Hochschule für Technik, Wirtschaft und Kultur Leipzig
Fakultät für Informatik, Mathematik und Naturwissenschaften

Masterarbeit

GPU-basierte Beschleunigung von MapReduce am Beispiel von OpenCL und Hadoop

Name:	Christof Pieloth
Matrikel-Nr.:	52674
Studiengang:	Informatik

Leipzig, den 31. Januar 2012

Betreuer: Prof. Dr.-Ing. Axel Schneider (HTWK Leipzig)
Dipl.-Inf. (FH) Marko Bauhardt (Datameer GmbH)

Abstract

Apache Hadoop ist ein Cluster-System zur verteilten und parallelen Verarbeitung von riesigen Datenmengen. Neben zahlreichen Funktionen stellt es ein MapReduce-Framework und ein verteiltes Dateisystem bereit, mit Hilfe dessen Programme (Jobs) zur Datenverarbeitung programmiert werden können. Für einfache Jobs genügt es bereits, die zwei Methoden `map` und `reduce` zu implementieren.

Open Computing Language (OpenCL) ist eine Programmierplattform mit zugehöriger Programmiersprache zur Programmierung paralleler Hardware wie Mehrkernprozessoren oder Grafikprozessoren (GPU). Wenn die Verarbeitung mit Hilfe von OpenCL auf Grafikprozessoren stattfindet, fällt dies in den Bereich von General Purpose Computation on Graphics Processing Unit (GPGPU). Somit wird die Grafikkarte neben ihrer Hauptaufgabe, der Berechnung zur Bildausgabe, für allgemeine Berechnungen genutzt, die zuvor nur von Hauptprozessoren ausgeführt werden konnten. Bei geeigneten Problemen und unter Verwendung paralleler Algorithmen kann mit Grafikprozessoren eine enorme Steigerung der Rechenleistung im Vergleich zu Hauptprozessoren erzielt werden.

Die vorliegende Arbeit beschreibt verschiedene Möglichkeiten, wie ein MapReduce-Job durch eine Grafikkarte beschleunigt werden kann. Anhand des k-Means-Verfahrens zur Clusteranalyse und einer numerischen Integration wird die Parallelisierung eines Tasks praktisch untersucht. Dabei werden die Berechnungen der `map`- oder `reduce`-Methode vom Grafikprozessor ausgeführt. Die Implementierungen verwenden die JNA-Bibliothek JavaCL zur Anbindung der OpenCL-API in Java. Am aufwendigsten ist die Entwicklung einer Speicherverwaltung zur Datenübertragung zwischen Haupt- und Grafikkartenspeicher. Eine Übertragung großer zusammenhängender Daten durch einen Puffer hat sich als performante Lösung bewährt. Die zwei Beispiel-Jobs mit GPU-Beschleunigung erreichten einen Speedup von 6,5 bis 7,5 im Vergleich zu der Implementierung ohne OpenCL.

Ich möchte mich an dieser Stelle bei all denen bedanken, die mich bei der Anfertigung meiner Masterarbeit unterstützt haben. In erste Linie sind meine Betreuer sowie die Mitarbeiter der Datameer GmbH zu erwähnen, die mir in fachlichen und technischen Fragen stets zur Seite standen. Für das Korrekturlesen dieser Arbeit gilt mein Dank Franziska Pieloth und Heike Thriene.

Ganz besonders bedanken möchte ich mich bei meiner Familie, die mich in allen bisherigen Vorhaben meines Lebens mit aller Kraft unterstützt hat.

Inhaltsverzeichnis

1. Einleitung

1.1. Gegenstand der Untersuchung

Die vorliegende Arbeit untersucht Themen der Parallelverarbeitung am Beispiel der Technologien Apache Hadoop und OpenCL. Die Programmiermodelle und Programmierschnittstellen (API) werden dabei genauer analysiert, um anhand dieser Untersuchungen mögliche Lösungsansätze für die Beschleunigung von Hadoop durch OpenCL zu erarbeiten.

1.1.1. Apache Hadoop

Apache Hadoop ist ein freies Cluster-System zur verteilten und parallelen Verarbeitung großer Datenmengen. Als Computercluster wird ein Verbund aus vernetzten Computern bezeichnet, über den die Rechenkapazität oder Verfügbarkeit im Vergleich zu einem einzelnen Computer erhöht werden soll. Die Entwicklung von Hadoop begann im Jahr 2005 und es ist seit Januar 2008 ein offizielles Projekt der Apache Software Foundation. Die Hauptbestandteile von Hadoop sind das verteilte Dateisystem HDFS und das MapReduce-Framework.

HDFS ist das Standarddateisystem von Hadoop und ermöglicht es große zusammenhängende Dateien zu speichern. Dafür werden die Dateien in Datenblöcke aufgeteilt und auf unterschiedliche Computer verteilt. Um eine hohe Verfügbarkeit beim Ausfall eines Computers sicherzustellen, werden alle Datenblöcke redundant im Cluster gespeichert.

MapReduce ist ein paralleles Programmiermodell für Berechnungen über große Datenmengen auf einem Computercluster. Hadoop stellt eine eigene Implementierung bereit, um damit Aufgaben für ein Hadoop-Cluster zu programmieren. Die Datenverarbeitung in MapReduce erfolgt in zwei aufeinanderfolgenden Phasen: Map und Reduce. Dabei bilden Key-Value-Paare die zentrale Datenstruktur in diesem Programmiermodell. Die Map-Phase erhält ein Key-Value-Paar als Eingabe und liefert nach der Verarbeitung ein oder mehrere Key-Value-Paare als Ausgabe. Anhand der Ausgabe der Map-Phase bildet das Framework Gruppen. Alle Paare mit gleichem Key werden einer Gruppe zugeordnet. Diese Gruppen werden anschließend von der Reduce-Phase verarbeitet, die ein oder mehrere Key-Value-Paare als Ausgabe liefert. In beiden Phasen können mehrere Key-Value-Paare bzw. mehrere Gruppen parallel berechnet werden.

1.1.2. Open Computing Language

Open Computing Language (OpenCL) ist ein offener Standard der Khronos Group zur Programmierung verschiedenartiger Parallelrechner wie Mehrkern- oder Grafikprozessoren (GPU). Die Spezifikation der ersten Version wurde im Dezember 2008 veröffentlicht. Die aktuelle Version 1.2 ist vom November 2011. Die Programmierung der Parallelrechner erfolgt mit der dazugehörigen OpenCL-API und Programmiersprache OpenCL C. Um auf einem Parallelrechner eine OpenCL-Anwendung ausführen zu können, werden für die Hardware OpenCL-Treiber vom Hersteller benötigt.

Da mit OpenCL auch Grafikkarten programmiert werden können, wird es der Technologie General Purpose Computation on Graphics Processing Unit (GPGPU) zugeordnet. Mit GPGPU können auf Grafikprozessoren „allgemeine Berechnungen" ausgeführt werden, die bisher nur von Hauptprozessoren verarbeitet werden konnten. Grafikprozessoren

bieten aufgrund der Hardwareentwicklung einen vielfach höheren Grad an Parallelität der Rechenoperationen als Mehrkernprozessoren. Bei geeigneten Problemen und unter Verwendung paralleler Algorithmen kann so eine enorme Steigerung der Rechenleistung im Vergleich zu Hauptprozessoren erzielt werden.

Allgemein besteht eine GPGPU-Anwendung aus einem Host- und einem Kernel-Teil. Der Host-Teil wird vom Hauptprozessor verarbeitet und stellt den Zugriff auf die Grafikkarte sicher. Der Kernel-Teil implementiert den Algorithmus für den Grafikprozessor. Dieser nutzt überwiegend die Datenparallelität aus, d.h. derselbe Code wird auf unterschiedlichen, voneinander unabhängigen Daten ausgeführt. Die Berechnungen werden zur Laufzeit von sogenannten Work-Items ausgeführt, die auf die Recheneinheiten verteilt werden. Ein Work-Item stellt dabei einen logischen Ausführungspfad dar und ist mit einem Thread vergleichbar. Um die Grafikkarte optimal ausnutzen zu können, muss außerdem das Speichermodell mit unterschiedlich schnellen und großen Speichern berücksichtigt werden.

1.2. Relevanz und Anlass der Untersuchung

Der Bedarf nach immer höherer Rechenleistung bleibt ungebrochen – insbesondere in Wirtschaft und Forschung. In der Forschung wird die Rechenleistung zum Beispiel für aufwendige Probleme der Mathematik und Simulationen naturwissenschaftlicher Phänomene benötigt. Die Wirtschaft verarbeitet häufig große Datenmengen zur Geschäftsanalyse oder zur Aufstellung von Prognosen. Eine höhere Rechenleistung ermöglicht es zum einen, Probleme in kürzerer Zeit zu lösen, und zum anderen, die Genauigkeit der Ergebnisse durch Zunahme weiterer Parameter zu erhöhen.

Die Taktfrequenz als ein Maß der Rechenleistung von Prozessoren stagniert seit der Jahrtausendwende nahezu. Das führte zu der Entwicklung paralleler Hardware wie Mehrkernprozessoren und Mehrprozessorsystemen. Wenn die Leistung eines einzelnen Computers nicht erhöht werden kann, besteht außerdem die Möglichkeit, mehrere Computer in Form eines Rechnerverbunds zur Leistungssteigerung zu nutzen. Computercluster sind in vielen Bereichen die bevorzugte Lösung, da die Rechenleistung durch Zunahme neuer Computer in den Verbund schnell und einfach erhöht werden kann. Ein Nachteil dieser Systeme ist jedoch der hohe Platz- und Energiebedarf.

Neue Hoffnungen den Bedarf zu stillen, liegen in der vergleichsweise jungen Technologie General Purpose Computation on Graphics Processing Unit (GPGPU). Diese Erwartungen sind in der massiven Parallelität der Grafikkarten begründet. Bisher werden spezielle Grafikkarten für das High Performance Computing überwiegend in Workstations eingesetzt. Doch auch im Bereich der Supercomputer hat die Beschleunigung durch Grafikkarten Einzug gehalten. In der Liste der 500 schnellsten Supercomputer vom November 2011 belegt das System „Tianhe-1A" mit über 7.000 Grafikkarten den zweiten Platz.[1] Mit vergleichsweise geringen Anschaffungskosten kann bei Computerclustern ein hoher Leistungszuwachs durch die Verwendung von Grafikprozessoren erreicht werden. Zusätzlich steigt der Platzbedarf nicht an, da vorhandene Computer erweitert werden anstatt neue hinzuzufügen. Allerdings sind nicht alle Probleme für die Verarbeitung auf der Grafikkarte geeignet und der Programmieraufwand ist im Vergleich zu verbreiteten Programmiermodellen[2] höher.

Ein immer häufiger genutztes Cluster-System zur Datenverarbeitung ist Hadoop. Die Stärken sind ein einfaches Programmiermodell, hohe Skalierbarkeit und freie Verfügbarkeit. Dank der hohen Speicherkapazität von modernen Festplatten kann bei geringen Anschaffungskosten bereits mit sehr wenigen Computern ein großer verteilter Speicher bereitgestellt werden. Im Gegensatz dazu ist die Steigerung der Rechenleistung sehr teu-

[1] `http://i.top500.org/system/176929`

[2] Zum Beispiel das Message Passing Interface oder MapReduce.

er, da diese auf dem herkömmlichen Weg nur unter Zunahme weiterer Computer erhöht werden kann. Die Anschaffung neuer Computer lässt sich unter gewissen Voraussetzungen vermeiden, wenn die Rechenleistung durch die Nutzung von Grafikkarten gesteigert wird.

Hadoop und OpenCL folgen unterschiedlichen Ansätzen in der Datenverarbeitung und Programmierung, deren Zusammenwirken nur wenig untersucht wurde. Bisherige Arbeiten erforschen überwiegend, wie die GPU-Programmierung durch das Programmiermodell MapReduce vereinfacht werden kann. Die in diesen Untersuchungen entwickelten MapReduce-Implementierungen sind aber speziell auf GPGPU angepasst. Im Gegensatz dazu stellt Hadoop eine vollständige Implementierung von MapReduce zur Verfügung, in die eine Grafikkarte eingebunden werden muss. Daher lassen sich nur wenige Erkenntnisse dieser Arbeiten auf Hadoop anwenden.

1.3. Zielsetzung

Das Ziel dieser Arbeit ist die Steigerung der Rechenleistung eines Hadoop-Clusters mit Hilfe einer Grafikkarte unter Nutzung von OpenCL. Als Maß der Rechenleistung dient die Ausführungszeit eines MapReduce-Jobs. Weiterhin ist der Programmieraufwand abzuschätzen, der benötigt wird, um eine Grafikkarte in das MapReduce-Framework und die Hadoop-API einzubinden.

Um mögliche Probleme und Einschränkungen zu erkennen, sind die Eigenschaften und Prinzipien beider Technologien zu untersuchen. Besonderes Interesse gilt der Datenorganisation von MapReduce im Vergleich zu GPGPU sowie den unterschiedlichen Programmiersprachen von Hadoop (Java) und OpenCL (C). Die Untersuchungen dienen als Grundlage, um verschiedene Lösungswege und Parallelisierungsstrategien zu suchen, die in Hinblick auf Machbarkeit, Aufwand und erwarteten Leistungszuwachs zu analysieren sind.

Der tatsächliche Programmieraufwand und Leistungszuwachs eines Lösungsweges soll anhand einer Beispielimplementierung praktisch untersucht werden. Eine abschließende Laufzeitanalyse ermittelt den Speedup und mögliche Faktoren, die die Performance negativ beeinflussen können.

1.4. Aufbau der Arbeit

Das Kapitel 2 führt den Begriff des parallelen Rechnens ein. Daneben werden wichtige Grundbegriffe und Prinzipien erläutert. Mittels dieser Grundlagen werden die Technologien Hadoop und OpenCL genauer vorgestellt.

Anschließend analysiert das Kapitel 3 die Eigenschaften und Konzepte der zwei Technologien, um diese in Beziehung zueinander zu setzen. Dabei dienen sowohl praktische Untersuchungen als auch theoretische Überlegungen zur Findung möglicher Lösungsansätze. Mit Hilfe der gewonnenen Erkenntnisse wird ein Lösungsansatz für eine Beispielimplementierung ausgewählt.

Die Kapitel 4 und 5 untersuchen den gewählten Ansatz anhand von zwei Beispiel-Jobs: dem k-Means-Verfahren zur Clusteranalyse und der numerische Integration. Jeder Algorithmus wird zu Beginn in Hinblick auf seine Parallelisierungsmöglichkeiten analysiert. Die Implementierungen und deren Einschränkungen werden mit UML-Diagrammen sowie einigen Quelltextauszügen vorgestellt. Eine abschließende Laufzeitanalyse ermittelt den praktischen Performancegewinn.

Am Ende dieser Arbeit werden die gewonnenen Erkenntnisse in Kapitel 6 zusammengefasst. Ein Ausblick diskutiert weitere interessante Aspekte, die aus offenen Fragen entstanden sind oder nicht genauer behandelt wurden. Außerdem werden aktuelle technische Entwicklungen aufgefasst und im Kontext dieser Arbeit betrachtet.

Bei der Beschreibung von Quelltexten werden spezielle Schreibweisen verwendet, um die Übersicht zu verbessern. Anweisungen und Programmausgaben sind in `Maschinenschrift` dargestellt. Längere Passagen sind vom Fließtext gelöst und grau unterlegt:

```
echo 'hello world' >> hello.txt
cat hello.txt
```

Die Syntax des verwendeten Pseudocodes orientiert sich an der Programmiersprache Python. In dieser werden Code-Blöcke nicht durch geschweifte Klammern, sondern durch Einrückungen gebildet.

Des Weiteren werden bei der allgemeinen Erklärung von Befehlen Variablen genutzt. Diese sind durch spitze Klammern gekennzeichnet und müssen bei der Anwendung durch einen entsprechenden Wert ersetzt werden. Der Name der Variable beschreibt deren Funktion. Zum Beispiel benötigt der Befehl zum Erstellen eines Ordners den Ordnernamen als Argument:

```
mkdir <directoryname>
```

Um einen Ordner mit dem Namen „foo" zu erstellen, muss die Variable `<directoryname>` durch `foo` ersetzt werden:

```
mkdir foo
```

Werte in eckigen Klammern sind mögliche Optionen und können entweder gesetzt oder nicht gesetzt werden, wobei die Klammern bei Verwendung nicht mit angegeben werden.

Die Arbeit ist als digitales Dokument auf der beiliegenden CD-ROM enthalten. Neben dem Quelltext befinden sich weitere Daten auf der CD-ROM, die im Anhang B aufgelistet sind.

2. Grundlagen

Dieses Kapitel erläutert wichtige Grundlagen mit Bezug auf die genutzten Technologien. In Abschnitt 2.1 wird der Begriff des parallelen Rechnens eingeführt und eine mögliche Klassifikation von Parallelrechnern vorgestellt. Anschließend werden das Cluster-Computing und das „General Purpose Computing on Graphics Processing Unit" genauer erklärt. Parallele Programme werden anders als sequenzielle Programme entwickelt, wodurch die für diese Arbeit wichtigen Programmiermodelle der parallelen Programmierung vorgestellt werden.

Nach den allgemeinen Grundlagen des parallelen Rechnens werden die Technologien Hadoop und OpenCL genauer erklärt und in die Welt der Parallelrechner eingeordnet.

2.1. Paralleles Rechnen

Um die Berechnungszeit eines Problems zu verkürzen, gibt es grundsätzlich drei Möglichkeiten:

1. Steigerung der Rechenleistung z.B. durch schnellere Prozessoren,
2. Optimierung der eingesetzten Algorithmen oder Einsatz besserer Algorithmen und
3. Nutzung von mehr als einem Rechnersystem zur Leistungssteigerung.

Die anhand der Taktfrequenz gemessene Leistungsgrenze moderner Mikroprozessoren wurde bereits kurz nach der Jahrtausendwende allmählich erreicht. Somit kann die Abarbeitungszeit von Problemen nicht mehr allein durch schnellere Prozessoren verkürzt werden. Dennoch steigen die Anforderungen an die Rechenleistung täglich, wie zum Beispiel durch den Wunsch nach genaueren Simulationen oder der Verarbeitung höherer Datenmengen, weiter an. Da auch Algorithmen nicht unendlich optimiert werden können, bleibt zur Leistungssteigerung nur die Nutzung von mehr als einem Rechnersystem übrig.

Hierbei ist die Grundidee, dass zwei parallele Recheneinheiten mit der Taktfrequenz f dieselbe Arbeit leisten, wie eine Recheneinheit mit der Taktfrequenz $2f$. Dazu muss der sequenzielle Befehlsstrom eines Prozessors in mehrere Befehlsströme aufgeteilt werden, die dann gleichzeitig und somit parallel, koordiniert zusammenarbeiten. Daraus kann der Begriff des Parallelrechners definiert werden [RR07, S.17]:

Ein Parallelrechner ist eine Ansammlung von Berechnungseinheiten (Prozessoren), die durch koordinierte Zusammenarbeit große Probleme schnell lösen können.

Das Lösen von Problemen mit Hilfe von Parallelrechnern wird als paralleles Rechnen bezeichnet. Dabei werden verschiedene Ziele verfolgt. Einerseits können Probleme in einer kürzeren Ausführungszeit erledigt werden, als dies durch eine Ausführung auf einer sequenziellen Recheneinheit möglich wäre. Andererseits kann dieses Potenzial auch dazu genutzt werden, um in derselben Zeit größere Aufgabenstellungen zu lösen, die zu genaueren Ergebnissen führen. Des Weiteren können einige Parallelrechner Probleme lösen, die von einer einzelnen Recheneinheit mit begrenzten Speicher nicht bearbeitet werden können. Dies sind vor allem Problemstellungen mit so großen Datenmengen, die eine Recheneinheit nicht speichern kann. Hierbei werden die Daten über geeignete Kommunikationswege auf die einzelnen Recheneinheiten verteilt. [RR07, S. 4]

2.1.1. Klassifikation von Parallelrechnern

Parallelrechner können anhand verschiedener Merkmale klassifiziert werden. Die Flynnsche Klassifizierung unterteilt Rechnerarchitekturen anhand der Befehls- und Datenströme in vier Klassen [RR07, S.18]:

SISD Single Instruction Single Data

MISD Multiple Instruction Single Data

SIMD Single Instruction Multiple Data

MIMD Multiple Instruction Multiple Data

Jeder dieser Klassen ist ein idealisierter Modellrechner zugeordnet, an dem das Funktionsprinzip erläutert wird.

Der **SISD-Modellrechner** entspricht dem klassischen von-Neumann-Rechnermodell mit einer Verarbeitungseinheit, einem Datenspeicher und einem Programmspeicher. Zu Beginn eines Verarbeitungsschritts wird ein Befehl, auch Instruktion genannt, aus dem Programmspeicher geladen und dekodiert. Anschließend werden benötigte Daten aus dem Datenspeicher geladen. Nun wird die Instruktion auf den geladenen Daten ausgeführt und das Resultat in den Datenspeicher zurückgeschrieben. Mit der Verarbeitung einer Instruktion wird auch nur ein Ergebnis berechnet.

Bei dem **MISD-Modellrechner** gibt es mehrere Verarbeitungseinheiten mit jeweils einem Programmspeicher, aber nur einem gemeinsamen Datenspeicher. Somit werden bei einem Verarbeitungsschritt unterschiedliche Instruktionen auf denselben Daten ausgeführt. Wenn ein Ergebnis in den Datenspeicher zurückgeschrieben werden soll, muss dessen Wert mit dem der anderen Recheneinheiten übereinstimmen. Aufgrund dieser Einschränkung gibt es eigentlich kein praktisches Rechnermodell, das in diese Klasse einzuordnen wäre. Einzig fehlertolerante Systeme können theoretisch diesem Prinzip zugeordnet werden.

Dem **SIMD-Modellrechner** kann ein Vektorrechner zugeordnet werden. Dieser besteht aus mehreren Verarbeitungseinheiten, mit jedoch nur einem Datenspeicher und einem Programmspeicher. In einem Verarbeitungsschritt erhält jede Verarbeitungseinheit die gleiche Instruktion aus dem Programmspeicher. Anschließend wird für jede Verarbeitungseinheit ein separates Datum aus dem Datenspeicher geladen. Nun werden die Instruktionen von den Verarbeitungseinheiten synchron und parallel ausgeführt. Abschließend werden die Ergebnisse in den Datenspeicher zurückgeschrieben. Nach der Verarbeitung einer Instruktion werden mehrere Ergebnisse berechnet.

Mehrkernprozessoren entsprechen dem **MIMD-Modellrechner**. Dieser hat einen Datenspeicher und mehrere Verarbeitungseinheiten von der jede einen eigenen Programmspeicher besitzt. Bei diesem Modellrechner können mehrere unabhängige Instruktionen asynchron und parallel auf unterschiedlichen Daten ausgeführt werden. Jede Verarbeitungseinheit kann unterschiedliche Resultate liefern und in den Datenspeicher zurückschreiben.

Gegenüber MIMD-Rechner haben SIMD-Rechner aufgrund von nur einem Befehlsstrom und der synchronen Abarbeitung den Vorteil, dass auf Programmebene keine Synchronisation erforderlich ist und somit diese einfacher zu programmieren sind. Jedoch ist dieses Berechnungsmodell nicht uneingeschränkt auf alle Probleme anwendbar. Ein praktischer Vorteil der MIMD-Rechner ist, dass das zugrunde liegende Berechnungsmodell weniger Einschränkungen besitzt. Auch wenn MIMD-Rechner aufgrund der erforderlichen Synchronisation einen höheren Programmieraufwand haben, sind Prozessoren bzw. Parallelrechner dieser Klasse am weitesten verbreitet. Dennoch wird das SIMD-Prinzip von vielen Parallelrechnern unterstützt.

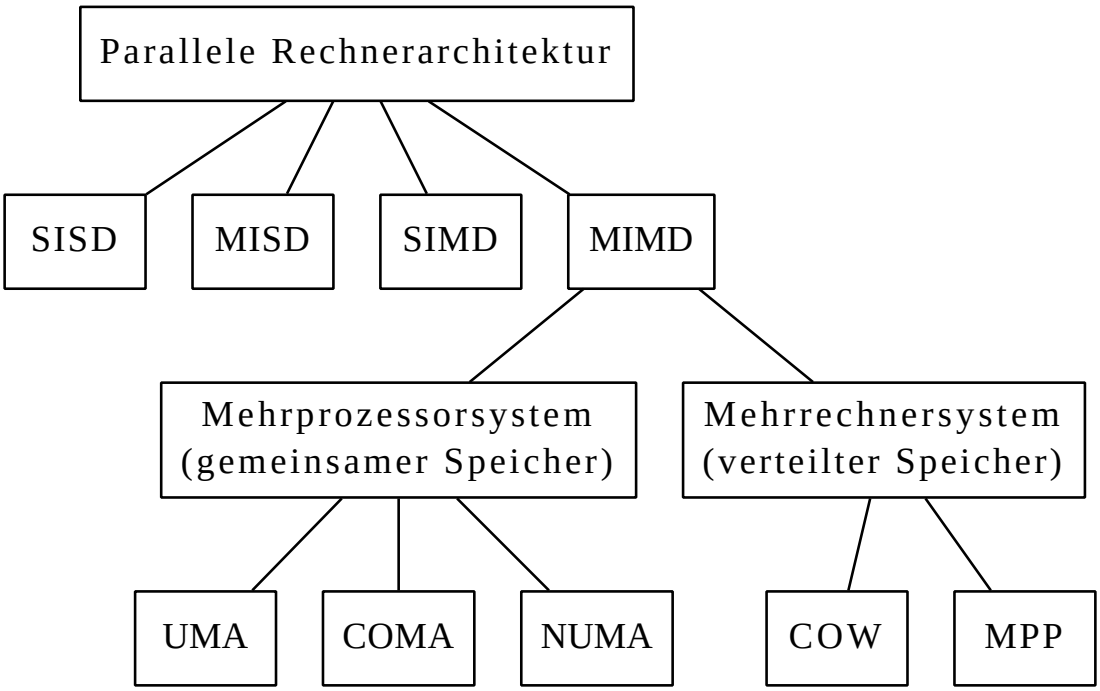


Abbildung 2.1.: Klassifikation von Rechnerarchitekturen nach Flynn und Tanenbaum

MIMD-Rechner existieren in vielen verschiedenen Ausprägungen, die anhand ihrer Speicherorganisation weiter unterteilt werden können. Grundsätzlich wird zwischen Rechnern mit gemeinsamen und verteilten Speicher unterschieden. Wie in Abbildung 2.1 zu sehen, werden diese Klassen wiederum unterteilt [TG99, S. 641]. Bei einem Rechnersystem mit gemeinsamen Speicher können alle Prozessoren über einen gemeinsamen Speicheradressraum auf einen globalen Speicher zugreifen. Der Datenaustausch und die Kommunikation bzw. Synchronisation der Prozessoren untereinander erfolgt über diesen globalen Speicher.

Eine Form der Mehrprozessorsysteme besitzt einen einheitlichen Speicherzugriff, auch **Uniform Memory Access** (UMA) genannt. Hierbei hat jeder Prozessor die gleiche Speicherübertragungszeit. Besitzen alle Prozessoren die gleiche Funktionalität und die gleiche Sicht auf das Gesamtsystem, werden diese Rechner als symmetrische Multiprozessoren (SMP) bezeichnet. Außer einem eigenen Cache, hat ein Prozessor in einem SMP-System keinen privaten Speicher. Bei UMA sind alle Prozessoren eng miteinander gekoppelt. In der Regel wird über einen zentralen Bus kommuniziert, wobei jedoch immer nur ein Prozessor auf den gemeinsamen Speicher zugreifen kann.

Prozessoren und deren Speicher können auch zu Gruppen zusammengesetzt werden, die loser gekoppelt sind als bei SMP. Jeder Prozessor besitzt mindestens einen lokalen Speicher, der physisch direkt diesem Prozessor zugeordnet ist. Da der Speicher physisch verteilt ist, über ein Verbindungsnetzwerk aber alle Prozessoren auf diesen zugreifen können, wird dieser als virtuell-gemeinsamer Speicher bezeichnet. Der Zugriff auf den eigenen lokalen Speicher eines Prozessors ist schneller als der Zugriff auf den Speicher eines anderen Prozessors. Diese Eigenschaft der unterschiedlichen Speicherzugriffszeiten wird als **Non-Uniform Memory Access** (NUMA) bezeichnet.

Systeme mit **Cache Only Memory Access** (COMA) sind in ihrem Aufbau mit NUMA-Systemen vergleichbar. Der Speicher wird jedoch als Cache benutzt, wodurch Daten redundant in dem verteilten Speicher vorliegen können.

Die zweite große Klasse von MIMD-Systemen sind Mehrrechnersysteme mit einem verteilten Speicher. Diese Systeme bestehen aus mehreren selbstständigen Recheneinheiten, Knoten oder Nodes genannt, mit eigenem Speicher und einem Verbindungsnetzwerk, das

diese Nodes physikalisch verbindet. Das Verbindungsnetzwerk ist in der Regel langsamer als das von Mehrprozessorsystemen. Die Kommunikation und der Datenaustausch zwischen den Nodes wird mit Hilfe von Nachrichten durchgeführt. Mehrrechnersysteme werden in **Massively Parallel Processors** (MPP) und **Cluster Of Workstations** (COW) unterschieden. Die Nodes eines COW bestehen aus Standard-Hardware und waren ursprünglich auch mit Standardnetzwerkkomponenten untereinander verbunden. Professionelle COW verwenden mittlerweile jedoch spezielle Hochgeschwindigkeitsnetzwerkkomponenten wie Infiniband.[1] Im Gegensatz zu COW sind MPP oft sehr teuer und bestehen aus herstellerspezifischer Hardware mit sehr vielen Nodes. MPP waren überwiegend im Bereich der sogenannten Supercomputer vertreten, da sie ursprünglich schnellere Verbindungsnetzwerke ermöglichten. Aufgrund der Entwicklung von Hochgeschwindigkeitsnetzwerken überwiegt mittlerweile der Anteil an COW im Supercomputerbereich.[2]

2.1.2. Cluster-Computing

Cluster-Computing ist das Lösen von Problemen auf Cluster Of Workstations (im Folgenden als Cluster bezeichnet). Cluster bestehen aus untereinander vernetzten Computern, welche sich nach außen als ein Computer präsentieren. Dieses Prinzip der parallelen Transparenz wird als Single System Image (SSI) bezeichnet, d.h. dem Benutzer ist im Idealfall nicht bekannt, dass er mit mehreren lose gekoppelten Computern arbeitet. Als Parallelrechner werden Cluster in die Klasse der MIMD-Systeme mit verteiltem Speicher eingeordnet.

Eine spezielle Cluster-Software überwacht die Gesamtfunktionalität, koordiniert die Zusammenarbeit zwischen den einzelnen Computern und stellt den Zugang zu dem System bereit. Die in einem Cluster befindlichen Computer werden als Knoten (engl. Nodes) bezeichnet. Die Cluster-Software unterscheidet hier meist zwischen sogenannten „Master Nodes" und „Slave Nodes". Die Überwachung und Verwaltung des Clusters wird vom Master Node übernommen. Die Slave Nodes, häufig auch „Worker" oder „Compute Nodes" genannt, stellen Ressourcen wie Rechenleistung oder Speicherplatz zur Verfügung.

Cluster werden im Programmiermodell des Nachrichtenaustausches programmiert. Häufig verwendete Bibliotheken sind Message Passing Interface (MPI) und Parallel Virtual Machine (PVM) [RR07, BM06]. Aber auch andere Modelle wie das in dieser Arbeit behandelte MapReduce gewinnen zunehmend an Bedeutung.

2.1.2.1. Aufstellungskonzepte

Ein Cluster wird als homogen bezeichnet, wenn alle Nodes das gleiche Betriebssystem verwenden und die gleiche Hardware-Architektur nutzen. Im Gegensatz dazu können in einem heterogenen Cluster unterschiedliche Betriebssysteme sowie Hardware-Architekturen verwendet werden.

Anhand der räumlichen Aufstellung wird unterschieden in [TG99, S. 687]:

- zentrale Cluster, häufig auch „Glass-House Cluster" genannt, und
- dezentrale Cluster, häufig auch als „Campus-Wide Cluster" bezeichnet.

Bei einem Glass-House Cluster befinden sich alle Nodes lokal nah beieinander und häufig zentral in einem Raum oder Server-Schrank. Diese Art der Cluster sind überwiegend homogen und oft sind die Computer sehr kompakt, um die Stellfläche und die Kabellänge zu reduzieren. Vorteile dieses Konzeptes sind eine bessere Erreichbarkeit bei Wartungsaufgaben und eine schnellere Vernetzung durch Nutzung von Hochgeschwindigkeitsnetzwerken.

[1] `http://www.top500.org/overtime/list/38/connfam`
[2] `http://www.top500.org/overtime/list/38/archtype`

Die Computer eines Campus-Wide Cluster sind über die Räume eines oder mehrerer Gebäude verteilt. Diese Dezentralität ist bei Stromausfällen oder Bränden von Vorteil, da so nicht der gesamte Cluster gefährdet ist. Durch die aufwendigere Vernetzung stellen Hochgeschwindigkeitsnetzwerke in der Regel keine Option dar. Des Weiteren ändert sich die Anzahl sowie die Hardware der Nodes häufig. Diese Heterogenität erhöht den Wartungs- und Programmieraufwand. Vertreter dieses Aufstellungskonzeptes sind „Feierabendcluster", bei denen die Rechenkapazitäten der Computer von Mitarbeitern bei Abwesenheit für den Cluster genutzt werden.

2.1.2.2. Anwendungsgebiete

Die Anwendungsgebiete von Clustern sind eng mit deren Zielen verknüpft [BM06, S. 31]:

- High Availability (HA) Cluster erhöhen die Verfügbarkeit,
- High Performance Computing (HPC) Cluster erhöhen die Rechenkapazitäten und
- High Throughput Cluster erhöhen die Durchsatzrate.

HA-Cluster werden mit dem Ziel eingesetzt, Dienste ausfallsicher zu betreiben. Dies können zum Beispiel Webserver, Datenbanken und Dateiserver sein, welche geringe, im besten Fall keine, Ausfallzeiten haben sollen. Die hohe Verfügbarkeit wird durch redundante und voneinander nahezu unabhängiger Hardware erreicht. Wenn einzelne Nodes ausfallen, darf dies nicht die noch intakten Nodes beeinflussen. Eine Cluster-Software erkennt die Ausfälle und veranlasst die betriebsbereiten Nodes die Dienste der ausgefallenen Nodes zu übernehmen.

HA-Cluster werden in Active/Passive und Active/Active Cluster unterteilt. Ein Active/Passive Cluster, auch „Failover Cluster" genannt, besteht aus mindestens zwei Nodes. Nur auf dem aktiven System, das Primärsystem, wird der Dienst ausgeführt. Die wartenden passiven Systeme sind im Normalbetrieb nicht in Verwendung und werden als Backup- oder Standby-System bezeichnet. Fällt nun das Primärsystem aus, muss das passive System gestartet und die Dienste auf dieses übertragen werden. Dieses ungeplante Wechseln im Fehlerfall wird als Failover bezeichnet. Die Dauer zwischen dem Ausfall eines Dienstes und dem Zeitpunkt, bis dieser wieder verfügbar ist, wird als Failover-Zeit bezeichnet. Da bei einem Ausfall die Dienste auf dem passiven Node erst gestartet und ggf. erst Daten kopiert werden müssen, ist die Failover-Zeit oft größer als bei Active/Active Clustern.

Bei einem Active/Active Cluster sind gleichzeitig mehrere Nodes aktiv. Die verwendete Ressource des Dienstes muss diese Konfiguration unterstützen, da die Daten auf mehreren Nodes verteilt sind. Dies ermöglicht bei einem Ausfall das schnelle Übernehmen des Dienstes von einem anderen Node, da die Daten bereits vorhanden sind und der Dienst schon gestartet ist. Bei der Ressourcenverwaltung wird zwischen Shared Nothing und Shared All unterschieden. In einem Shared-Nothing-System besitzt jeder Node eine eigene Ressource, auf die nur dieser Node Zugriff hat. Dadurch müssen keine Sperrmechanismen umgesetzt werden, das sich positiv auf die Performance auswirkt. Bei Shared All wird die Ressource von mehreren Nodes gemeinsam genutzt, weshalb konkurrierende Zugriffe durch Sperrmechanismen verwaltet werden müssen. Andererseits können Daten optimaler auf die vorhandenen Ressourcen verteilt werden.

HPC-Cluster werden genutzt, um aufwendige Rechenaufgaben, auch Jobs genannt, zu lösen. Der Cluster stellt hierbei eine vielfach höhere Rechenleistung zur Verfügung als ein normaler Computer, wodurch die Jobs schneller oder in vertretbarer Zeit genauere Lösungen berechnet werden können. Die hohe Rechenleistung des Clusters setzt sich aus der Rechenleistung jedes einzelnen Nodes im Cluster zusammen. Durch das Hinzufügen von neuen Nodes kann die Rechenleistung mit wenig Aufwand erhöht werden. Einzig

die Geschwindigkeit des Verbindungsnetzwerks stellt in der Regel eine Beschränkung der maximal möglichen Rechenleistung dar. Die Rechenleistung eines HPC-Clusters kann auf zwei Weisen genutzt werden. Einerseits kann ein Job in mehrere Teilaufgaben, Tasks genannt, unterteilt werden, welche dann auf mehrere Nodes verteilt und parallel berechnet werden. Jedoch bietet nicht jedes Problem diese Möglichkeit der parallelen Zerlegung. Andererseits können auch mehrere Jobs durch einen Job-Scheduler auf die Nodes verteilt werden. Jeder Node berechnet dann autonom und unabhängig von anderen Nodes seinen zugewiesenen Job. Der Job-Scheduler hat außerdem oft weitere Aufgaben: Er überwacht die Ausführung jedes Jobs und startet gegebenenfalls fehlerhafte Jobs neu. Des Weiteren berücksichtigt er Aspekte der Lastverteilung, der Ressourcenzuteilung und Prioritäten der einzelnen Jobs in einer Warteschlange.

High Throughput Cluster sind vergleichbar mit HPC-Clustern und unterscheiden sich in erster Linie von den Jobs, die sie verarbeiten. Im Gegensatz zu HPC-Clustern werden viele kleine, aber ähnliche Jobs mit unterschiedlichen Parametern verarbeitet. Vordergründiges Ziel ist es, die Anzahl der zu verarbeitenden Jobs pro Zeiteinheit bei gleichbleibender Verarbeitungszeit zu erhöhen, kurz die Erhöhung der Durchsatzrate. Dies sind zum Beispiel die Transaktionen pro Sekunde bei einem Datenbanksystem oder die Anfragen pro Sekunde an einen Webserver. Da dies oft mit einer effizienten Lastverteilung einhergeht, werden diese Cluster auch als „Load Balancing Cluster" bezeichnet. Außerdem haben solche Cluster ähnliche Eigenschaften wie HA-Cluster, da sie Ausfälle einzelner Nodes ebenfalls ohne Ausfallzeiten bewältigen müssen.

2.1.3. General Purpose Computing on Graphics Processing Unit

General Purpose Computation on Graphics Processing Unit (GPGPU) bedeutet übersetzt „allgemeine Berechnung auf Grafikprozessoren". Die vereinfachte Aufgabe einer Grafikkarte ist die Steuerung der Bildschirmausgabe in einem Computer. Der Begriff „allgemeine Berechnung" ist im Zusammenhang mit der Aufgabe einer Grafikkarte in einem Computer zu betrachten. Im Gegensatz zu Hauptprozessoren (engl. Central Processing Unit, CPU), die einen möglichst großen Umfang an Funktionen bereitstellen sollen, sind die Funktionen und die damit verbundenen Berechnungen eines Grafikprozessors (engl. Graphics Processing Unit, GPU) speziell auf dieses Einsatzgebiet zugeschnitten. Aufgrund dieser technischen Einschränkungen war es mit den ersten Grafikkarten nicht annähernd möglich, vergleichbare Berechnungen wie die eines Hauptprozessors durchzuführen.

Mit der Entwicklung von grafischen Benutzeroberflächen und 3D-Grafik erweiterte sich auch der Funktionsumfang und die Leistung von Grafikkarten. Im Jahr 2000 wurde mit Direct3D 8.0 von Microsoft das Konzept der Shader eingeführt.[3] Diese ermöglichen die hardwareunabhängige Programmierung zur Berechnung von zum Beispiel Beleuchtungseffekten oder Geometriedaten. Durch die Weiterentwicklung dieses Konzepts konnten auf Grafikkarten zunehmend immer aufwendigere Operationen umgesetzt werden.

Da die Berechnungen von Grafikkarten ein hohes Potenzial an Parallelität bieten, verlief die Hardwareentwicklung entscheidend anders als bei Hauptprozessoren. So können zum Beispiel Bildpunkte als Vektoren mit drei Werten für Rot, Grün und Blau dargestellt werden. Soll nun ein Bildpunkt $p_1 =< r_1, g_1, b_1 >$ den Bildpunkt $p_2 =< r_2, g_2, b_2 >$ durch die Funktion $p_1 \cdot p_2 =< r_1 \cdot r_2, g_1 \cdot g_2, b_1 \cdot b_2 >$ modifizieren, kann dies unter Nutzung des SIMD-Prinzips in einem Rechenschritt durchgeführt werden. Diese Eigenschaft nutzten die Hersteller bei der Entwicklung immer leistungsstärkerer und hoch parallelisierter Grafikkarten massiv aus. Aktuelle Grafikkarten besitzen mehrere hundert sogenannter Stream-Prozessoren.[4] In Bezug auf die Klassifikation von Tanenbaum stellen Grafikkar-

[3] `http://www.microsoft.com/Presspass/press/2000/nov00/directxlaunchpr.mspx`

[4] Zum Beispiel die NVIDIA Quadro 6000 mit 448 CUDA Recheneinheiten:
`http://www.nvidia.de/object/product-quadro-6000-de.html`

ten eine Mischform aus SIMD- und MIMD-Systemen dar. Ein Stream-Prozessor arbeitet nach dem SIMD-Prinzip. Zusätzlich sind mehrere Stream-Prozessoren in voneinander unabhängige Gruppen mit eigenem lokalen Speicher eingeteilt (siehe Abschnitt 2.1.3.3). Daher sind GPUs auch dem MIMD-Prinzip mit einer Speicherarchitektur, die mit NUMA vergleichbar ist, zu zuordnen [Sch10, S. 199].

Im Gegensatz dazu fand unter den Hauptprozessorherstellern ein regelrechtes „Megahertz-Rennen" statt. Aufgrund der steigenden Taktfrequenz und der kleiner werdenden Strukturen schien vor allem die hohe Wärmeentwicklung die Leistungsgrenzen der damaligen Prozessoren aufzuzeigen. Als Ausweg aus dieser Entwicklung stellte AMD im Jahr 2004 den ersten Dual-Core-Prozessor mit x86-Befehlssatz vor [HÖ8]. Aktuelle Mehrkernprozessoren besitzen bis zu 16 physikalische Kerne.[5] Im Vergleich mit aktuellen Grafikkarten ist dies jedoch eine sehr geringe Anzahl.

2.1.3.1. Technologien

Der Begriff GPGPU beinhaltet alle Technologien, die es ermöglichen Grafikkarten auch zur Berechnung außerhalb des Anwendungsgebietes der Bildschirmausgabe zu nutzen. Zur Zeit gibt es hauptsächlich drei verbreitete Technologien:

- CUDA von NVIDIA,
- OpenCL von Apple und der Khronos Group und
- Direct Compute von Microsoft.

Aufgrund von cleveren Marketing seitens NVIDIA und guten Entwicklungswerkzeugen zur Zeit der Markteinführung ist CUDA bei Entwicklern sehr beliebt. Aber auch OpenCL ist durch den offenen Standard und der Hardwareunabhängigkeit weit verbreitet. Hingegen findet Microsofts Direct Compute beim High-Performance-Computing und in Spezialsoftware kaum Anwendung.

2.1.3.2. Programmiermodell

Ein GPGPU-Programm enthält zwei Bestandteile: das Host-Programm und den Kernel. Als Host wird das System bezeichnet, auf dem die Anwendung ausgeführt wird, welche die Grafikkarte nutzt. Der auf einer oder mehreren Grafikkarten ausgeführte Code wird als Kernel bezeichnet. Ein Kernel ist immer in einer Host-Anwendung eingebettet und kann nicht als eigenständiges Programm ausgeführt werden.

Die Aufgabe des Host ist es, die Ressourcen zu organisieren, indem er die vorhandenen Grafikkarten abfragt und die Aufgaben sowie Daten auf diese verteilt. Nach erfolgreicher Berechnung werden die Daten aus dem Grafikspeicher in den Hauptspeicher geladen und ggf. weiterverarbeitet.

Bei GPGPU wird hauptsächlich die Datenparallelität ausgenutzt, d.h. derselbe Code wird auf unterschiedlichen, voneinander unabhängigen Daten ausgeführt. Dieser Kernel implementiert den Algorithmus für den Grafikprozessor. Zur Ausführung eines Kernels muss ein Indexraum angegeben werden, um die Anzahl der Kernel-Instanzen festzulegen. Jede Kernel-Instanz wird als Work-Item bezeichnet und ist genau einem Index, der Global ID, zugewiesen.

Zusätzlich können Work-Items in Work-Groups organisiert werden, deren Indexraum dieselbe Dimension wie die der Work-Items besitzen muss. Jede Work-Group besitzt eine Group ID aus ihrem Indexraum. Neben der eindeutigen Global ID besitzt jedes Work-Item eine Local ID. Die Local ID ist innerhalb einer Gruppe eindeutig. Mit Hilfe der Local

[5] Zum Beispiel der AMD Opteron mit 16 Kernen:
`http://www.amd.com/de/products/server/processors/6000-series-platform/6200/Pages/6200-series-processors.aspx`

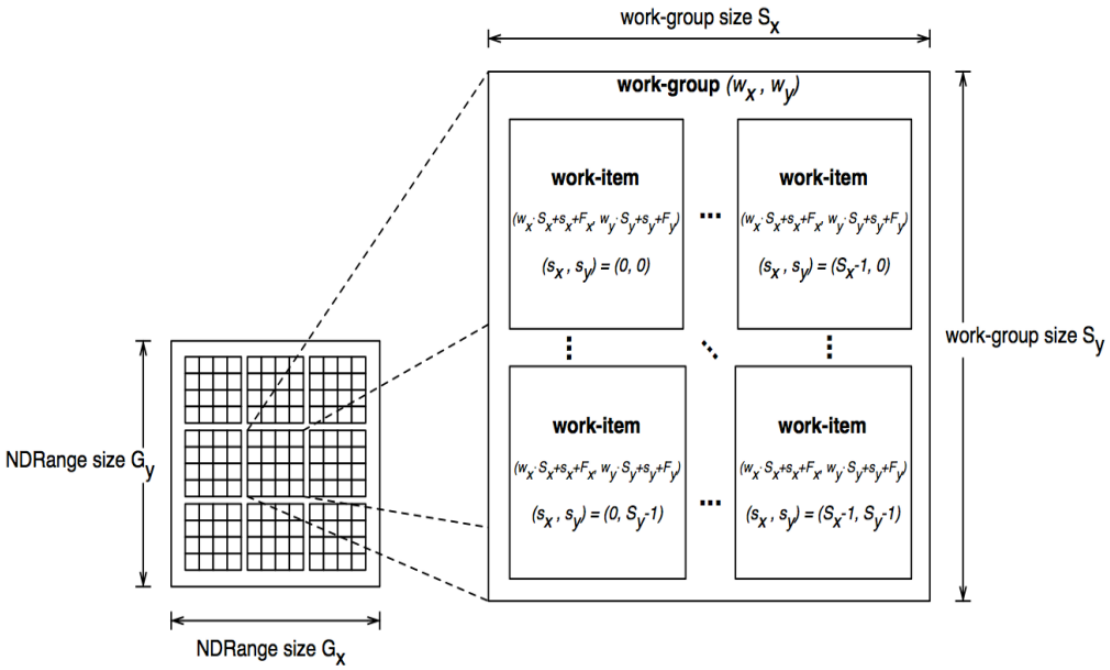


Abbildung 2.2.: Aufteilung von Work-Items in Work-Groups (Quelle: Khronos Group)

und der Group ID kann wiederum die eindeutige Global ID eines Work-Items berechnet werden.

In Abbildung 2.2 ist ein globaler 2-dimensionaler Indexraum (G_x, G_y) abgebildet. Die Anzahl der Work-Items ergibt sich aus dem Produkt von G_x und G_y. Zusätzlich sind alle Work-Items in Work-Groups mit dem Indexraum (S_x, S_y) organisiert. Die Anzahl der Work-Items einer Work-Group ist das Produkt aus S_x und S_y. Die Anzahl der Work-Groups pro Dimension kann durch $(W_x, W_y) = (\frac{G_x}{S_x}, \frac{G_y}{S_y})$ berechnet werden.

2.1.3.3. Speichermodell

In Abbildung 2.3 auf der nächsten Seite ist das allgemeine Speichermodell einer Grafikkarte dargestellt. Die Grafikkarte wird als Compute Device bezeichnet. Das System, in dem ein oder mehrere Compute Devices arbeiten, heißt Host[6]. Host Memory bezeichnet den Hauptspeicher eines Computers. Kein Work-Item hat direkten Zugriff auf diesen Speicher.[7] Aus dem Host Memory kann der Host Daten in den Global Memory und Constant Memory kopieren. Alle Work-Items können Daten aus dem Global Memory lesen und schreiben, jedoch keinen Speicher allozieren. Im Constant Memory können alle Work-Items Speicher allozieren und Daten lesen.

Wie in Abschnitt 2.1.3.2 auf der vorherigen Seite beschrieben können mehrere Work-Items in Work-Groups zusammengefasst werden. Alle Work-Items einer Work-Group teilen sich einen gemeinsamen Local Memory mit Lese- und Schreibzugriff. Jedes Work-Item kann Speicher im Local Memory allozieren, der anschließend von allen Work-Items dieser Work-Group genutzt werden kann. Jedes Work-Item besitzt seinen eigenen Private Memory, auf den es vollen Zugriff hat. Der Host hat auf den Local und Private Memory keinen Zugriff.

Außerdem sollte beachtet werden, dass der Global bzw. Constant Memory langsamer als der Local Memory ist. Die unterschiedlichen Zugriffszeiten und die Verarbeitung nach dem SIMD-Prinzip ermöglichen viele Speicheroptimierungen [Sch10, S. 203], die in dieser

[6] Diese Terminologie wird in Abschnitt 2.3 auf Seite 31 genauer erklärt.
[7] Ausnahmen gibt es bei OpenCL-Implementierungen.

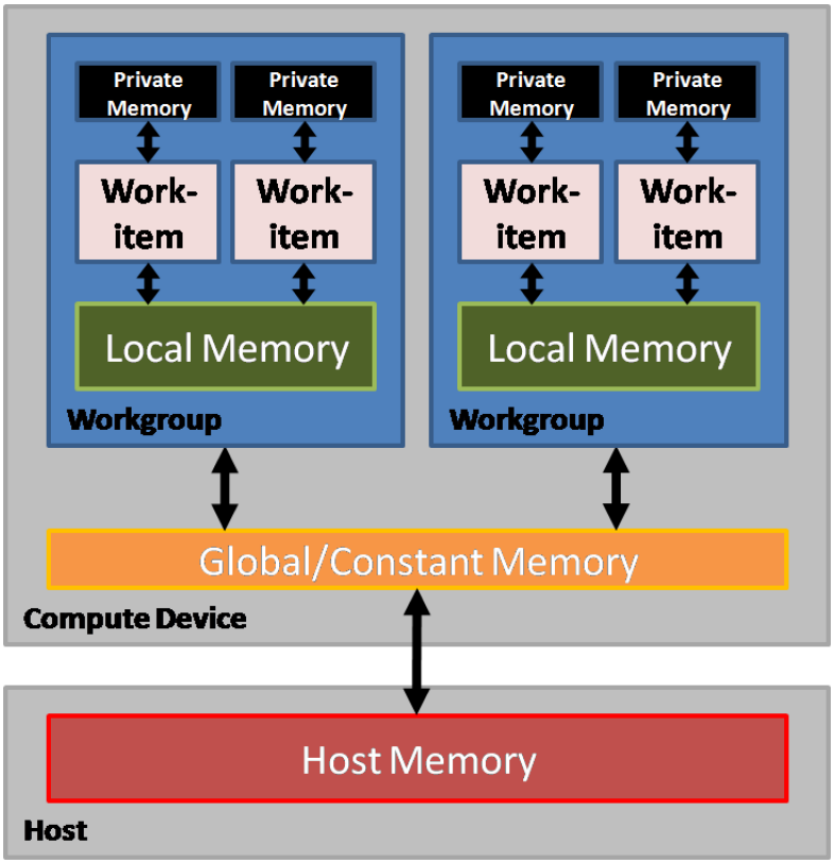


Abbildung 2.3.: Allgemeines Speichermodell einer Grafikkarte (Quelle: AMD)

Arbeit jedoch nicht vorgestellt werden.

2.1.4. Programmiermodelle der parallelen Programmierung

Die Art und Weise, wie eine Rechnerarchitektur programmiert werden kann, wird in Programmiermodellen abstrahiert und durch Ausblenden von Details vereinfacht dargestellt. Aus den klassischen Modellen wie der prozeduralen, objektorientierten und funktionalen Programmierung können weitere Modelle für bestimmte Problemfelder entwickelt werden.

Die Abgrenzung zwischen sogenannten Frameworks, Programmierschnittstellen (API) und Programmiermodellen ist nicht immer eindeutig. Ein Framework stellt Funktionalitäten bereit und gibt die Softwarearchitektur vor, da es häufig den Kontrollfluss der Anwendung bereits umsetzt. Der Programmierer entwickelt lediglich bestimmte Komponenten und bindet diese in das Programmiergerüst ein. Im Gegensatz dazu stellt eine API nur Funktionalitäten bereit, ohne einen konkreten Kontrollfluss vorzugeben. Eine API lässt bei der Entwicklung mehr Freiheiten als ein Framework, jedoch muss hierbei auch mehr implementiert werden. Programmiermodelle für bestimmte Problemfelder können mit Hilfe von APIs leichter umgesetzt oder durch konkrete Frameworks bereits implementiert sein.

Anschließend werden Programmiermodelle zur parallelen Programmierung vorgestellt, mit deren Hilfe die Parallelität von Parallelrechnern ausgenutzt werden kann. Die Wahl eines konkreten Modells ist von der vorhandenen Rechnerarchitektur oder dem zu parallelisierenden Problem abhängig, teilweise sogar von beiden.

2.1.4.1. Datenparallelität

Bei der Datenparallelität werden dieselben Operationen auf unterschiedlichen Elementen einer Datenstruktur ausgeführt [RR07, S. 122]. Wenn die Daten voneinander unabhängig sind, so können auf diesen Daten inhärent parallele Operationen ausgeführt werden [BM06, S. 37].

Zur Veranschaulichung der Datenparallelität dient die Vektoraddition: $\overrightarrow{c} = \overrightarrow{a} + \overrightarrow{b}$. Die Dimension des Vektors sei durch n gegeben und somit enthält jeder Vektor n Elemente. Bei einem sequenziellen Programm werden n Additionen nacheinander ausgeführt:

```
for(int i=0; i < n; i++)
  c[i] = a[i] + b[i];
```

Da die Anweisung `c[i] = a[i] + b[i]` aber unabhängig von dem verwendeten Index `i` ist, können die Anweisungen für jeden konkreten Wert von `i` parallel ausgeführt werden. Diesem Vorgehen liegt das SIMD-Modell zugrunde und es wird von vielen Prozessoren unterstützt. Um eine Vektoraddition anhand des SIMD-Modells zu parallelisieren, bieten einige Programmiersprachen sogenannte Vektoranweisungen an. Hierbei wird die Vektoranweisung beim Kompilieren automatisch in ihre unabhängigen Anweisungen zerlegt und für die Ausführungseinheiten des Prozessors vorbereitet. In Fortran 90 ist eine parallele Vektoraddition sehr einfach:

```
c=a+b
```

Da das SIMD-Prinzip bei modernen Prozessoren durch spezielle Maschineninstruktionen, z.B. den Streaming SIMD Extensions (SSE), umgesetzt wird, können damit nur einfache Datenstrukturen wie Vektoren und einfache Operationen verarbeitet werden. Für komplexere Datenstrukturen und MIMD-Rechnerarchitekturen kann eine Datenparallelität durch das SPMD-Konzept (Single Program Multiple Data) ausgenutzt werden [RR07, S. 123]. Dabei werden ein vollständiges Programm oder nur Teilstücke eines Programms parallel von allen Prozessoren ausgeführt. Die Zuteilung der Daten an den entsprechenden Prozessor findet hierbei über Identifikatoren statt. Als Identifikatoren können zum Beispiel Prozessornummern oder Thread-Nummern verwendet werden. Die Zuteilung der Programminstanzen an die Ausführungseinheiten (Prozessoren, Nodes in einem Cluster o.ä.) wird bei der Verwendung von Threads dynamisch durch das Betriebssystem, bei einem Cluster dynamisch durch die Cluster-Software oder statisch im Programmcode vorgenommen. In Abhängigkeit von der vorhandenen Rechnerarchitektur, können beim SPMD-Konzept weitere Programmiermodelle Anwendung finden, um die Datenparallelität auf die Hardware abzubilden. Eine parallele Vektoraddition nach dem SPMD-Konzept kann z.B. so umgesetzt werden:

```
int pid = getPid();
int pidCount = getMaxPids()
for(int i = pid; i < n; i = i + pidCount)
  c[i] = a[i] + b[i];
```

Bei diesem Beispiel sind die Erzeugung der parallelen Programminstanzen und die Synchronisation zwischen diesen nicht berücksichtigt. Dieses Programmstück wird für jede Ausführungseinheit genau einmal gestartet. Anschließend wird der Variable `pid` ein fortlaufender Wert zugewiesen, der in nur einer Programminstanz vorkommt und als Identifikator verwendet wird. `pidCount` enthält die Anzahl der gestarteten Programminstanzen. Da die Schleife mit dem Index `pid` startet und als Schrittweite die Anzahl der Programminstanzen genutzt wird, verwendet jede Programminstanz andere Daten.

2.1.4.2. Funktionsparallelität durch Threads

Bei der Funktionsparallelität wird die Unabhängigkeit zwischen verschiedenen Funktionen, Aufgaben bzw. Programmabschnitten ausgenutzt. Im Gegensatz zur Datenparallelität wird kein paralleler Datenfluss, sondern ein paralleler Kontrollfluss erzeugt. Ein Programmstück, das unabhängig von einem anderen ausgeführt werden kann, wird als Task bezeichnet. Bei der Funktionsparallelität wird das Programm in mindestens zwei unabhängige Tasks zerlegt. Zwischen Tasks können verschiedene Abhängigkeiten bestehen, welche die parallele Ausführung beeinflussen. Wenn ein Task zum Beispiel das Endergebnis eines anderen Tasks benötigt, kann dieser nur dann gestartet werden, wenn das Endergebnis berechnet und gespeichert wurde. Um falsche Ergebnisse zu vermeiden, dürfen solche Tasks niemals parallel sondern nur sequenziell ausgeführt werden. Die Ausführungsreihenfolge und Abhängigkeiten müssen bei der Funktionsparallelität explizit durch den Programmierer vorgegeben werden. Diese Koordinierung, oder auch Synchronisation genannt, kann durch Sperren gemeinsamer Variablen, Haltepunkten und Signalen erreicht werden.

Damit die Tasks auch wirklich gleichzeitig abgearbeitet werden können, müssen sie auf die vorhandenen Ausführungseinheiten abgebildet werden. Auf MIMD-Systemen mit gemeinsamen Speicher können hierfür leichtgewichtige Prozesse, Threads genannt, verwendet werden. Für jedes Programm, das gestartet wird, erzeugt das Betriebssystem einen Prozess. Mit Hilfe des Prozesses kann das Betriebssystem die Ausführung eines Programms verwalten. Dies ist notwendig, da ein Programm selten ohne Unterbrechung ausgeführt wird. Weil mehrere Programme um Hardwareressourcen konkurrieren, verwaltet das Betriebssystem den Zugriff und erlaubt jedem Programm, eine Ressource eine bestimmte Zeit lang zu nutzen. Damit die Ausführung eines Programms an derselben Stelle fortgesetzt werden kann, an der es unterbrochen wurde, muss der Zustand der Programmausführung gesichert werden. Ein Prozess enthält alle wichtigen Daten, die für dieses Vorgehen notwendig sind.

Bei MIMD-Systemen wird jeder Prozess durch das Betriebssystem einer Recheneinheit zugeordnet, wodurch diese parallel ausgeführt werden. Da das Sichern und Wiederherstellen eines Prozesses sehr viel Zeit in Anspruch nimmt und Tasks oft nur sehr kurze Zeit existieren, ist es nicht sehr performant für jeden Task einen eigenen Prozess zu starten. Da ein Task in der Regel immer einem Programm zugeordnet ist, kann für jeden Task ein Thread genutzt werden. Threads können nur aus einem Prozess heraus gestartet werden und teilen sich viele Ressourcen mit dem Prozess, aus dem sie erzeugt wurden. Deswegen ist der Verwaltungsaufwand für Threads in der Regel geringer und schneller als der für Prozesse, da nicht für jeden Thread der komplette Zustand gesichert werden muss.

Ein typisches Anwendungsbeispiel für Threads ist die Trennung zwischen der grafischen Benutzeroberfläche und den Berechnungen, die durch diese angestoßen werden. Bei Programmstart wird ein Thread erzeugt, der die Benutzeroberfläche verwaltet. Wenn der Benutzer nun über die Oberfläche eine Datei öffnen will, kann dies bei großen Dateien sehr lange dauern und die Oberfläche würde solange blockieren, bis dieser Vorgang beendet ist. Um dem entgegenzuwirken wird dieser Vorgang in einem Thread ausgeführt. Dadurch kann der Benutzer mit der Oberfläche interagieren, während die Datei geöffnet wird. Damit die Oberfläche den Dateiinhalt nach Beendigung des Tasks anzeigen kann, müssen die zwei Threads untereinander kommunizieren. Erst wenn der Thread, der die Oberfläche anzeigt, ein „Fertigsignal" von dem Thread, der die Datei öffnet, erhält, dürfen die entsprechenden Operationen zur Anzeige ausgeführt werden.

2.1.4.3. Nachrichtenaustausch

Der Nachrichtenaustausch ist eine Kommunikationsform und weniger ein Programmiermodell. Dabei findet die Datenübertragung und die Synchronisation durch das Versenden

und Empfangen von Nachrichten statt. Diese Kommunikationsform wird hauptsächlich bei Systemen mit verteiltem Speicher, wie zum Beispiel Clustern, eingesetzt, um Daten zwischen zwei getrennten Adressräumen auszutauschen. Da der Datenaustausch und die Synchronisation über gemeinsame Variablen schneller ist, wird der Nachrichtenaustausch bei Systemen mit gemeinsamen Speicher kaum verwendet. Mit dem Nachrichtenaustausch können sowohl Datenparallelität, als auch Funktionsparallelität ausgenutzt werden. Wie bei Threads muss der Programmierer die Synchronisation und den Datenaustausch explizit, also durch das Senden und Empfangen von Nachrichten, vorgeben.

Die Kommunikation findet immer zwischen zwei Prozessen statt, die auf demselben System oder auf getrennten Systemen ausgeführt werden. Für den Nachrichtenaustausch zwischen zwei Prozessen müssen Sende- und Empfangsoperation als Paar auftreten. Die Kommunikation kann blockierend oder nicht-blockierend stattfinden. Bei einer blockierenden Übertragung wartet der Absender solange an der Sendeoperation, bis die Nachricht vom Empfänger entgegengenommen wurde. Der Empfänger wartet solange bei der Empfangsoperation, bis die Nachricht vollständig empfangen wurde. Somit dient die blockierende Nachrichtenübertragung zur Synchronisation und Datenübertragung. Beim nicht-blockierenden Senden wird die Nachricht versendet, ohne zu prüfen oder zu warten, ob bzw. bis der Empfänger diese entgegengenommen hat. Dennoch kann zu einem späteren Zeitpunkt überprüft werden, ob der Empfänger die Nachricht entgegengenommen hat. Der Empfänger führt ebenfalls die nicht-blockierende Empfangsoperation aus. Wenn keine Nachricht erhalten wurde, also der Sender noch keine Nachricht versendet hat, wird das Programm weiter ausgeführt und es muss zu einem späteren Zeitpunkt erneut der Nachrichteneingang geprüft werden. Bei einer nicht-blockierenden Nachrichtenübertragung findet also keine Synchronisation statt, da nicht bekannt ist, an welcher Stelle im Programm sich der andere Prozess gerade befindet.

Für eine effiziente Übertragung und komfortable Programmierung gibt es drei Transferarten:

1. Punkt-zu-Punkt-Kommunikation
2. Punkt-zu-Mehrpunkt-Kommunikation
3. Mehrpunkt-zu-Mehrpunkt-Kommunikation

Die **Punkt-zu-Punkt-Kommunikation** verbindet genau zwei Prozesse untereinander. Sendet ein Prozess Nachrichten an mehr als einen Empfänger, spricht man von **Punkt-zu-Mehrpunkt-Kommunikation**. Wenn mehrere Prozesse im selben Kontext eine Punkt-zu-Mehrpunkt-Kommunikation ausführen, d.h. jeder Prozess an jeden sendet und jeder Prozess von jedem eine Nachricht empfängt, findet eine **Mehrpunkt-zu-Mehrpunkt-Kommunikation** statt.

Die Übertragungsdauer einer Mehrpunkt-Kommunikation kann mit Hilfe eines aufspannenden Baumes verkürzt werden. Diese Umsetzung ist vergleichbar mit dem Schneeballeffekt. Eine Mehrpunkt-Kommunikation startet dabei in der Wurzel des Baumes und führt dann die Übertragungen Tiefe für Tiefe aus. Alle Knoten einer Tiefe führen die Übertragung zueinander parallel aus. Damit steigt die Anzahl der parallelen Übertragungen mit wachsender Tiefe an. Eine Punkt-zu-Mehrpunkt-Kommunikation kann zum Beispiel als eine Folge von Punkt-zu-Punkt-Kommunikationen umgesetzt werden. Dieses Vorgehen kann optimiert werden, indem jeder Prozess, der bereits eine Nachricht empfangen hat, diese Nachricht ebenfalls an einen noch ausstehenden Prozess versendet. Dieses Vorgehen ist in Abbildung 2.4 auf der nächsten Seite dargestellt. Die Kommunikation startet bei Prozess 1. Alle durchgezeichneten Kanten führen eine Punkt-zu-Punkt-Kommunikation aus. Alle Einzeltransfers in einer Iteration werden parallel ausgeführt. Bereits nach der dritten Iteration haben alle Prozesse die Nachricht empfangen. [RR07, S. 143]

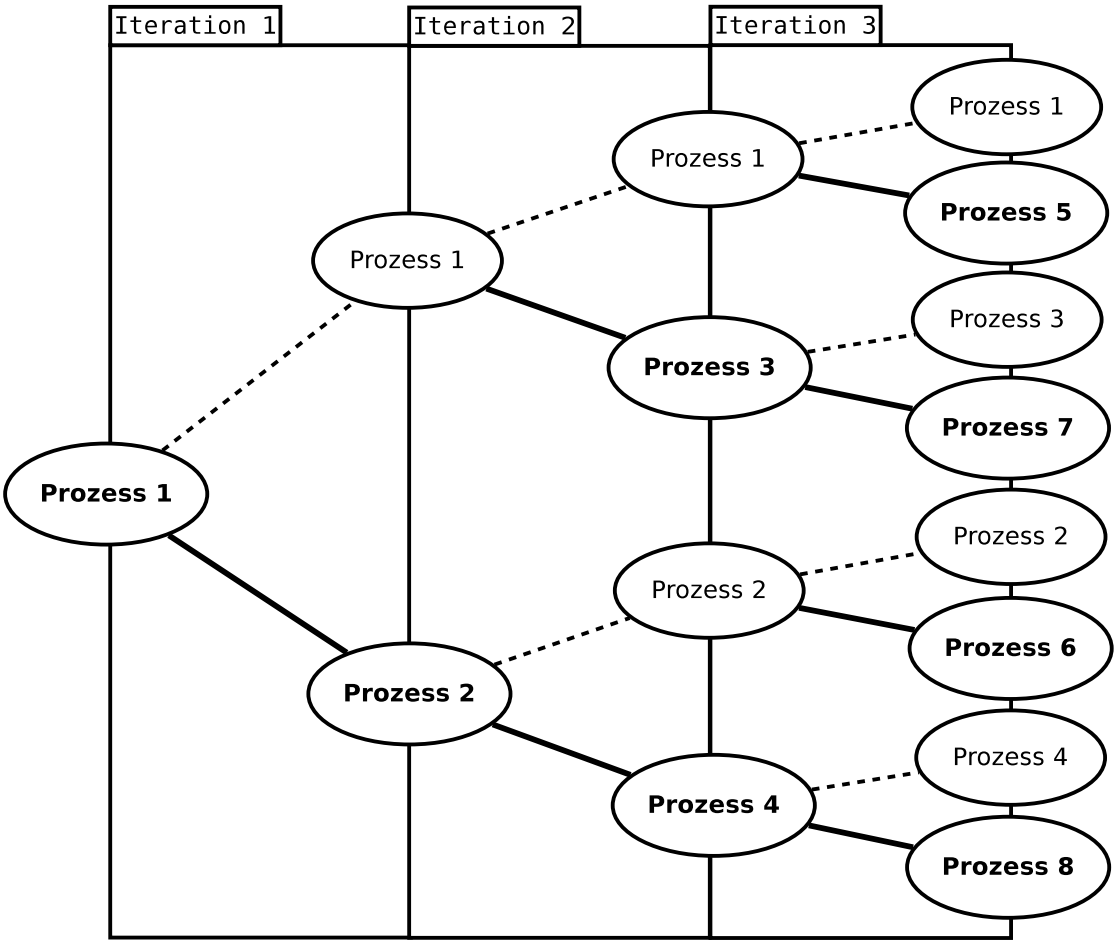


Abbildung 2.4.: Optimierte Punkt-zu-Mehrpunkt-Kommunikation mit 8 Prozessen

2.2. Apache Hadoop

Hadoop ist ein Cluster-System zur verteilten und parallelen Verarbeitung von großen Datenmengen[8]. Das System ist in den Bereich der HPC-Cluster einzuordnen. Seit Januar 2008 ist es ein „Top Level Project" von Apache, das aus dem Suchmaschinenprojekt Apache Nutch entstanden ist [Lam10, S.19]. Hadoop stellt für den Benutzer drei Komponenten zur Verfügung:

- ein Job-Managementsystem,
- ein verteiltes Dateisystem und
- das Programmiermodell MapReduce mit dem dazugehörigen Framework.

Das Job-Managementsystem verwaltet alle Jobs und Tasks, die auf dem Cluster ausgeführt werden. Es erkennt fehlerhafte Jobs und startet diese gegebenenfalls neu. Das verteilte Dateisystem ist der Datenspeicher von Hadoop. Alle Daten, die durch Hadoop verarbeitet werden, befinden sich verteilt auf den Nodes und müssen nicht von einem externen Speichersystem gelesen werden. Das spart lange Datenübertragungen zu Beginn eines Jobs. Ein Job wird im Programmiermodell MapReduce programmiert, wofür ein umfangreiches Framework zur Verfügung steht. MapReduce erweitert das SPMD-Konzept, indem es die Kommunikation und Datenzuteilung teilweise bereits vollständig implementiert.

Gewöhnliche HPC-Cluster sind eher ungeeignet für Probleme mit großen Datenmengen und vielen externen Speicherzugriffen, da die Datenübertragung über das Verbindungsnetzwerk um ein Vielfaches langsamer ist, als der Zugriff auf lokale Daten [BM06, S. 32]. Bei einigen Anwendungen können die Daten nur in kleineren Teilen von einem externen Speicher geladen werden, weil ein Node nicht alle Daten im Hauptspeicher oder auf der Festplatte speichern kann. Dieses Problem wird bei Hadoop gelöst, indem die Daten durch das verteilte Dateisystem auf allen Nodes verteilt gespeichert sind. Dadurch befindet sich ein Großteil der Daten schon lokal auf einem Node und muss nicht über ein langsames Netzwerk von einem externen Speicher geladen werden.

Hadoop wird häufig im Kontext von NoSQL erwähnt. Unter dem Begriff NoSQL (Not only SQL) werden Technologien zusammengefasst, die den Zugriff auf Datenbanken ermöglichen, die einen nicht-relationalen Ansatz verfolgen. Bei relationalen Datenbanken liegen die Daten strukturiert in Form von Tabellen vor, die sich gegenseitig referenzieren können. Die Verarbeitung der Daten einer relationalen Datenbank wird überwiegend mit der Datenbanksprache SQL (Structured Query Language) programmiert. In Anlehnung an Datenbanksysteme wird das verteilte Dateisystem von Hadoop wie eine Datenbank genutzt. Da diese jedoch beliebige und unstrukturierte Daten enthält, kann eine Datenbanksprache wie SQL nicht verwendet werden. Stattdessen wird eine „Datenbankanfrage" in Form eines Jobs gestellt. Die „Datenbanksprache" ist in diesem Fall MapReduce.

2.2.1. Komponenten

Die Komponenten von Hadoop können zwei Bereichen zugeordnet werden:

- verteilter Speicher und
- verteiltes Rechnen.

Jeder dieser Bereiche arbeitet in Form einer Master-Slave-Architektur. Hierbei wird zwischen Computern oder Anwendungen unterschieden, die Aufgaben verwalten bzw. zuweisen und denen, die die Aufgaben ausführen. Alle Komponenten treten in Form von

[8] Yahoo! hat mit Hadoop ein Petabyte Daten erfolgreich sortiert:
`http://developer.yahoo.com/blogs/hadoop/posts/2009/05/hadoop_sorts_a_petabyte_in_162/`

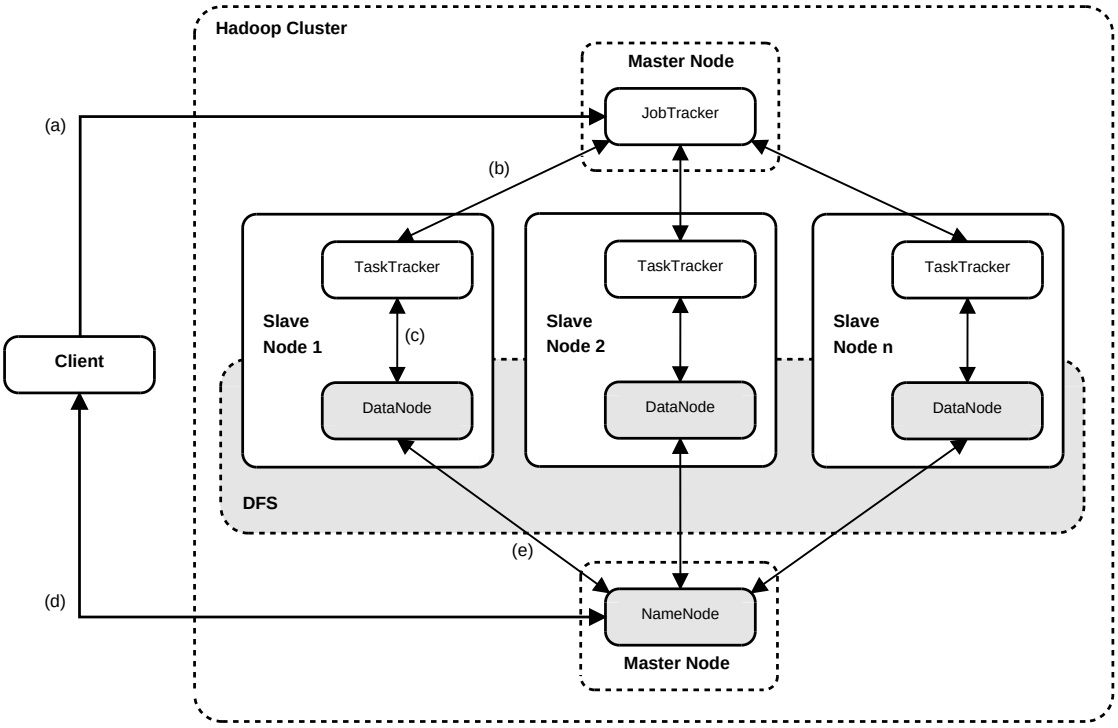


Abbildung 2.5.: Zusammenspiel der Komponenten eines Hadoop-Clusters

Anwendungsdiensten auf, die auf Computern in einem Netzwerk verteilt sind. Ein Computer in diesem Netzwerk wird als Node bezeichnet. Besteht die Hauptaufgabe eines Nodes in der Verwaltung des Clusters, wird dieser als Master Node bezeichnet. Dieser Node verteilt Aufgaben an sogenannte Slave Nodes, die diese Aufgaben bearbeiten. Eine Aufgabe als Ganzes wird im Folgenden als Job bezeichnet. Je nach Aufbau einer Aufgabe, kann diese in kleinere Teilaufgaben aufgeteilt werden. Diese Teilaufgaben heißen Tasks.

Abbildung 2.5 illustriert das Zusammenspiel der Komponenten eines Hadoop-Clusters. Im Bereich des verteilten Rechnens fungiert der Dienst JobTracker als Master und der Dienst TaskTracker als Slave. Ein Benutzer, auch Client genannt, reicht einen Job auf dem Master Node ein (a). Der Job wird vom Dienst JobTracker entgegengenommen. Dieser unterteilt den Job in mehrere Tasks und weist diese den Slave Nodes zu (b). Der TaskTracker empfängt die Tasks und führt diese aus. Dazu können Daten aus dem verteilten Dateisystem (engl. Distributed File System, DFS) geladen werden (c).

Der Dienst NameNode arbeitet im Bereich des verteilten Speichers als Master und verteilt die Daten an die DataNodes, die als Slaves fungieren. Alle DataNode-Dienste bilden das DFS, das vom NameNode verwaltet wird. Dieser nimmt Daten vom Client entgegen (d) und verteilt diese auf die DataNodes (e).

Je nach Größe des Clusters und Hardwareausstattung der Master Nodes können die Dienste JobTracker und NameNode gemeinsam auf einem Node oder verteilt auf zwei Nodes betrieben werden.

2.2.2. Verteilter Speicher

Das **Hadoop Distributed File System** (HDFS) ist das Standarddateisystem von Hadoop.[9] Es ist ein verteiltes und skalierbares Dateisystem für sehr große Datenmengen. Da das HDFS kein eigenständiger Dienst ist, setzt es sich aus dem NameNode zur

[9] Neben HDFS unterstützt Hadoop auch weitere Dateisysteme. Zusätzlich stellt die Hadoop-API eine abstrakte Klasse für Erweiterungen bereit.

Verwaltung und mindestens einem DataNode als Datenspeicher zusammen.

Für das Dateisystem kann eine Block Size und ein Replication Factor konfiguriert werden. Anhand der Block Size werden zusammenhängende Daten in Datenblöcke aufgeteilt, die auf unterschiedlichen DataNodes gespeichert werden. Dadurch können einerseits Daten gespeichert werden, deren Größe die physische Kapazität eines Datenträgers übersteigt, andererseits kann so die Performance durch parallele Zugriffe erhöht werden. Um einen Datenverlust durch einen Hardwareausfall auszuschließen, werden die Dateien und Datenblöcke redundant gespeichert. Die Anzahl der Duplikate ist durch den Replication Factor definiert.

Der **NameNode** verwaltet alle Dateieinträge des HDFS, er organisiert die Aufteilung der Dateien in Blöcke, welche Nodes diese speichern und überwacht den Status des Dateisystems. Falls ein DataNode ausfällt, registriert dies der NameNode und verweist automatisch auf die redundanten Datenblöcke auf anderen DataNodes. Da die Aufgaben des NameNodes sehr speicher- und I/O-intensiv sind, läuft in der Regel nur dieser Dienst auf dem Node. Von allen Diensten im Hadoop-Cluster, ist nur der Ausfall des NameNodes für die Verfügbarkeit problematisch. Aufgrund der hohen Auslastung werden nur sogenannte Snapshots (Speicherauszüge) zur Sicherung auf einem **Secondary NameNode** gespeichert. Im Gegensatz zum NameNode werden auf dem Secondary NameNode keine Echtzeitänderungen des HDFS verarbeitet. Solange kein neuer NameNode mit Hilfe der Snapshots eingerichtet ist, sind die Daten auf den DataNodes unbrauchbar, da keine Informationen über zusammenhängende Datenblöcke vorhanden sind. Des Weiteren sind alle neuen Daten und Änderungen seit dem letzten Snapshot verloren. Der NameNode und der Secondary NameNode können manuell als Active/Passive Cluster konfiguriert werden.

Ein **DataNode**-Dienst wird von jedem Slave Node in einem Hadoop-Cluster bereitgestellt. Diese dienen als Datenspeicher im HDFS und speichern die Datenblöcke auf ihr lokales Dateisystem. Bei einem Dateizugriff stellt der Client zunächst eine Anfrage an den NameNode, um die Anzahl und den Speicherort der Datenblöcke abzufragen. Anschließend kommuniziert der Client direkt mit den entsprechenden DataNodes. Außerdem kommunizieren DataNodes untereinander, um Datenblöcke zu duplizieren. Zusätzlich wird mit dem NameNode kommuniziert, um ihn lokale Änderungen mitzuteilen und um Dateioperationen (Erstellen, Löschen, Umbenennen) abzufragen. Damit ein Ausfall eines DataNodes registriert werden kann, muss dieser in festgelegten Zeitabständen eine Anfrage (Heartbeat) an den NameNode senden, der diese protokolliert. Wird in der vorgegebenen Zeit kein Heartbeat von einem DataNode empfangen, so wird dies vom NameNode als Ausfall interpretiert.

2.2.3. Verteiltes Rechnen

Ein Hadoop-Job wird mit Hilfe des Programmiermodells MapReduce (vgl. 2.2.4) programmiert. Ein Job ist nur eine andere Bezeichnung für eine Anwendung, die einen Algorithmus zur Datenverarbeitung umsetzt. Im Kontext eines Cluster-Systems heißt das, dass diese Anwendung nur ein geschlossener Auftrag von vielen ist.

Der Client reicht einen Job über den **JobTracker** ein. Hierbei wird in der Regel nur der ausführbare Code übermittelt und nicht die zu verarbeitenden Daten, da diese aus dem HDFS gelesen werden. Der JobTracker ermittelt nun, welche Daten aus dem HDFS benötigt werden und auf welchen DataNodes sich diese befinden. Anschließend wird der Job unter Berücksichtigung der Speicherorte in Tasks unterteilt, die an die TaskTracker der zugehörigen DataNodes zugewiesen werden. Der JobTracker erkennt, wenn einzelne Tasks oder Jobs fehlschlagen und startet diese automatisch neu.

Ein **TaskTracker** ist für die Ausführung einzelner Tasks verantwortlich. Er überwacht die Ausführung und informiert den JobTracker über den aktuellen Status. Analog zum DataNode sendet der TaskTracker einen Heartbeat an den JobTracker, damit dieser Aus-

fälle erkennen kann. Jeder Slave Node kann in Abhängigkeit von der Hardwareausstattung mehrere Tasks parallel ausführen. Die Anzahl der maximalen Tasks kann für jeden Slave Node individuell an die Hardware angepasst werden.

Aufgrund der Aufteilung eines Jobs in mehrere Tasks, muss ein Job nicht explizit, z.B. durch Threads, parallelisiert werden. Diese Parallelisierung ist implizit durch Hadoop und das MapReduce-Framework gegeben. Lastet ein Job den Cluster nicht vollständig aus, können weitere Jobs gestartet werden. Das Ausführen voneinander unabhängiger Aufgaben wird als Funktionsparallelität bezeichnet. Bei Hadoop sind diese Funktionen zum einen unterschiedliche Jobs als Einheit, zum anderen aber auch mehrere Tasks von unterschiedlichen Jobs, die entweder parallel auf mehreren Nodes sowie auf einem Node parallel ausgeführt werden. Die Aufteilung der Daten in Datenblöcke auf mehrere DataNodes und der Jobs in einzelne Tasks verdeutlicht die Datenparallelität, die Hadoop massiv nutzt. Hierbei wird der gleiche Code auf unterschiedlichen Daten ausgeführt (vgl. SPMD). Da die zu verarbeitenden Daten in der Regel größer sind als der Code, berücksichtigt Hadoop die Datenlokalität. Ein Task wird vorzugsweise immer auf dem Slave Node ausgeführt, auf dem sich der zu verarbeitende Datenblock befindet. Dadurch wird der Kommunikationsaufwand innerhalb des Clusters reduziert.

2.2.4. MapReduce

MapReduce bezeichnet einerseits ein Programmiermodell zur Parallelverarbeitung und andererseits Frameworks, die Grundfunktionalitäten für Anwendungen bereitstellen, die dieses Programmiermodell nutzen. Hadoop stellt neben den Funktionen zum Cluster-Management ein MapReduce-Framework bereit, mit dem Jobs zur Datenverarbeitung programmiert werden können.

Vereinfacht besteht ein MapReduce-Job aus zwei Phasen:

1. der Map-Phase und
2. der Reduce-Phase.

Diese Phasen werden nacheinander ausgeführt. Die Map-Phase wird vor der Reduce-Phase ausgeführt und reicht die bearbeiteten Daten an diese weiter. Die Eingabedaten werden vom MapReduce-Framework als Schlüssel-Wert-Paare, folgend als Key-Value-Paare bezeichnet, eingelesen und der Map-Phase übergeben. Diese verarbeitet jedes Key-Value-Paar und reicht die bearbeiteten Key-Value-Paare an das MapReduce-Framework weiter. Dieses sammelt für jeden Key alle Values in einer Liste und übermittelt diese Listen der Reduce-Phase. In der Reduce-Phase wird jede Liste verarbeitet und anschließend dem MapReduce-Framework übergeben, welches die Daten speichert. Die Art und Weise der Verarbeitung in den Phasen wird vom Programmierer des Jobs vorgegeben. In Tabelle 2.1 sind die Ein- und Ausgabedaten der einzelnen Phasen aufgelistet.

Phase	Eingabe	Ausgabe
Map	$< Key_1, Value_1 >$	$List(< Key_2, Value_2 >)$
Reduce	$< Key_2, List(Value_2) >$	$List(< Key_3, Value_3 >)$

Tabelle 2.1.: Ein- und Ausgabedaten der Map- und Reduce-Phase

2.2.4.1. MapReduce-Framework

MapReduce-Jobs werden überwiegend in Java geschrieben, da die Hadoop-API ebenfalls in dieser Programmiersprache implementiert ist. Andere Programmiersprachen können mit Hilfe von Hadoop Streaming und Hadoop Pipes verwendet werden. Da die API bereits

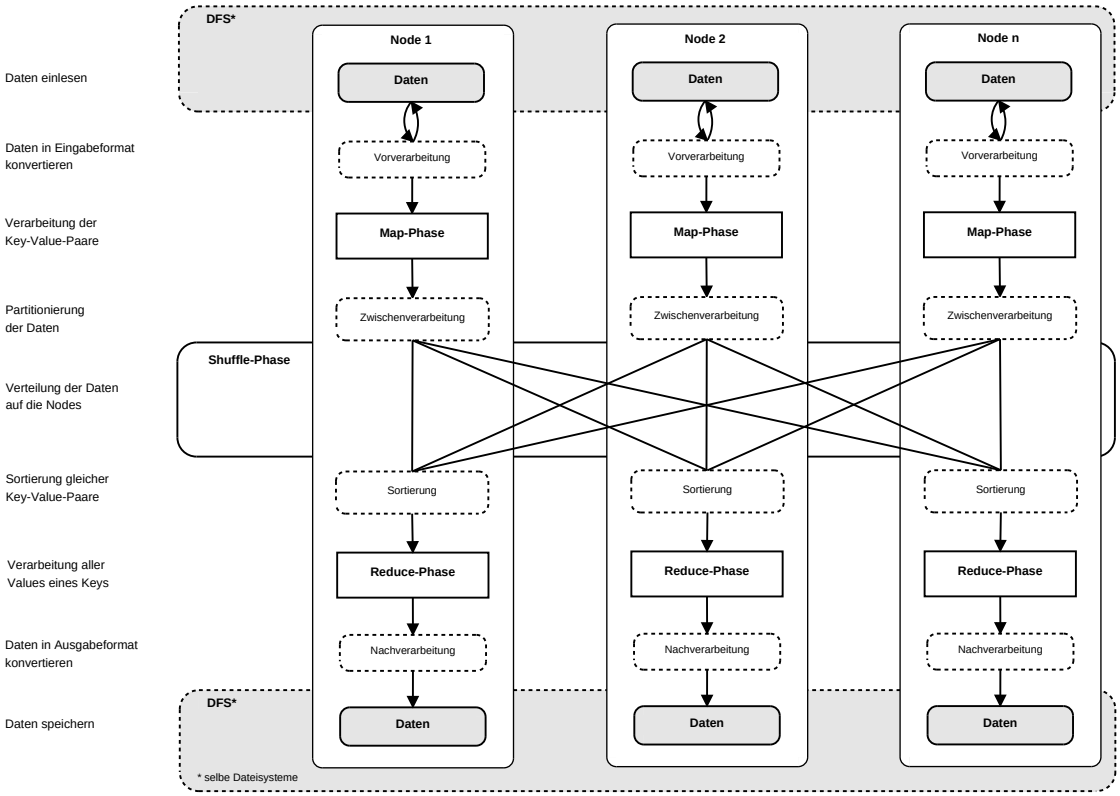


Abbildung 2.6.: Datenfluss eines MapReduce-Jobs

verschiedene Ein- und Ausgabeformate bereitstellt, muss für einen einfachen Job für jede Phase nur eine Klasse erweitert und eine Methode implementiert werden:

- Klasse für Map-Phase:
  `org.apache.hadoop.mapreduce.Mapper`
- Methode für Map-Phase:
  `void map(KEYIN key, VALUEIN value, Mapper.Context context)`
- Klasse für Reduce-Phase:
  `org.apache.hadoop.mapreduce.Reducer`
- Methode für Reduce-Phase:
  `void reduce(KEYIN key, Iterable<VALUEIN> values, Reducer.Context context)`

Das Einlesen der Daten, die Kommunikation zwischen den Nodes sowie das Speichern der Ausgabe wird durch das MapReduce-Framework übernommen. Für komplexe Jobs müssen häufig einzelne Phasen im Datenfluss des Frameworks angepasst werden. Abbildung 2.6 zeigt den Datenfluss eines MapReduce-Jobs. Typischerweise liegen die Daten auf mehreren Slave Nodes im HDFS verteilt. Das Dateiformat dieser Daten ist beliebig und wird durch Hadoop nicht vorgegeben. Bei der Vorverarbeitung werden die Daten vom DataNode gelesen. In Abhängigkeit vom Dateiformat wird entschieden, wie die Daten unterteilt und einzelnen Tasks zugewiesen werden. Zusätzlich müssen die bereits unterteilten Daten in das gewünschte Eingabeformat konvertiert werden. Unterstützt Hadoop das vorhandene Dateiformat nicht, kann für dieses eine eigene Vorverarbeitung implementiert werden.

Jedes Key-Value-Paar wird durch die Map-Methode verarbeitet. Der Methode ist dabei nur das übergebene Key-Value-Paar bekannt. Sie besitzt in der Regel keine Informationen über bereits verarbeitete Daten und wird kontinuierlich mit neuen Daten aufgerufen. Bearbeitete Paare werden dem MapReduce-Framework übergeben.

In der Zwischenverarbeitung werden Paare in Gruppen eingeteilt. Alle Values mit gleichem Key werden derselben Gruppe hinzugefügt. Um den Kommunikationsaufwand in der Shuffle-Phase zu reduzieren, kann optional aus den Values einer Gruppe ein Zwischenergebnis berechnet werden.

In der anschließenden Shuffle-Phase werden die Gruppen an Slave Nodes verteilt. Gruppen mit gleichem Key werden immer an denselben Slave Node versendet. Vor der Übertragung wird jedem Slave Node mitgeteilt, an welchen Slave Node eine Gruppe übertragen wird.

Da ein Slave Node während der Shuffle-Phase mehrere Gruppen mit gleichen und unterschiedlichen Keys empfangen kann, werden die Gruppen anhand des Keys sortiert. Anschließend können Gruppen mit gleichem Key zusammengeführt werden.

Für jede dieser Gruppen wird nun die Reduce-Methode aufgerufen, um alle Values mit gleichem Key zu verarbeiten. Alle bearbeiteten Key-Value-Paare werden dem MapReduce-Framework übergeben. In der Nachverarbeitung werden diese Paare in das definierte Ausgabeformat konvertiert und im HDFS gespeichert.

Die Abbildung 2.6 stellt nur den Datenfluss eines MapReduce-Jobs dar. Anders als der Datenfluss zwischen den einzelnen Phasen werden diese selbst nicht streng sequenziell nacheinander ausgeführt. Ab einer bestimmten Menge an verarbeiteten Daten in der Map-Phase wird von Hadoop die Shuffle- und anschließend die Reduce-Phase eingeleitet, währenddessen gleichzeitig weiter Daten in der Map-Phase verarbeitet werden.

2.2.4.2. Beispiel: WordCount

Als Beispiel eines MapReduce-Jobs soll die Anzahl aller Wörter in mehreren Dokumenten ermittelt werden. Die Dokumente sind im HDFS über mehrere DataNodes verteilt und jeder DataNode kann ebenfalls mehrere Dokumente enthalten.

In der Map-Phase liest jeder Slave Node die lokalen Dokumente zeilenweise ein. Aus jeder Zeile werden die Wörter anhand von Leer- und Satzzeichen extrahiert. Für jedes Wort in einer Zeile wird ein Key-Value-Paar erzeugt, wobei das Wort selbst als Key und der Wert 1 als Value verwendet wird.

Da ein Wort in mehreren Dokumenten vorkommen kann und die Dokumente über die Nodes verteilt sind, müssen identische Wörter in der Reduce-Phase immer an dieselben Slave Nodes übertragen werden. Indem die Map-Phase alle Wörter als Keys überträgt, wird dies durch das Framework sichergestellt. Die Reduce-Methode addiert nun alle empfangenen Values eines Wortes auf und speichert sie. Dieser Algorithmus ist im folgenden Pseudocode umgesetzt:

```
map(key, line):
  words[] = line.split(' ', '.', ',', '!', '?')
  foreach(word in words):
    mapPut(word, 1)
}

reduce(key, values[]):
  count = 0
  foreach(value in values)
    count += value
  reducePut(key, count)
}
```

Als Beispieldaten sind drei Dokumente gegeben, die auf ebenfalls drei Nodes verteilt sind:

```
doc1 = "ich gehe morgen zur arbeit." // Node 1
doc2 = "die arbeit ist morgen fertigzustellen." // Node 2
doc3 = "morgen gehe ich ins kino und nicht zur arbeit." // Node 3
```

In der Map-Phase entstehen diese Ein- und Ausgaben:

```
node1_in = ("doc1", "ich gehe morgen zur arbeit.")
node1_out = ("ich", 1), ("gehe", 1), ("morgen", 1), ("zur", 1), ↩
   ("arbeit", 1)
node2_in = ("doc2", "die arbeit ist morgen fertigzustellen.")
node2_out = ("die", 1), ("arbeit", 1), ("ist", 1), ("morgen", ↩
   1), ("fertigzustellen", 1)
node3_in = ("doc3", "morgen gehe ich ins kino und nicht zur ↩
   arbeit.")
node3_out = ("morgen", 1), ("gehe", 1), ("ich", 1), ("ins", 1), ↩
   ("kino", 1), ("und", 1), ("nicht", 1), ("zur", 1), ↩
   ("arbeit", 1)
```

Die Shuffle-Phase erhält alle Key-Value-Paare als Eingabe:

```
shuffle_in = ("ich", 1), ("gehe", 1), ("morgen", 1), ("zur", ↩
   1), ("arbeit", 1), ("die", 1), ("arbeit", 1), ("ist", 1), ↩
   ("morgen", 1), ("fertigzustellen", 1), ("morgen", 1), ↩
   ("gehe", 1), ("ich", 1), ("ins", 1), ("kino", 1), ("und", ↩
   1), ("nicht", 1), ("zur", 1), ("arbeit", 1)
```

Aus dieser Eingabe werden anschließend Gruppen gebildet:

```
// shuffle_out
grp01 = ("ich", [1, 1])
grp02 = ("gehe", [1, 1])
grp03 = ("morgen", [1, 1, 1])
grp04 = ("zur", [1, 1])
grp05 = ("arbeit", [1, ,1, 1])
grp06 = ("die", [1])
grp07 = ("ist", [1])
grp08 = ("fertigzustellen", [1])
grp09 = ("ins", [1])
grp10 = ("kino", [1])
grp11 = ("und", [1])
grp12 = ("nicht", [1])
```

Diese Gruppen werden möglichst gleichmäßig auf die Nodes verteilt, die diese in der Reduce-Phase verarbeiten:

```
node1_in = ("ich", [1, 1]), ("gehe", [1, 1]), ("morgen", [1, 1, ↩
   1]), ("zur", [1, 1])
node1_out = ("ich", 2), ("gehe", 2), ("morgen", 3), ("zur", 2)
node2_in = ("arbeit", [1, 1, 1]), ("die", [1]), ("ist", [1]), ↩
   ("fertigzustellen", [1])
node2_out = ("arbeit", 3), ("die", 1), ("ist", 1), ↩
   ("fertigzustellen", 1)
node3_in = ("ins", [1]), ("kino", [1]), ("und", [1]), ("nicht", ↩
   [1])
node3_out = ("ins", 1), ("kino", 1), ("und", 1), ("nicht", 1)
```

Nachdem das Framework die Daten im HDFS gespeichert hat, kann das Ergebnis gelesen werden:

```
ich 2
gehe  2
morgen  3
zur 2
arbeit  3
die 1
ist 1
```

```
fertigzustellen 1
ins 1
kino  1
und 1
nicht 1
```

2.3. Open Computing Language

Open Computing Language (OpenCL) ist ein offener Standard der Khronos Group, der die einheitliche Nutzung von CPUs, GPUs und anderen Prozessoren spezifiziert. Dazu wird die Programmiersprache OpenCL C (im Folgenden OpenCL-C) mit der Syntax von ISO C99 genutzt. OpenCL-C selbst ist eine Untermenge von ISO C99 mit Erweiterungen zur Parallelverarbeitung. Die erste Version wurde im Dezember 2008 von der Khronos Group veröffentlicht. Die aktuelle Version 1.2 ist vom November 2011.

Anwendungen, die OpenCL nutzen, können ohne Änderungen auf allen Geräten ausgeführt werden. Vorausgesetzt, dass auf dem System ein OpenCL-Treiber installiert ist und das Gerät sowie der Treiber unterstützen die verwendete Version. Einschränkungen der Plattformunabhängigkeit gibt es bei der Verwendung von Erweiterungen, die laut Standard nicht unterstützt werden müssen, und hardwarespezifischen Optimierungen. Erweiterungen unterscheiden sich in optionale Erweiterungen aus dem Standard und von Herstellern definierte Erweiterungen. Letztere sollten nur genutzt werden, wenn die verwendete Hardware sicher eingeschränkt werden kann.

Viele namhafte Hersteller stellen Implementierungen in Form von Treibern und Programmierplattformen für ihre Geräte bereit, wie zum Beispiel:

- NVIDIA (Grafikkarten)[10],
- AMD (Grafikkarten und x86-Prozessoren mit SSE ab Version 2.x)[11],
- Intel (x86-Prozessoren mit SSE ab Version 4.1)[12] und
- IBM (POWER6 und Cell Prozessoren)[13].

Somit ist OpenCL nicht nur auf Grafikkarten beschränkt, kann aber dennoch für GPGPU genutzt werden.

In diesem Kapitel wird das Plattformmodell und das allgemeine Programmiermodell aus Abschnitt 2.1.3.2 auf Seite 16 in Bezug auf OpenCL genauer erläutert.

2.3.1. Plattformmodell

Das von OpenCL definierte Plattformmodell ist in Abbildung 2.7 auf der nächsten Seite dargestellt. Es besteht aus dem sogenannten Host, einem Computer an den ein oder mehrere Compute Devices angeschlossen sind. Compute Devices sind Geräte die OpenCL unterstützen, wie zum Beispiel Grafikkarten oder Hauptprozessoren im Verbund mit ihren Hauptspeicher. Diese sind weiter unterteilt in ein oder mehrere Compute Units. Bei einer Grafikkarte sind die Compute Units die Shader und bei einem Mehrkernprozessor die einzelnen Prozessorkerne. Eine Compute Unit enthält wiederum ein oder mehrere Processing Elements. Diese werden in der Regel als SIMD-Einheiten genutzt.

Aufgrund des offenen Standards und der vielen unterstützten Geräte können Geräte und Implementierungen von unterschiedlichen Herstellern in einem Host verfügbar sein. Diese können mit Hilfe der OpenCL-Programmierschnittstelle abgefragt werden. In diesem Fall tritt zwischen dem Host und der Compute Devices noch die Plattform in Form eines Treibers. So können in einem Computer z.B. die Compute Devices in Form eines CPUs von AMD und einer GPU von NVIDIA vorhanden sein. In diesem Fall sind die OpenCL-Plattformen bzw. Treiber von AMD und NVIDIA vorhanden.

[10] `http://developer.nvidia.com/category/zone/cuda-zone`
[11] `http://developer.amd.com/zones/OpenCLZone/`
[12] `http://software.intel.com/en-us/articles/vcsource-tools-opencl-sdk/`
[13] `https://www.ibm.com/developerworks/community/groups/service/html/communityview?communityUuid=80367538-d04a-47cb-9463-428643140bf1`

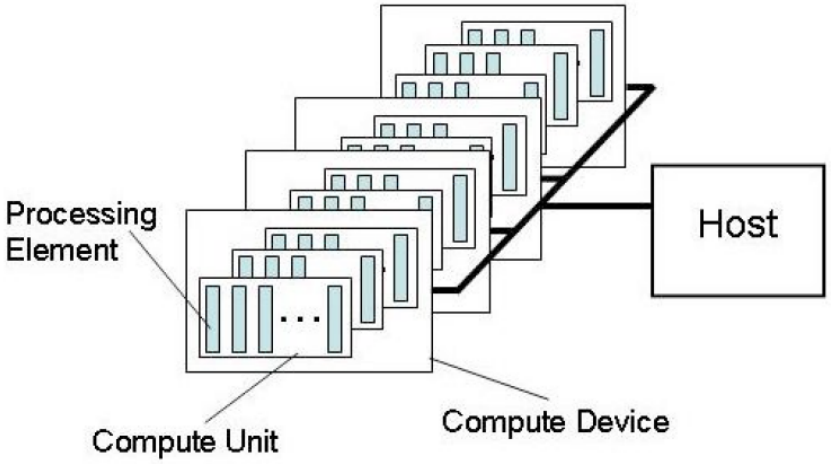


Abbildung 2.7.: Plattformmodell von OpenCL (Quelle: Khronos Group)

2.3.2. Programmiermodell

Das Programmiermodell von OpenCL unterscheidet sich kaum von dem in Abschnitt 2.1.3.2 auf Seite 16 beschriebenen Modell.

Die Programmierschnittstelle von OpenCL enthält Methoden für die Host-Anwendung und den Kernel (siehe [Mun10]). Die API ist in C implementiert, jedoch bietet die Khronos Group offiziell eine C++ Wrapper API an. Alle Methoden für die Host-Anwendung sind im Namensraum `cl` definiert. Darüber können zum Beispiel Eigenschaften über die Devices abgefragt oder Kernel initialisiert und gestartet werden.

OpenCL-Kernel werden in OpenCL-C programmiert, das Methoden, Schlüsselwörter und Datentypen zur Verfügung stellt. Methoden zur Abfrage der jeweiligen IDs, zur Synchronisation und für mathematische Funktionen sind nur einige Beispiele. Folgende Schlüsselwörter können bei der Deklaration von Kernel und Variablen genutzt werden:

__kernel Eine Methode mit dem Präfix `__kernel` kennzeichnet eine Kernel-Methode und kann aus der Host-Anwendung aufgerufen werden.

__global Daten mit dem Präfix `__global` werden im Global Memory gespeichert.

__constant Daten mit dem Präfix `__constant` werden im Constant Memory gespeichert.

__local Daten mit dem Präfix `__local` werden im Local Memory gespeichert. Kernel-Argumente können zwar diesen Präfix benutzen, dürfen dann aber nur den Wert `NULL` übergeben bekommen.

Kernel werden in OpenCL-Kernel und native Kernel unterschieden. OpenCL-Kernel werden erst zur Laufzeit kompiliert. Es ist üblich, die Kernel als String in der Host-Anwendung zu definieren oder sie in eine separate Datei mit dem Suffix `.cl` auszulagern. OpenCL-Kernel sind somit bis auf die zuvor beschriebenen Einschränkungen auf allen OpenCL-Implementierungen lauffähig. Native Kernel werden nicht zur Laufzeit kompiliert und sind von der Implementierung abhängig, da die Hersteller unterschiedliche Zwischensprachen verwenden.

Nachfolgend wird anhand einer Vektoraddition der Ablauf einer OpenCL-Anwendung in C++ beschrieben. Der Kernel ist in der Datei `kernel.cl` gespeichert:

```
__kernel void addVec(__global int* vecC, const __global int* ↩
    vecA, const __global int* vecB, const unsigned int size) {
  unsigned int w = get_global_id(0);
  if(w >= size)
```

```
    return;
  vecC[w] = vecA[w] + vecB[w];
}
```

1. Abfrage der vorhandenen OpenCL-Plattformen.

```
std::vector<cl::Platform> platforms;
cl::Platform::get(&platforms);
```

2. Abfrage aller Plattformen nach Compute Devices vom Typ GPU. Anschließend das erste Device aus der Liste wählen, um auf diesem zu rechnen.

```
std::vector<cl::Device> devices;
std::vector<cl::Device> devTmp;
for (std::vector<cl::Platform>::iterator it = ↩
    platforms.begin(); it != platforms.end(); ++it) {
  it->getDevices(CL_DEVICE_TYPE_GPU, &devTmp);
  devices.insert(devices.end(), devTmp.begin(), devTmp.end());
  devTmp.clear();
}
cl::Device device = devices.front();
```

3. Die Ressourcen zur Ausführung eines Kernels werden in einem `Context` zusammengefasst.

```
cl::Context context = cl::Context(devices);
```

4. Eine `CommandQueue` koordiniert die Ausführung der Kernel.

```
cl::CommandQueue cmdQ = cl::CommandQueue(context, device, ↩
    CL_QUEUE_PROFILING_ENABLE);
```

5. Laden des Programms aus der Datei `kernel.cl` und Erstellen des Kernels `addVector`.

```
std::string src = readFile("kernel.cl");
cl::Program::Sources source;
source.push_back(std::make_pair(src.data(), src.length()));
cl::Program program = cl::Program(context, source);
try {
  program.build(devices);
} catch(cl::Error& err) {
  return EXIT_FAILURE;
}
cl::Kernel kernel = cl::Kernel(program, "addVector");
```

6. Erstellen der OpenCL-Datenstrukturen und Kopieren der Daten aus der Host-Anwendung auf den Grafikkartenspeicher. Neben den skalaren Datentypen unterstützt OpenCL die Datentypen `Buffer`, um 1-dimensionale Felder oder Vektoren zu speichern, und `Image`, um 2- oder 3-dimensionale Texturen, Frame-Buffer oder Bilder zu speichern.

```
cl_int status = CL_SUCCESS; // Fehlercode
cl::Buffer aBuffer(context, CL_MEM_READ_ONLY | ↩
    CL_MEM_COPY_HOST_PTR, sizeof(int) * vecA.size(), ↩
    &vecA[0], &status);
cl::Buffer bBuffer(context, CL_MEM_READ_ONLY | ↩
    CL_MEM_COPY_HOST_PTR, sizeof(int) * vecB.size(), ↩
    &vecB[0], &status);
```

```
cl::Buffer cBuffer(context, CL_MEM_WRITE_ONLY, sizeof(int) ↩
    * vecC.size(), NULL, &status);
```

7. Argumente für den Kernel setzen.

```
status = kernel.setArg(0, cBuffer);
status = kernel.setArg(1, aBuffer);
status = kernel.setArg(2, bBuffer);
status = kernel.setArg(3, vecC.size());
```

8. Kernel zur Ausführung an die `CommandQueue` senden.

```
cl::KernelFunctor func = kernel.bind(cmdQ, ↩
    cl::NDRange(vecC.size()), cl::NDRange(1));
cl::Event event = func();
event.wait();
cmdQ.finish();
```

9. Ergebnis aus dem Grafikspeicher zurück in den Hauptspeicher kopieren.

```
status = cmdQ.enqueueReadBuffer(cBuffer, true, 0, ↩
    sizeof(int) * vecC.size(), &vecC[0]);
```

3. Vorbetrachtung

Dieses Kapitel beleuchtet unterschiedliche Aspekte der Aufgabenstellung. Es werden die Eigenschaften und Konzepte der zwei Technologien, Hadoop und OpenCL, genauer untersucht und in Beziehung zueinander gesetzt. Einige Eigenschaften und Konzepte werden durch Laufzeittests analysiert. Außerdem werden mögliche Probleme aufgezeigt und verschiedene Lösungsansätze diskutiert. Mit Hilfe der gewonnenen Erkenntnisse werden konkrete Lösungswege für die weiteren Untersuchungen dieser Arbeit ausgewählt.

3.1. Parallelisierungsstrategien

Unabhängig von Hadoop gibt es einige Projekte, die das MapReduce-Programmiermodell verwenden, um die Programmierung von parallelen Rechnerarchitekturen zu vereinfachen. Eines der bekanntesten Projekte ist „Phoenix" [RRP+07], das ein MapReduce-Framework für Mehrprozessorsysteme entwickelt. Phoenix vereinfacht die Arbeit mit POSIX[1] Threads, indem die Datenverteilung und Synchronisation vom MapReduce-Framework übernommen werden. Dieses Vorgehen ist mit der Funktion des TaskTrackers von Hadoop vergleichbar, welcher einzelne Tasks durch Java-Threads parallel ausführt. Außerdem gibt es mehrere Untersuchungen, die das MapReduce-Programmiermodell nutzen, um die Programmierung von Grafikkarten zu vereinfachen [SO11, HFL+08]. Ähnlich wie Phoenix, übernimmt das Framework vom Projekt „Mars" die Datenverteilung auf die GPU sowie die Ausführung und Synchronisation der Kernel. Hierbei wird die Funktionsparallelität der Grafikkarte genutzt, um mehrere Tasks parallel auf dieser auszuführen.

Die vorgestellten Untersuchungen implementieren ihr eigenes MapReduce-Framework. In dieser Arbeit soll jedoch ein bestehendes Framework durch eine Grafikkarte beschleunigt werden. Aufbau und Datenfluss von Hadoop bieten verschiedene Möglichkeiten, eine Grafikkarte einzubinden. Diese unterscheiden sich in der Komplexität bei der Umsetzung und der Schicht im Framework, in der die Grafikkarte eingebunden werden soll.

3.1.1. Einbindung der GPU in das Task Scheduling

Die Einfachheit der parallelen Verarbeitung von MapReduce besteht darin, dass ein Task als sequenzielles Programm implementiert wird. Die Parallelität wird erreicht, indem mehrere Tasks parallel auf unterschiedlichen Nodes und mehreren Prozessoren ausgeführt werden. Auf einem Node koordiniert der TaskTracker die Ausführung der Tasks. Dieser hat Kenntnis über das vorhandene Rechnersystem und die möglichen unabhängigen Ausführungseinheiten. Der TaskTracker startet für jeden Task einen Thread, der durch das Scheduling des Betriebssystems einen Prozessorkern zugeteilt wird. Die Anzahl der gleichzeitig ausgeführten Tasks kann begrenzt werden und richtet sich in der Regel nach der Anzahl der vorhandenen Prozessorkerne.

Indem die Grafikkarte als ein Mehrkernprozessor angesehen wird, kann sie in diesen Vorgang eingebunden werden. Da die Grafikkarte vom Betriebssystem nicht als Hauptprozessor verwendet wird und die Threads nicht ohne Änderung auf dieser ausgeführt werden können, müssen die Tasks und der TaskTracker erweitert werden. So muss auch für die Grafikkarte eine maximale Anzahl an Tasks konfiguriert werden, die parallel verarbeitet werden können. Neben der Implementierung für die CPU benötigt jeder Task

[1] Portable Operating System Interface (POSIX) ist eine standardisierte API zur Interaktion zwischen Anwendungen und dem Betriebssystem.

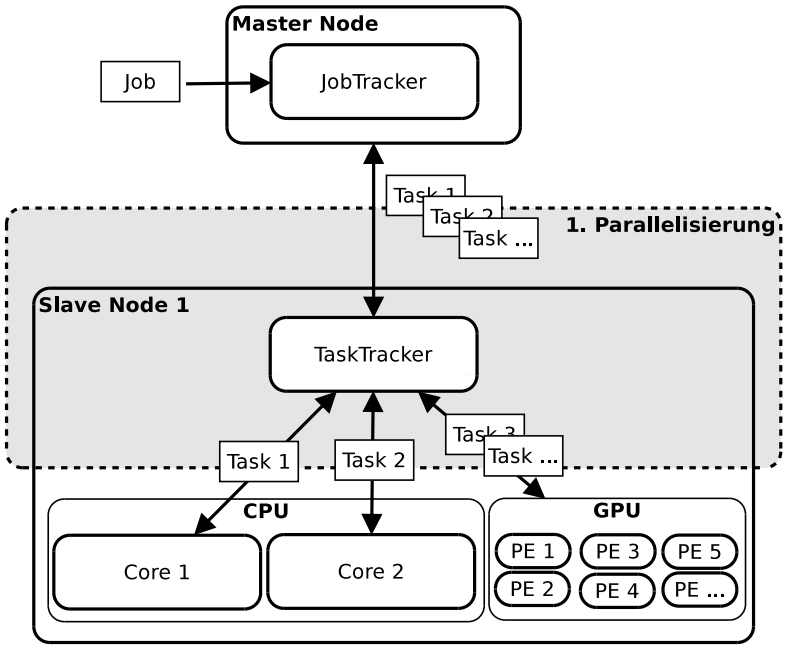


Abbildung 3.1.: Einbindung der GPU in das Task Scheduling

zusätzlich eine Implementierung für die GPU, die vom TaskTracker je nach Bestimmung der Ausführungseinheit ausgewählt werden muss.

Analog zum Hauptprozessor wird ein Task für die Grafikkarte als sequenzielles Programm implementiert und mehrfach parallel ausgeführt. Die Nutzung der Grafikkarte als Mehrkernprozessor ist auch die Idee hinter dem Projekt „Mars" [HFL+08]. Das Scheduling von Tasks zwischen CPU und GPU auf Hadoop wurde in „Hybrid Map Task Scheduling on GPU-based Heterogeneous Clusters" [SSM] untersucht. Der Schwerpunkt dieser Untersuchung ist das Verhältnis zwischen der Ausführungszeit eines Tasks auf der CPU und der GPU. Das Scheduling ist dann optimal, wenn die Differenz der beiden Summen aus der Ausführungszeit aller Tasks auf der CPU bzw. GPU am geringsten ist. Wenn ein Task auf der Grafikkarte um den Faktor 10 schneller als auf dem Hauptprozessor ist, werden 10 mal mehr Tasks der GPU als der CPU zur Verarbeitung zugewiesen.

Abbildung 3.1 stellt dar, wie die Tasks auf die unterschiedlichen Recheneinheiten verteilt werden. Der JobTracker unterteilt einen Job anhand der Datenblöcke im HDFS in mehrere Tasks und weist diese den zugehörigen TaskTrackern zu. Anhand des Scheduling auf den TaskTracker werden die einzelnen Tasks den Prozessorkernen oder der Grafikkarte zugewiesen. Der Job wird nur einmal parallelisiert, indem er in mehrere Tasks unterteilt wird.

3.1.2. Parallelisierung eines Tasks

Eine weitere Möglichkeit besteht darin, einen aufwendigen Task durch Parallelisierung zu beschleunigen. Für eine bessere Auslastung von Mehrkernprozessoren ist dieses Vorgehen bei Hadoop nicht nötig, da dafür nur mehrere Tasks gestartet werden müssen, die durch das Betriebssystem den Prozessorkernen zugeteilt werden.

Um das Potential einer Grafikkarte zur Beschleunigung eines Tasks nutzen zu können, muss der Task explizit für die Grafikkarte programmiert werden. Anders als in 3.1.1 werden nicht mehrere sequenzielle Tasks auf der GPU ausgeführt, sondern nur ein Task, der im besten Fall alle Ausführungseinheiten der Grafikkarte nutzt. Die Effizienz dieser Methode ist stark von der Funktions- oder Datenparallelität abhängig, die ein Task zur Parallelisierung bietet. In der Map-Phase sind mit dem Key-Value-Paar nur wenig Daten zu verarbeiten, weshalb eine Grafikkarte nur schwer ausgelastet werden kann. Die Reduce-

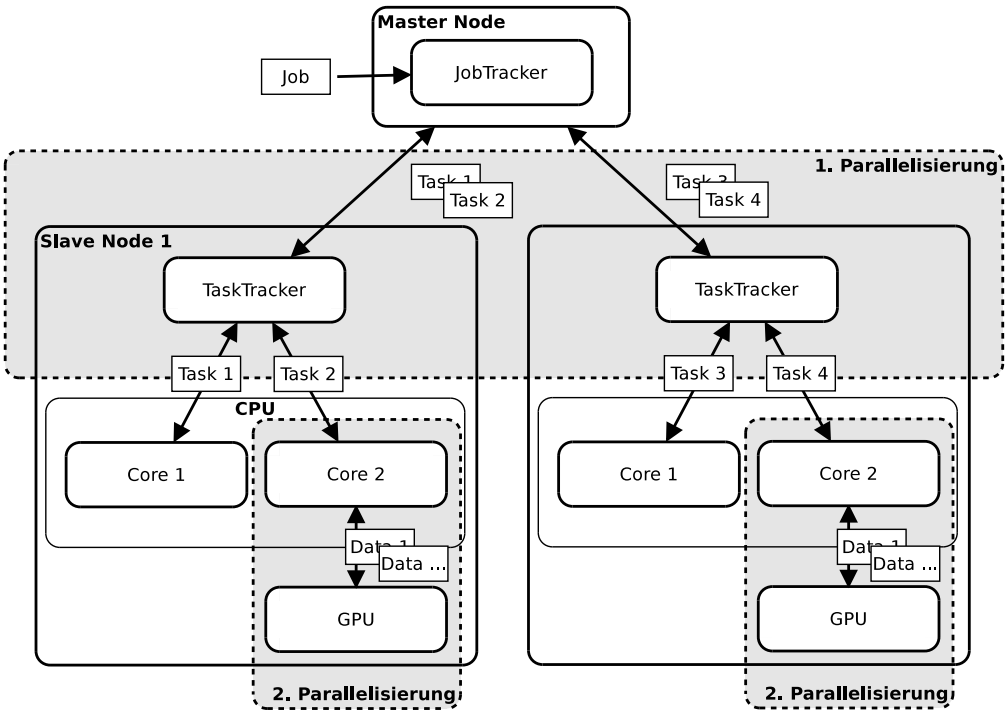


Abbildung 3.2.: Parallelisierung eines Tasks

Phase eignet sich besser für eine Berechnung auf der GPU, da für einen Key mehrere Values zu verarbeiten sind. Die Datenverarbeitung findet in den Methoden `map()` und `reduce()` statt, aus diesem Grund muss auch in diesen die GPU genutzt werden.

Bei diesem Konzept findet eine 2-stufige Parallelisierung statt, die in Abbildung 3.2 verdeutlicht ist. In der ersten Stufe wird ein Job anhand der Datenblöcke im HDFS in mehrere Tasks unterteilt. Anschließend wird ein Task aufgrund seiner Funktions- oder Datenparallelität in der zweiten Stufe parallel auf der GPU verarbeitet. Mit jeder Stufe wird eine Parallelverarbeitung schwieriger, weil die parallelen Teilaufgaben immer kleiner werden und diese weniger Daten verarbeiten müssen.

Da die meisten Grafikkarten bisher nur einen Kernel ausführen können, dürfen auf einem Node nur so viele Tasks mit GPU-Unterstützung gestartet werden, wie Grafikkarten vorhanden sind, um eine gegenseitige Blockierung der Tasks zu vermeiden. Auf einem System mit 16 Prozessorkernen und zwei Grafikkarten bleiben somit 14 Prozessorkerne ungenutzt, wenn nur Tasks mit GPU-Unterstützung ausgeführt werden. Dies kann entweder durch ein Scheduling ähnlich wie in 3.1.1 oder durch eine dynamische Auswahl der Implementierung (CPU oder GPU) im Task vermieden werden.

3.1.3. Auswahl einer Parallelisierungsstrategie

Aufgrund des zeitlichen Rahmens für diese Arbeit wird nur eine der vorgestellten Strategien genauer untersucht. Die Einbindung der GPU in das Task Scheduling ist sehr aufwendig. Dafür muss der TaskTracker von Hadoop erweitert werden, um die Grafikkarte an den Datenfluss anzubinden und ein Scheduling zur optimalen Auslastung der Prozessorkerne und Grafikkarte zu erreichen. Zusätzlich muss jeder Task für die CPU und GPU implementiert werden. Diese Strategie wird nicht untersucht, weil Teilaspekte bereits in unterschiedlichen Projekten umgesetzt wurden und sie sehr zeitaufwendig ist.

Stattdessen wird die Parallelisierung eines Tasks genauer betrachtet. Hierbei werden die Implementierungen für die CPU mit denen für die GPU verglichen. Es werden also entweder alle Tasks auf der CPU oder auf der GPU ausgeführt, jedoch nicht gemischt auf beiden. Ein Scheduling oder eine dynamische Auswahl der Implementierungen wird nicht

umgesetzt, da dies zwar das Rechnersystem besser auslastet, aber keine Erkenntnisse über die Parallelisierung eines Tasks für die GPU liefert.

3.2. Nutzung von OpenCL in einem MapReduce-Job

Die Grafikkarte kann nicht direkt aus Java heraus genutzt werden, da sie in OpenCL, also in den Programmiersprachen C und OpenCL-C, programmiert werden muss. Das MapReduce-Framework bietet mit Hadoop Streaming und Hadoop Pipes zwei Möglichkeiten, beliebige Programmiersprachen zu verwenden. In beiden Varianten muss der gesamte Job in der gewählten Programmiersprache implementiert werden. Dabei können nur wenige Funktionen aus dem Framework genutzt werden. Java Native Access (JNA) und Java Native Interface (JNI) ermöglichen den Zugriff auf plattformspezifische Programmbibliotheken, wie zum Beispiel der OpenCL-API, direkt aus Java heraus. Weil der gesamte Job dabei in Java programmiert wird, können alle Funktionen der Hadoop-API genutzt werden.

3.2.1. Hadoop Streaming

Hadoop Streaming ist eine Funktion, die durch das Framework bereitgestellt wird, um MapReduce-Jobs in beliebigen Programmiersprachen zu erstellen. Dabei ist es egal, ob diese Jobs im Binärformat oder als ausführbares Skript vorliegen. Ein Streaming-Job wird über ein Java Archive (JAR) mit den Eingabeparametern für den Ein- und Ausgabeordner sowie die Programme für den Mapper und Reducer gestartet:

```
$HADOOP_HOME/bin/hadoop jar $HADOOP_HOME/hadoop-streaming.jar
  -input <input dir> -output <output dir>
  -mapper <mapper program> -reducer <reducer program>
```

Die Steuerung und Datenübertragung zwischen Mapper bzw. Reducer und dem JAR findet über die Standardeingabe (`STDIN`) und Standardausgabe (`STDOUT`) statt. Das ist mit der Ausführung von Unix-Programmen in der Shell vergleichbar [Lam10, Pol89]:

```
cat <input file> | <mapper program> | sort | <reducer program> ↩
   > <output file>
```

Die Anwendung `cat` liest die Eingabedatei ein und leitet die Ausgabe an den Mapper weiter. Dieser verarbeitet die Daten, die über die Standardeingabe gelesen werden, und sendet diese an die Standardausgabe. Anschließend sortiert `sort` die Ausgabe vom Mapper und leitet diese an den Reducer weiter. Auch der Reducer liest die Daten über `STDIN` ein und gibt diese nach der Verarbeitung über `STDOUT` aus. Die Dateiumlenkung `>` schreibt die Ausgabedaten in die Zieldatei. Anstatt der Unix-Befehle `cat`, `|`, `sort` und `>` werden bei Hadoop Streaming die Funktionen des Frameworks genutzt.

Aufgrund vieler Einschränkungen ist Hadoop Streaming eher für einfache Jobs geeignet, von denen im besten Fall schon Skripte für die Unix-Shell existieren. Da die Daten über `STDIN` und `STDOUT` übertragen werden, liegen diese nur als Strings vor und die Eingabe muss immer manuell in Key-Value-Paare konvertiert werden. Außerdem werden die Daten in der Shuffle-Phase zwar anhand des Keys sortiert, können aber nicht als getrennte Gruppen an den Reducer übertragen werden. Deswegen muss der Reducer selbständig erkennen, wenn er einen neuen Key bzw. eine neue Gruppe mit neuen Values über `STDIN` einliest. Diese Funktionen werden normalerweise vom MapReduce-Framework übernommen.

3.2.2. Hadoop Pipes

Durch Hadoop Pipes können Jobs in der Programmiersprache C++ entwickelt werden. Die Kommunikation zwischen dem C++-Programm und dem Hadoop-System findet über

Sockets statt.[2] Dazu stellt Hadoop Header-Dateien und Bibliotheken bereit. Ein Job muss mindestens die folgenden Klassen und Methoden von `hadoop/Pipes.hh` implementieren:

```
class Mapper: public Closable {
public:
  virtual void map(MapContext& context) = 0;
};

class Reducer: public Closable {
public:
  virtual void reduce(ReduceContext& context) = 0;
};
```

Die Objekte `MapContext` und `ReduceContext` stellen Methoden bereit, um ähnlich wie im Java-Framework mit Hadoop zu interagieren:

- `context.getInputKey()` liefert den aktuell eingelesenen Key
- `context.getInputValue()` liefert den Value des Keys
- `context.nextValue()` liest den nächsten Value der Gruppe ein (nur in `ReduceContext` verfügbar)
- `context.emit()` übergibt ein Key-Value-Paar an das MapReduce-Framework

Zusätzlich gibt es weitere Methoden, um Informationen über den Verarbeitungsstatus eines Jobs zu setzen oder auszulesen.

Dank dieser Methoden müssen die Key-Value-Paare nicht manuell, wie bei Hadoop Streaming, aus dem Eingabestrom ermittelt werden. Außerdem enthält jeder Methodenaufruf von `reduce()` nur die Daten einer Gruppe. Somit können mit `context.nextValue()` alle Values eines Keys gelesen werden, wodurch eine manuelle Gruppenerkennung überflüssig ist.

Im Vergleich mit Hadoop Streaming bieten Hadoop Pipes eine höhere Funktionalität und einfachere Anwendung. Durch die Verwendung von Methoden anstatt Ein- und Ausgabeströmen werden einige Fehlerquellen bei der Datenübertragung ausgeschlossen.

3.2.3. Java Native Access & Java Native Interface

JNA und JNI sind Java-Programmbibliotheken, die den Zugriff auf plattformspezifische Programmbibliotheken ermöglichen. Mit JNA ist es möglich, auf dem System vorhandene Bibliotheken zu laden und direkt Funktionen aus diesen aufzurufen. Dabei muss kein nativer Code geschrieben werden. Da der Java-Compiler die nativen Funktionsaufrufe nicht prüft, können Laufzeitfehler auftreten, wenn zum Beispiel die genutzte Methode von der plattformspezifischen Programmbibliothek nicht bereitgestellt wird oder die Datentypen nicht übereinstimmen. Dieses Problem umgeht JNI, indem eine Header-Datei aus dem Java-Programm generiert werden muss. Diese Header-Datei muss anschließend mit den gewünschten Funktionen und Bibliotheksaufrufen implementiert und kompiliert werden. JNI hat dadurch den Nachteil, dass zusätzlich plattformspezifischer Code geschrieben werden muss.

Mit beiden Technologien ist es möglich, OpenCL unabhängig vom MapReduce-Framework in Java-Programmen verwenden zu können. Für OpenCL gibt es zwei oft genutzte Bibliotheken, die das bereits umsetzen:

- JOCL[3] nutzt JNI und

[2] `http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/mapred/pipes/package-summary.html`

[3] `http://www.jocl.org/`

- JavaCL[4] nutzt JNA zur Anbindung von OpenCL.

Die API von JOCL gleicht der OpenCL-API in C. Es werden statische Methoden und Indizes als Rückgabewerte verwendet. JOCL nutzt die Objektorientierung von Java nicht und folgt der prozeduralen Programmierung. Anders als JOCL, stellt JavaCL einen objektorientierten Zugang zur OpenCL-API bereit, wodurch es sich komfortabler in Java nutzen lässt.

3.2.4. Auswertung

Die vorgestellten Möglichkeiten zur Anbindung von OpenCL an Hadoop werden anhand eines Beispiel-Jobs genauer untersucht.

Aus einer Menge von Wetterdaten soll für jedes Jahr die Höchsttemperatur ermittelt werden. Die Wetterdaten liegen in Tabellenform vor. Jede Spalte enthält einen Wert, zu dem unter anderen die Wetterstation, die Temperatur und das Jahr zählen. Eine Tabellenzeile enthält die Jahreszusammenfassung einer Wetterstation. Hadoop liest diese Daten zeilenweise ein und die Map-Phase ermittelt das Jahr und die Jahreshöchsttemperatur. Um die Höchsttemperatur eines Jahres ermitteln zu können, müssen alle Temperaturen eines Jahres der gleichen Gruppe zugeordnet werden. Indem das Jahr als Key genutzt wird, findet diese Zuordnung automatisch statt. Abschließend ermittelt die Reduce-Phase mit Hilfe der Grafikkarte aus allen Werten eines Jahres das Maximum.

In Abbildung 3.3 auf der nächsten Seite sind die Laufzeiten der verschiedenen Implementierungen für unterschiedliche Datenmengen auf dem PC dargestellt (Testsystem siehe Anhang A). Der Job mit der Kennzeichnung „Hadoop" nutzt kein OpenCL und dient als Referenz. Es ist zu erkennen, dass alle OpenCL-Varianten langsamer sind als die Referenzimplementierung. Aufgrund der linearen Zeitkomplexität des Problems ist es für GPGPU eher ungeeignet [Pie11]. Da das Interesse jedoch auf die verschiedenen Anbindungen gerichtet ist, wird die Auswertung hierdurch nicht beeinflusst. Im Diagramm ist zuerkennen, dass JOCL und JavaCL kaum langsamer als die Referenzimplementierung sind. Sowohl Hadoop Streaming als auch Hadoop Pipes sind bei 64 MB Daten mindestens 30 Sekunden und bei 256 MB schon 40 Sekunden langsamer als die JavaCL-Variante. JNA und JNI sind also performanter als die zwei Anbindungsmöglichkeiten von Hadoop.

Wenn eine spezifische Programmbibliothek in Java nicht verfügbar ist, kann das Erstellen eines JNA- oder JNI-Zugriffs jedoch sehr aufwendig werden. In diesen Fällen bietet es sich an, Hadoop Streaming oder Hadoop Pipes zu nutzen. Da die Programmierung durch die bereitgestellten Funktionen von Hadoop Pipes komfortabler ist, sollte für die Programmiersprache C++ immer Hadoop Pipes genutzt werden.

Für die weiteren Untersuchungen dieser Arbeit stehen JNA und JNI zur Auswahl, da in diesen Technologien bereits OpenCL-Implementierungen existieren und sie performanter als die von Hadoop bereitgestellten Anbindungsmöglichkeiten sind. Des Weiteren ist ein vollständiger Zugriff auf die Funktionen der Hadoop-API nur in Java möglich. Mit der objektorientierten Umsetzung von JavaCL lässt sich dieses in Java pragmatischer nutzen als JOCL und wird somit für weitere Untersuchen verwendet. Aus dieser Auswahl resultiert der in Abbildung 3.4 auf der nächsten Seite dargestellte Software-Stack, den ein MapReduce-Job nutzt.

[4] `http://code.google.com/p/javacl/`

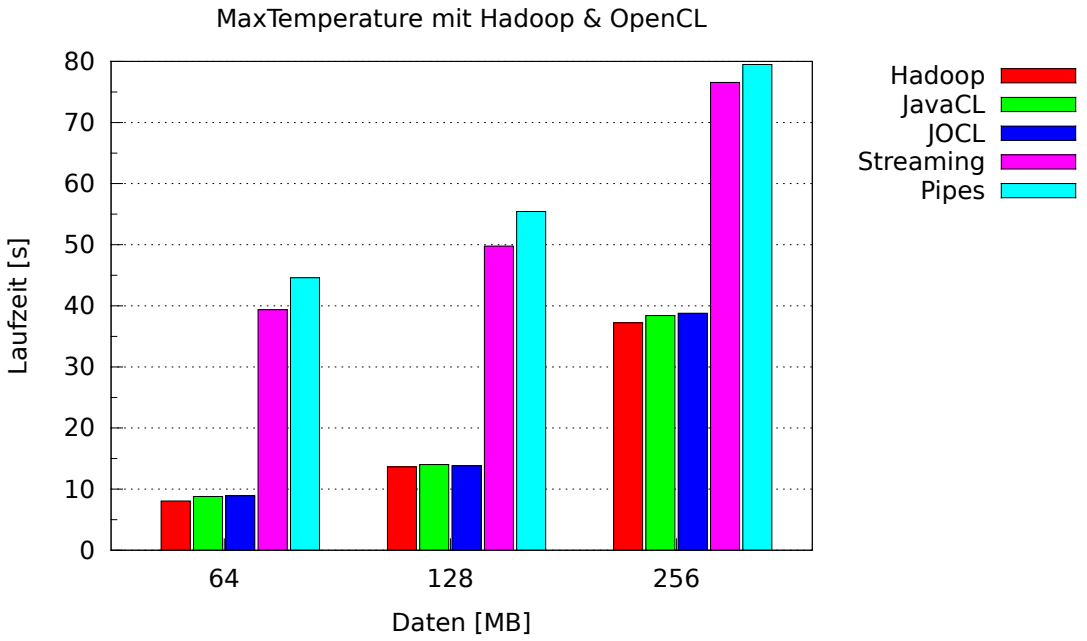


Abbildung 3.3.: Laufzeiten der verschiedenen Implementierungen vom MaxTemperature-Job

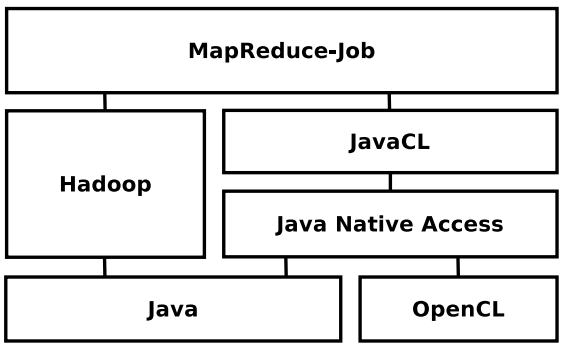


Abbildung 3.4.: Software-Stack eines MapReduce-Jobs mit JavaCL

3.3. Laufzeiten der OpenCL-API

Wie bereits in der Ausarbeitung [Pie11] beschrieben, müssen einige Besonderheiten bei OpenCL berücksichtigt werden. Bevor ein Code auf der Grafikkarte ausgeführt werden kann, müssen zunächst vorhandene Geräte abgefragt und anschließend benötigte Objekte erstellt werden.

Für die sieben wichtigsten Operationen wurden die Bearbeitungszeiten mit der JavaCL-Bibliothek gemessen (vgl. Tabelle 3.1). Die Ergebnisse sind die Mittelwerte aus fünf Testläufen auf den Testsystemen PC und EC2, deren Eigenschaften im Anhang aufgelistet sind. Bis der erste Kernel auf einer Grafikkarte ausgeführt werden kann, vergehen auf dem PC mindestens 528,4 ms und auf der EC2 mindestens 1.887 ms. Diesen Zeiten berücksichtigen jedoch noch keine Datenübertragung. Ein MapReduce-Job sollte alle benötigten OpenCL-Objekte nur einmal laden und ab diesen Zeitpunkt wiederverwenden. Die Initialisierung sollte also nicht in der `map`- bzw. `reduce`-Methode stattfinden, da diese dann für jedes Key-Value-Paar ausgeführt wird. Für diesen Zweck sieht das MapReduce-Framework die `setup`- und `cleanup`-Methode vor. `setup` wird beim Start eines Tasks einmalig vor `map` bzw. `reduce` aufgerufen. Am Ende eines Tasks wird `cleanup` aufgerufen.

Messung	Zeit auf PC (ms)	Zeit auf EC2 (ms)
Plattformabfrage	73,0	617,8
Geräteabfrage	< 1	< 1
Context-Erstellung	2,0	1.255,8
CommandQueue-Erstellung	249,8	0,6
Programm-Erstellung	101,8	6,4
Kernel-Erstellung	101,8	6,4
Buffer-Erstellung	< 1	< 1

Tabelle 3.1.: Laufzeiten wichtiger OpenCL-API-Funktionen

Einen weiteren Einfluss auf den Geschwindigkeitsvorteil einer Grafikkarte gegenüber einem Hauptprozessor hat die Datenübertragung. Vor der Berechnung müssen die Eingabedaten aus dem Hauptspeicher auf den Grafikkartenspeicher geschrieben und nach der Berechnung die Ausgabedaten aus dem Grafikkartenspeicher gelesen werden. Nur wenn die Übertragungszeit schneller als die Berechnung auf dem Hauptprozessor ist, kann ein Geschwindigkeitsvorteil bei der Berechnung auf der GPU erzielt werden. Die Map-Phase verarbeitet jedes Key-Value-Paar einzeln, wodurch viele Daten in einem separaten Kopiervorgang auf die Grafikkarte übertragen werden. Auch in der Reduce-Phase können die Values nur nacheinander gelesen werden, sodass ebenfalls viele Kopiervorgänge stattfinden. Da einzelne Kopiervorgänge sehr langsam sind, werden normalerweise mehrere Daten mit Hilfe eines Feldes in einem Vorgang übertragen. Der Laufzeitunterschied zwischen vielen Einzelübertragungen und einer zusammenhängenden Übertragung wurde für das Lesen aus und das Schreiben auf den Grafikkartenspeicher gemessen. Die Ergebnisse aus Tabelle 3.2 auf der nächsten Seite sind die Mittelwerte aus fünf Messungen.

Die Messungen belegen, dass viele einzelne Übertragungen wesentlich langsamer sind, als wenn die gleiche Datenmenge in einer Übertragung gelesen oder geschrieben werden würde. So dauert das Lesen von 65.536 int-Werten durch ebenso viele Lesevorgänge aus dem Grafikkartenspeicher 5.758,6 ms auf dem PC. Hingegen werden nur 4,8 ms benötigt, wenn die gleiche Datenmenge zusammenhängend als ein Feld gelesen wird. Schlussfolgernd sollte das Lesen oder Schreiben von einzelnen Daten vermieden und stattdessen viele Daten zusammenhängend als Feld übertragen werden. Dies kann zum Beispiel durch einen Puffer realisiert werden, indem Daten in diesem gesammelt und erst dann zusammenhängend übertragen werden, wenn dieser gefüllt ist.

Messung	Zeit auf PC (ms)	Zeit auf EC2 (ms)
Lesen: 128 Einzelwerte	18,6	86,2
Lesen: Feld mit 128 Werten	0,2	1,2
Lesen: 65.536 Einzelwerte	5.758,6	66.706,6
Lesen: Feld mit 65.536 Werten	4,8	4,2
Schreiben: 128 Einzelwerte	67,2	100,2
Schreiben: Feld mit 128 Werten	0,6	0,8
Schreiben: 65.536 Einzelwerte	29.332,0	43.817,2
Schreiben: Feld mit 65.536 Werten	3,8	4,0

Tabelle 3.2.: Laufzeiten für das Lesen/Schreiben von int-Werten durch mehrere Operationen und durch eine Operation mit Hilfe eines Feldes

3.4. Unterschiedliche Datenorganisation

Ein MapReduce-Job hat eine andere Datenorganisation und anderen Datenfluss als ein Programm, das für einen „konventionellen" Computer (SISD-Prinzip) programmiert wird. Für eine gute Performance muss aber ein Weg gefunden werden, der sowohl MapReduce als auch GPGPU gerecht wird.

3.4.1. Datenorganisation einer GPGPU-Anwendung

In der Regel lesen herkömmliche Programme alle Eingabedaten vor der Verarbeitung in den Hauptspeicher. Bei diesem Vorgang kann mit Funktionen wie `fseek` und `lseek` an unterschiedliche Stellen in der Eingabedatei gesprungen werden [CF90], wodurch ein wahlfreier Zugriff auf jedes einzelne Datum möglich ist. Spätestens nach dem Einlesen ist die genaue Größe der Datenmenge bekannt. Bei der weiteren Verarbeitung ist der wahlfreie Zugriff von der gewählten Datenstruktur abhängig.

Wie in Abschnitt 3.3 beschrieben, sieht OpenCL die Übertragung mit Puffern in Form von Feldern vor. Dies hat neben einer höheren Performance den Vorteil, dass die Größe der Eingabemenge bzw. die Anzahl der Daten immer bekannt ist. Für die Verteilung der Daten zwischen Haupt- und Grafikkartenspeicher ist die Kenntnis über die Datengröße wichtig. Der Grafikkartenspeicher ist häufig kleiner als der Hauptspeicher (siehe Anhang A), weshalb große Eingabemengen nur in mehreren Teilen von der GPU verarbeitet werden können. Ist die Größe der Eingabedaten bekannt, können diese auf einfache Weise so unterteilt werden, dass die Grafikkarte gut ausgelastet werden kann. Mit Hilfe der OpenCL-API kann die Größe des Grafikkartenspeichers zur Laufzeit eines Programms ermittelt werden.

Der Pseudocode aus Algorithmus 3.1 setzt dieses Vorgehen um. In der ersten Zeile wird die Eingabedatei `inputFile` in den Hauptspeicher gelesen. Die Variable `gpuSize` erhält die Größe des Grafikkartenspeichers. Über die Variablen `start` und `count` werden die Eingabedaten in Blöcke unterteilt. Die Größe eines Blocks ist das Minimum aus der Größe der Eingabemenge und des Grafikkartenspeichers. Wenn die Eingabedaten kleiner als der Grafikkartenspeicher sind, können diese in einem Block übertragen werden. Nachdem ein Datenblock mit der Methode `openCL.copyTo()` auf die Grafikkarte kopiert wurde, kann die Berechnung auf der GPU mit `openCL.execute()` gestartet werden. Anschließend werden die Ergebnisse mit `openCL.copyFrom()` in den Hauptspeicher kopiert und ggf. zusammengetragen.

3.4.2. Einfache Datenorganisation eines MapReduce-Jobs mit GPGPU

Das MapReduce-Framework liest die Daten aus dem HDFS in Form eines Datenstroms. Ein Datenstrom ist eine kontinuierliche Abfolge von Datensätzen, deren Ende während

```
1  input[] = read(inputFile)
2  inputSize = size(input)
3  gpuSize = openCL.getRamSize(device)
4  start = 0
5  count = gpuSize
6  while(start < inputSize):
7     count = min(count, inputSize - start)
8     openCL.copyTo(device, input, start, count)
9     openCL.execute(kernel);
10    openCL.coptyFrom(device, result, start, count)
11    start += count
12 accumulate(result)
```

Algorithmus 3.1: Blockweise Zerlegung der Daten und Berechnung auf der GPU

der Verarbeitung nicht abzusehen ist. Somit ist die Größe der Eingabemenge zur der Laufzeit unbekannt. Aufgrund des verteilten Dateisystems und der Aufteilung der Daten in Blöcke ist die Bestimmung der Datengröße nur mit sehr viel Aufwand möglich. Doch auch wenn die Größe der Eingabedaten bekannt wäre, sieht das MapReduce-Framework in der Map-Phase nur die sequenzielle Verarbeitung eines Key-Value-Paares vor. Obwohl in der Reduce-Phase für einen Key eine Liste von Values verarbeitet wird, ist auch hier die Größe der Daten unbekannt, da der Zugriff auf die Liste nur durch einen Iterator möglich ist. Aufgrund der Eigenschaft eines Datenstroms stellt der Iterator keine Methode zur Verfügung, um die Größe der Liste zu ermitteln. Außerdem können die Values nur einmal sequenziell gelesen werden. Eine Einzelübertragung, wie sie in der Map- und der Reduce-Phase ausgeführt werden müsste, ist für eine GPU jedoch ungeeignet (siehe Tabelle 3.2 auf der vorherigen Seite).

Würden alle Key-Value-Paare der Map-Phase und alle Values eines Keys in der Reduce-Phase im Hauptspeicher zwischengespeichert werden, könnten diese Daten zusammenhängend auf den Grafikkartenspeicher übertragen werden. Der Pseudocode aus Algorithmus 3.2 speichert die Daten der Map-Phase zwischen. Anders als vom MapReduce-Framework vorgesehen, speichert die `map`-Methode nur alle Daten in einer Liste, ohne diese zu verarbeiten. Wenn alle Daten aus dem HDFS gelesen wurden, ruft das Framework die Methode `cleanup()` auf. Diese konvertiert die Liste zu einem Feld, um dieses anschließend auf den Grafikkartenspeicher zu übertragen.[5] Nach der Berechnung auf der GPU werden die Ergebnisse gelesen und dem Framework übergeben. Da die `reduce`-Methode der Reduce-Phase für jede Gruppe separat aufgerufen wird, werden die Methoden `setup()` und `cleanup()` nicht benötigt, wie in Algorithmus 3.2 zu sehen ist.

Dieses einfache Vorgehen ist jedoch nicht praxistauglich. Auf einem Node werden in der Regel so viele Daten verarbeitet, dass diese nicht komplett in den Hauptspeicher zwischengespeichert werden können. Des Weiteren ist die Speicherorganisation sehr aufwendig. Aufgrund der unbekannten Datenmenge muss eine Liste zur Zwischenspeicherung verwendet werden, die anschließend in ein Feld mit fester Größe kopiert werden muss:

```
input[] = List.toArray(list)
```

Obwohl in der `reduce`-Methode nur Teile der Eingabemenge gespeichert werden müssen, überwiegt auch hier der Aufwand zur Speicherorganisation. Mit jedem Aufruf von `reduce()` steigt die Speicherauslastung der Java Virtual Machine (JVM). Erst wenn diese einen Schwellenwert übersteigt oder kein freier Speicher zur Verfügung steht, sucht die Garbage Collection (GC), die automatische Speicherbereinigung der JVM, nach unbenutzten Objekten, um diese zu löschen. Dieser Vorgang würde in Abhängigkeit von der Größe der Eingabedaten und des verfügbaren Hauptspeichers kontinuierlich gestartet

[5] Bei großen Datenmengen können diese auch in Blöcke unterteilt und dann übertragen werden.

```
Mapper:
  setup():
    list = List()

  map(key, value):
    list.add(key, value)

  cleanup():
    input[] = List.toArray(list)
    openCL.copyTo(device, input)
    openCL.execute(kernel)
    results[] = openCL.copyFrom(device)
    foreach(r in results):
      mapPut(r.key, r.value)

Reducer:
  reduce(key, iterator):
    list = List()
    foreach(value in iterator):
      list.add(key, value)
    input[] = List.toArray(list)
    openCL.copyTo(device, input)
    openCL.execute(kernel)
    results[] = openCL.copyFrom(device)
    foreach(r in results):
      reducePut(r.key, r.value)
```

Algorithmus 3.2: Einfache Zwischenspeicherung in der Map- und Reduce-Phase

werden und einen erheblichen Teil der Rechenzeit in Anspruch nehmen.

Diese einfache Datenorganisation hat somit zwei negative Eigenschaften:

- Speicherüberlauf bei zu großer Eingabemenge und
- eine hohe Anzahl von GC-Aufrufen.

Da Hadoop für die Verarbeitung von riesigen Datenmengen entwickelt und eingesetzt wird, sollte von einer Implementierung kein Speicherlimit vorgegeben werden, um Speicherüberläufe zu verhindern. Obwohl die GC die funktionelle Abarbeitung des Jobs nicht beeinflusst, kann durch eine schlechte Datenorganisation die Laufzeit erheblich beeinflusst werden.

3.4.3. Optimierte Datenorganisation eines MapReduce-Jobs mit GPGPU

Die Einschränkungen aus Abschnitt 3.4.2 sind stark voneinander abhängig. Der Aufwand der Speicherbereinigung richtet sich nach der Anzahl der nicht mehr benötigten Objekte, die zu löschen sind, und dem belegten Speicher. Die Speicherorganisation in Algorithmus 3.2 ist sehr aufwendig, da die Felder `input` und `result` sehr viel zusammenhängenden Speicher benötigen. Dies verursacht viele GC-Aufrufe, auch wenn noch genügend Speicher verfügbar ist, da der Speicher im Laufe der Zeit fragmentiert und somit nur kleine Speicherbereiche alloziert werden können. Die Fragmentierung wird verstärkt, indem mit jedem `reduce`-Aufruf neue Felder unterschiedlicher Größe alloziert werden.

Die Fragmentierung und Speicherauslastung kann durch die Wiederverwendung der Felder minimiert werden. Nur wenn ein bereits alloziertes Feld zu klein ist, muss ein neues alloziert werden. Da bei diesem Vorgehen die Feldgröße nicht mit den enthaltenen Daten übereinstimmen muss, benötigt jedes Feld eine Variable, um die aktuelle Anzahl

```
Buffer:
  count = 0
  size = 65536
  data[] = Array(size)

  put(key, value):
    if(count < size)
      data[count++] = key
      data[count++] = value
      return TRUE
    else
      count = count % size
      return FALSE

  isFull():
    return count >= size

  hasData():
    return count > 0

  reset():
    count = 0
```

Algorithmus 3.3: Puffer mit automatischen Rücksetzen

der enthaltenen Daten zu zählen. Eine feste Feld- bzw. Puffergröße vereinfacht die Wiederverwendung und verhindert einen drohenden Speicherüberlauf. Sowohl in der `map`- als auch der `reduce`-Methode wird der Puffer befüllt. Wenn der Puffer vollständig befüllt ist oder keine Eingabedaten mehr vorhanden sind, wird er auf den Grafikkartenspeicher kopiert. Die GPU verarbeitet anschließend die Daten. Nachdem die Resultate der GPU gelesen wurden, kann der Puffer neu befüllt werden und der Vorgang wiederholt sich solange, bis alle Daten verarbeitet wurden. In der Map-Phase ist diese blockweise Verarbeitung problemlos möglich, da alle Key-Value-Paare unabhängig voneinander sind. Hingegen können in der Reduce-Phase für jede Gruppe mehrere Teilergebnisse berechnet werden, wenn die Anzahl der Values die Puffergröße übersteigt. Aus diesen Teilergebnissen muss abschließend ein Endergebnis berechnet werden. Da eine „herkömmliche" `reduce`-Methode die Values ebenfalls nur nacheinander lesen kann, ist dieses Vorgehen keine Einschränkung gegenüber einer Implementierung ohne GPU-Unterstützung. Der Pseudocode in Algorithmus 3.3 setzt einen Puffer um, der kontinuierlich über die `put`-Methode gefüllt werden kann. Wenn der Puffer voll ist, liefert diese Methode den Wert `FALSE` und der Aufrufer kann zum Beispiel eine Berechnung starten.

Mit der Verwendung dieses Puffers können die Probleme aus Abschnitt 3.4.2 gelöst werden. Der Algorithmus 3.4 auf Seite 48 verwendet den in 3.3 vorgestellten Puffer für die Map-Phase. Mit jedem Aufruf von `map()` wird der Puffer gefüllt und wenn dieser keine Daten mehr speichern kann, wird mit `computeOpenCL()` die Berechnung auf der GPU gestartet. Am Ende jeder Map-Phase können im Puffer noch nicht verarbeitete Daten vorhanden sein. Dies wird in der `cleanup`-Methode geprüft und ggf. eine abschließende Verarbeitung gestartet. Die Verarbeitung der Reduce-Phase ist mit der Map-Phase vergleichbar, jedoch werden alle Values eines Keys in der `reduce`-Methode verarbeitet. Die Funktionalität von `cleanup()` wird somit am Ende jeder `reduce`-Methode benötigt. Außerdem muss der Puffer vor der Verarbeitung einer neuen Gruppe zurückgesetzt werden, da sonst Values unterschiedlicher Keys vermischt werden würden.

Um die zwei Algorithmen übersichtlich darstellen zu können, wurde auf die Wiederverwendung der Ergebnisfelder verzichtet. Außerdem berechnet die Reduce-Phase für

jeden Value ein neues Ergebnis, ohne diese zu einem Endergebnis zu akkumulieren. Die Performance und Auslastung der Grafikkarte ist von der Speicherbelegung und dem Rechenaufwand abhängig. Für einfache Akkumulationsoperationen ist der Kopieraufwand häufig größer als der Rechenaufwand, auch wenn genügend Daten vorhanden sind. Im besten Fall hat der Puffer dieselbe Größe wie der Grafikkartenspeicher, da so der Speicher immer ausreichend gefüllt ist. Da jeder Task auf einem Node in einer JVM ausgeführt wird, ist der verfügbare Speicher in einer Hadoop-Standardkonfiguration auf 1 GB begrenzt. Dieser Speicher muss für alle Daten der Map- oder Reduce-Phase ausreichen. Aus diesem Grund ist der Puffer häufig um ein Vielfaches kleiner als der verfügbare Grafikkartenspeicher (siehe Anhang A). Bei Akkumulationsoperationen können mehrere Pufferinhalte hintereinander auf die Grafikkarte kopiert werden. Erst wenn der Grafikkartenspeicher belegt ist, wird die Verarbeitung der angesammelten Daten gestartet. Für typische Akkumulationsoperationen, wie Summe oder Maximum, sind diese Funktionen mit Hilfe einer Zwischenschicht leicht zu implementieren.

```
Mapper:
  setup():
    buffer = Buffer()

  map(key, value):
    if(!buffer.put(key, value))
      computeOpenCL(buffer)
      buffer.put(key, value)

  computeOpenCL(buffer):
    openCL.copyTo(device, buffer.data, 0, buffer.count)
    openCL.execute(kernel)
    results[] = openCL.copyFrom(device)
    foreach(r in results)
      mapPut(r.key, r.value)

  cleanup():
    if(buffer.hasData)
      computeOpenCL(buffer)
      buffer.reset()

Reducer:
  setup():
    buffer = Buffer()

  reduce(key, iterator):
    buffer.reset()
    foreach(value in iterator):
      if(!buffer.put(key, value))
        computeOpenCL(buffer)
        buffer.put(key, value)
    if(buffer.hasData)
      computeOpenCL(buffer)

  computeOpenCL(buffer):
    openCL.copyTo(device, buffer.data, 0, buffer.count)
    openCL.execute(kernel)
    results[] = openCL.copyFrom(device)
    foreach(r in results)
      reducePut(r.key, r.value)
```

Algorithmus 3.4: Map- und Reduce-Phase mit Puffer fester Größe

4. Beispielimplementierungen

Dieses Kapitel beschreibt die Umsetzung von zwei Beispiel-Jobs: das k-Means-Verfahren zur Clusteranalyse und die numerische Integration. Anhand dieser Jobs werden die zuvor erarbeiteten Erkenntnisse praktisch erprobt. Der Schwerpunkt dieser Untersuchung liegt dabei nicht auf den speziellen Eigenschaften der Lösungsverfahren, sondern in ihrer algorithmischen Struktur und den daraus resultierenden Parallelisierungsmöglichkeiten unter Berücksichtigung von MapReduce und GPGPU. Die Implementierungen werden anhand ausgewählter Quelltextbeispiele und UML-Diagramme erklärt. Der gesamte Quelltext ist auf der beiliegenden CD-ROM hinterlegt.

4.1. OpenCL-Komponenten

Bereits im Masterprojekt [Pie11] hat der Autor auf die Notwendigkeit hingewiesen, dass die Entwicklung von OpenCL-Anwendungen schrittweise und in unabhängigen Modulen erfolgen sollte. Die kleinstmögliche funktionelle Komponente einer OpenCL-Anwendung aus Sicht des Host ist der Kernel. Dieser implementiert die Parallelverarbeitung auf der Grafikkarte. Jedoch muss auch die Host-Anwendung die Implementierung des Kernels berücksichtigen, um zum Beispiel die Daten im richtigen Format zu liefern und die benötigte Anzahl von Work-Items zu starten. Ein Programmierer kommt somit an zwei Ebenen mit OpenCL in Berührung. Die unterste Ebene ist der in OpenCL-C programmierte Kernel. Die darüber liegende zweite Ebene implementiert mit Hilfe eines Kernels eine komplexe Operation. In dieser Ebene werden die Daten gegebenenfalls in ein anderes Datenformat konvertiert, auf die Grafikkarte oder von ihr kopiert und der Kernel mit der richtigen Anzahl von Work-Items und Work-Groups gestartet.

Aus softwaretechnischer Sicht ist es wünschenswert, wenn in beiden Ebenen möglichst kleine und unabhängige Komponenten gebildet werden können. Dieses Prinzip der losen Kopplung vereinfacht die Entwicklung und Wartbarkeit, da einzelne Komponenten weniger komplex sind und mit Hilfe von Unit-Tests, aufgrund der geringen Abhängigkeit zu anderen Komponenten, leichter getestet werden können. Obwohl damit komplexe Algorithmen durch die Komposition verschiedener Komponenten leichter programmiert werden können, hat dieses Prinzip für OpenCL einen Nachteil: Jede Komponente startet einen Kernel und konvertiert gegebenenfalls Daten. Im Vergleich zu einem komplexen Kernel beeinflusst dieser Mehraufwand die Performance negativ.

In großen Softwareprojekten stehen die Belange der Wartbarkeit und Wiederverwendung einzelner Komponenten oft über denen der Performance. Mit den zwei Schnittstellen `ICLKernel` und `ICLBufferedOperation` werden die OpenCL-Ebenen abstrahiert.

4.1.1. Ebene der OpenCL-Kernel

Trotz der Verwendung von JavaCL muss ein OpenCL-Kernel weiterhin in OpenCL-C programmiert werden. Eine Typüberprüfung der Parameter an dieser Schnittstelle zwischen dem JavaCL-Aufruf und dem OpenCL-Kernel findet erst zur Laufzeit und nicht während der Kompilierung statt. Anders als in Java üblich müssen die Argumente mit einer Positionsangabe durch die Methode `CLKernel.setArg(int, CLMem)` übertragen werden. Dadurch ist keine nutzbare Schnittstelle vorhanden, die durch Typen und Namen der Argumente dem Programmierer eine korrekte Nutzung vorgibt.

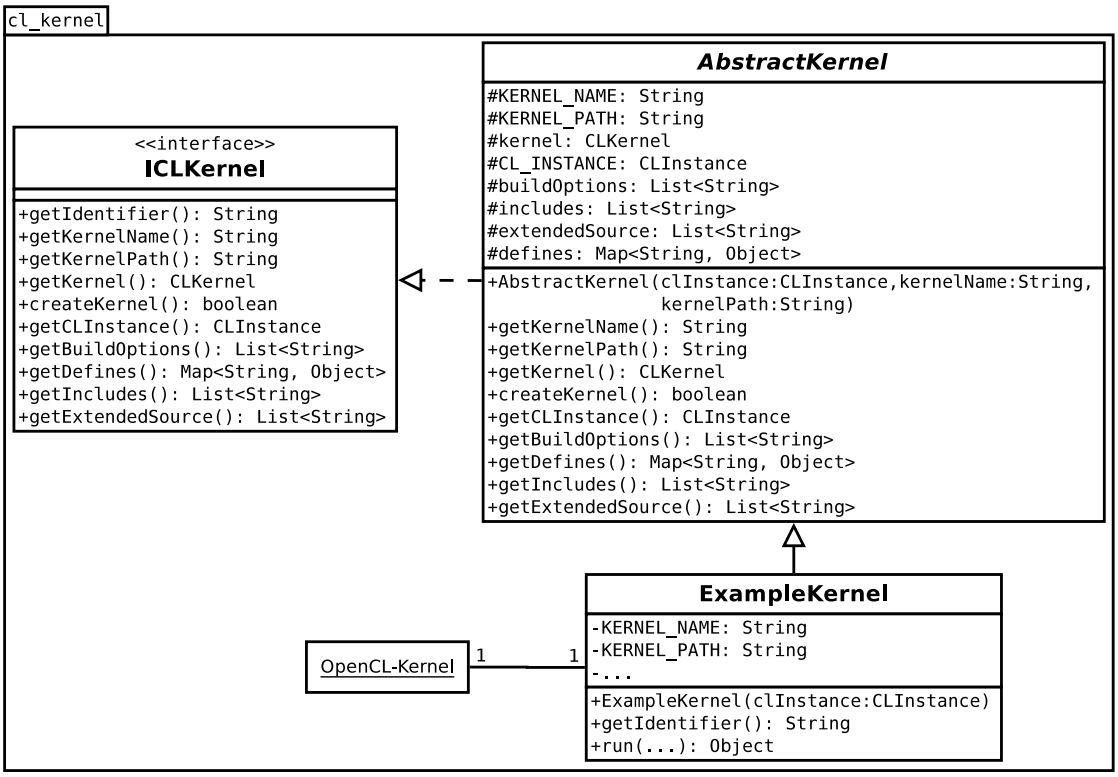


Abbildung 4.1.: Klassendiagramm zur Veranschaulichung der Abstraktion zwischen dem in OpenCL-C geschriebenen Kernel und der zugeordneten Klasse in Java

Mit dem Entwurfsmuster Adapter kann eine neue Schnittstelle für einen OpenCL-Kernel in Java erstellt werden. Adapter werden auch häufig als „Wrapper" (dt. Verpackung) bezeichnet, weil sie ein Programmstück „umhüllen" und oft um Funktionalitäten erweitern. Der Adapter dient zur Übersetzung der typsicheren Java-Schnittstelle in die unkomfortablere OpenCL-Schnittstelle. Für jeden OpenCL-Kernel muss also eine Klasse als Adapter in Java implementiert werden.

Die Abbildung 4.1 zeigt die gewählte Struktur für das beschriebene Vorgehen. Die Klasse `ExampleKernel` ist der Adapter für genau einen OpenCL-Kernel. Die Methode `ExampleKernel.run()` stellt die Schnittstelle dar, die vom Programmierer genutzt wird. Da die Klasse in Java implementiert ist, kann dieser Aufruf bei der Kompilierung geprüft werden. Die korrekte Verwendung des OpenCL-Kernels muss von `ExampleKernel` sichergestellt werden, weshalb diese Klasse immer vom Entwickler des OpenCL-Kernels erstellt werden sollte. Die fehlerfreie Verbindung zwischen dem Java-Kernel und dem OpenCL-Kernel kann mit Unit-Tests leicht überprüft werden.

Das Erstellen eines OpenCL-Kernels folgt immer dem gleichen Ablauf und unterscheidet sich nur in unterschiedlichen Parametern wie Defines oder Includes. Wenn die benötigten Parameter für einen Kernel bekannt sind, kann die Erstellung durch eine zentrale Methode durchgeführt werden. Dafür muss ein Java-Kernel die Schnittstelle `ICLKernel` mit der Methode `createKernel()` implementieren. Die abstrakte Klasse `AbstractKernel` implementiert einige Methoden von `ICLKernel` und stellt benötigte Variablen bereit. Damit nicht jeder Java-Kernel alle Methoden von `ICLKernel` neu implementieren muss, können diese von `AbstractKernel` erben. Eine konkrete Implementierung muss somit nur die `run`- und `getIdentifier`-Methode umsetzen. Zur Erstellung des OpenCL-Kernels benötigte Parameter müssen im Konstruktor in die entsprechenden Datenstrukturen hinzugefügt werden: `buildOptions`, `includes`, `extendedSource` und `defines`.

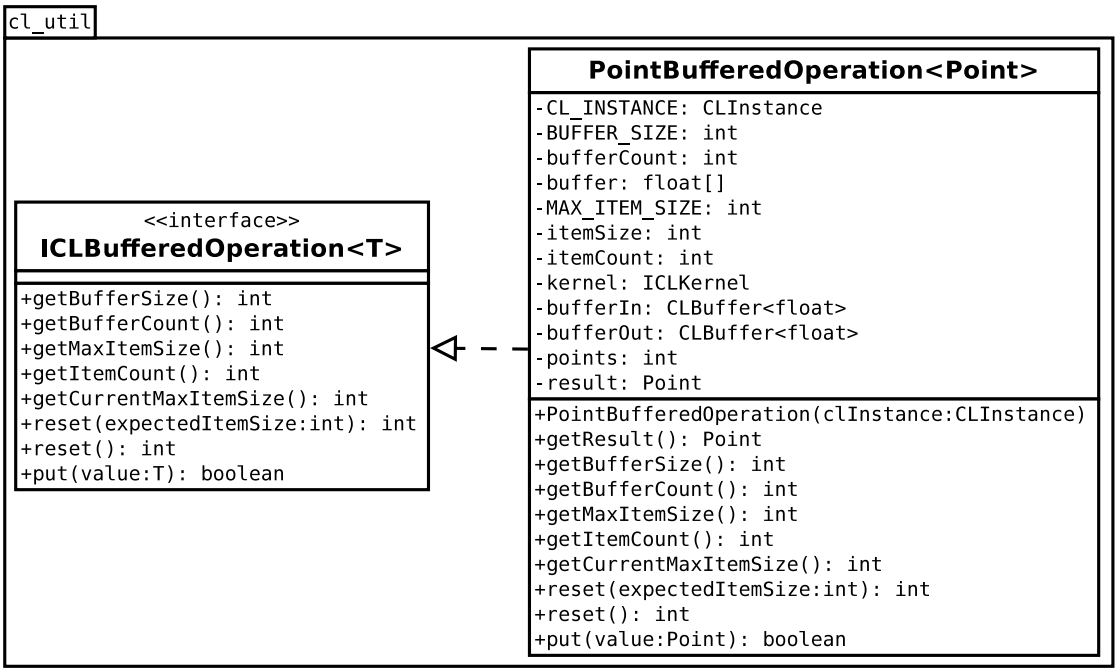


Abbildung 4.2.: Klassendiagramm für die Schnittstelle `ICLBufferedOperation` mit einer Beispielimplementierung

4.1.2. Ebene der OpenCL-Operationen

Die Ebene der OpenCL-Operationen reduziert die Komplexität einer Operation unter Nutzung der GPU. Diese Ebene versteckt die Arbeit mit der Grafikkarte, zu der Datenübertragungen und Datenkonvertierungen gehören, und die in Abschnitt 3.4 vorgestellten Probleme mit der Datenorganisation. Eine konkrete OpenCL-Operation muss dafür die Schnittstelle `ICLBufferedOperation` implementieren. Hierbei handelt es sich um einen Algorithmus, der mit einen oder mehreren Kernel eine komplexe Operation ausführt. Wie das „Buffered" (dt. gepuffert) im Namen verdeutlichen soll, muss die in Abschnitt 3.4.3 vorgestellte Datenorganisation mit Hilfe eines Puffers implementiert werden. Die Nutzung einer solchen GPU-Operation mit Hilfe der Schnittstelle `ICLBufferedOperation` setzt in einem MapReduce-Job keine Kenntnisse von OpenCL voraus.

In Abbildung 4.2 ist die Schnittstelle mit ihren Methoden und einer Beispielimplementierung dargestellt. Der Anwender einer Implementierung nutzt diese Methoden:

- `put()` zum Hinzufügen der Daten in den Puffer,
- `reset()` um den Puffer und Zwischenergebnisse zurückzusetzen und
- `getResult()` um das Ergebnis zu berechnen.

Da jede Implementierung unterschiedliche Endergebnisse liefern kann und andere Argumente benötigt, ist `getResult()` nicht in der Schnittstelle definiert.

Alle anderen Methoden geben vordergründig einen Weg für den Entwickler einer OpenCL-Operation vor. Die Schnittstelle `ICLBufferedOperation` unterscheidet zwischen `bufferSize` und `itemSize`. Diese Unterscheidung ist für die Serialisierung der Objekte wichtig. Dabei werden die Objekte in skalare Datentypen zerlegt. Analog dazu werden bei einer Deserialisierung aus skalaren Datentypen Objekte erstellt. Dieses Vorgehen muss vor bzw. nach jeder GPU-Übertragung durchgeführt werden, wenn mit Objekten gearbeitet wird. Die `itemSize` ist die maximale Anzahl an Objekten, die der Puffer speichern kann. Die tatsächliche Puffergröße unterscheidet sich von der `itemSize`, da ein Objekt aus mehreren skalaren Datentypen bestehen kann, und wird mit der `bufferSize` angegeben. Die

Puffergröße ist vom physikalisch verfügbaren Grafikkartenspeicher abhängig und wird in der Regel im Konstruktor ermittelt. Die `itemSize` ist von der Puffergröße und der Serialisierung abhängig. Der initialisierte Puffer `buffer` wird nur neu alloziert, wenn mit der `reset`-Methode eine andere Puffergröße angegeben wird. Die Methode prüft außerdem, ob die gewünschte Größe auf der Grafikkarte alloziert werden kann und verkleinert diese wenn nötig.

Vor dem Hinzufügen prüft `put()` anhand von `itemCount` und `itemSize`, ob der Puffer noch nicht gefüllt ist. Falls dies zutrifft, wird das Objekt serialisiert im `buffer` gespeichert und anschließend die Zählvariablen, `itemCount` und `bufferCount`, inkrementiert. Bei einem vollen Puffer unterscheidet sich das Vorgehen von der konkreten Operation. Wenn in dieser mit Teilergebnissen gerechnet werden kann, wird der Puffer auf die Grafikkarte kopiert, das Teilergebnis berechnet, die Zählvariablen zurückgesetzt und anschließend das Objekt serialisiert im Puffer gespeichert, wobei die alten Werte überschrieben werden. Dadurch können Daten in unbestimmter Anzahl verarbeitet werden. Ein abschließendes `getResult()` berechnet aus dem Teilergebnis und den noch im Puffer befindlichen Daten das Endergebnis. Wenn die Operation keine Teilergebnisse nutzen kann, wird das Objekt nicht zum Puffer hinzugefügt und `put()` liefert den Rückgabewert `false`. Nun kann der Anwender entscheiden, ob das Ergebnis ohne die fehlenden Objekte mit `getResult()` berechnet werden soll oder der Puffer, wenn möglich, vergrößert werden muss.

Wenn der verfügbare Speicher der JVM kleiner als der der Grafikkarte ist und die konkrete Operation dies zulässt, kann eine intelligente Implementierung vor der Berechnung auf der GPU mehrere Pufferinhalte auf den Grafikkartenspeicher übertragen (siehe Abschnitt 3.4.3). Dadurch wird die Grafikkarte besser ausgelastet und die Anzahl der auszuführenden Kernel sinkt.

Anhand der Klasse `PointBufferedOperation` wird eine konkrete Implementierung erläutert. `PointBufferedOperation` berechnet den Mittelpunkt einer Menge von 3-dimensionalen Punkten in einem euklidischen Raum. Hierzu wird für jede Dimension die Summe aus allen Punkten berechnet und anschließend mit der Anzahl der Punkte dividiert:

$$p_M = \frac{1}{|P|}\sum_{i=1}^{|P|} p_i \in P,\ P \subseteq \mathbb{R}^3,\ P \mathrel{\widehat{=}} Punktmenge,\ p_M \mathrel{\widehat{=}} Mittelpunkt \tag{4.1}$$

Da eine fortlaufende Addition stattfindet, kann die Mittelpunktberechnung Teilergebnisse nutzen. Dazu wird die Variable `result` für die Aufsummierung und die Variable `points` für die Anzahl der Punkte genutzt. Die Klasse `Point` mit den Attributen `x`, `y` und `z` repräsentiert einen Punkt. Jeder Punkt wird serialisiert, indem die x-, y-, z-Koordinaten nacheinander in dem Feld `buffer` gespeichert werden. Der Konstruktor von `PointBufferedOperation` ermittelt anhand des physikalisch verfügbaren Grafikkartenspeichers und der Dimension die Puffergröße. Bei drei Dimensionen ist die `bufferSize` um den Faktor 3 größer als die `itemSize`. Nach der Initialisierung von `PointBufferedOperation` ist das Feld `buffer` alloziert, die Zählvariablen haben den Wert 0 und der Punkt `result` ist mit der Koordinate $(0, 0, 0)$ initialisiert. Die `reset`-Methode setzt den Zustand zurück und alloziert gegebenenfalls ein neues Feld. Falls der Puffer noch nicht gefüllt ist, fügt `put()` einen Punkt serialisiert hinzu und inkrementiert die Zählvariablen (`bufferCount`, `itemCount`, `points`). Wenn der Puffer voll ist, wird dieser auf die Grafikkarte übertragen und der OpenCL-Kernel summiert die Dimensionen aller Punkte auf. Dieses Teilergebnis wird wiederum im Host auf `result` addiert. Anschließend werden die Zählvariablen (`bufferCount`, `itemCount`) zurückgesetzt. Mit einem rekursiven Aufruf von `put()` und dem Punkt als Argument wird dieser in den nun zurückgesetzten Puffer hinzugefügt. Der Aufruf von `getResult()` berechnet mit den noch im Puffer befindlichen Punkten das neue Teilergebnis. Jede Dimension des Teilergebnisses wird nun durch die Anzahl der Punkte (`points`) dividiert. Das Endergebnis wird für die Rückgabe zwischengespeichert, um mit `reset()` alle Variablen zurücksetzen zu können. Abschließend liefert `getResult()` das zwi-

schengespeicherte Endergebnis als Rückgabewert. Nach jedem Aufruf von `getResult()` können direkt neue Punkte mit `put()` hinzugefügt werden, um deren Mittelpunkt zu berechnen.

4.2. Clusteranalyse mit k-Means

Clusteranalyse ist ein Verfahren zur Entdeckung von Gruppen (engl. Cluster) in Datenbeständen. Ziel einer Clusteranalyse ist das Bilden von Gruppen aus einer Menge von Objekten, die in einer definierten Eigenschaft möglichst ähnlich zueinander sind, sich jedoch von Objekten anderer Gruppen möglichst stark unterscheiden sollen. Für jede Eigenschaft muss eine Metrik angegeben werden, um die Ähnlichkeit zwischen Objekten im Bezug auf die auszuwertende Eigenschaft messen zu können. Anwendungsbereiche der Clusteranalyse sind zum Beispiel das automatisierte Klassifizieren von Dokumenten im Data-Mining oder die Mustererkennung in der Bildverarbeitung.

Ein leicht verständliches Verfahren zur Clusteranalyse ist k-Means. Der k-Means-Algorithmus ordnet Objekte einer vorher definierten Anzahl von k Gruppen zu. Das iterative Verfahren benötigt als Eingabe die Objektmenge, die Gruppen und die zu verwendende Metrik. Die Gruppen werden dabei als Zentren bzw. Schwerpunkte (engl. Centroid) angegeben. Als Ausgabe liefert k-Means die Zentren der Gruppen und die Zuordnung der Objekte zu diesen. Ein Objekt wird dem Zentrum zugewiesen, bei dem sich die Ergebnisse der Metrik am geringsten voneinander unterscheiden. Nach dieser Zuordnung wird in jeder Iteration für jede Gruppe das Mittel (engl. Mean) berechnet, welches das neue Zentrum bildet. Das Verfahren terminiert, wenn sich die Zentren nicht mehr verändern oder eine Grenze an Iterationen erreicht ist. Das Ergebnis von k-Means ist von den vorher definierten Zentren abhängig. Diese können entweder zufällig aus der Eingabemenge oder mit Hilfe einer Heuristik bestimmt werden.

Das k-Means-Verfahren wird für eine Menge von n-dimensionalen Punkten in einem euklidischen Raum implementiert. Als Metrik wird der euklidische Abstand zwischen einem Punkt und einem Zentrum verwendet. Der Algorithmus 4.1 auf der nächsten Seite stellt eine Umsetzung des k-Means-Verfahrens für das vorgestellte Problem dar. In den Zeilen 1 bis 4 werden die Punkte eingelesen und die Methode `generateCentroids()` liefert k zufällig ausgewählte Gruppenzentren aus der Punktmenge. Anschließend wird in den Zeilen 6 bis 15 für jeden Punkt das Zentrum mit dem geringsten Abstand gesucht. Die Berechnung des euklidischen Abstands zwischen zwei Punkten wird von der Methode `distance()` implementiert. Die Datenstruktur `groups` verwaltet die Zuweisung der Punkte zu einer Gruppe, die durch ein Centroid identifiziert wird. In den Zeilen 16 bis 18 wird für jede Gruppe das neue Zentrum ermittelt. Die `means`-Methode berechnet aus einer Menge von Punkten deren Mittelpunkt, welcher in `centroids` gespeichert wird. Abschließend prüft `checkFinish()`, ob der Algorithmus beendet werden soll. Das Ergebnis des Verfahrens ist in `centroids` und `groups` gespeichert. Der vorgestellte Algorithmus muss nun für die Anforderungen von MapReduce und GPGPU angepasst werden.

4.2.1. Analyse

Der Algorithmus 4.1 auf der nächsten Seite ist in zwei Phasen unterteilt:

- Phase 1 ermittelt für jeden Punkt das Zentrum mit dem geringsten Abstand und ordnet den Punkt der zugehörigen Gruppe zu.
- Phase 2 berechnet für jede Gruppe ein neues Zentrum.

Im Vergleich zur Initialisierung und der Abbruchbedingung haben diese Teile aufgrund der Iterationen den höchsten Rechenaufwand. Die asymptotischen Laufzeiten der beiden Phasen sind von den folgenden Variablen abhängig:

```
1  // Initialisierung
2  points[] = read(inputFile)
3  centroids[] = generateCentroids(points, k)
4  stop = false
5  while(not stop):
6    // Phase 1: Suche geringsten Abstand für Gruppenzuordnung
7    groups{} = Map()
8    foreach(p in points):
9      min = MAX
10     cc = NULL
11     foreach (c in centroids):
12       if(distance(c, p) < min)
13         min = distance(c, p)
14         cc = c
15     groups.put(cc, p)
16   // Phase 2: Berechne neue Zentren für jede Gruppe
17   foreach(c in centroids):
18     c = mean(groups.get(c))
19   // Iterationsabbruch?
20   stop = checkFinish()
```

Algorithmus 4.1: Pseudocode des k-Means-Algorithmus zur Analyse einer Punktmenge

- d Dimension des euklidischen Raumes
- n Anzahl der Punkte
- k Anzahl der Gruppen

In der ersten Phase müssen die Abstände zwischen allen Punkten und allen Zentren berechnet werden. Der euklidische Abstand zwischen den d-dimensionalen Punkten x und y ist durch diese Formel definiert:

$$dist(x, y) = \sqrt{\sum_{i=1}^{d}(x_i - y_i)^2}$$

Die Laufzeit der Funktion $dist(x, y)$ ist von der Dimension abhängig. Somit ergibt sich für die erste Phase die asymptotische Laufzeit von:

$$\Theta(d, n, k) = n \cdot k \cdot d$$

Die Laufzeit der zweiten Phase ist von der Berechnung des neuen Zentrums abhängig. Das Zentrum für m Punkte aus einer Punktmenge P wird mittels folgender Formel berechnet:

$$mean(P) = \frac{1}{m}\sum_{i=1}^{m} p_i \in P \tag{4.2}$$

Die Funktion $mean(P)$ muss für jede Gruppe, und somit k-mal, ausgeführt werden. Da jeder Punkt genau einer Gruppe zugeordnet ist, finden auch genau n Additionen statt. Zusätzlich wird für jede Gruppe eine Division ausgeführt. Beide Operationen müssen für jede Dimension berechnet werden. Daraus resultiert für die Phase 2 die folgende asymptotische Laufzeit:

$$\Theta(d, n, k) = d \cdot n + d \cdot k = d \cdot (n + k)$$

Unter den Annahmen $d \ll n$, $k \ll n$, $d \geq 1$ und $k \geq 2$ gilt diese Ungleichung:

$$\begin{aligned} d \cdot (n + k) &< n \cdot k \cdot d \\ n + k &< n \cdot k \\ Laufzeit_{Phase\,2} &< Laufzeit_{Phase\,1} \end{aligned}$$

Die meisten Berechnungen beim k-Means-Verfahren finden somit in der ersten Phase statt. Mit diesen Kenntnissen kann der Algorithmus nun auf die zwei verfügbaren Ebenen der Parallelität und deren Programmiermodelle abgebildet werden. In der ersten Phase kann für jeden Punkt unabhängig von den anderen Punkten die minimale Distanz zu den Gruppenzentren ermittelt werden. Die Eingabemenge kann somit beliebig unterteilt und den einzelnen Verarbeitungseinheiten zugewiesen werden. In einem Hadoop-Cluster ist die Eingabemenge (die Punkte) im HDFS abgelegt. Die Map-Phase liest diese ein und verteilt die Eingabemenge anhand der Blöcke im verteilten Dateisystem implizit an alle beteiligten Nodes. Für die Phase 1 müssen alle Gruppenzentren zentral in einem gemeinsamen Speicher für alle Nodes verfügbar sein. Die `map`-Methode erhält als Value einen Punkt und ermittelt das am geringsten entfernte Gruppenzentrum mit Hilfe der Grafikkarte. Der Punkt (als Value) wird zusammen mit dem Zentrum (als Key) der Reduce-Phase übergeben. Das MapReduce-Framework fügt nun jeden Punkt anhand des Keys einer Gruppe zu. Dadurch erhält die `reduce`-Methode alle Punkte einer Gruppe und kann für diese das neue Zentrum berechnen. Ob diese Berechnung auf der GPU oder CPU schneller ist, muss praktisch ermittelt werden. Um das Ergebnis im HDFS zu speichern, übermittelt die `reduce`-Methode dem Framework abschließend jeden Punkt der Gruppe als Value zusammen mit dem neuen Zentrum als Key. Ein Job, folglich das Durchlaufen einer Map- und einer Reduce-Phase, entspricht einer Iteration des k-Means-Verfahrens. Da in der Regel eine Iteration kein ausreichend genaues Ergebnis liefert, müssen mehrere Jobs verkettet werden.

4.2.2. Umsetzung

Ein Punkt als zentraler Datentyp wird durch die Schnittstellen `IPoint` und `ICPoint` repräsentiert. `IPoint` stellt als Basistyp einen n-dimensionalen Punkt dar. `ICPoint` erweitert diesen Basistyp, um diesen Punkt einem Gruppenzentrum zuweisen zu können. Zusätzlich muss noch die Schnittstelle `WritableComparable` zum Lesen, Schreiben und Vergleichen implementiert werden, da das MapReduce-Framework die Punkte als Keys und Values verwendet. Im Algorithmus kann die Klasse `PointWritable` verwendet werden, da sie die Schnittstellen `ICPoint` und `WritableComparable` implementiert. Das Klassendiagramm in Abbildung 4.3 auf der nächsten Seite stellt den Aufbau der Datentypen grafisch dar. Die Trennung und Vererbung der Schnittstellen ermöglicht die Wiederverwendung einzelner Komponenten des Algorithmus ohne Bezug auf das MapReduce-Framework.

Zur Eingabe und Ausgabe dient eine formatierte Textdatei, in der pro Zeile entweder nur ein Punkt oder ein Punkt mit dem zugewiesenen Gruppenzentrum gespeichert ist. Das Gruppenzentrum steht immer am Zeilenanfang und ist durch einen Tabulator vom darauf folgenden Punkt getrennt. Zur Serialisierung und Deserialisierung eines Punktes wird die Hilfsklasse `Points` mit den folgenden Methoden genutzt:

- `String createString(IPoint<Float>)` serialisiert einen Punkt zu einem String und
- `IPoint<Float> createPoint(String)` erstellt aus einem String einen Punkt.

Die String-Repräsentation eines Punktes hat die Form:

```
<Dimension 1>;<Dimension 2>;...;<Dimension n>
```

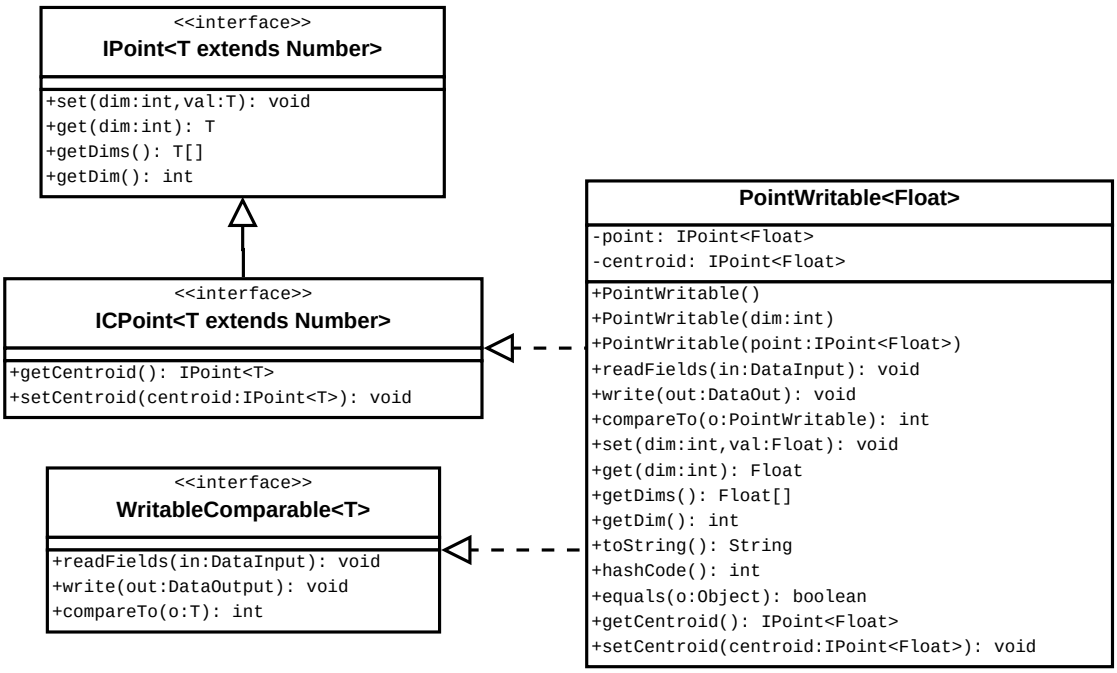


Abbildung 4.3.: Aufbau des Datentyps für einen Punkt

Damit die `map`-Methode direkt den Datentyp `PointWritable` als Argument erhält, benötigt das Framework zum Einlesen der Eingabedatei noch ein `InputFormat` mit zugehörigem `RecordReader`. Dieses wird durch die Klasse `PointInputFormat` implementiert und enthält die zwei wichtigen Methoden zur Deserialisierung:

```
public boolean nextKeyValue() throws IOException {
  // ...; this.line = in.readLine(...); ...
}

public NullWritable getCurrentKey() {
  return this.key;
}

public PointWritable getCurrentValue() {
  this.value = new ⤸
     PointWritable(Points.createPoint(this.line.toString()));
  return this.value;
}
```

Da beim Start von k-Means den Punkten noch keine Zentren zugewiesen sind, liefert `getCurrentKey()` nur einen Nullwert. Analog zur Eingabedatei benötigt die Ausgabedatei ein `OutputFormat` mit zugehörigem `RecordWriter`. Anders als im `InputFormat` werden neben den Punkten auch die Gruppenzentren berücksichtigt. Die Klasse `PointOutput` setzt die dafür benötigten Methoden um. Die `write`-Methode serialisiert die Key-Value-Paare:

```
public synchronized void write(PointWritable key, PointWritable ⤸
    value) throws IOException {
  String s;
  if (key != null) {
    s = Points.createString(key);
    out.write(s.getBytes(utf8));
  }
  if (key != null && value != null)
    out.write(keyValueSeparator);
  if (value != null) {
```

```
    s = Points.createString(value);
    out.write(s.getBytes(utf8));
  }
  out.write(newline);
}
```

Die Klasse `KMeansHadoop` wird durch Hadoop geladen und startet anschließend den Job. Ein k-Means-Job benötigt folgende Parameter, die vor dem Start für den Hadoop-Cluster und den Algorithmus konfiguriert werden müssen:

- Job-Bezeichnung:

```
job.setJobName(<job name>);
```

- Eingabedatei:

```
PointInputFormat.setInputPaths(job, <input path>);
```

- Eingabedatei für Gruppenzentren:

```
DistributedCache.addCacheFile(<centroids path>, ↩
   job.getConfiguration());
```

- Ausgabedatei:

```
PointOutputFormat.setOutputPath(job, <output path>);
```

- Auswahl der CPU- oder GPU-Implementierung:

```
job.setMapperClass(<map implementation for CPU or GPU>);
job.setReducerClass(<reduce implementation for CPU or GPU>);
```

- Anzahl der Iterationen
- Bestimmen der Eingabe- und Ausgabeformate:

```
job.setInputFormatClass(PointInputFormat.class);
job.setOutputFormatClass(PointOutputFormat.class);
```

- Bestimmen der Datentypen für Keys und Values:

```
job.setMapOutputKeyClass(PointWritable.class);
job.setMapOutputValueClass(PointWritable.class);
job.setOutputKeyClass(PointWritable.class);
job.setOutputValueClass(PointWritable.class);
```

Damit alle Nodes lesenden Zugriff auf die Gruppenzentren haben, muss der Cluster einen gemeinsamen Speicher bereitstellen. Hierzu sieht das Framework den „Distributed Cache" vor und kopiert alle benötigten Daten aus dem HDFS auf die Nodes, bevor ein Job gestartet wird.[1] Die notwendige Verkettung der Jobs ist durch eine Schleife realisiert, die die definierte Anzahl von Iterationen durchläuft. Damit der nächste Job Zugriff auf die neuen Gruppenzentren hat, wird als Eingabedatei die Ausgabedatei des vorherigen Jobs verwendet. Ein neuer Job wird erst gestartet, wenn alle Tasks beendet und alle Daten auf dem HDFS gespeichert wurden. Da nur die Map-Phase die Punkte mit zugewiesenen Gruppenzentrum als Ausgabe liefert (siehe Tabelle 4.1), wird abschließend ein Job ohne Reduce-Phase gestartet, um das Ergebnis im HDFS zu speichern.

[1] `http://hadoop.apache.org/common/docs/current/mapred_tutorial.html#DistributedCache`

Phase	Eingabe	Ausgabe
Map	$< Null, Point_1 >, List(Centroids)$	$< Centroid_1, Point_1 >$
Reduce	$< Centroid_1, List(Points) >$	$< Centroid_2, Null >$

Tabelle 4.1.: Ein- und Ausgabedaten der Map- und Reduce-Phase von k-Means

Der Datenfluss für die MapReduce-Implementierung von k-Means ist in Tabelle 4.1 verdeutlicht. Die `map`-Methode erhält als Value einen Punkt aus den Eingabedaten und hat über den `DistributedCache` Zugriff auf die Gruppenzentren. Für diesen Punkt sucht `map()` nun das zugehörige Gruppenzentrum. Anschließend werden das Zentrum als Key und der Punkt als Value dem Framework als Ausgabe übergeben. Die `reduce`-Methode erhält als Key ein Gruppenzentrum und als Value alle Punkte, die diesem Zentrum zugewiesen wurden. Für diese Punkte berechnet `reduce()` das neue Zentrum und übergibt nur dies dem Framework, um den Kommunikationsaufwand zu reduzieren. Nach der Reduce-Phase sind nur die neuen Gruppenzentren im HDFS gespeichert.

Die Implementierungen der Map- und Reduce-Phase sind zusammen als statisch innere Klassen in `KMMapperReducer` für die CPU und in `KMMapperReducerCL` für die GPU zu finden. Die `setup`-Methode liest zu Beginn eines Tasks die Gruppenzentren in den Hauptspeicher ein. Die GPU-Implementierung ermittelt ein verfügbares OpenCL-Gerät und initialisiert `CLPointFloat`, eine Implementierung der in Abschnitt 4.1.2 vorgestellten Schnittstelle `ICLBufferedOperation<ICPoint<Float>>`. Anschließend werden die Gruppenzentren mit der Methode `CLPointFloat.prepareNearestPoints()` auf den Grafikkartenspeicher kopiert. Die Operation `CLPointFloat` verwendet den Java-Kernel `PointFloatNearestIndex`. Der Java-Kernel benötigt die Punkte und die Gruppenzentren als Eingabe. Als Ausgabe liefert er für jeden Punkt den Index des zugewiesenen Gruppenzentrums. Die Verwendung von Indizes, anstatt der Punkte mit den Zentren, verringert die Menge der zu übertragenden Daten, weil diese nur noch von der Anzahl der Punkte und nicht zusätzlich noch von der Dimension abhängig ist. Um eine korrekte Zuordnung über die Indizes gewährleisten zu können, muss sichergestellt werden, dass sich die Indizes der Punkte und Zentren in den Datenstrukturen im Haupt- und Grafikkartenspeicher nicht ändern. Für jeden Punkt im Grafikkartenspeicher wird ein Work-Item gestartet, das den minimalen Abstand zwischen dem Punkt und allen Zentren sucht. Um die GPU optimal auslasten zu können, müssen mindestens so viele Punkte verarbeitet werden, wie Ausführungseinheiten auf der Grafikkarte vorhanden sind. Die `map`-Methode fügt die Punkte über `CLPointFloat.put()` dem Puffer hinzu. Wenn dieser voll ist, werden die Punkte auf der GPU verarbeitet und zusammen mit den errechneten Zentren anschließend dem MapReduce-Framework übergeben. Noch nicht verarbeitete Punkte werden abschließend in der `cleanup`-Methode verarbeitet.

Die Datenorganisation für die Verarbeitung auf der GPU ist für eine `reduce`-Methode unkomplizierter, da alle Daten für eine Gruppe vorhanden sind und nicht durch mehrere Methodenaufrufe gesammelt werden müssen. Anhand der Formel 4.1 auf Seite 52 wird das neue Zentrum aus den Punkten einer Gruppe berechnet. Für die GPU bieten sich hierbei vor allem zwei Möglichkeiten zur Parallelisierung an:

1. Für jede Dimension wird ein Work-Item gestartet, welches die Werte dieser Dimension für alle Punkte addiert.

2. Jede Dimension wird separat mit Hilfe einer parallelen Reduktion bzw. einer „Fan-In-Summation" berechnet [BSP10][GO96, S. 87].

Beide Möglichkeiten haben Vor- und Nachteile. Die erste Variante ist sehr leicht zu implementieren, jedoch wird die Parallelität der Grafikkarte nur bei großen Dimensionen gut ausgenutzt. Die zweite Variante ist nur von der Anzahl der zu addierenden Punkte

abhängig, welche in der Regel sehr groß ist.[2] Jedoch ist die Implementierung des Kernels schwieriger. Wenn der Kernel so programmiert ist, dass er nur ein Feld bzw. eine Dimension berechnet, kann er für andere Operationen wiederverwendet werden. Allerdings muss dann für jede Dimension ein separater Kernel gestartet werden. Die zweite Möglichkeit wurde praktisch umgesetzt und genauer untersucht.

Die Berechnung auf der GPU dauert zusammen mit der Datenübertragung jedoch länger als eine Implementierung für die CPU. Verantwortlich dafür ist die lineare Laufzeit für die Datenübertragung bei vergleichsweise wenig Rechenaufwand, die bei der CPU-Implementierung entfällt. Folglich nutzt die Reduce-Phase keine Grafikkarte.

4.2.3. Probleme und Einschränkungen

Die vorhandene Implementierung von k-Means für MapReduce mit GPU-Unterstützung enthält nur wenige Einschränkungen. In der Map-Phase müssen alle Gruppenzentren und mindestens ein Punkt in den Haupt- und Grafikkartenspeicher geladen werden können. Eine optimale Auslastung der Grafikkarte beginnt jedoch erst, wenn mindestens so viele Punkte verarbeitet werden, wie physische Ausführungseinheiten vorhanden sind.

Die Serialisierung und Deserialisierung verursacht weitere Probleme und Einschränkungen, die in einer Testimplementierung ohne MapReduce beobachtet wurden. Ohne MapReduce müssen die Punkte einer Gruppe für die zweite Phase zusammengestellt werden, da die erste Phase jedem Punkt ein Zentrum zuweist und keine Datenstrukturen für die Gruppen erstellt. Die Testimplementierung verwendet dafür in der zweiten Phase eine `HashMap` mit dem Zentrum als Key und einer Liste von Punkten als Value. Die Liste eines Zentrums speichert die einer Gruppe zugeordneten Punkte:

```
HashMap<IPoint<Float>, List<ICPoint<Float>>> groups = new ↩
    HashMap<IPoint<Float>, List<ICPoint<Float>>>();
for(ICPoint<Float> p : points) {
  if(!groups.containsKey(p.getCentroid()))
    groups.put(p.getCentroid(), new LinkedList<ICPoint<Float>>());
  groups.get(p.getCentroid()).add(p);
}
```

Ursprünglich lieferte der Kernel `PointFloatNearestIndex` die Zentren als Punkte, also die Werte aller Dimensionen, und nicht als Index. Dadurch muss nun für jeden Punkt beim Setzen des Zentrums mit der Methode `ICPoint.setCentroid()` ein neues Objekt erstellt werden. Somit existieren nach Abschluss der ersten Phase genau n Objekte, die jedoch nur k Gruppenzentren repräsentieren ($k \ll n$). Beim Zusammenstellen der Gruppen nimmt der Methodenaufruf `groups.get()` dadurch jedoch mehr Zeit in Anspruch, da sich die Objektreferenzen inhaltlich gleicher Zentren unterscheiden. Eine `HashMap` überprüft die Keys in der Regel nur anhand der `hashCode`-Methode eines Objektes. Enthält die `HashMap` bereits ein Objekt mit demselben Hashcode des übergebenen Keys, liefert diese den dazu gehörigen Value. Wenn sich jedoch die Objektreferenz des in der `HashMap` enthaltenen Keys von dem übergebenen unterscheidet, werden diese zwei Objekte zusätzlich durch die `equals`-Methode eines Objektes verglichen.[3] Dadurch finden zusätzlich $n - k$ `equals`-Aufrufe statt, da sich die Objektreferenzen der Zentren aller Punkte unterscheiden. In Abhängigkeit der Komplexität von `equals()` kann dieser Mehraufwand deutlich messbar sein. Dieser Umstand wurde in der Testimplementierung beobachtet, als die Phasen 1 und 2 jeweils für sich betrachtet mit der GPU performanter waren als die CPU-Implementierung, jedoch die Gesamtlaufzeit deutlich langsamer war. Da sich die Implementierung für die CPU und GPU in nur einer Methode unterscheidet,

[2] Es werden mehr Punkte verarbeitet, als physische Recheneinheiten der GPU vorhanden sind.

[3] Der Quelltext der Java-Bibliotheken ist im Java Development Kit enthalten oder kann unter `http://download.java.net/jdk6/source/` heruntergeladen werden.

konnte die Ursache nur mit einem Profiler gefunden werden. Die Änderung der Referenz bei einer Serialisierung und Deserialisierung muss beim GPGPU besonders berücksichtigt werden, wenn die Objekte der Ausgabe weiter verwendet werden. Eine Alternative stellt die Verwendung von Indizes dar. Dies hat den Vorteil, dass einerseits die zu übertragene Datenmenge sinkt und andererseits bestehende Querverweise nicht aktualisiert werden müssen. Zusätzlich können bei der Serialisierung nicht benötigte Objektattribute ausgelassen werden. Aufgrund dieser Beobachtung liefert der überarbeitete Kernel `PointFloatNearestIndex` die Indizes der als Eingabe übergebenen Gruppenzentren. Da die Objekte der Gruppenzentren noch im Hauptspeicher verfügbar sind, können diese anhand der Indizes ausgewählt und den Punkten zugeordnet werden. Somit befinden sich auch nach der Berechnung auf der GPU nur noch k Objekte der Gruppenzentren mit den ursprünglichen Referenzen im Hauptspeicher, wodurch der Mehraufwand entfällt.

Aufgrund der Verwendung von Indizes müssen alle Objekte, die über diese identifiziert werden sollen, im Hauptspeicher verbleiben. Dies hat zur Folge, dass nur der kleinere der beiden Speicher, Haupt- oder Grafikkartenspeicher, ausgelastet werden kann. Im besten Fall sollte immer der Grafikkartenspeicher komplett gefüllt werden, um der GPU genügend Daten zum Rechnen bereitzustellen. Eine unkonfigurierte JVM stellt einem Java-Programm 64 MB Hauptspeicher zur Verfügung. Die Testsysteme verfügen jedoch über mindestens 1 GB Grafikkartenspeicher. Somit können entweder nur 64 MB verwendet werden oder die JVM muss mit mehr Speicher gestartet werden. Beide Alternativen führen nur bedingt zum Ziel, da sich neben den Eingabedaten, in Form von Objekten, noch der Puffer sowie weitere Daten der Anwendung im Hauptspeicher befinden. Dies führte bei Tests zu einer `OutOfMemException`, da sich der tatsächlich verfügbare Speicher nur schwer abschätzen lässt und die GC recht schnell nicht mehr genügend Speicher bereitstellen kann. Der Aufwand zur Umsetzung einer komplexen Speicherverwaltung ist sehr groß, da die Anzahl der Dimensionen, der Gruppenzentren, der verfügbare Speicher der JVM und der Grafikkarte sowie sonstige Daten des Algorithmus berücksichtigt werden müssen. Deswegen ist in der vorliegenden Implementierung die Anzahl der Punkte, die von einem Java-Kernel berechnet werden, auf $65536/Dimension$ beschränkt.

4.3. Numerische Integration

Unter dem Begriff der numerischen Integration werden Verfahren zur näherungsweisen Berechnung von bestimmten Integralen zusammengefasst. Eine Funktion $F(x)$ wird als unbestimmtes Integral der Funktion $f(x)$ bezeichnet, wenn die Eigenschaft $F'(x) = f(x)$ erfüllt ist. Außerdem heißt die Funktion $F(x)$ im Intervall $[a, b]$ Stammfunktion von $f(x)$, wenn für alle $x \in (a, b) \subseteq \mathbb{R}$ die Beziehung $F'(x) = f(x)$ gilt. Bestimmte Integrale werden zur Berechnung des Flächeninhalts verwendet, der zwischen dem Graphen von $f(x)$ und der x-Achse liegt. Diese Fläche kann mit Hilfe des Hauptsatzes der Differenzial- und Integralrechnung ermittelt werden [DE03]:

$$\int_a^b f(x)dx = [F(x)]_a^b = F(b) - F(a)$$

In der Praxis ist es jedoch oft nicht möglich, eine Stammfunktion zu einer Funktion $f(x)$ zu bilden. Häufig ist sogar die Funktion selbst unbekannt, wenn zum Beispiel nur eine Wertetabelle der Form $(x_i, f(x_i))$ vorliegt. Das Integral kann in diesen Fällen nur näherungsweise durch eine numerische Integration berechnet werden. Eine allgemeine Form der Näherung besteht darin, die Funktion $f(x)$ durch eine andere Funktion n-ten Grades zu ersetzen, deren Integral berechnet werden kann:

$$\int_a^b f(x)dx \doteq \sum_{i=0}^{n} \alpha_i f(x_i),\ x_i \in (a, b) \subseteq \mathbb{R}$$

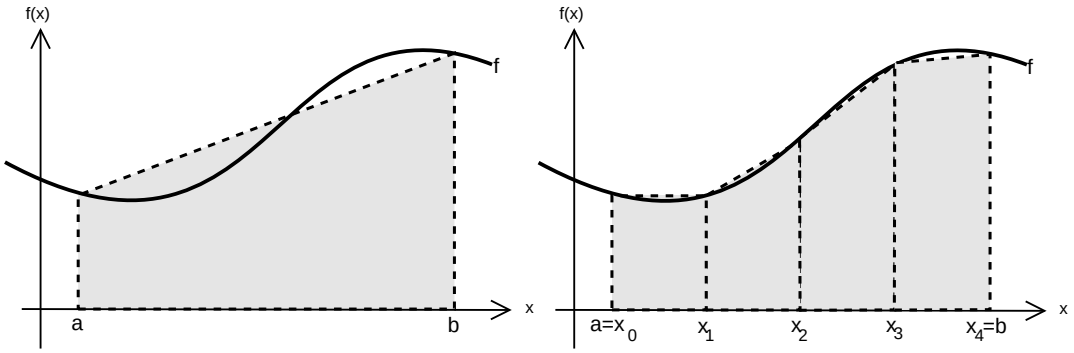


Abbildung 4.4.: Trapezregel für das Intervall $[a, b]$ (links) und Trapezregel mit vier Teilintervallen (rechts)

Eine Möglichkeit besteht darin, die Funktion $f(x)$ durch eine lineare Funktion mit dem Startpunkt a und dem Endpunkt b zu ersetzen. Dadurch entsteht ein Trapez, dessen Fläche der Näherung des Integrals von $f(x)$ entspricht. Diese, als Trapezregel bekannte, Approximation ist auf der linken Seite der Abbildung 4.4 veranschaulicht. Das Integral berechnet sich aus der Formel:

$$\int_a^b f(x)dx \doteq \frac{(b-a)}{2}\left[f(a)+f(b)\right] \tag{4.3}$$

Auf der linken Abbildung von 4.4 ist zu erkennen, dass die Trapezregel sehr ungenau sein kann. Um diese Ungenauigkeit einzuschränken, gibt es zwei Ansätze:

1. Verwendung von Polynomen höheren Grades oder
2. Zerlegung des Intervalls $[a, b]$ in n Teilintervalle.

In der Praxis ist die Zerlegung in Teilintervalle meist zufriedenstellender, da n fast beliebig erhöht werden kann und somit die Genauigkeit steigt. Das Integral für das Intervall $[a, b]$ berechnet sich aus der Summe der Integrale der Teilintervalle. Auf der rechten Seite der Abbildung 4.4 ist dieses Vorgehen veranschaulicht. Das Intervall $[a, b]$ wird in vier Teilintervalle $[x_{i-1}, x_i]$ mit $i = 1, ..., 4$ zerlegt, wobei $x_0 = a$ und $x_4 = b$ ist. Werden die Teilintervalle auf die Formel 4.3 angewendet, errechnet sich die Approximation aus:

$$\begin{aligned}\int_a^b f(x)dx &\doteq \sum_{i=1}^{n}\frac{h}{2}\left[f(x_{i-1})+f(x_i)\right]\\ \int_a^b f(x)dx &\doteq h(\frac{f(a)}{2}+f(a+h)+...+f(a+(n-1)h)+\frac{f(a+nh)}{2})\end{aligned} \tag{4.4}$$

Die Länge eines Teilintervalls ist durch $h = \frac{b-a}{n}$ gegeben. [GO96, S. 172]

4.3.1. Analyse

Eine numerische Integration ist sehr effizient und leicht parallelisierbar. Die n Teilintervalle werden auf p Nodes verteilt, sodass jeder Node $\frac{n}{p}$ Teilintervalle berechnet. Aus den Teilintervallen bildet jeder Node eine Summe, die er einem Master Node übermittelt. Die Summe der empfangenen Teilsummen ergibt das approximierte Integral. Die zu übertragene Datenmenge ist aufgrund der wenigen Übergabeparameter für einen Node sehr gering:

- die Funktion f,

- Start des Teilintervalls für den Node x_p,
- Länge eines Teilintervalls h und
- Anzahl der zu berechnenden Teilintervall n_p.

Bis auf die Funktion f sind alle Parameter skalare Datentypen. Angenommen x_p und h werden als `double` und n_p als `int` deklariert, so müssen an jeden Node nur 20 Byte, zuzüglich der Daten für die Funktion, übertragen werden.

Grundsätzlich kann dieses Vorgehen gut auf das MapReduce-Programmiermodell abgebildet werden. Die Map-Phase berechnet für jedes der n Teilintervalle das Integral und übergibt diese dem Framework. Mit Hilfe einer Combiner-Klasse kann vor der Reduce-Phase die Teilsumme der Teilintervalle eines Nodes gebildet werden. Die Teilsummen aller Nodes werden anschließend in der Reduce-Phase von einem Node addiert. Als Ergebnis liefert dieser das approximierte Integral.

Problematisch bei der MapReduce-Implementierung von Hadoop ist jedoch, dass diese für die Arbeit mit großen Datenmengen optimiert ist und eine numerische Integration nur sehr wenig und kleine Eingabedaten benötigt. So ist vorgesehen, dass die Key-Value-Paare für die Map-Phase eigentlich anhand der Datenblöcke im HDFS gebildet werden. Prinzipiell benötigt eine numerische Integration keine Daten aus dem Dateisystem, da alle Parameter beim Start des Algorithmus angegeben werden. Mit Hilfe einer speziellen Implementierung der Schnittstelle `InputFormat` könnten Daten anhand der Startparameter generiert werden, anstatt diese aus dem HDFS zu lesen. Da alle Nodes jedoch unabhängig voneinander diese Implementierung ausführen, wäre zur korrekten Aufteilung eine Kommunikation zwischen den Nodes nötig. Andererseits besteht die Möglichkeit, dass ein Job vor der Unterteilung in Tasks eine Datei mit den Startparametern für jeden Node im HDFS ablegt. Die Implementierung von `InputFormat` muss die Daten der Datei dann anhand der Zeilen und nicht anhand der Blöcke im HDFS auf die Nodes verteilen.

Darüber hinaus könnte der Anwender Teilintervalle angeben, die aus dem HDFS gelesen werden. Befinden sich in einem Teilintervall viele Extremwerte, steigt der Fehler, der durch das Abschneiden der Flächen entsteht. Dieser Fehler kann durch eine weitere Unterteilung des betreffenden Teilintervalls minimiert werden, falls diese Bereiche des Funktionsgraphs bekannt sind und festgelegt werden.

Der bisher diskutierte Ansatz berechnet die Teilintervalle in der Map-Phase und somit findet auch die GPU-Berechnung auf dieser statt. Für die weitere Untersuchung ist aber ein Anwendungsfall interessant, in dem eine GPU-Verarbeitung in der Reduce-Phase stattfindet. Da dann die Reduce-Phase die Integrale berechnet, kann die Map-Phase zum Erstellen und Verteilen der Intervalle genutzt werden. Um die Berechnung auf mehrere Nodes zu verteilen, müssen die Intervalle anhand von Keys in Gruppen eingeteilt werden. Als Ausgabe liefert der Job dann jedoch für jede Gruppe ein Ergebnis. Deswegen wird dem Anwender die Möglichkeit gegeben, mehrere Intervalle durch einen Key zu kennzeichnen. Alle Intervalle mit dem gleichen Key werden derselben Gruppe zugeordnet. Dadurch ist es möglich, Integrale verschiedener Intervalle für eine Funktion zu berechnen. Um Abschneidefehler zu minimieren, kann ein Intervall in mehrere Teilintervalle mit gleichem Key unterteilt werden.

4.3.2. Umsetzung

Die Klasse `NumericalIntegrationNamed` konfiguriert den Job, bevor die Tasks gestartet werden:

- Job-Bezeichnung:

```
job.setJobName(<job name>);
```

- Eingabedatei:

```
TextInputFormat.setInputPaths(job, <intput path>);
```

- Ausgabedatei:

```
TextOutputFormat.setOutputPath(job, <output path>);
```

- Funktion $f(x)$:

```
job.getConfiguration().set("FUNCTION", <function>);
```

- Exponent (optional):

```
job.getConfiguration().set("EXPONENT", <function>);
```

- Anzahl der Teilintervalle n:

```
job.getConfiguration().set("RESOLUTION", <n>);
```

- Auswahl der CPU- oder GPU-Implementierung:

```
job.setMapperClass(<map implementation for CPU or GPU>);
job.setReducerClass(<reduce implementation for CPU or GPU>);
```

- Bestimmen der Eingabe- und Ausgabeformate:

```
job.setInputFormatClass(TextInputFormat.class);
job.setOutputFormatClass(TextOutputFormat.class);
```

- Bestimmen der Datentypen für Keys und Values:

```
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(FloatIntervalWritable.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(FloatIntervalWritable.class);
```

Als Funktionen können nur Implementierungen der Schnittstelle `IMathFunction` verwendet werden. Diese Schnittstelle definiert zwei Methoden:

- `Float getValue(Float)` liefert den Funktionswert an der Stelle x und
- `String getOpenCLFunction()` liefert die in OpenCL-C implementierte `getValue`-Methode.

Der Rückgabewert von `getOpenCLFunction()` ist für die GPU-Unterstützung nötig. Der `String` wird beim Kompilieren des OpenCL-Kernels als Define übergeben und muss dadurch nicht für jede neue Funktion angepasst werden.

Das Eingabeformat der Implementierung ist `TextInputFormat`. Damit können Textdateien aus dem HDFS gelesen werden. Jede Zeile der Eingabe enthält ein Intervall mit dem Bezeichner:

```
<Bezeichner>\t[<a>;<b>]
```

Die Map-Phase konvertiert die im Value enthaltene Zeile in die gewünschten Datentypen `Text` und `FloatIntervalWritable`:

```
protected void map(LongWritable key, Text value, ⤸
    NIMapper.Context context) throws IOException, ⤸
    InterruptedException {
  IIntervalNamed<String, Float> interval = ⤸
      Intervals.createFloatIntervalNamed(value.toString());
  context.write(new Text(interval.getIdentifier()), new ⤸
      FloatIntervalWritable(interval));
}
```

Die Hilfsklasse `Intervals` stellt Methoden bereit, um aus einem `String` ein Objekt vom Typ `IIntervalNamed` zu erstellen und umgekehrt. Ein Intervall mit einem Startpunkt a, einem Endpunkt b und dem Bezeichner wird durch die Schnittstellen `IInterval` und `IIntervalNamed` repräsentiert. Die Klasse `FloatIntervalWritable` implementiert die Schnittstellen `IInterval<Float>` und zusätzlich `Writable`, um als Value für das Framework verwendet werden zu können.

Nach der Map-Phase sortiert und gruppiert das Framework die Intervalle anhand der Bezeichner. Die `reduce`-Methode erhält nur Intervalle mit dem gleichen Bezeichner. Die Schnittstelle `INumericalIntegration` definiert zwei Methoden, um das approximierte Integral einer Funktion zu berechnen:

- Setzen der zu berechnenden Funktion:

  ```
  void setFunction(IMathFunction<Float>);
  ```

- Näherungsweise Berechnung des Integrals im vorgegebenen Intervall mit n Teilintervallen:

  ```
  Float getIntegral(IInterval<Float> interval, int n);
  ```

Die Schnittstelle `INumericalIntegrationMulti` erweitert `INumericalIntegration`, damit auf der GPU mehrere Intervalle parallel berechnet werden können. Der Puffer der GPU-Implementierung `TrapeziumIntegrationMultiCL` wird in der Reduce-Phase mit Intervallen einer Gruppe gefüllt. Anschließend wird die Verarbeitung auf der Grafikkarte mit n Work-Items gestartet, von denen jedes genau ein Teilintegral von jedem Intervall berechnet. Die Teilintegrale eines Intervalles werden abschließend durch eine Fan-In-Summation auf der Grafikkarte addiert. Falls nicht alle Intervalle der Gruppe auf der Grafikkarte berechnet werden können, startet die `reduce`-Methode mehrere GPU-Durchläufe und summiert die Zwischenergebnisse dieser Berechnungen.

4.3.3. Probleme und Einschränkungen

Die umgesetzte numerische Integration kann nur Funktionen verwenden, die die Schnittstelle `IMathFunction` implementieren. Die Eingabe benutzerspezifischer Funktionen kann mit Hilfe der Klasse `ClassLoader` ermöglicht werden. Damit kann zur Laufzeit eine Klasse dynamisch instanziiert werden, die zum Beispiel im Distributed Cache bereitgestellt wird. Als zusätzliche Hürde muss der Entwickler alle Funktionen in Java und in OpenCL-C implementieren, damit diese dynamisch mit dem OpenCL-Kernel kompiliert werden können. Dieser Umstand erhöht das potenzielle Risiko von Laufzeitfehlern, da der Java-Compiler die OpenCL-Funktion nicht syntaktisch überprüfen kann.

Die Genauigkeit der Berechnung sollte sich zwischen der CPU- und GPU-Implementierung kaum unterscheiden, da aktuelle Grafikkarten die OpenCL-Erweiterung `cl_khr_fp64` zur doppelten Genauigkeit unterstützen. Die GPU-Implementierung besitzt sogar bessere Rundungsfehlereigenschaften, da im Gegensatz zum CPU eine Fan-In-Summation verwendet wird [GO96, S. 86].

Der gewählte Lösungsweg mit der Berechnung in der Reduce-Phase ist für dieses Problem eher ungeeignet, da die Map-Phase die Daten nur einliest und konvertiert, ohne eine Berechnung auszuführen. Prinzipiell startet Hadoop implizit mehrere Mapper, um auf diesen verteilt rechnen zu können. Die Ergebnisse der Mapper werden in der Regel durch einen Reducer zusammengeführt. Der ursprüngliche Lösungsweg mit der Berechnung in der Map-Phase ist dafür besser geeignet. Doch auch in dieser Implementierung würde Hadoop nur einen Mapper starten, da die numerische Integration keine oder nur geringe Daten aus dem HDFS benötigt. Diese Einschränkung kann nur durch das in der Analyse beschriebene `InputFormat` gelöst werden.

Dennoch hat diese Implementierung gezeigt, dass es für MapReduce unbedeutend ist, in welcher der beiden Phasen eine GPU-Berechnung stattfindet. Welche Phase durch eine GPU beschleunigt werden soll, ist viel mehr vom konkreten Problem abhängig. Aufgrund der Gruppen und der höheren Datenmenge kann eine Grafikkarte in der Reduce-Phase jedoch pragmatischer genutzt werden.

5. Laufzeitanalyse

Der praktische Performancegewinn des k-Means-Verfahrens und der numerischen Integration wird in diesen Kapitel untersucht. Vor der Laufzeitanalyse werden die genauen Testbedingungen definiert und vorgestellt. Anschließend werden die Laufzeiten der einzelnen Phasen der implementierten MapReduce-Jobs sowie der Speedup analysiert.

5.1. Testaufbau

Die Messungen werden auf einem Node einer Amazon Elastic Compute Cloud (Amazon EC2) mit dem Instance-Typ „Cluster GPU Quadruple Extra Large" ausgeführt. Die technischen Daten des Instance-Typs sind in Anhang A aufgeführt.

Auf diesem Node ist ein Hadoop-Cluster in einer Pseudo-Distributed-Konfiguration eingerichtet. In dieser Konfiguration besteht der Cluster aus nur einem Node und alle Dienste werden von diesem Node ausgeführt. Obwohl dies nicht wirklich der Definition eines Clusters entspricht, ist diese Konfiguration für die Anwendungsbeispiele ausreichend. Die Verwendung eines Clusters mit mindestens zwei Nodes erschwert die Messungen, da eine hohe Datenmenge generiert werden muss, um eine gleichmäßige Verteilung der Daten auf dem HDFS zu erreichen. Durch diese hohe Datenmenge steigt zusätzlich die Laufzeit jeder Testreihe und somit die Kosten der gemieteten Amazon EC2 Instanz. Des Weiteren skaliert ein Hadoop-Cluster mit einem Hochleistungsnetzwerk bei steigender Datenmenge nahezu linear, wenn ein weiterer Slave Node hinzugefügt wird.[1] Dazu müssen die Eingabedaten lediglich in mindestens genauso viele Blöcke unterteilt werden können, wie physikalische Recheneinheiten im Cluster vorhanden sind. Daraus resultiert die Annahme, dass ein Cluster einen Job um den Faktor x schneller ausführt, wenn alle Nodes die Tasks des Jobs ebenfalls um den Faktor x schneller bearbeiten. Aufgrund der gewählten Parallelisierungsstrategie (siehe 3.1.3) und der statischen Zuteilung der Implementierung wird die Anzahl der Map- und Reduce-Tasks auf 1 festgelegt, um sicherzustellen, dass die Grafikkarte auch nur von einem Map- oder Reduce-Task genutzt wird. Somit wird von den vorhandenen zwei Grafikkarten auch nur eine genutzt, wodurch ein direkter Vergleich zwischen einem Prozessorkern und einer GPU erleichtert wird.

Da in der verwendeten Konfiguration keine Netzwerkübertragung stattfindet, entfällt eine Ursache für mögliche Messungenauigkeiten. Das Task Scheduling des Betriebssystems hat ebenfalls nur einen geringen Anteil an Messungenauigkeiten, weil auf dem Node keine weiteren Benutzerprogramme ausgeführt werden und die notwendigen Dienste auf acht Prozessorkerne verteilt werden können. Um weitere Einflüsse auszugleichen, wird für jeden Testlauf der Durchschnitt aus drei Messungen gebildet.

Die folgenden Auswertungen verwenden den Begriff Speedup als Synonym für Geschwindigkeitsvorteil zwischen der Implementierung mit und ohne GPU-Unterstützung. Theoretisch gibt der Speedup an, um welchen Faktor die Ausführung eines Programms beschleunigt werden kann, wenn dieses anstatt auf einem Einprozessorsystem auf einem Mehrprozessorsystem ausgeführt wird. Bei einem Mehrprozessorsystem berechnet sich dieser aus dem Quotienten der Gesamtlaufzeit des Algorithmus auf einem Prozessorkern und der Gesamtlaufzeit auf mehreren Kernen:

[1] Der Skalierbarkeit sind obere Grenzen gesetzt, da die Übertragungsgeschwindigkeit des Netzwerks im Gegensatz zur Rechenleistung nicht erhöht wird.

$$S_p(n) = \frac{T_1(n)}{T_p(n)} \quad \text{mit} \quad \begin{array}{l} S_p(n) \widehat{=} \text{ Speedup bei einem System mit } p \text{ Prozessoren} \\ T_1(n) \widehat{=} \text{ Laufzeit auf einem Einprozessorsystem} \\ T_p(n) \widehat{=} \text{ Laufzeit auf einem System mit } p \text{ Prozessoren} \end{array}$$

Weiterhin gilt im Allgemeinen $1 \leq S_p(n) \leq p$ [RR07, S. 168]. Die untere Grenze von $S_p(n) = 1$ kann in Ausnahmefällen, zum Beispiel aufgrund einer schlechten Implementierung oder zu hohem Kommunikationsaufwand, unterboten werden. In Abhängigkeit der verwendeten Algorithmen wird zwischen dem absoluten und relativen Speedup unterschieden: Der absolute Speedup setzt den besten bekannten sequenziellen Algorithmus auf einem Einprozessorsystem mit dem parallelen Algorithmus auf einem Mehrprozessorsystem in Beziehung. Hingegen sind beim relativen Speedup beliebige Algorithmen für das Einprozessorsystem erlaubt. Häufig wird dafür der parallele Algorithmus auf nur einem Prozessor ausgeführt. [Sch10, S. 7]

Für die weiteren Untersuchungen wird der relative Speedup folglich aus dem Quotienten der Gesamtlaufzeit des Algorithmus ohne GPU-Unterstützung und der Gesamtlaufzeit mit GPU-Unterstützung berechnet:

$$S(n) = \frac{T_{CPU}(n)}{T_{GPU}(n)} \quad \text{mit} \quad \begin{array}{l} S(n) \widehat{=} \text{ Speedup bei einem System mit GPU-Unterstützung} \\ T_{CPU}(n) \widehat{=} \text{ Laufzeit auf ohne GPU-Unterstützung} \\ T_{GPU}(n) \widehat{=} \text{ Laufzeit auf mit GPU-Unterstützung} \end{array}$$

Die folgenden Diagramme entsprechen nicht der üblichen Darstellung des Speedups, mit den Recheneinheiten auf der Abszissen- und dem Speedup auf der Ordinatenachse. Stattdessen ist auf der Abszissenachse eine Eingabegröße, und nicht die Anzahl der verwendeten Stream-Prozessoren, aufgetragen. Anders als bei Mehrprozessorsystemen ist es bei GPGPU nicht üblich die Recheneinheiten über Parameter anzugeben, da die GPU nur effizient als „Ganzes" programmiert werden kann.

5.2. k-Means

Die k-Means-Implementierung besitzt vier Eingabevariablen:

n Anzahl der Punkte

k Anzahl der Gruppenzentren

d Dimension der Punkte und Gruppenzentren

i Anzahl der Iterationen

Die Laufzeit wird für verschiedene Größen und Kombinationen der Variablen untersucht. In der vorliegenden Implementierung sind die Punkte und Gruppenzentren die Eingabedaten, die Hadoop aus dem HDFS liest. Da n in der Regel um ein Vielfaches größer ist als k, ist davon auszugehen, dass die Punkte anhand ihrer Unterteilung in Blöcke im HDFS auf die Nodes verteilt werden. Deswegen wird diese Eingabegröße in Megabyte und nicht als Anzahl der Punkte angegeben. Die Anzahl der Punkte lässt sich somit aus der Dimension und der Dateigröße ermitteln. Die Eingabedaten für die Testläufe werden durch die Methode `Points.generate()` zufällig erzeugt. Dabei werden ungefähr 60 Prozent der zu erzeugenden Punkte um k zufällig erzeugte Zentren verteilt. Die restlichen 40 Prozent werden ohne Berücksichtigung von Gruppenzentren zufällig im Raum verteilt.

Die Anzahl der Iterationen bis der Algorithmus terminiert, ist von den Eingabedaten abhängig. Da n und k während der Laufzeit konstant sind und die Anzahl der Operationen einer Iteration nur von diesen und nicht von den Werten der Punkte abhängen,

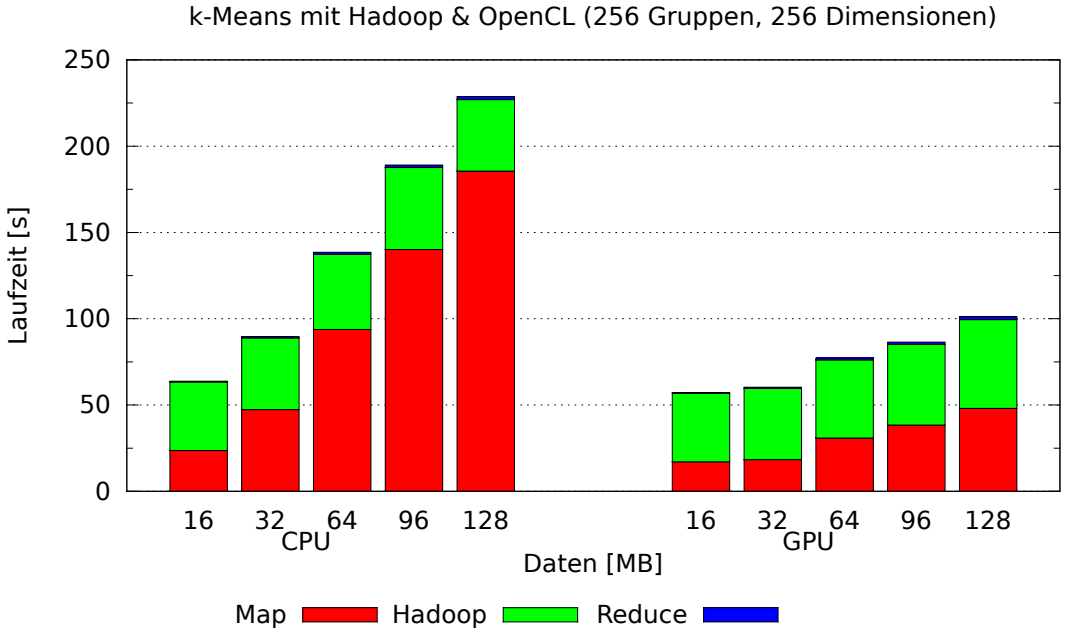


Abbildung 5.1.: Laufzeiten der einzelnen Phasen für das k-Means-Verfahren mit und ohne GPU-Unterstützung

steigt die Laufzeit im Bezug auf i linear an:

$$t(i) = i \cdot t(1) \tag{5.1}$$

Um verwertbare Messergebnisse zu erhalten, muss jeder Testlauf die gleiche Anzahl an Iterationen ausführen. Für die Laufzeitmessungen wird die Anzahl der Iterationen auf 1 festgelegt, um die Dauer der Testläufe nicht unnötig zu erhöhen. Es werden 16, 32, 64, 96 und 128 MB zufällig generierte Punkte als Eingabe genutzt.

Die Abbildung 5.1 stellt die Laufzeiten für 256 Gruppen und 256 Dimensionen dar. Auf der linken Seite sind die Laufzeiten für die Implementierung ohne GPU-Unterstützung und auf der rechten Seite mit GPU-Unterstützung abgebildet. Die Laufzeit ist zusätzlich in die einzelnen Phasen eines MapReduce-Jobs unterteilt. Der Hadoop-Anteil beinhaltet die Zeit für das Starten und Beenden der Jobs sowie die Zeit für den Datenaustausch zwischen der Map- und Reduce-Phase. In dem Säulendiagramm sind die unterschiedlichen asymptotischen Laufzeiten der Map- und Reduce-Phase gut zu erkennen. Zum Beispiel benötigt die Map-Phase der CPU-Implementierung für 128 MB ungefähr 150 Sekunden, wohingegen die Reduce-Phase weniger als 10 Sekunden benötigt. Außerdem ist zu erkennen, dass der Hadoop-Anteil bei steigender Eingabegröße nicht wesentlich ansteigt, obwohl mehr Daten übertragen werden müssen. Da die Testläufe nur auf einem Node ausgeführt wurden, fand die Datenübertragung zwischen der Map- und Reduce-Phase über die Festplatte, wenn nicht sogar nur über den Hauptspeicher statt. Diese Speichermedien benötigen für die Übertragung von 32 MB weniger als eine Sekunde, wodurch dies im Diagramm kaum sichtbar ist. Der Vergleich der Laufzeiten zwischen CPU und GPU zeigt, dass k-Means mit GPU-Unterstützung bei 128 MB mehr als doppelt so schnell ist als ohne GPU-Unterstützung. Des Weiteren steigt die Laufzeit der GPU-Implementierung flacher an als die der CPU-Implementierung.

Die Abbildung 5.2 auf Seite 70 zeigt in zwei Diagrammen den Speedup der GPU-Implementierung mit konstanter Dimension und konstanter Gruppenanzahl. Die GPU-Implementierung ist bei 16 MB, 64 Gruppen und 64 Dimensionen langsamer als die CPU-Implementierung. Für diese Eingabegrößen ist die Berechnung der Map-Phase auf

der CPU schneller als die nötige Organisation für die Berechnung auf der Grafikkarte. Steigt jedoch eine der drei Eingabegrößen an, steigt auch der Rechenaufwand und die Verarbeitung auf der GPU wird praktikabler. Bereits mit 256 Gruppen und nur zwei Dimensionen beträgt der Speedup bei 128 MB fast 2. Auf dem Diagramm mit konstanter Gruppenanzahl ist zu erkennen, dass ab 32 MB der Speedup bei 64 Dimensionen höher ist als der für 256 Dimensionen. Da die asymptotische Laufzeit durch $\Theta(d, n, k) = n \cdot k \cdot d$ bestimmt ist, sollte der Speedup für 256 Dimensionen, bei konstanter Gruppenanzahl, immer höher sein als der für 64 Dimensionen. Dieses Verhalten ist in der Implementierung begründet. Die maximale Anzahl der Punkte, die auf den Grafikkartenspeicher übertragen werden, beträgt $65.536/\mathit{Dimension}$. Bei 64 Dimensionen werden 1.024 und bei 256 Dimensionen 256 Punkte übertragen. Für jeden Punkt wird genau ein Work-Item gestartet, das einer Ausführungseinheit der Grafikkarte zugewiesen wird. Die Grafikkarte des EC2-Testsystems besitzt 448 Stream-Prozessoren, die bei 64 Dimensionen und somit 1.024 Punkten gut ausgelastet werden. Bei 256 Dimensionen werden nur 256 Work-Items gestartet, wodurch 192 Stream-Prozessoren ungenutzt bleiben. Trotz der sehr beschränkten Ausnutzung des Grafikkartenspeichers wird bei 64 Dimensionen, 1.024 Gruppen und 128 MB ein Speedup von 6,5 erreicht. Es ist anzunehmen, dass dieser durch eine bessere Speicherverwaltung weiter erhöht werden kann. Wenn mehr Grafikkartenspeicher genutzt wird, sinkt die Anzahl der gestarteten Datenübertragungen und der zu startenden Kernel. Dadurch sinkt der Kommunikationsaufwand zwischen der Grafikkarte und der Host-Anwendung. Des Weiteren können die Gruppenzentren im Local Memory der Grafikkarte gespeichert werden, wenn die Anzahl der Zentren klein genug ist. Dadurch werden langsame Zugriffe auf den Global Memory reduziert.

Im Vergleich zu anderen Implementierungen ist ein Speedup von 6,5 noch gering. Die Implementierung der Arbeit „A Parallel Implementation of K-Means Clustering on GPUs" [FRCC08] hat einen Speedup von 13. Einen maximalen Speedup von 40 hat die Implementierung der Arbeit „Accelerating K-Means on the Graphics Processor via CUDA" [ZG09]. Dabei ist jedoch zu berücksichtigen, dass beide Implementierungen ohne Einschränkungen durch ein Programmiermodell und Framework für die Grafikkarte optimiert werden konnten. Diese Implementierungen benötigen zum Beispiel keinen Puffer, da die Punkte zusammenhängend eingelesen und auf den Grafikkartenspeicher übertragen werden. Dadurch ist die maximale Anzahl der Punkte auch nicht durch die Puffergröße begrenzt. Zuletzt muss bei einem Speedup verschiedener Rechnersysteme auch die Performance der in Relation gesetzten Hardware berücksichtigt werden.

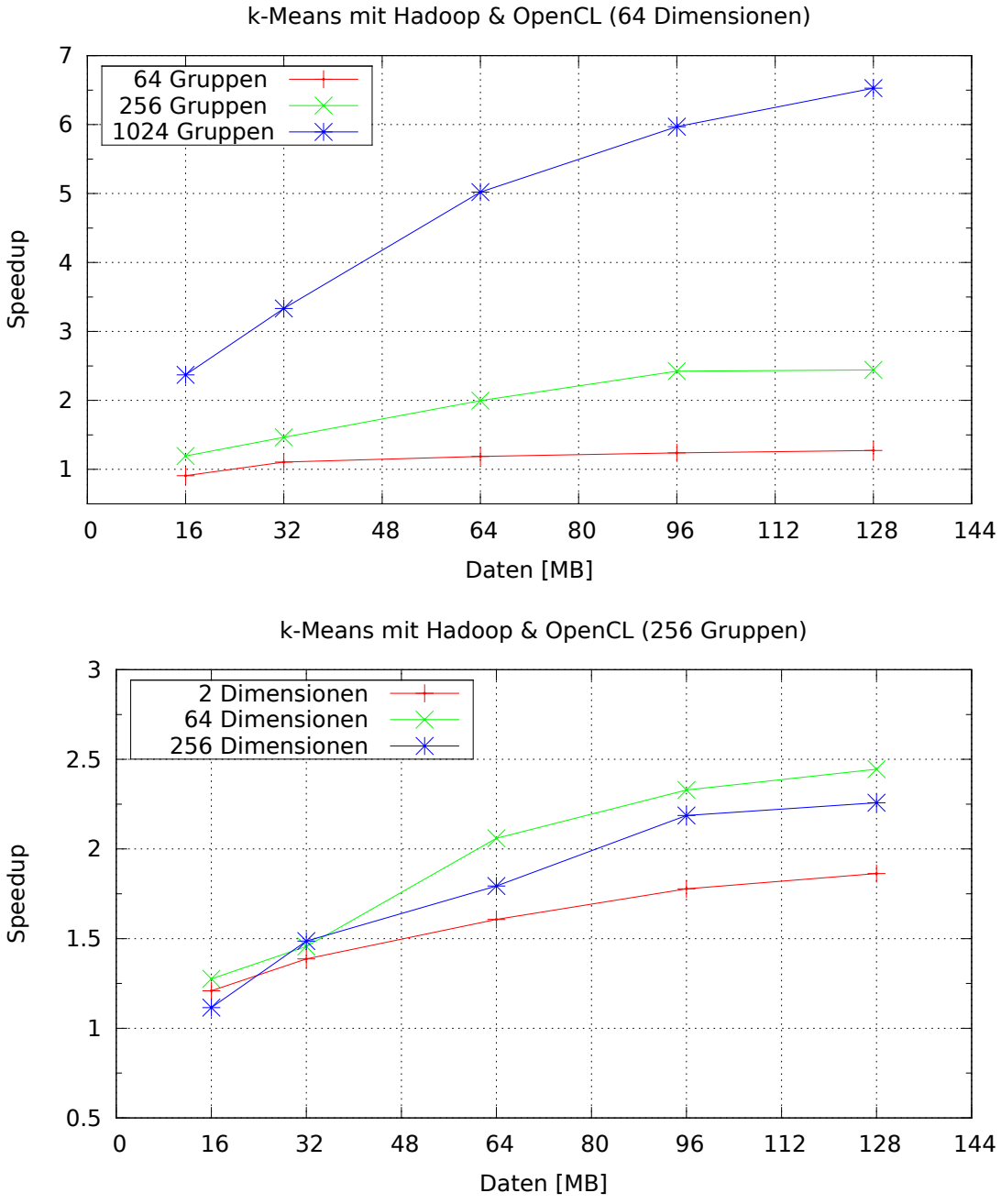


Abbildung 5.2.: Speedup des k-Means-Verfahrens für verschiedene Eingabegrößen

5.3. Numerische Integration

Von der Funktion:

$$f(x) = 3x^3 + 2x^2 + x$$

wird näherungsweise das Integral bestimmt und die benötigte Zeit gemessen. Die Laufzeit der Implementierung ist von zwei Eingabegrößen abhängig:

- Anzahl der Teilintervalle n , auch Auflösung genannt, und
- Anzahl der zu berechnenden Intervalle c.

Die definierten Intervalle sind in einer Datei im HDFS gespeichert. Von jedem in der Datei befindlichen Intervall wird das Integral mit n Teilintervallen näherungsweise berechnet. Die Eingabedatei wird mit Hilfe der Methode `NIData.generateIntervals()` generiert. Diese erstellt aus einem Intervall $[a, b]$ genau c Intervalle der Form:

$$I_1[a, k_1],\ I_2[k_1, k_2],\ ...,\ I_c[k_{c-1}, b]$$

Jedes der gebildeten Intervalle hat die Länge $\frac{b-a}{c}$. Für die Messungen wird das Intervall $[-10, 10]$ in unterschiedlich viele Intervalle geteilt und zur Eingabe im HDFS gespeichert.

Die Abbildung 5.3 auf der nächsten Seite zeigt die Laufzeiten für eine feste Auflösung von 100.000 und einer unterschiedlichen Anzahl von Intervallen. Da sowohl die Implementierung für die CPU als auch für die GPU die gleiche Datenmenge verarbeiten, ist der Anteil für die Initialisierung des Jobs, das Sortieren und Übertragen der Key-Value-Paare (als Hadoop-Phase gekennzeichnet) bei beiden Implementierungen gleich groß. Die Map-Phase wird nur zum Einlesen und Konvertieren der weniger als 25 KB großen Eingabedatei verwendet. Aufgrund des sehr geringen Aufwands ist diese Phase nicht im Diagramm wahrnehmbar. Den größten Rechenaufwand hat die Reduce-Phase mit der Approximation der Integrale. Im Gegensatz zur GPU-Implementierung ist bei der Implementierung für die CPU auf der linken Seite eine deutliche Steigerung der Laufzeit bei wachsender Anzahl der Intervalle zu erkennen.

Hingegen scheint die Grafikkarte eine konstante Zeit zur Berechnung zu benötigen. Erst bei genauerer Betrachtung der Messergebnisse in Tabelle 5.1 auf der nächsten Seite ist zu erkennen, dass auch die GPU-Implementierung mit steigender Eingabemenge mehr Zeit benötigt. Die Spalte „Reduce-Methode" enthält die Laufzeit der Methode `reduce()` inklusive der GPU-Berechnung ohne die Methoden `setup()` und `cleanup()`. Die Differenz der Laufzeiten der Reduce-Phase zwischen 50 und 1.000 Intervallen beträgt nur ungefähr 192 ms. Diese Differenz, zwischen der gemessenen minimalen und maximalen Eingabemenge, ist um mehr als das 44-fache geringer als die Differenz der CPU-Implementierung zwischen 250 und 500 Intervallen. Der Unterschied von einigen hundert Millisekunden, bei einer Gesamtlaufzeit von mindestens 25 Sekunden, wird zusätzlich durch Messungenauigkeiten in allen drei Phasen abgeschwächt, wodurch das Laufzeitverhalten der Grafikkarte im Diagramm nur schwer darstellbar ist.

Die Spalte „Reduce-Phase" beinhaltet die gesamte Reduce-Phase, d.h. alle drei Methoden. Es ist zu erkennen, dass `reduce()` nur einen sehr geringen Anteil an der gesamten Reduce-Phase hat. In Abschnitt 3.3 wurde gemessen, dass auf dem Testsystem EC2 ungefähr 1887 ms zur Initialisierung der OpenCL-Komponenten benötigt werden. Die Reduce-Phase führt die Initialisierung einmalig in der `setup`-Methode durch. Die Differenz der Laufzeiten zwischen der gesamten Reduce-Phase und der Reduce-Methode beträgt im Durchschnitt 3.510,4 ms. Dies zeigt, dass die gemessene Zeit für die Initialisierung tatsächlich benötigt wird und im vorliegenden Fall sogar mehr Zeit beansprucht, als die eigentliche Rechenarbeit.

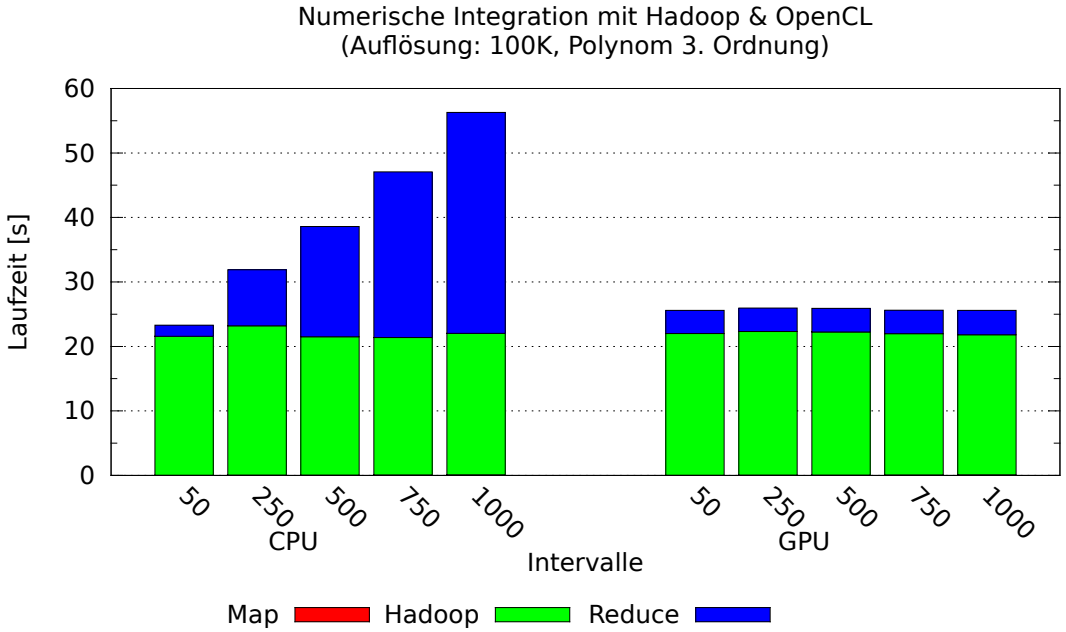


Abbildung 5.3.: Laufzeiten der einzelnen Phasen für die numerische Integration mit und ohne GPU-Unterstützung

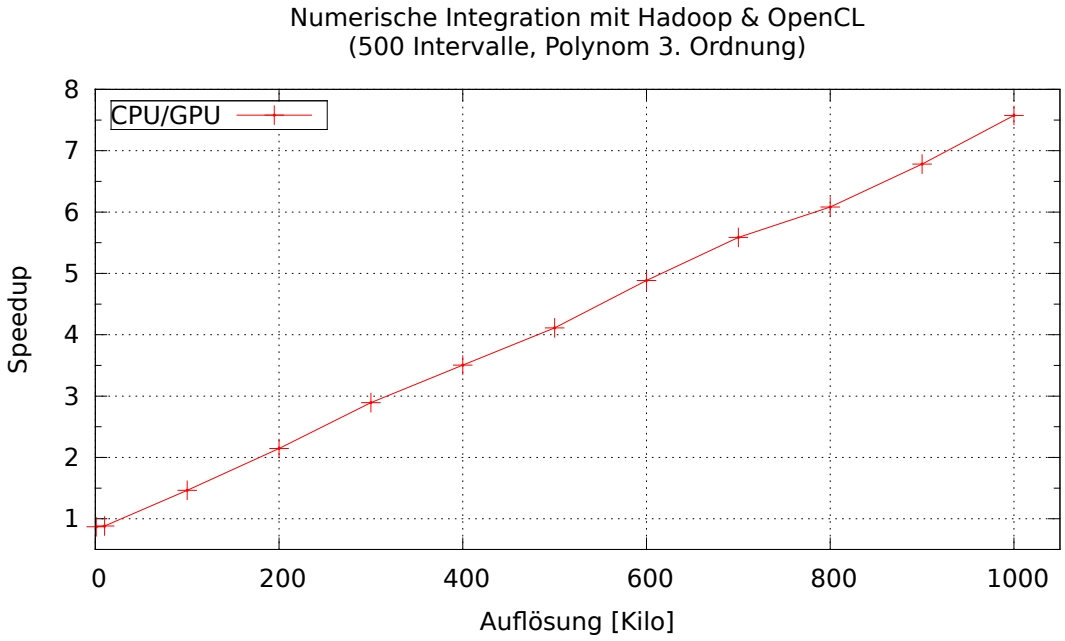
Intervalle	Map-Methode	Map-Phase	Reduce-Methode	Reduce-Phase	Gesamt
50	7,00	16,00	55,00	3.580,00	25.583,33
250	25,33	36,33	118,00	3.644,00	2.5947,00
500	41,00	56,33	137,67	3.668,00	25.916,33
750	57,00	72,00	212,00	3.640,67	25.619,33
1000	69,33	90,00	230,33	3.772,33	25.588,00

Tabelle 5.1.: Laufzeiten der einzelnen Phasen für die numerische Integration mit GPU-Unterstützung ($n = 100.000$, Zeit in Millisekunden).

Die Diagramme der Abbildung 5.4 auf der nächsten Seite zeigen den Speedup mit einer konstanten Anzahl an Intervallen und einer konstanten Auflösung. Auf beiden Diagrammen ist zu erkennen, dass bei der minimalen Eingabegröße ein Speedup von unter 1 erreicht wird und die GPU-Unterstützung die Laufzeit somit negativ beeinflusst. Die Implementierung startet für jedes Teilintervall genau ein Work-Item, wodurch die GPU bereits ab einer Auflösung von 448 physikalisch ausgelastet ist. Das untere Diagramm zeigt jedoch, dass die GPU-Implementierung selbst bei 100.000 Teilintervallen langsamer als die CPU-Implementierung ist. Der Rechenaufwand eines Work-Items reicht somit nicht aus, um den Organisationsaufwand für die GPU-Berechnung zu rechtfertigen. Dieser Rechenaufwand kann entweder durch eine komplexere Funktion $f(x)$ oder durch die Verarbeitung mehrerer Intervalle erreicht werden. Auf dem Diagramm mit der konstanten Auflösung ist zu erkennen, dass ab ungefähr 100 Intervallen ein Geschwindigkeitsvorteil mit der GPU-Unterstützung erreichbar ist. Mit wachsender Anzahl der Intervalle steigt der Speedup jedoch nur schwach an. Bei der Verdopplung der Intervalle von 250 auf 500 steigt der Speedup nur um 0,25 auf 1,5 an. Erst bei ungefähr 900 Intervallen ist die GPU-Implementierung doppelt so schnell wie das Pendant auf der CPU. Die Anzahl der Intervalle hat somit nur einen geringen Einfluss auf den Speedup.

Das obere Diagramm zeigt den Speedup für eine steigende Auflösung bei konstanter Anzahl der Intervalle. Darin ist zu erkennen, dass die Auflösung einen größeren Einfluss auf den Speedup hat, als die Anzahl der Intervalle. Der Speedup steigt um ungefähr 1,5 auf 3,5, wenn die Auflösung von 200.000 auf 400.000 verdoppelt wird. Ein Speedup von 7,5 konnte bei 500 Intervallen und einer Million Teilintervallen erreicht werden.

Der Speedup kann jedoch nicht beliebig gesteigert werden, indem zum Beispiel die Anzahl der Teilintervalle erhöht wird. Einerseits kann die Trapezregel bereits ab 1.000 Teilintervallen eine mittlere Genauigkeit erreichen und andererseits verringert sich die Länge h der Teilintervalle mit steigendem n, wodurch Fehler aufgrund der festen Maschinengenauigkeit auftreten [Pie10, GO96, S. 180]. Somit ist eine höhere Anzahl der Teilintervalle nur bei einem längeren Intervall sinnvoll. Ebenfalls ist die Angabe mehrerer Intervalle für eine Funktion $f(x)$ ein eher konstruierter Anwendungsfall. Praktisch wird ein variables h nicht statisch vor der Berechnung angegeben, sondern dynamisch während der Laufzeit ermittelt.



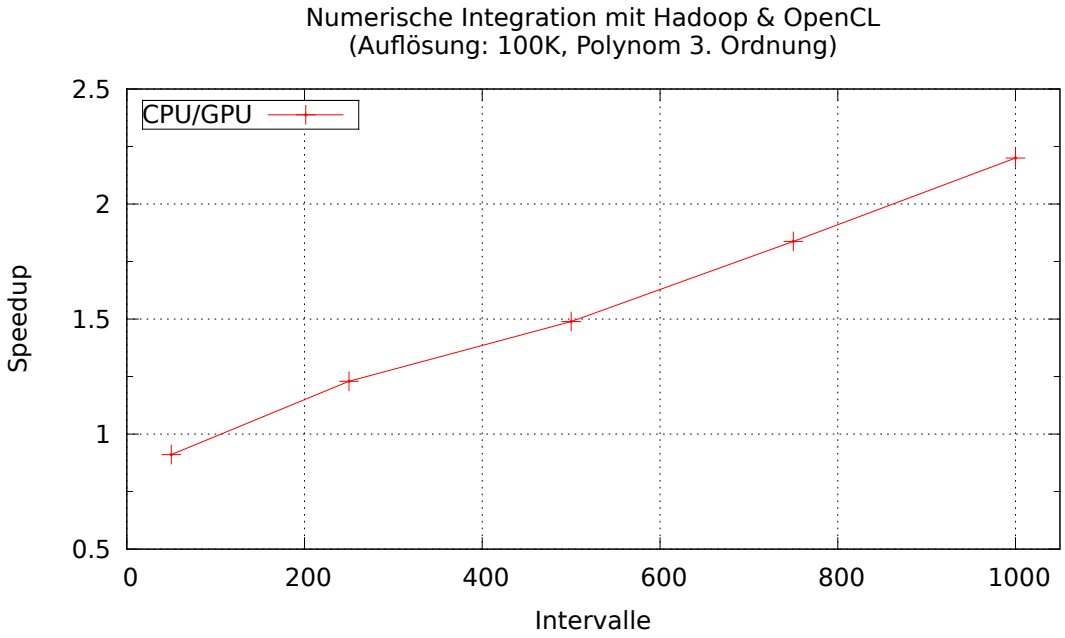


Abbildung 5.4.: Speedup der numerischen Integration für verschiedene Eingabegrößen

6. Schlusswort

6.1. Zusammenfassung

6.1.1. Paralleles Rechnen

Die in dieser Arbeit untersuchten Technologien sind in den Bereich der Parallelverarbeitung einzuordnen. Der Grundgedanke ist die Annahme, dass zwei parallele Recheneinheiten mit der Taktfrequenz f die selbe Arbeit leisten können, wie eine Recheneinheit mit der Taktfrequenz $2f$. Um diese Parallelität ausnutzen zu können, muss der sequenzielle Befehlsstrom des Algorithmus in mindestens zwei Befehlsströme aufgeteilt werden. Unter Zunahme weiterer Recheneinheiten kann die Leistung theoretisch beliebig erhöht werden. In der Praxis unterliegt diese Annahme jedoch mehreren Einschränkungen, die durch Hardwarearchitekturen und Programmiermodelle vorgegeben sind.

Abschnitt 2.1 erläutert die Grundlagen des parallelen Rechnens. Ein allgemeiner Vergleich paralleler Hardwarearchitekturen ist mit Hilfe der Flynnschen Klassifikation möglich. Diese unterteilt Rechnerarchitekturen, anhand ihrer Art und Weise wie sie Befehls- und Datenströme verarbeiten, in vier Klassen. Parallelrechner werden der SIMD- oder MIMD-Klasse zugeordnet. Parallelrechner die dem Prinzip von Single Instruction Multiple Data (SIMD) zu zuordnen sind, besitzen mindestens zwei Verarbeitungseinheiten mit nur einem Daten- und Programmspeicher. Dadurch ist es möglich, dass jede Verarbeitungseinheit die selbe Instruktion aus nur einem Befehlsstrom parallel auf unterschiedlichen Daten ausführt. Anders als bei SIMD werden bei dem Prinzip von Multiple Instruction Multiple Data (MIMD) unabhängige Befehlsströme auf voneinander unabhängigen Daten ausgeführt. Dafür sind mindestens zwei Verarbeitungseinheiten mit jeweils einem eigenen Programmspeicher notwendig. Da sich Rechnersysteme, die der MIMD-Klasse zu zuordnen sind, sehr stark voneinander unterscheiden können, unterteilt Tanenbaum diese anhand ihrer Speicherarchitektur in Systeme mit gemeinsamen und verteilten Speicher.

Die Art und Weise wie ein Rechnersystem programmiert werden kann, ist von der Hardwarearchitektur abhängig. Die wesentlichen Grundprinzipien der Programmierung eines Rechnersystems können durch Programmiermodelle verallgemeinert werden. Ein Programmiermodell reduziert die vorhandene Hardwarearchitektur auf die charakteristischen Merkmale, um die Programmierung durch das Ausblenden von Details zu vereinfachen. Viele der aktuell verbreiteten parallelen Programmiermodelle basieren auf einem oder mehreren der in Abschnitt 2.1.4 vorgestellten Modelle. Wenn jedes Datum einer Datenstruktur, im Kontext der Problemstellung, nicht in Beziehung mit einem anderen steht, so ist die Reihenfolge der Verarbeitung nicht fest vorgeben. Aufgrund dessen können diese Daten parallel verarbeitet werden, das heißt dieselben Operationen werden auf unterschiedlichen Elementen der Datenstruktur ausgeführt. Diese Art der Verarbeitung wird als Datenparallelität bezeichnet. Alle SIMD-Architekturen bedienen sich dieser Parallelität, die den Daten innewohnt. Jedoch nicht alle Problemstellungen können mit diesem Modell gelöst werden. Häufig sind die Daten nicht unabhängig oder der Algorithmus lässt solch eine Verarbeitung nicht zu. Neben den Daten, enthält der Algorithmus oft selbst voneinander unabhängige Befehlsketten. Diese Befehlsketten können in der Regel zu Funktionen oder Aufgaben (Tasks) abstrahiert werden. Jede Aufgabe wird dann durch einen Thread oder Prozess ausgeführt. Das Betriebssystem verteilt die Threads auf die verfügbaren Recheneinheiten, die die Aufgaben parallel verarbeiten. Aufgrund der Bildung von Funktionen und Aufgaben heißt dieses Modell Funktions- oder

auch Task-Parallelität. Alle parallel ausgeführten Teile eines Programmes müssen bei der Task-Parallelität an verschiedenen Stellen der Ausführung miteinander kommunizieren. Diese Kommunikation dient entweder der Synchronisation oder dem Datenaustausch. Eine Kommunikationsform bei der dies durch das Senden und Empfangen von Nachrichten stattfindet, wird Nachrichtenaustausch genannt. Bei einer Kommunikation treten Sende- und Empfangsoperation immer als Paar auf. Dabei erfolgt ein Nachrichtenaustausch zwischen zwei Prozessen, die auf demselben System oder auf getrennten Systemen ausgeführt werden. Diese Kommunikationsform wird bei Mehrcomputersystemen eingesetzt, um Daten zwischen den getrennten Speicherbereichen der Recheneinheiten zu übertragen. Aber auch der Einsatz bei Mehrprozessorsystemen ist möglich.

Mit den eingeführten Kenntnissen werden die Technologien Apache Hadoop und Open Computing Language (OpenCL) erklärt. In Abschnitt 2.2 wird das Cluster-System Apache Hadoop vorgestellt. Ein Cluster ist ein Verbund aus mehreren Computern (Nodes), die sich dem Anwender gegenüber als nur ein Computer präsentieren. Diese Rechnerverbünde sind MIMD-Systeme mit verteilten Speicher. Apache Hadoop ist ein Vertreter der High Performance Computing (HPC) Cluster. HPC-Cluster werden genutzt um aufwendige Rechenaufgaben (Jobs) zu lösen, bei denen die Berechnung auf einem Computer entweder zu viel Zeit in Anspruch nehmen würde oder aber die Datenmenge nicht von einen Computer allein verarbeitet und gespeichert werden kann. Das Cluster-System Hadoop ist auf die Verarbeitung von riesigen Datenmengen spezialisiert. Dafür stellt es verschiedene Dienste bereit, die den Bereichen verteilter Speicher und verteiltes Rechnen zugeordnet werden können. Den Bereich des verteilten Rechnens ist ein zentrales Job-Managementsystem zu zuordnen. Dieses nimmt Aufgaben entgegen und zerlegt sie in Teilaufgaben, die dann von den Nodes bearbeitet werden. Hauptkomponente des verteilten Speichers ist das Hadoop Distributed File System (HDFS). Das HDFS kann große zusammenhängende Daten verteilt im Cluster speichern, indem es diese beim Schreiben in Blöcke zerlegt und anschließend auf die Nodes verteilt. Aufgrund dieser Zerlegung entsteht eine Datenparallelität, die durch das Programmiermodell MapReduce ausgenutzt wird. Mit dem gleichnamigen Framework werden parallele und verteilte Jobs für ein Hadoop-Cluster entwickelt. Die Abarbeitung eines MapReduce-Jobs erfolgt in zwei Phasen mit Key-Value-Paaren als zentrale Datenstruktur. Die Verarbeitung startet mit der Map-Phase. Dabei werden Daten aus dem HDFS gelesen und in Form von Key-Value-Paaren der `map`-Methode als Eingabe übergeben. Diese Methode verarbeitet den Key mit dem zugehörigen Value und liefert ein neues Key-Value-Paar als Ausgabe. Alle Ausgaben der parallel verarbeiteten `map`-Methoden werden vom Cluster entgegengenommen und anhand des Keys in Gruppen eingeteilt. Jede Gruppe ist durch ihren Key gekennzeichnet und enthält alle Values, die zusammen mit diesem Key ausgegeben wurden. Darauf folgt die Reduce-Phase. Während dieser Phase verarbeitet die `reduce`-Methode eine zusammengestellte Gruppe und liefert ein oder mehrere Key-Value-Paare als Ergebnis. Abschließend werden alle Ergebnisse vom Cluster gesammelt und sortiert im HDFS gespeichert.

Der Abschnitt 2.1.3 führt die noch recht junge Technologie General Purpose Computation on Graphics Processing Unit (GPGPU) ein. Sie ermöglicht die Verwendung von Grafikkarten zur Lösung von Problemen, die bisher nur durch Hauptprozessoren bearbeitet werden konnten. Grafikkarten sind sehr auf das Aufgabenfeld der Bildverarbeitung spezialisiert, das ein hohes Maß an Parallelität bietet. Um dieses Potenzial effizient ausnutzen zu können, entwickelten sich Grafikkarten mit steigenden Leistungsanforderungen zu hochgradig paralleler Hardware. Die zunehmende Popularität von 3D-Grafik, durch Videospiele oder CAD-Anwendungen, erweiterte den Funktionsumfang in Form von zusätzlichen Befehlen zur freien Programmierung von 3D-Effekten. Aufgrund der Entwicklung immer umfangreicherer Befehlssätze können auch Probleme anderer Aufgabenfelder das Leistungspotenzial moderner Grafikkarten nutzen. Der Begriff GPGPU umfasst so-

wohl die Idee der Nutzung von Grafikkarten für allgemeine Berechnungen, als auch alle Technologien die dies ermöglichen. Eine offene Schnittstelle und Bibliothek ist OpenCL. Der Abschnitt 2.3 erklärt wie damit Grafikkarten programmiert werden. Eine OpenCL-Anwendung besteht aus einem Host- und einem Kernel-Teil. Eine OpenCL-Anwendung ist mit einer normalen Anwendung für den Hauptprozessor vergleichbar, nur dass aufwendige Teile vom Grafikprozessor (GPU) und nicht vom Hauptprozessor (CPU) berechnet werden. Der Host-Teil der Anwendung stellt die Schnittstelle zwischen CPU und GPU dar, indem er alle benötigten Komponenten initialisiert und die Verarbeitung auf der GPU startet. Der Kernel-Teil wird in der Programmiersprache OpenCL-C programmiert und führt die Berechnung auf der Grafikkarte aus. Das Programmiermodell entspricht überwiegend der Datenparallelität, auch wenn eine Funktionsparallelität genutzt werden kann. Im Gegensatz zu einer CPU enthält eine GPU hunderte Recheneinheiten. Die Berechnungen werden zur Laufzeit von sogenannten Work-Items ausgeführt, die auf die Recheneinheiten verteilt werden. Ein Work-Item stellt dabei einen logischen Ausführungspfad dar und ist mit einem Thread vergleichbar. Eine Synchronisation zwischen Work-Items ist nur innerhalb einer Work-Group möglich, in die mehrere Work-Items gruppiert werden können. Jedes Work-Item verfügt über einen globalen und gruppeninternen Index, der zum Beispiel zur Datenauswahl heran gezogen werden kann. Der größte Unterschied zwischen der CPU- und GPU-Programmierung ist das 4-schichtige Speichermodell. Der Host-Teil nutzt, wie konventionelle Programme auch, den Hauptspeicher. Für eine GPU-Verarbeitung müssen alle benötigten Daten aus dem Hauptspeicher auf den Grafikkartenspeicher (Global Memory) kopiert werden. Jedes Work-Item hat Lese- und Schreibzugriff auf alle Daten im, vergleichsweise langsamen, Grafikkartenspeicher. Um ein vielfaches kleiner ist der Speicher, der jeder Work-Group zur Verfügung steht (Local Memory). Auf diesen schnelleren Speicher haben alle Work-Items einer Gruppe vollen Zugriff. Außerdem besitzt jedes Work-Item einen eigenen privaten Speicher (Private Memory). Nach der Berechnung auf der GPU müssen alle Ergebnisse zur Weiterverarbeitung durch den Host-Teil vom Grafikkartenspeicher in den Hauptspeicher kopiert werden.

6.1.2. Problemanalyse

Mit Hilfe der gesammelten Grundlagen von Hadoop und OpenCL untersucht das Kapitel 3, welche Möglichkeiten und Einschränkungen bestehen, beide Technologien miteinander zu kombinieren. Ein Hadoop-Cluster bietet zwei Ansatzpunkte zur Beschleunigung durch eine Grafikkarte. Wenn die Grafikkarte als ein Mehrkernprozessor mit mehreren hundert Prozessorkernen aufgefasst wird, kann diese in das Task Scheduling des Clusters eingebunden werden. Der Vorteil von diesem Anstatz ist, dass der MapReduce-Job nicht zusätzlich für die Grafikkkarte parallelisiert werden muss, sondern als sequenzielles Programm in OpenCL-C implementiert wird. Der TaskTracker kann die GPU jedoch nicht ohne eine Erweiterung als Zusatzprozessor nutzen, da er für jeden Task nur einen Thread startet ohne diesen explizit einen Prozessorkern zu zuweisen. Der TaskTracker verwaltet nur die Anzahl und Fortschritte der gestarteten Tasks. Die Zuweisung der Threads auf die Prozessorkerne übernimmt das Betriebssystem. Für eine GPU-Unterstützung muss vom TaskTracker die GPU-Implementierung der Tasks geladen und aktiv auf der Grafikkarte gestartet werden. Dieser Ansatz ist sehr aufwendig, da neben der zusätzlichen GPU-Implementierung auch der TaskTracker erweitert werden muss. Eine einfachere Möglichkeit besteht darin, den Task für die Verarbeitung auf der GPU zu parallelisieren. Dies ist bei MapReduce eigentlich nicht üblich, da Mehrkernprozessoren durch das Ausführen mehrerer Tasks ausgelastet werden. Bei diesen Vorgehen findet eine 2-stufige Parallelverarbeitung statt. In der ersten Stufe sind die Daten aufgrund der Blöcke im HDFS bereits unterteilt und können von unabhängigen Tasks bearbeitet werden. Unter Ausnutzung der im Task enthaltenen Funktions- oder Datenparallelität wird dieser für die Grafikkarte parallelisiert. Die Effizienz der zweiten Stufe ist stark vom konkreten Job abhängig. In

der Map-Phase sind mit dem Key-Value-Paar nur wenig Daten zu verarbeiten, weshalb eine Grakkarte nur schwer ausgelastet werden kann. Die Reduce- Phase eignet sich besser für eine Berechnung auf der GPU, da für einen Key mehrere Values zu verarbeiten sind. Obwohl diese Art der Parallelisierung einfacher als die Einbindung in das Task Scheduling ist, müssen für diese Umsetzung weitere Aspekte untersucht werden.

Da das MapReduce-Framework in der Programmiersprache Java programmiert ist, werden in Abschnitt 3.2 geeignete Schnittstellen zur Anbindung der in C verfügbaren OpenCL-Bibliothek analysiert. Hadoop stellt über das Framework zwei Funktionen bereit, um MapReduce-Jobs auch in anderen Programmiersprachen implementieren zu können. Mit Hadoop Streaming kann jede beliebige Programmiersprache genutzt werden, da die Kommunikation und Datenübertragung über die Standardeingabe und Standardausgabe stattfindet. Hadoop Pipes stellt für C++ eine pragmatischere Schnittstelle bereit, die einige Fehlerquellen von Hadoop Streaming ausschließt. Unabhängig vom MapReduce-Framework ermöglichen Java Native Access (JNA) und Java Native Interface (JNI) den Zugriff auf plattformspezifische Programmbibliotheken wie OpenCL. Ein großer Vorteil der letzten beiden Möglichkeiten ist, dass der gesamte Job fast ausschließlich in Java implementiert werden kann und somit alle Funktionen der Hadoop-API verfügbar sind. Anhand eines Beispiel-Jobs wurde die Performance und Pragmatik aller vier Möglichkeiten genauer untersucht. Am schnellsten kann OpenCL mit der JNA-Bibliothek JavaCL angebunden werden, die zusätzlich eine sehr verständliche API bereitstellt und deswegen für alle weiteren Untersuchungen verwendet wird.

Um den Einfluss der JavaCL-Bibliothek auf einen MapReduce-Job besser einschätzen zu können, wurden in Abschnitt 3.3 die Bearbeitungszeiten wichtiger OpenCL-Operationen gemessen. Für die Initialisierung aller benötigten OpenCL-Komponenten vergehen auf den Testsystemen 528,4 und 1.887 ms. Schlussfolgernd sollten diese Operationen nur einmalig ausgeführt werden. In jeder Phase kann dafür die `setup`- und `cleanup`-Methode verwendet werden. Außerdem wurde die Lese- und Schreibgeschwindigkeit zwischen Haupt- und Grafikkartenspeicher gemessen. Aus diesen Messungen ist abzulesen, dass viele Einzelübertragung im Vergleich zu größeren zusammenhängenden Übertragungen langsamer sind. So benötigt auf dem PC das Lesen von 128 Einzelwerten 18,6 ms. Hingegen werden 65.536 zusammenhängende Werte in nur 4,8 ms gelesen. Für einen MapReduce-Job bedeutet dies, dass Key-Value-Paare in Puffern gesammelt und anschließend zusammenhängend übertragen werden sollten.

Die Datenorganisation der beiden Technologien unterscheidet sich stark voneinander. In herkömmlichen Anwendungen ist die zu verarbeitende Datenmenge bekannt oder kann ermittelt werden. Dadurch kann eine zusammenhängende Datenübertragung auf den Grafikkartenspeicher stattfinden. MapReduce liefert die Key-Value-Paare jedoch als Datenstrom, wodurch die zu verarbeitende Datenmenge nicht abgeschätzt werden kann. Die durch den Datenstrom vorgegebene Einzelübertragung auf den Grafikkartenspeicher ist sehr ineffizient und kann durch eine Zwischenspeicherung vermieden werden. Dabei ist zu berücksichtigen, dass die Datenmenge eines Jobs in der Regel nicht vollständig in den Haupt- und Grafikkartenspeicher geladen werden kann. Der Abschnitt 3.4 beschreibt wie diese Daten mit Hilfe eines Puffers effizienter übertragen werden können. Ein Puffer speichert eine kleinere Datenmenge zwischen und überträgt diese auf die Grafikkarte, wenn er gefüllt ist. Hierbei ist zu beachten, dass Speicherbereiche in Java nicht explizit freigegeben werden können, da dies eine Garbage Collection übernimmt. Aufgrund des zusammenhängenden Speichers eines Puffers, würden viele neue Puffer den Speicher schnell füllen und zyklisch eine aufwendige Speicherbereinigung auslösen. Indem ein Puffer durch einfaches Überschreiben der Inhalte wiederverwendet wird, kann der Aufwand einer Speicherbereinigung erheblich gesenkt werden.

6.1.3. Praktische Untersuchungen

Die bisher nur unabhängig voneinander betrachteten Untersuchungen werden in Kapitel 4 und 5 auf zwei Beispielimplementierung angewendet. Aus der Entwicklung gingen die allgemeinen Schnittstellen `ICLKernel` und `ICLBufferedOperation` hervor, um die Abhängigkeit der verschiedenen OpenCL-Komponenten zu verringern. Die Schnittstelle `ICLKernel` ermöglicht eine typsichere Verbindung zwischen Java und einen OpenCL-Kernel. Zusätzlich wird sichergestellt, dass der Kernel mit korrekten Parametern initialisiert und gestartet wird. `ICLBufferedOperation` nutzt eine oder mehrere Implementierungen von `ICLKernel`, um eine komplexe Operation umzusetzen. Außerdem müssen alle Objekte für die GPU-Berechnung serialisiert und die Ergebnisse gegebenenfalls deserialisiert werden. Durch diese Schnittstelle wird dem Anwender die Arbeit mit einer gepufferten OpenCL-Operation erleichtert.

Als eine Beispielimplementierung wurde die Clusteranalyse mit dem k-Means-Verfahren ausgewählt. Ziel einer Clusteranalyse ist das Bilden von Gruppen aus einer Menge von Objekten, die in einer definierten Eigenschaft möglichst ähnlich zueinander sind, sich jedoch von Objekten anderer Gruppen möglichst stark unterscheiden sollen. k-Means benötigt als Eingabe k Gruppen, die in Form von Zentren angegeben und vor dem Start bestimmt werden. In jeder Iteration werden die Objekte anhand einer Metrik einem Zentrum zugeordnet. Anschließend werden von allen gebildeten Gruppen neue Zentren errechnet und als Eingabe für die nächste Iteration genutzt. Das Verfahren terminiert, wenn sich die Zentren nicht mehr ändern oder eine definierte Anzahl von Iterationen durchlaufen wurde. Das Verfahren wurde für eine Menge von n-dimensionalen Punkten in einem euklidischen Raum implementiert. Die Map-Phase erhält als Eingabe die Punkte der Eingabemenge als Value. Die Gruppenzentren werden über einen gemeinsamen Speicher eingelesen. Jeder Punkt kann unabhängig und parallel einem Gruppenzentrum zugewiesen werden. Als Ausgabe liefert die Map-Phase einen Punkt (als Value) mit dem zugewiesenem Gruppenzentrum (als Key). Die Reduce-Phase erhält als Eingabe die gebildeten Gruppen und errechnet für jede Gruppe unabhänig das neue Gruppenzentrum. In der Map-Phase ist die Verarbeitung von nur einem Punkt (Value) auf der GPU langsamer als auf dem CPU. Daher werden mehrere Punkte in einem Puffer gesammelt und parallel auf der Grafikkarte verarbeitet. Die Reduce-Phase konnte durch eine GPU-Unterstützung nicht beschleunigt werden. Die Beschleunigung durch die Grafikkarte ist stark von den Eingabegrößen abhängig. Wenn mindestens so viele Punkte wie physikalisch vorhandene Recheneinheiten der Grafikkarte verarbeitet werden, steigt der Speedup mit steigender Dimension und Gruppenanzahl. Der maximal gemessene Speedup beträgt 6,5. Bei der Entwicklung wurde ein Nachteil der Serialisierung und Deserialisierung festgestellt. Durch die Serialisierung werden Objekte in skalare Datentypen zerlegt. Wenn die serialisierten Daten durch eine Deserialisierung zurück in Objekte konvertiert werden, entstehen neue Objektreferenzen. Somit existieren im Speicher zwei Objekte mit demselben Inhalt. Das ist in der Regel unerwünscht, da alle über die Laufzeit aufgebauten Datenstrukten auf alte Objekte verweisen. Schlussfolgernd werden bei der Serialisierung und Deserialisierung Zusatzinformationen benötigt, um aus den Daten keine neuen Objekte zu erzeugen, sondern bereits existierende Objekte zu aktualisieren. Unter Verwendung solcher Zusatzinformationen, wie zum Beispiel Indizes, genügt es, wenn nur die zur GPU-Berechnung benötigten Objektattribute übertragen werden.

Neben der Clusteranalyse wurde eine numerische Integration implementiert. Hierbei wird das Integral der Funktion $f(x)$ im Intervall $[a, b]$ näherungsweise berechnet, indem die Funktion $f(x)$ durch eine einfacher zu berechnende Funktion n-ten Grades ersetzt wird. Bei der verwendeten Trapezregel dient die Formel für den Flächeninhalt eines Trapez als Grundlage. Um die Genauigkeit der Berechnung zu steigern, wird das Intervall $[a, b]$ in n Teilintervalle zerlegt. Auf jedes Teilintervall wird die Trapezregel angewendet. Die Summe der n Teilergebnisse ist das approximierte Integral der Funktion $f(x)$ im In-

tervall $[a, b]$. Die numerische Integration kann gut auf das MapReduce-Modell abgebildet werden. In der Map-Phase werden parallel alle Teilintervalle berechnet. Anschließend berechnet die Reduce-Phase die Summe. Dieses Vorgehen kann aber nicht ohne Änderung auf die MapReduce-Implementierug von Hadoop angewendet werden, weil die Parallelität von der Anzahl und Verteilung der Datenblöcke im HDFS abhängig ist. Da eine numerische Integration keine Daten aus dem HDFS benötigt, wurde ein erweiterter Anwendungsfall konstruiert. Über eine Eingabedatei im HDFS können mehrere Intervalle für die Integration angegeben werden. Dies ermöglicht die Berechnung mehrerer Integrale unterschiedlicher Intervalle für eine Funktion. Zusätzlich kann ein Intervall manuell in Teilintervalle zerlegt werden, um die Genauigkeit in festgelegten Bereichen zu erhöhen. Des Weiteren wurde die Aufteilung der MapReduce-Phasen geändert, um den Einfluss einer GPU-Unterstützung in der Reduce-Phase zu untersuchen. Dabei wird die Map-Phase für das Einlesen und Konvertieren der Eingabedatei verwendet. Die Reduce-Phase führt die numerische Integration für alle Intervalle aus. Die GPU-Implementierung sammelt mehrere Intervalle in einem Puffer und überträgt diese auf den Grafikkartenspeicher. Die GPU startet bei der Verarbeitung n Work- Items, von denen jedes genau ein Teilintegral von jedem Intervall berechnet. Die Teilintegrale eines Intervalles werden abschließend durch eine Fan-In-Summation auf der Grafikkarte addiert. Die Laufzeitanalyse ergab, dass die GPU-Implementierung mit steigender Anzahl der Teilintervalle besser skaliert, als mit steigender Anzahl der Eingabeintervalle. Außerdem ist der Anstieg der Laufzeit bei steigender Eingabegröße für die GPU-Implementierung im Vergleich zur CPU-Implementierung sehr gering. Der MapReduce-Job mit GPU-Unterstützung erreicht bei 500 Intervallen und einer Million Teilintervallen einen Speedup von 7,5.

6.2. Fazit

Die vorliegende Arbeit hat gezeigt, dass ein MapReduce-Job auf einem Hadoop-Cluster unter Verwendung einer Grafikkarte beschleunigt werden kann. Dabei ist die Beschleunigung stark von der Problemstellung und der Implementierung abhängig.

Eine einfache Einbindung von Grafikkarten stellt die Parallelisierung eines Tasks dar, da hierfür das MapReduce-Framework nicht erweitert werden muss. Um die OpenCL-Bibliothek zur GPU-Programmierung nutzen zu können, stehen drei Ansätze zur Verfügung. Als besonders performant und komfortabel hat sich die JNA-Bibliothek JavaCL herausgestellt. Mit dieser Bibliothek kann bis auf den OpenCL-Kernel alles in Java implementiert werden. Die Anbindung einer Grafikkarte in MapReduce ist mit diesen Ansatz sehr einfach.

Um jedoch eine hohe Beschleunigung zu erzielen, müssen einige Details berücksichtigt werden. Dadurch steigt der Programmieraufwand im Vergleich zu einem MapReduce-Job ohne GPU-Unterstützung stark an. Die Initialisierung von OpenCL-Komponente sollte immer in den beiden Methoden `setup()` und `cleanup()` der jeweiligen MapReduce-Phase stattfinden. Am aufwendigsten hat sich die Entwicklung einer Speicherverwaltung herausgestellt. Die Übertragung einzelner Daten zwischen Haupt- und Grafikkartenspeicher ist sehr teuer. Eine Übertragung großer zusammenhängender Daten durch einen Puffer hat sich als performante Lösung bewährt. Die Implementierung einer gepufferten Speicherorganisation, die den gesamten Grafikkartenspeicher ausnutzen kann, ist aufgrund der dynamischen Speicherverwaltung der JVM sehr aufwendig.

Die Grafikkarte kann ohne Einschränkungen bei der Programmierung in beiden Phasen verwendet werden. Welche Phase durch die GPU unterstützt werden soll, ist hauptsächlich von der Problemstellung abhängig. Grundsätzlich kann die GPU in der Reduce-Phase besser und pragmatischer ausgelastet werden, da die `reduce`-Methode in der Regel viele Values verarbeiten muss. Hingegen verarbeitet die Map-Phase nur einzelne Key-Value-Paare, wodurch der Grafikkarte häufig zu wenig Daten für eine effiziente Verarbeitung

zur Verfügung stehen. Eine Lösung ist das Sammeln von Daten über mehrere Methodenaufrufe von `map()` hinweg. Dies stellt jedoch eine eher provisorische Lösung des Problems dar.

6.3. Ausblick

Einige Überlegungen und Ansätze dieser Arbeit wurden nur prototypisch implementiert oder theoretisch betrachtet. Die gepufferte Speicherverwaltung der Implementierung nutzt nur sehr wenig Grafikkartenspeicher aus. Durch eine bessere Lösung könnte der gesamte Grafikkartenspeicher genutzt und somit die Beschleunigung gesteigert werden. Die neuen Konzepte sollten möglichst generisch und flexibel implementiert werden, um diese in verschiedenen MapReduce-Jobs wiederverwenden zu können.

Die Einbindung der Grafikkarte in das Task Scheduling bietet großes Potenzial, weil nun der Task nicht zusätzlich für die Grafikkarte parallelisiert werden muss. Stattdessen wird die Funktionsparallelität der Grafikkarte genutzt, wodurch der OpenCL-Kernel als sequenzieller Algorithmus in OpenCL-C implementiert werden kann. Ein Work-Item verarbeitet dabei, ähnlich wie die CPU-Implementierung, mehrere Eingaben für die `map`- und `reduce`-Methode. Die Implementierung des OpenCL-Kernels ist dadurch sehr einfach. Da für die gesamte Map- bzw. Reduce-Phase nur ein OpenCL-Kernel gestartet wird, kann dieser kontinuierlich mit genügend Daten versorgt werden. Der TaskTracker benötigt für jeden Job ein `InputFormat`, um die Daten gepuffert und serialisiert auf den Grafikkartenspeicher zu übertragen. Dies gilt analog für das `OutputFormat`. Außerdem muss der TaskTracker die Anzahl der laufenden Tasks verwalten und auf die physikalisch verfügbaren Recheneinheiten der Grafikkarte einschränken. Dieser Ansatz erfordert aber viel Entwicklungsarbeit, da viele Teile von Hadoop erweitert werden müssen.

Dieser Aufwand könnte sich jedoch lohnen. Die aktuelle Entwicklung von Grafikkarten und Manycore-Prozessoren, wie zum Beispiel das Intel Produkt „Knights Corner"[1], zeigt, dass Rechenleistung in Zukunft vermutlich immer häufiger durch Zusatzkarten gesteigert werden wird. Im Bereich der Manycore-Prozessoren ist anzunehmen, dass diese den x86-Befehlssatz nutzen und somit als vollwertige Prozessoren vom Betriebssystem verwendet werden könnten. Folglich muss auch Hadoop nicht erweitert werden. Falls jedoch die Entwicklung von Grafikkarten für das High Performance Computing anhält, müssen weiterhin Schnittstellen wie OpenCL verwendet werden. Eine Unterstützung von OpenCL durch die Hadoop-API, würde die Programmierung von GPU-beschleunigten MapReduce-Jobs erheblich erleichtern. Die Veröffentlichung von OpenCL-Treibern und SDKs von namenhaften Prozessorherstellern bekräftigt das Interesse an diesen Hardwarearchitekturen.

Aber nicht nur Hadoop muss um Funktionalitäten erweitert werden. Die meisten Hadoop-Anwender führen String-basierte Jobs, wie zum Beispiel die Auswertung von Log-Dateien, auf ihren Clustern aus.[2] Dafür ist OpenCL bisher nicht geeignet, da nur der Datentyp `char` unterstützt wird und keine Methoden zur String-Verarbeitung verfügbar sind. Das Projekt „Mars" hat für ihre Untersuchung eine eigene String-Bibliothek für Grafikkarten implementiert [HFL+08]. An dieser Stelle sollte der offizielle Standard erweitert werden, um das Einsatzgebiet von OpenCL zu erweitern. Das Projekt „Aparapi"[3] von AMD hat das Ziel einer „nahtlosen" Einbindung von OpenCL in Java. So soll der Java-Entwickler den OpenCL-Kernel in Java schreiben können und der explizite Datentransfer zwischen Haupt- und Grafikkartespeicher soll implizit vom Compiler bzw. der Laufzeitumgebung verwaltet werden. Die API von Aparapi würde die Verwendung

[1] `http://www.intel.com/content/www/de/de/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html`

[2] `http://wiki.apache.org/hadoop/PoweredBy`

[3] `http://developer.amd.com/zones/java/aparapi`

der GPU in Hadoop stark vereinfachen, unabhängig davon ob die Grafikkarte über den TaskTracker oder direkt im Task genutzt wird.

Abbildungsverzeichnis

Algorithmenverzeichnis

Abkürzungsverzeichnis

Amazon EC2	Amazon Elastic Compute Cloud
API	Application Programming Interface
COMA	Cache Only Memory Access
COW	Cluster Of Workstations
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DFS	Distributed File System
GC	Garbage Collection
GPGPU	General Purpose Computation on Graphics Processing Unit
GPU	Graphics Processing Unit
HA	High Availability
HDFS	Hadoop Distributed File System
HPC	High Performance Computing
JAR	Java Archive
JNA	Java Native Access
JNI	Java Native Interface
JVM	Java Virtual Machine
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
MPI	Message Passing Interface
MPP	Massively Parallel Processors
NoSQL	Not only SQL
NUMA	Non-Uniform Memory Access
OpenCL	Open Computing Language
POSIX	Portable Operating System Interface
PVM	Parallel Virtual Machine
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data

SMP	Symmetric Multiprocessor
SPMD	Single Program Multiple Data
SQL	Structured Query Language
SSE	Streaming SIMD Extensions
SSI	Single System Image
UMA	Uniform Memory Access
UML	Unified Modeling Language

Literaturverzeichnis

[BM06] Bauke, Heiko ; Mertens, Stephan: *Cluster Computing - Praktische Einführung in das Hochleistungsrechnen auf Linux-Clustern.* Springer, 2006

[BSP10] Bannerman, Marcus ; Strobl, Severin ; Pöschel, Thorsten: *Supercomputing on Graphics Cards - An Introduction to OpenCL and the C++ Bindings: Part 6.* 2010

[CF90] Clauss, Matthias ; Fischer, Günther: *Programmieren mit C.* VEB Verlag Technik Berlin, 1990

[DE03] Dobner, Hans-Jürgen ; Engelmann, Bernd: *Analysis 2 - Integralrechnung und mehrdimensionale Analysis.* Fachbuchverlag Leipzig, 2003

[FRCC08] Farivar, Reza ; Rebolledo, Daniel ; Chan, Ellick ; Campbell, Roy: *A Parallel Implementation of K-Means Clustering on GPUs*, 2008

[GO96] Golub, Gene ; Ortega, James M.: *Scientific Computing.* Teubner Verlag, 1996

[HÖ8] Hülsbömer, Simon: Der x86-Prozessor wird 30 - wie Intel dank IBM alle Gipfel stürmte. In: *Computerwoche* (2008), Juni. `http://www.computerwoche.de/hardware/notebook-pc/1866928/index8.html`. – Abgerufen: 16.03.2011

[HFL+08] He, Bingsheng ; Fang, Wenbin ; Luo, Qiong ; Govindaraju, Naga K. ; Wang, Tuyong: *Mars: A MapReduce Framework on Graphics Processors*, 2008

[Lam10] Lam, Chuck: *Hadoop in Action.* Manning Publications, 2010

[Mun10] Munshi, Aaftab ; Khronos OpenCL Working Group (Hrsg.): *The OpenCL Specification - Version: 1.1.* Khronos OpenCL Working Group, September 2010

[Pie10] Pieloth, Christof: *Multiprozessor-Systeme & -Programmierung - Aufgabe 3: Numerische Integration*, Juni 2010

[Pie11] Pieloth, Christof: *Masterprojekt - Untersuchung der OpenCL-Programmierplattform zur Nutzung für rechenintensive Algorithmen*, August 2011

[Pol89] Polze, Christoph: *UNIX-Werkzeuge zur Programmentwicklung.* VEB Verlag Technik Berlin, 1989

[RR07] Rauber, Thomas ; Rünger, Gudula: *Parallele Programmierung.* Springer, 2007

[RRP+07] Ranger, Colby ; Raghuraman, Ramanan ; Penmetsa, Arun ; Bradski, Gary ; Kozyrakis, Christos ; Stanford University (Hrsg.): *Phoenix: Evaluating MapReduce for Multi-core and Multiprocessor Systems.* Stanford University, 2007

[Sch10] Schüle, Josef: *Paralleles Rechnen*. Oldenbourg Wissenschaftsverlag GmbH, 2010

[SO11] Stuart, Jeff A. ; Owens, John D. ; University of California (Hrsg.): *Multi-GPU MapReduce on GPU Clusters*. University of California, 2011

[SSM] Shirahata, Koichi ; Sato, Hitoshi ; Matsuoka, Satoshi ; Tokyo Institute of Technology, Japan Science and technology Agency, National Institute of informatics (Hrsg.): *Hybrid Map Task Scheduling on GPU-based Heterogeneous Clusters*. Tokyo Institute of Technology, Japan Science and technology Agency, National Institute of informatics

[TG99] Tanenbaum, Andrew S. ; Goodman, James: *Computerarchitektur*. Prentice Hall, 1999

[ZG09] Zechner, Mario ; Granitzer, Michael ; Know-Center (Hrsg.): *Accelerating K-Means on the Graphics Processor via CUDA*. Know-Center, 2009

A. Testsysteme

Attribut	Desktop-PC	HPC mit Amazon EC2
Kurzbezeichnung	PC	EC2
Betriebssystem	Linux (x86_64)	Linux (x86_64)
Kernel-Version	2.6.37.6-0.7-desktop	2.6.32.46-0.3-default
Prozessor (CPU)	AMD Phenom II X4 955	2x Intel Xeon CPU X5570
Kerne pro CPU	4	4 + 4 (Hyperthreading)
Taktfrequenz (CPU)	3,2 GHz	2,93 GHz
Hauptspeicher	4 GB DDR3-1333	22 GB (Typ unbekannt)
Grafikkarte (GPU)	AMD Radeon HD 5850	2x NVIDIA Tesla M2050
Leistungsfähigkeit 64-Bit (GPU)	418 GFLOPS	515 GFLOPS
Leistungsfähigkeit 32-Bit (GPU)	2,09 TFLOPS	1,03 TFLOPS
Speicher (GPU)	1 GB	3 GB
Stream-Prozessoren (GPU)	1440	448
OpenCL Compute Units	18	14
OpenCL-Version	1.1	1.0
OpenCL-Treiber	CAL 1.4.1523	270.41.06
Hadoop-Version	0.20.2	0.20.2

B. Inhalt der CD-ROM

`Abbildungen\`	Abbildungen aus der Masterarbeit
`Binärdateien\`	Externe Bibliotheken und Programme
`Internetnachweise\`	Kopien verwendeter Internetnachweise
`Literatur\`	Digital verfügbare Literatur
`Messergebnisse\`	Messergebnisse der Laufzeitmessungen
`Quelltext\`	Quelltext der Implementierungen
`Masterarbeit.pdf`	Masterarbeit als digitales Dokument im PDF-Format

Eidesstattliche Versicherung

Ich versichere, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen oder anderen Quellen entnommen sind, sind als solche eindeutig kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form noch nicht veröffentlicht und noch keiner Prüfungsbehörde vorgelegt worden.

Ort, Datum	Unterschrift