CUDA Occupancy Calculator* kann die GPU auf der Tesla-Hardware theoretisch nur zu 75% und auf der Fermi-Hardware zu 100% ausgelastet werden.

	10x40x20	10x40x40	20x40x20	10x40x80	10x80x40	40x80x40
Tesla	2,26	4,37	4,16	7,43	7,43	27,12
Fermi	1,49	2,69	2,64	4,46	4,41	-[1]
MPI	0,85	1,67	1,67	3,31	3,31	13,23

Tabelle 6.8: Die Laufzeiten der Testfälle mit dem BELL-Kernel in Sekunden.

[1] Konnte nicht berechnet werden, da nicht genügend Arbeitsspeicher auf der Grafikkarte vorhanden ist.

In Tabelle 6.9 sind die Laufzeiten des Kernel *BELL V2* aufgelistet. Im Vergleich zum Kernel *BELL* haben sich die Laufzeiten stark verschlechtert. Damit ist diese Imple-

mentierung auf der Tesla-Hardware im Vergleich zur MPI-Implementierung um den Faktor 4 langsamer und auf der Fermi-Hardware um den Faktor 2 langsamer. Die Analyse mit dem *Compute Visual Profiler* zeigt, dass der Multiplikationskernel dieser Implementierung 28 Register benötigt. Nach dem *CUDA Occupancy Calculator* kann die GPU auf der Tesla-Hardware theoretisch nur zu 50% und auf der Fermi-Hardware zu 67% ausgelastet werden. Damit erklärt sich der große Performanzverlust. Durch die schlechte Auslastung der GPU konnte der verringerte Datentransfer zum *global memory* der GPU keine Beschleunigung des Kernels bewirken. Die hohe Komplexität des Kernels hat zu einer Verschlechterung der Laufzeiten geführt. Im nächsten Schritt sollte die Komplexität des Kernels und der Datentransfer zum *global memory* der GPU während der Ausführung eines Kernels verringert werden. Deshalb sollte in der nächsten Implementierung der *shared memory* der GPU verwendet werden.

	10x40x20	10x40x40	20x40x20	10x40x80	10x80x40	40x80x40
Tesla	3,96	7,25	7,24	13,12	13,48	48,55
Fermi	2,06	3,75	3,75	6,66	6,62	-[1]
MPI	0,85	1,67	1,67	3,31	3,31	13,23

Tabelle 6.9: Die Laufzeiten der Testfälle mit dem BELL V2-Kernel in Sekunden.

[1] Konnte nicht berechnet werden, da nicht genügend Arbeitsspeicher auf der Grafikkarte vorhanden ist.

6.9 SpMV mit Blocked ELL-Datensatz unter Verwendung von shared memory

Eine Verbesserung der SpMV mit dem Blocked Ellpack Datensatz kann durch die Verwendung von *shared memory* erreicht werden. Die Einträge der Matrix in den *shared memory* zu laden ist nicht sinnvoll, da die Werte der Matrix pro Iteration nur einmal benötigt werden. Viele Einträge des Vektors werden mehrfach benötigt, deshalb ist es sinnvoll, Teile des Vektors in den *shared memory* zu laden. Allerdings passt nicht der ganze Vektor in den *shared memory*, deshalb ist eine Vorauswahl der benötigten Werte zutreffen.

6.9.1 Aufbereitung der Informationen für den shared memory

Der Vektor der SpMV kann nicht vollständig in den *shared memory* geladen werden, da der Vektor viel zu groß ist. Es muss ein Bereich des Vektors bestimmte werden, der aus dem *global memory* in den *shared memory* geladen wird. Dieser Bereich wird auch Daten beinhalten, die nicht für die SpMV benötigt werden, allerdings können diese nicht aussortiert werden, da sonst eine Indizierung nicht mehr möglich ist. Dieser

Nachteil sollte durch die mehrfache Verwendung der Werte im *shared memory* ausgeglichen werden. Welcher Bereich in den *shared memory* geladen werden muss, wird während der Initialisierung der Daten bestimmt. Deshalb benötigt das *Struct Lblock-Matrix* zusätzlich zwei Attribute für diese Implementierung: ein Integer-Array mit drei Werten pro *work-group* und ein Pointer auf den *global memory* der GPU, an dem das Integer-Array gespeichert wird. Pro *work-group* müssen drei Werte ermittelt werden. Der niedrigste und der höchste Index für die benötigten Werte des Vektors. Desweiteren muss die Anzahl der zu kopierenden Werte pro *work-item* ermittelt werden. Für die Berechnung dieser Informationen müssen mit dem Index für den Anfang einer Zeile im Blockspaltenindex und dem Blockzeilenindex die Indizes der benötigten Werte des Vektors pro Zeile bestimmt werden. Mehrere Zeilen werden zu einer *work-group* zusammengeschlossen. Für die *work-group* wird der niedrigste Index des Vektors bestimmt und im ersten Wert einer Dreiergruppe im Integer-Array gespeichert. Für jede *work-group* werden Dreiergruppen erzeugt, die aufsteigend nach der *work-group ID* im Integer-Array gespeichert werden. Der höchste Index des Vektors für eine *work-group* wird im zweiten Wert der Dreiergruppe gespeichert. Im dritten Wert wird die Anzahl der Werte gespeichert, die ein *work-item* in den *shared memory* kopieren muss.

6.9.2 Implementierungsdetails

Die Implementierung dieses Kernels wird im weiteren Text mit *BELL V3* bezeichnet. Die *Defines* `#define USE_BELL_V3 ON` und `USE_CENTRAL_DATASTORE ON` müssen für die Verwendung dieser Implementierung aktiviert werden. Die Partitionierung der Daten ist identisch mit der Implementierung *BELL*. In Abbildung 6.16 ist die Partitionierung dargestellt. Die Anzahl der *work-items* wird von der Anzahl der Zeilen der Matrix bestimmt. Die Größe der *work-groups* variiert. Die *work-groups* Größe sollte immer ein Vielfaches von 5 (Anzahl der Zeilen eines BCSR-Blocks) und 32 (Größe eines warps) sein. Die Einträge der Matrix, die Einträge des Vektors, der Blockspaltenindex, die Einträge des Ergebnisvektors, den Blockzeilenindex, das Array mit den Informationen für den *shared memory*, die Anzahl der Blockspalten und die Anzahl der Blockzeilen werden vom Kernel als Übergabeparameter benötigt. Die Einträge des Vektors müssen im Host-Code des Kernels in den *global memory* der GPU geladen werden. Der benötigte Speicher für die Einträge des Ergebnisvektors werden im Host-Code im *global memory* der GPU alloziert und mit einem Kernel werden alle Einträge auf Null gesetzt. Der Kernel *BELL* wird um die Verwendung von *shared memory* erweitert. Im Kernel muss ein Array mit fester Größe definiert werden, dessen Speicher im *shared memory* alloziert wird. Arrays mit dynamischen Größen können in OpenCL nicht erstellt werden. Zuerst muss jede Instanz eines Kernels eine bestimmte Anzahl

von Werten des Vektors in den *shared memory* kopieren. Welche Werte und wie viel Daten jede Kernelinstanz kopieren muss wird während der Initialisierung der Daten berechnet (siehe 6.9.1). Damit sichergestellt werden kann, dass alle Kernelinstanzen das Kopieren der Werte in den *shared memory* abgeschlossen haben, bevor die Berechnungen beginnen, muss eine Synchronisation der *work-items* einer *work-group* nach dem Kopieren stattfinden. Dies wird mit einer *barrier* realisiert, die den *shared memory* überwacht realisiert. Der Zugriff auf die Werte des Vektors während der Multiplikation muss vom *global memory* auf den *shared memory* umgestellt werden. Der niedrigste Index der benötigten Werte des Vektors wird vom Index für den Zugriff auf den Vektor subtrahiert, damit die Adressierung im *shared memory* funktioniert. Das Ergebnis einer Zeile der Matrix wird direkt im Ergebnisvektor gespeichert. Der Ergebnisvektor wird nach der Berechnung der SpMV vom Host-Code aus dem *global memory* GPU ausgelesen und im Speicher des Hosts gespeichert.

6.9.3 Resultat

Für diese Implementierung konnten keine Messungen durchgeführt werden, da der *shared memory* viel zu klein für die Daten von TRACE ist. Unterschiedlich große *work-groups* konnten diese Problem nicht beseitigen. In Tabelle 6.10 sind die benötigten Einträge für unterschiedlich große *work-groups* und unterschiedliche Testfälle dargestellt. Die Tesla-Architektur bietet maximal 16 KB *shared memory*, dass sind maximal 4096 Gleitkommazahlen oder maximal 2048 komplexe Gleitkommazahlen. Die Fermi-Architektur bietet maximal 48 KB *shared memory*, dass sind maximal 12.288 Gleitkommazahlen oder maximal 6144 komplexe Gleitkommazahlen. Der kleinste Testfall benötigt schon bei einer *work-group* Größe von 80 *work-items* 12.415 komplexe Gleitkommazahlen. Auch die Unterteilung der Zeilen der Matrix in Unterabschnitte während der Berechnung, wie dies im Kernel *ELL* realisiert ist, wird nicht weiterhelfen. Dafür sind die benötigten Datenmengen zu groß und würden auch den positiven Effekt des *shared memory* verringern. Die Bereiche des Vektors, die von den unterschiedlichen Zeilen benötigt werden, überlappen sich und würden den positiven Effekt des *shared memorys* vergrößern. Durch die Einführung von Unterabschnitten würden die überlappenden Bereiche verringert oder auch verschwinden. Die Verwendung von *shared memory* für die SpMV ist mit der derzeitigen Hardware nicht realistisch.

6.10 SpMV mit CUDA

Die beste Implementierung der SpMV soll mittels CUDA implementiert werden. Damit soll gezeigt werden, wie viel Performanz durch die zusätzliche Abstraktionsschicht von OpenCL verloren geht. Der Quellcode der Kernel kann unverändert übernommen

Größe einer work-group	Anzahl Werte für Testfall 10x40x20
80	12.415
160	13.375
320	13.555
480	13.735

Tabelle 6.10: Anzahl zu kopierende Werte in den Shared Memory für unterschiedliche work-group Größen.

werden. Es muss nur der Host-Quellcode angepasst werden.

6.10.1 Implementierungsdetails

Die Implementierung dieses Kernels wird im weiteren Text mit *CUDA BELL V4* bezeichnet. Der Kernel *BELL* wird von OpenCL nach CUDA portiert. Der Host-Code muss von den OpenCL-Befehlen auf die CUDA-Befehle umgestellt werden. An der Logik der Implementierung werden keine Änderungen vorgenommen. Am Kernel-Code müssen minimale Änderungen vorgenommen werden. Die Methoden-Deklaration des Kernels muss modifiziert werden. Die *Defines* `#define USE_BELL_V4 ON` und `USE_CENTRAL_DATASTORE ON` müssen für die Verwendung dieser Implementierung aktiviert werden. Der *Defines* `#define USE_OPENCL OFF` muss deaktiviert werden.

6.10.2 Resultat

Diese Implementierung konnte noch mal einen deutlichen Performanzgewinn erzielen. Auf der Tesla-GPU konnte eine Beschleunigung von 1,5 Sekunden gegenüber der Implementierung *BELL* erreicht werden. Auf der Fermi-GPU konnte eine Beschleunigung von 1 Sekunde erreicht werden. Dies zeigt, dass die Abstraktionsschicht von OpenCL ein wenig Performanz kostet. Damit ist fast die Performanz der MPI-Implementierung erreicht. Diese Implementierung ist auf der Tesla-GPU um den Faktor 1,82 langsamer und auf der Fermi-GPU um den Faktor 1,04 langsamer. Die Analyse mit dem *Compute Visual Profiler* zeigt, dass der Multiplikationskernel dieser Implementierung 20 Register benötigt. Nach dem *CUDA Occupancy Calculator* kann die GPU auf der Tesla-Hardware theoretisch nur zu 75% und auf der Fermi-Hardware zu 100% ausgelastet werden. Obwohl keine logischen Änderungen an den Kerneln vorgenommen wurden, wird ein Register mehr benötigt. Dies hat aber keine Auswirkung auf die theoretische Auslastung der GPU.

	10x40x20	10x40x40	20x40x20	10x40x80	10x80x40	40x80x40
Tesla	1,94	3,39	3,23	5,92	6,04	21,06
Fermi	1,28	2,10	2,02	3,44	3,45	-
MPI	0,85	1,67	1,67	3,31	3,31	13,23

Tabelle 6.11: Die Laufzeiten der Testfälle mit dem Kernel *CUDA BELL V4* in Sekunden.

6.11 Probleme

Im Laufe der Implementierung und der Tests sind verschiedene Probleme festgestellt worden. Diese erforderten detaillierte und zeitaufwendige Untersuchungen. Einige verlangsamten den Entwicklungsprozess.

6.11.1 Unterschiedliche Ergebnis-Vektoren

Bei der Auswertung der Simulationserbenisse konnten mehrfach Unterschiede zwischen den Ergebnissen der MPI-Variante und der OpenCL-Variante festgestellt werden. Zum Vergleichen wurden die Ergebnisvektoren in eine Text-Datei geschrieben und mittels eines Diff-Tools verglichen. Die Rundungsregeln beim Überschreiten des Anzeigebereichs einer Gleitkommazahl sorgten für Abweichungen. Auf der GPU werden die Zahlen, die zu klein sind (Unterschreitung des Exponenten -37) auf 0 gerundet. Auf der CPU wird auf die kleinste anzeigbare Zahl gerundet. Ein weiteres Problem wird durch die unterschiedliche Aufsummierung der Gleitkommazahlen bewirkt. Dies wird durch die Verwendung der unterschiedlichen Datenformate notwendig. Ein Validierung der Ergebnisse wird erschwert, da auch bei korrekter Berechnung die Ergebnisse abweichen können. Dies hängt mit der Genauigkeit von Gleitkommazahlen zusammen. Im BCSR-Format werden Zwischensummen aus 5 Ergebnissen gebildet und am Ende bilden alle Zwischensummen einer Zeile das Ergebnis eines Wertes im Ergebnisvektor. Im ELL-Format werden keine oder zumindest weniger Zwischensummen gebildet, dies kann bei Gleitkommazahlen zu unterschiedlichen Ergebnissen führen. Deshalb müssen weitere Methoden zur Validierung der Ergebnisse verwendet werden. Die Verwendung von Gleitkommazahlen mit doppelter Genauigkeit können Indizien für die Richtigkeit des Ergebnisses liefern. Bei Gleitkommazahlen mit doppelter Genauigkeit sollten die Abweichungen im Ergebnis erst wesentlich später in den Nachkommastellen auftreten. Im konkreten Tesfall traten die Abweichungen bei der Verwendung von Gleitkommazahl mit einfacher Genauigkeit nach der 7 Nachkommastelle auf, bei der Verwendung von Gleitkommazahlen mit doppelter Genauigkeit traten die Abweichungen erst an der 15 Nachkommastelle auf.

6.11.2 Große Zeitdifferenzen bei der Block Verarbeitung der MPI-Variante

Die Zeitmessungen für die MPI-Variante des TRACE-Codes zeigten extreme Abweichungen in den Laufzeiten einzelner Blöcke. Die ersten 3 Blöcke, die auf dem System erzeugt wurden, hatten eine bis zu 10-fach längere Laufzeit als die übrigen 5 Blöcke. Die 8 Blöcke der Testfälle sind Identisch, deshalb sollten alle Blöcke ähnliche Laufzeiten aufweisen. Dieses Phänomen konnte auch bei unterschiedlichen Testszenarien mit unterschiedlicher Blockanzahl pro CPU-Kern festgestellt werden. Auch auf unterschiedlichen Mehrkern-Arbeitsplatzrechnern konnte dieses Problem beobachtet werden. Auf einem CPU-Cluster hingegen wurde dieses Problem nicht festgestellt. Dort wurden ähnliche Laufzeiten für alle Blöcke aufgezeichnet. Das Problem muss mit der Hardware der Arbeitsplatzrechners oder mit den Allokationsstrategien der Linux-Kernels zusammenhängen. Die getesteten Systeme besitzen mehrere CPUs, die auf mehrere Sockel verteilt sind und jeweils eigene Arbeitsspeicherbänke besitzen. Eventuell werden die langsameren Blöcke auf dem Arbeitsspeicher der anderen CPU alloziert, und die Datentransfers sind daher wesentlich langsamer.

6.11.3 Fehlende Debugging Möglichkeiten unter Linux

In Linux-Systemen gibt es keine Debugging Möglichkeit für die OpenCL-Kernel (siehe Kapitel 3.6). Dies erschwerte die Fehlersuche erheblich. Die Fehlersuche wurde mit kleinen Beispielmatrizen durchgeführt und dem Programm geDEbugger. Diese Beispiele konnten per Hand nachgerechnet werden und mit dem Ergebnis der Simulation verglichen werden.

7 Zusammenfassung und Diskussion der Ergebnisse

Im Diagramm 7.1 sind die Laufzeiten der unterschiedlichen OpenCL-Kernel auf der *NVIDIA Tesla C1060* dargestellt. Im Diagramm 7.2 sind die Laufzeiten der unterschiedlichen OpenCL-Kernel auf der *NVIDIA Quadro 5000* dargestellt. In den Diagrammen 7.1, 7.2 und der Tabelle 7.1 kann man erkennen, dass die Fermi GPU ihre Hardware Vorteile bei fast allen Implementierungen ausnutzen kann. Nur bei den Kerneln *BCSR 1a* und *BCSR 1b* sind die Laufzeiten auf der Fermi-Hardware schlechter. Es ist gut zu erkennen, dass die verschiedenen Optimierungsansätze zu einer Leistungssteigerung führen.

Ein weiterer Metrik sind die erreichten GFlop/s der unterschiedlichen Implementierungen. Die Anzahl der Rechenoperationen wurde in einem separaten Testlauf gezählt. Für die Implementierungen mit dem BCSR-Datenformat oder dem BELL-Datenformat mit dem Testfall *10x40x80* bei 100 Iterationen wurden $7,548 \cdot 10^{10}$ Re-

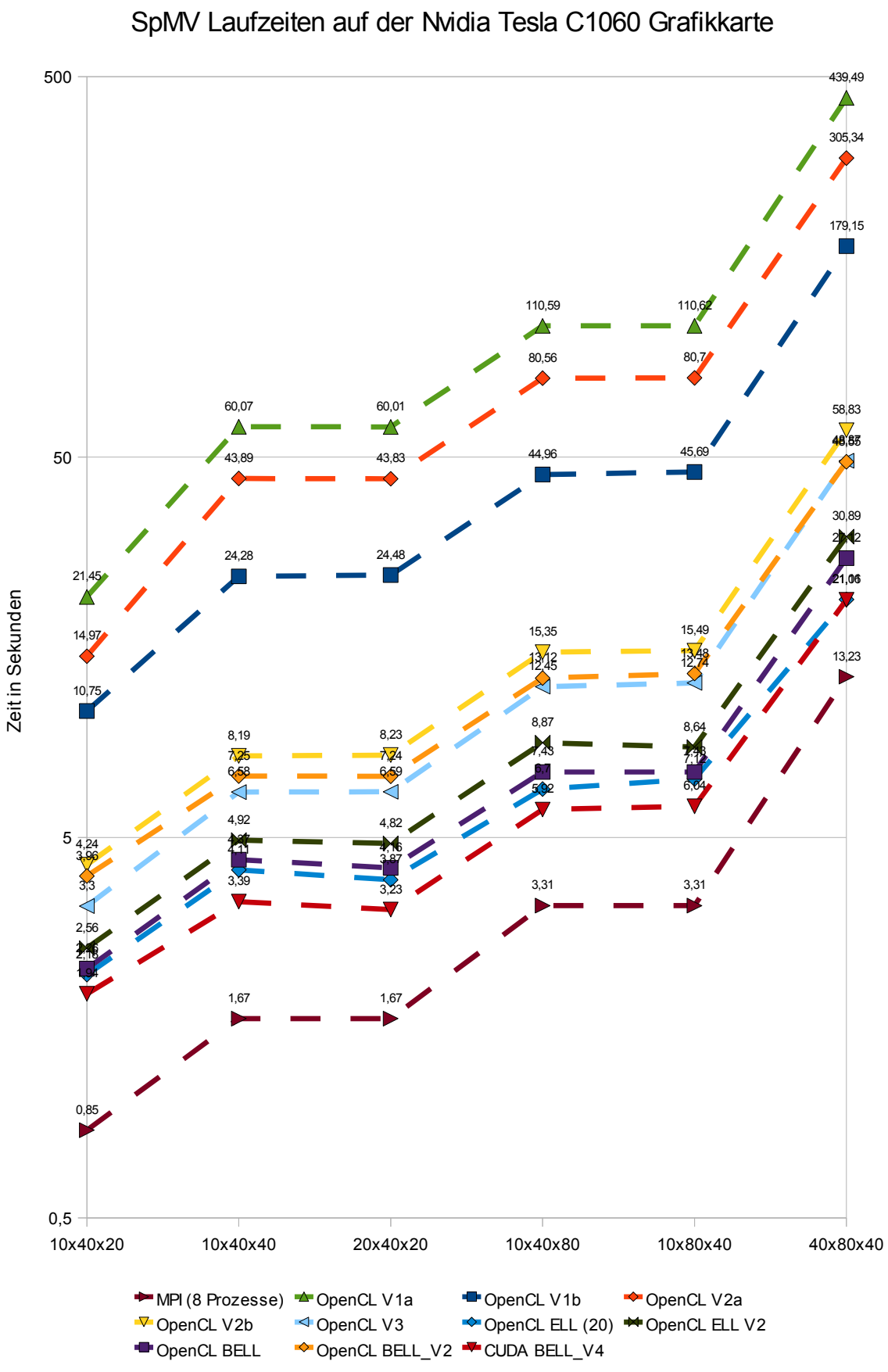

Abbildung 7.1: Vergleich der Laufzeiten aller TRACE-Varianten (Tesla C1060).

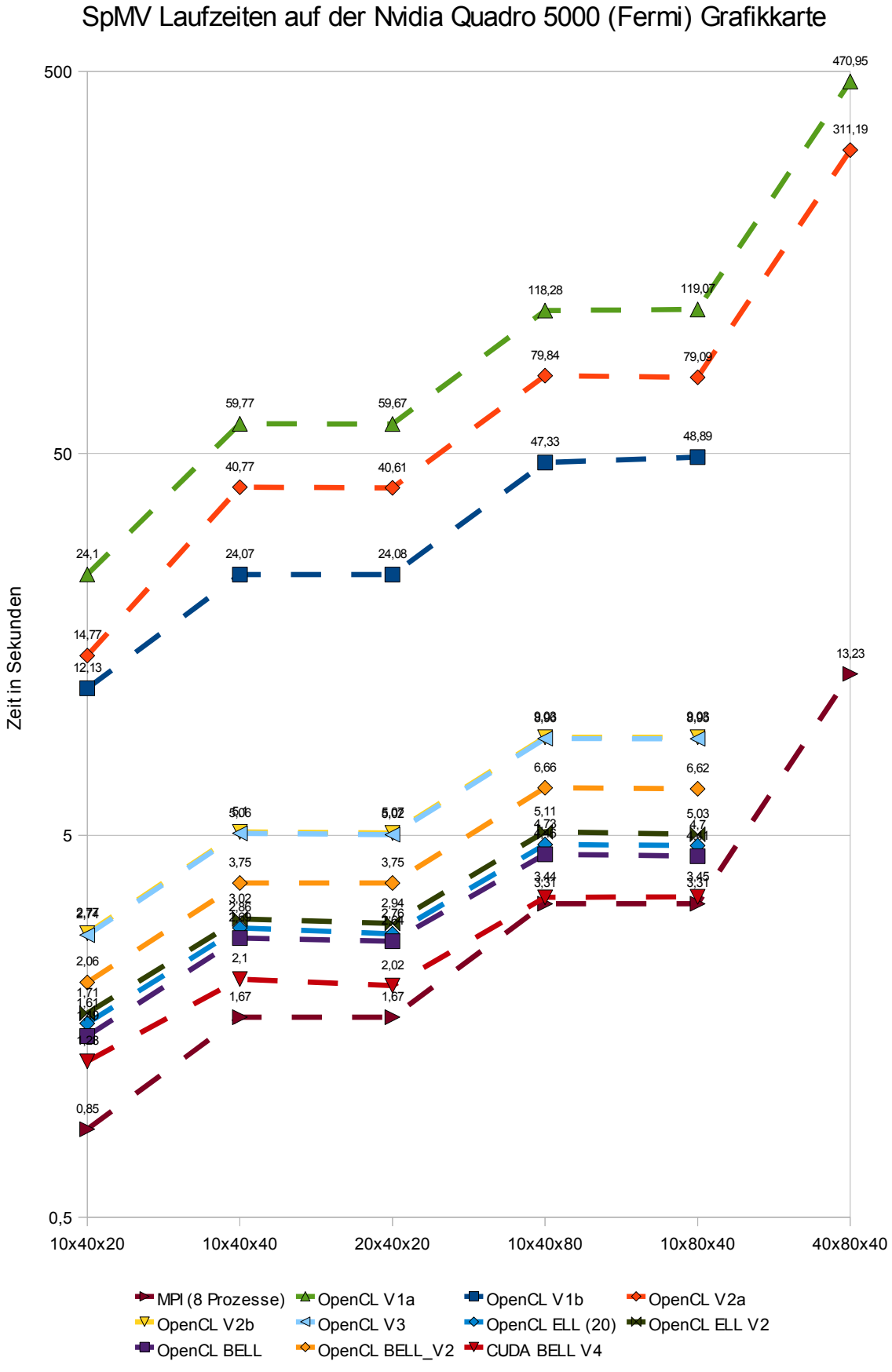

Abbildung 7.2: Vergleich der Laufzeiten aller TRACE-Varianten (Quadro 5000).

	CPU	Tesla C1060	Quadro 5000
MPI	3,31	-	-
OpenCL BCSR V1a	-	110,59	118,28
OpenCL BCSR V1b	-	44,96	47,33
OpenCL BCSR V2a	-	80,56	79,84
OpenCL BCSR V2b	-	15,35	9,03
OpenCL BCSR V3	-	12,45	8,96
OpenCL ELL	-	6,70	4,73
OpenCL ELL V2	-	8,87	5,11
OpenCL BELL	-	7,43	4,46
OpenCL BELL V2	-	13,12	6,66
CUDA BELL V4	-	5,92	3,44

Tabelle 7.1: Die Laufzeiten der Kernel für den Testfall 10x40x80 auf beiden Grafikkarten in Sekunden.

chenoperationen gezählt. Bei den gleichen Bedingungen wurden für die Implementierungen mit dem ELL-Datenformat $6,2672 \cdot 10^{10}$ Rechenoperationen gezählt.

Beim Vergleich der Tabellen 7.1 und 7.2 fällt auf, dass auf der Tesla-Hardware der Kernel *ELL* kürzere Laufzeiten aufweist als der Kernel *BELL*. Die Performanz (GFlops/s) für die Kernel *BELL* sind auf beiden Hardware-Architekturen besser als die Performanz für die Kernel *ELL*. Die Laufzeiten sind bei diesem Vergleich offensichtlich kein Maß für die Performance, da die Anzahl der Operationen verschieden ist.

	CPU	Tesla C1060	Quadro 5000
MPI	22,8 GFlop/s	-	-
OpenCL BCSR V1a	-	0,68 GFlop/s	0,64 GFlop/s
OpenCL BCSR V1b	-	1,68 GFlop/s	1,59 GFlop/s
OpenCL BCSR V2a	-	0,94 GFlop/s	0,95 GFlop/s
OpenCL BCSR V2b	-	4,92 GFlop/s	8,36 GFlop/s
OpenCL BCSR V3	-	6,06 GFlop/s	8,42 GFlop/s
OpenCL ELL	-	9,35 GFlop/s	13,24 GFlop/s
OpenCL ELL V2	-	7,07 GFlop/s	12,26 GFlop/s
OpenCL BELL	-	10,16 GFlop/s	16,92 GFlop/s
OpenCL BELL V2	-	5,75 GFlop/s	11,33 GFlop/s
CUDA BELL V4	-	12,75 GFlop/s	21,94 GFlop/s

Tabelle 7.2: Die erreichten GFlop/s der Kernel auf den beiden Grafikkarten.

In dem Paper [ChoiJ-2010] erreicht Jee W. Choi mit seiner *BCSR*-Implementierung mit den vergleichbaren Testfällen *Ship* und *Harbour* 14 bis 18 GFLOP/s. Mit seiner *BELL*-Implementierung erreicht er 18 bis 29 GFLOP/s. Er verwendet die gleiche Grafikkarte *Tesla C1060*. Wieso konnten keine ähnlichen Ergebnisse erzielt werden? Jee W. Choi verwendet in seinen Implementierungen Blockstrukturen von 2x2 oder 4x4 Gleitkommazahlen. In TRACE werden Blockstrukturen von 5x5 komplexen Gleitkom-

mazahlen verwendet. Diese großen Blockstrukturen wirken sich negativ auf die Performanz aus, dies wurde auch schon im vorherigen Abschnitt diskutiert. Da TRACE mit komplexen Gleitkommazahlen rechnet, müssen pro benötigten Wert doppelt so viele Daten geladen werden wie bei der Implementierung von Jee W. Choi. Dies wirkt sich ebenfalls negativ auf die Performanz aus. Jee W. Choi verwendet CUDA und nicht OpenCL, dies beschleunigt die Implementierung zusätzlich. Durch OpenCL wird eine zusätzliche Abstraktionsschicht eingeführt, die zusätzlich Zeit kostet.

Die beste Performanz liefert der Kernel *CUDA BELL V4* mit 21,94 GFlop/s auf der Fermi-Hardware. Dieser ist damit aber immer noch langsamer als die MPI Implementierung, die 22,8 GFlop/s auf 8 Prozessorkernen erreicht. Damit ist die beste Implementierung um den Faktor 1,04 langsammer als die MPI-Variante; die GPU-implementierung ist nur unwesentlich langsamer als MPI-Implementierung.

Allerdings sehe ich noch Potenzial für weitere Optimierungen an den Kerneln *BELL*, *BELL V2* und *CUDA BELL V4*. Es könnte noch eine Variante mit Unterabschnitten implementiert werden, wie dies beim Kernel *ELL* geschehen ist. Diese Variante konnte aus Zeitgründen leider nicht mehr implementiert werden. Negativ auf die Performanz wirken sich die sehr großen BCSR-Blöcke (5x5 Werte) aus. Im Falle des *BELL V2* Kernel ist die Komplexität so groß geworden, dass die GPU nur noch zu 50% bzw. zu 67% ausgelastet werden kann.

In naher Zukunft sollte TRACE nicht für die SpMV auf OpenCL umgestellt werden. Sollten in Zukunft bessere GPUs zur Verfügung stehen, kann ein Umstieg in Betracht gezogen werden. Allerdings sollten vorher die Performanzmessungen wiederholt werden. Ein Wechsel des Datenformats für die Matrizen müsste dann ebenfalls erfolgen, da für größere Simulationen eine doppelte Datenhaltung nicht zu empfehlen ist.

8 Ausblick

Die Implementierung der Skalarmultiplikationen für den Unterraum des GMRES-Algorithmus verspricht noch eine weitere Performanzsteigerung. Da die Vektoren des Unterrraums auf der GPU berechnet werden, könnten diese für die Berechnung des Unterraums auf der GPU gespeichert werden. Damit würde das Kopieren der Vektoren und der Ergebnisvektoren zwischen Host und GPU entfallen. Dies könnte eine zusätzliche Beschleunigung bewirken. Müssten die Vektoren für die Skalarmultiplikationen auf die GPU kopiert werden, ist kein Performanzgewinn zu erwarten. Der Aufwand für den Datentransfer wäre zu hoch im Vergleich zu den geringen Rechenoperationen, die parallelisiert werden könnten.

Sollten in Zukunft GPUs mit wesentlich mehr *shared memory* zur Verfügung stehen, könnte der Kernel *BELL V3* weiter untersucht werden. Durch die Nutzung des *shared memory* lies sich wahrscheinlich eine gute Performanzsteigerung erzielen.

Die Implementierung *BELL V2* besitzt noch Potenzial für besser Ergebnisse. Mit diesem Kernel kann die Tesla-GPU theoretisch nur zu 50% und die Fermi-GPU theoretisch nur zu 67% ausgelastet werden. Sollten neue GPUs mehr Register pro *work-item* zur Verfügung haben, sollte eine deutliche Beschleunigung des Kernels messbar sein.

Abkürzungsverzeichnis

ALU	**A**rithmetic **L**ogic **U**nit
BCRS	**B**lock **C**ompressed **R**ow **S**torage
BCSR	**B**lock **C**ompressed **S**parse **R**ow
BELL	**B**locked **ELL**pack
BiCGStab	**Stab**ilized **Bi**-**C**onjugate **G**radient
BLAS	**B**asic **L**inear **A**lgebra **S**ubprograms
CFD	**C**omputational **F**luid **D**ynamics
CG	**C**onjugate **G**radient
CGNS	**C**FD **G**eneral **N**otation **S**ystem
COO	**Coo**rdinate format
CPU	**C**entral **P**rocessing **U**nit
CSR	**C**ompressed **S**parse **R**ow
CUDA	**C**ompute **U**nified **D**evice **A**rchitecture
DDR	**D**ouble **D**ata **R**ate
DIA	**Dia**gonal Sparse Matrix
DLR	**D**eutsches Zentrum für **L**uft- und **R**aumfahrt e.V.
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
ECC	**E**rror-**C**orrecting **C**ode
ELL	**ELL**pack-Itpack
FLOPS	**F**loating **P**oint **O**perations **P**er **S**econd
GCC	**G**NU **C**ompiler **C**ollection
GDB	**G**NU **D**e**b**ugger
GDDR	**G**raphics **D**ouble **D**ata **R**ate
GMRES	**G**eneralized **M**inimum **Res**idual
GNU	**G**NU is **n**ot **U**nix
GPGPU	**G**eneral-**P**urpose Computation on **G**raphics **P**rocessing **U**nit
GPU	**G**raphics **P**rocessing **U**nit
HPC	**H**igh **P**erformance **C**omputing
I/O	**I**nput/**O**utput
ILU	**I**ncomplete **LU**-Decomposition
ILU(p)	**ILU** with **Level of Fill**
ILUT	**ILU** with **T**hreshold
MILU	**M**odified **ILU**
MPI	**M**essage **P**assing **I**nterface
NVCC	**NV**IDIA **C**UDA **C**ompiler Driver
OpenCL	**Open C**omputing **L**anguage
OpenGL	**Open G**raphics **L**ibrary

RAM	**R**andom **A**ccess **M**emory
RANS	**R**eynolds-**A**veraged-**N**avier-**S**tokes
SATA	**S**erial **A**dvanced **T**echnology **A**ttachment
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
SpMV	**Sp**arse-**M**atrix **V**ector multiplication
SSOR	**S**ymmetric **S**uccessive **O**ver-**R**elaxation
STL	**S**tandard **T**emplate **L**ibrary
TDR	**T**imeout **D**etection and **R**ecovery
TRACE	**T**urbo machinery **R**esearch **A**erodynamic **C**omputational **E**nvironment
URANS	**U**nsteady **R**eynolds-**A**veraged-**N**avier-**S**tokes
VBCSR	**V**ariable **B**lock **C**ompressed **S**parse **R**ow
ViennaCL	**Vienna C**omputing **L**ibrary

Literatur

[Amdah-1967] Amdahl, Gene: *Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities*. In: AFIPS Conference Proceedings Volume 30, American Federation of Information Processing Societies, 1967 Spring Joint Computer Conference, Atlantic City USA, 1967, S. 483–485.

[CGNSS-2011] CGNS Steering Committee: *CFD General Notation System: What is CGNS?*. http://cgns.sourceforge.net/WhatIsCGNS.html, Januar 2011.

[ChoiJ-2010] Choi, Jee W.; et al.: *Model-driven Autotuning of Sparse Matrix-Vector Multiply on GPUs*. In: Proceedings of the 2010 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Association for Computing Machinery, 2010 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Bangalore India, 2010, Seite 115-126.

[Garla-2010] Garland, Michael; Bell, Nathan: *Cusp-libary: Generic Parallel Algorithms for Sparse Matrix and Graph Computations*. http://code.google.com/p/cusp-library/, Oktober 2010.

[Graph-2010] Graphic Remedy: *gDEBugger CL*. http://www.gremedy.com/gDEBuggerCL.php, Dezember 2010.

[Haken-2011] Hakenesch, Peter R. : *Aerodynamik des Flugzeugs: Numerische Strömungssimulation*. http://www.lrz.de/~hakenesch/aerodynamik/K3_Folien.pdf, Januar 2011.

[Hicke-2011] Hickel, Stefan: *Angewandte Strömungssimulation: Gittergenerierung*. http://www.aer.mw.tum.de/fileadmin/tumwaer/www/pdf/lehre/angewandte_cfd/V4.pdf, Januar 2011.

[Hober-2010] Hoberock, Jared; Bell, Nathan: *Thrust: A Parallel Template Library*. http://code.google.com/p/thrust/, Oktober 2010.

[Kersk-2010] Kersken, Hans-Peter; et al.: *Time-linearized and Time-accurate 3D RANS Methods for Aeroelastic Analysis in Turbomachinery*. In: Proceedings of ASME Turbo Expo 2010: Power for Land, Sea and Air, American Society of Mechanical Engineers, ASME Turbo Expo 2010, Glasgow United Kingdom, 2010.

[Kersk-2011] *Zur Verfügung gestellt von Dr. rer. nat. Hans-Peter Kersken.*

[Khron-2010] Khronos OpenCL Working Group: *The OpenCL Specification*. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf, Version 1.1 Revision 33, Oregon USA, Juni 2010.

[Khron-2010a] Khronos Group: *OpenGL: The Industry's Foundation for High Performance Graphics*. http://www.opengl.org/, Oktober 2010.

[KirkD-2010] Kirk, David B.; Hwu, Wen-mei W.: *Programming Massively Parallel Processors: A Hands-on Approach*. Morgen Kaufmann Publisher, Burlington USA, 2010.

[Micro-2010] Microsoft Corporation: *DirectX Developer Center*. http://msdn.microsoft.com/en-us/directx/default, Oktober 2010.

[Micro-2011] Microsoft Corporation: *Timeout Detection and Recovery of GPUs through WDDM*. http://www.microsoft.com/whdc/device/display/wddm_timeout.mspx, Januar 2011.

[Moore-2011] Moore, Ronald: *Parallel and Distributed Computing: Introduction*. http://www.fbi.h-da.de/index.php?eID=tx_nawsecuredl&u=0&file=fileadmin/personal/r.moore/ParallelnDistributed/I-Introduction-20101026.pdf&t=1295429116&hash=46ccce468161d3f1492f0edb309dff45, Januar 2011.

[NVIDI-2010] NVIDIA Corporation: *Compute Visual Profiler: User Guide*. http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/VisualProfiler/Compute_Visual_Profiler_User_Guide.pdf, Oktober 2010.

[NVIDI-2011] NVIDIA Corporation: *CUDA GPU Occupancy Calculator*. http://developer.download.nvidia.com/compute/cuda/3_2_prod/sdk/docs/CUDA_Occupancy_Calculator.xls, Januar 2011.

[NVIDI-2011a] NVIDIA Corporation: *NVIDIA Parallel Nsight Power of GPU Computing: Simplicity of Visual Studio*. http://www.nvidia.com/object/parallel-nsight.html, Januar 2011.

[NVIDI-2011b] NVIDIA Corporation: *NVIDIA OpenCL Best Practices Guide*. http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/OpenCL_Best_Practices_Guide.pdf, Januar 2011.

[NVIDI-2011c] NVIDIA Corporation: *OpenCL Programming Guide for the CUDA Architecture*. http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/OpenCL_Programming_Guide.pdf, Januar 2011.

[NVIDI-2011d] NVIDIA Corporation: *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. http://www.nvidia.de/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, April 2011.

[NVIDI-2011e] NVIDIA Corporation: *NVIDIA OpenCL Best Practices Guide*. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf, März 2011.

[NVIDI-2011f] NVIDIA Corporation: *OpenCL Programming Guide for the CUDA Architecture*. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf, März 2011.

[RuppK-2010] Rupp, Karl: *ViennaCL*. http://viennacl.sourceforge.net, Oktober 2010.

[SaadY-2003] Saad, Yousef: *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia USA, 2003

[TOP50-2011] TOP500 Project: *Top 500 Supercomputer Site - November 2010*. http://www.top500.org/lists/2010/11, April 2011.