Universität zu Köln

Institut für Linguistik

Sprachliche Informationsverarbeitung

Bachelorarbeit

im Fach Informationsverarbeitung

Parallelisierung des k-Means-Algorithmus zum Clustern von Dokumenten

vorgelegt von

Alena Tabea Geduldig

Inhalt

1 Einleitung 1

1.1 Text Mining – Vom Text zur Information 1

1.2 Document Clustering 1

1.3 General Purpose Computation in Graphics Processing Units 1

1.4 Ziele der Arbeit 2

2 Clusteranalyse 3

2.1 Document Clustering 4

2.1.1 Dokumentvektoren und tf-idf-Werte 4

2.1.2 Ähnlichkeitsmaße für Vektoren 5

2.2 Der K-Means-Algorithmus 6

2.3 Das Laufzeitproblem 8

2.4 Zusammenfassung und Überleitung 9

3 General Purpose Computation on Graphics Processing Units 10

3.1 Motivation – *The Concurrency Revolution* 11

3.2 CPU und GPU: Rechnerarchitekturen im Vergleich 12

3.3 GPGPU in der Praxis 13

3.4 Die GPGPU Schnittstelle OpenCL 15

3.4.1 Das Plattformmodell 15

3.4.2 Das Ausführungsmodell 16

3.4.3 Das Speichermodell 16

3.4.4 Das Programmiermodell 17

3.5 Zusammenfassung und Überleitung 18

4 Document Clustering auf der GPU 19

4.1 Implementation 20

4.1.1 Parallelisierung der Ähnlichkeitsberechnungen 20

4.1.2 Speicherorganisation 22
4.1.3 Parallelisierung der Zentrumsberechnungen 23
4.1.4 Zusammenfassung 25
4.2 Performancevergleich 26
5 Fazit und Ausblick 31
Literaturverzeichnis 33

1 Einleitung

1.1 Text Mining – Vom Text zur Information

Geschriebene Sprache ist in unserem Kulturkreis schon seit tausenden von Jahren das Medium in dem Wissen festgehalten, bearbeitet und weitergegeben wird. Der Text ist somit ein bedeutender Wissensrohstoff, der mit der Entwicklung des Internets in großer und stetig wachsender Menge verfügbar ist. Aus der Vielzahl an digitalen Texten Wissen zu extrahieren und nutzbar zu machen, ist das Ziel von Text Mining Verfahren. Eine Herausforderung stellt hierbei die unstrukturierte Form der Texte dar. Anders als strukturierte Daten einer Datenbank, stellen Texte un- beziehungsweise semistrukturierte Daten dar, die eine umfangreiche Aufbereitung erfordern.

1.2 Document Clustering

Document Clustering ist ein Text Mining Verfahren, das Texte auf Grundlage ihrer inhaltlichen Zusammengehörigkeit strukturiert. Das Ziel liegt darin, thematisch eng verwandte Texte zu Gruppen zusammenzufassen und damit gleichzeitig von inhaltlich entfernten Texten abzugrenzen. Das Clustern von Textdokumenten ist nur eine von vielen Anwendungsmöglichkeiten einer Clusteranalyse. Im Allgemeinen findet die Clusteranalyse zur Strukturierung ganz unterschiedlicher Objekte in verschiedenen Disziplinen Anwendung. Doch auch wenn eine Clusteranalyse auf unterschiedlichen Objekten durchgeführt wird, bleiben die zu Grunde liegenden Clustering-Algorithmen in ihrer Funktionalität dieselben. Ein sehr gebräuchlicher Clustering-Algorithmus ist der K-Means-Algorithmus. Gegenüber anderen Verfahren zeichnet er sich durch seine vergleichsweise unkomplizierte und schnelle Vorgehensweise aus. Dennoch bleibt das Clustern – speziell von Textdokumenten – ein äußerst komplexes und zeitintensives Verfahren. Dies liegt daran, dass Textdokumente, auch nach angemessener Aufbereitung, durch sehr umfangreiche Daten repräsentiert werden, die erheblichen Einfluss auf die Laufzeit eines Clustering-Algorithmus haben.

1.3 General Purpose Computation in Graphics Processing Units

Vor dem Hintergrund der Laufzeitproblematik einer Clusteranalyse wird in dieser Arbeit ein Programmierverfahren vorgestellt, das erst durch die Hardwareentwicklung der letzten Jahre möglich wurde: General Purpose Computation on Graphics Processing Units (GPGPU). Die Graphics Processing Unit (GPU) bezeichnet die zentrale Recheneinheit auf einer Grafikkarte. Ihre Aufgabe ist es, die zentrale Recheneinheit eines Computers (CPU) durch die Übernahme aufwändiger Grafikberechnungen zu entlasten. In den letzten Jahren begannen Entwickler, Grafikprozessoren auch für allgemeine nicht-grafische Zwecke zu verwenden. Den Anlass dazu

gaben insbesondere zwei Entwicklungen: Erstens stockte die Entwicklung der Performanceleistung von CPUs. Dies lag daran, dass eine weitere massive Erhöhung der Taktfrequenzen der CPUs technisch nicht mehr realisierbar war. Zweitens entwickelten sich die GPUs zeitgleich zu immer aufwändigeren Parallelprozessoren, die die CPU bald in ihrer Rechenpower überholt hatten. Zu verdanken ist dies ihrer zielgerichteten Optimierung auf Grafikberechnungen, die insbesondere durch die Computerspielindustrie vorangetrieben wurde. Das Ziel von GPGPU ist es, Programmabläufe durch die gezielte Nutzung der parallelen Rechenpower der GPU zu beschleunigen

1.4 Ziele der Arbeit

Diese Arbeit befasst sich mit der Fragestellung, ob auch das Clustern von Textdokumenten von diesem Verfahren profitieren und in Zukunft effizienter umgesetzt werden kann. Genauer geht es um die Frage, ob der K-Means-Algorithmus an die parallele Verarbeitungsweise und die spezielle Architektur moderner Grafikprozessoren so anpassbar ist, dass Textdokumente dadurch schneller geclustert werden können als auf herkömmliche Weise auf der CPU.

Um diese Frage beantworten zu können, werden in Kapitel 2 zunächst die Grundlagen der Clusteranalyse im Allgemeinen, sowie die des Document Clusterings im Speziellen erläutert. Hierbei wird auch deutlich werden, warum gerade das Clustern von Textdokumenten enorm hohe Laufzeiten zur Folge hat. In Kapitel 3 wird die GPU Programmierung beleuchtet. Nachdem ein Überblick über dessen technische Grundlagen und Möglichkeiten gegeben wurde, wird die GPGPU-Schnittstelle OpenCL vorgestellt, ein plattformunabhängiges Framework zum Erstellen GPU-gestützter Anwendungen. In Kapitel 4 werden die zuvor erläuterten Themen schließlich zusammengeführt. Es wird eine Strategie entwickelt, wie ein GPU-basierter K-Means-Algorithmus zum Clustern von Textdokumenten am effizientesten realisiert werden kann. Diese Strategie wird schließlich durch eine eigene Implementierung mit OpenCL umgesetzt. Im letzten Kapitel wird überprüft, ob die OpenCL-basierte K-Means-Variante tatsächlich ein effizienteres Document Clustering ermöglicht. Um die zentrale Fragestellung abschließend beantworten zu können, wird ein direkter Performancevergleich des GPU-gestützten K-Means Algorithmus mit einer herkömmlichen auf der CPU ausgeführten Variante durchgeführt.

2 Clusteranalyse

Die Clusteranalyse ist ein Verfahren zur Ermittlung von Ähnlichkeitsstrukturen innerhalb großer unstrukturierter Datensätze. Ein Clustering-Algorithmus teilt eine große ungeordnete Menge gleichartiger Objekte in verschiedene Teilmengen (Cluster) auf. Die Bildung solcher Cluster wird von zwei Prinzipien bestimmt (Bacher 1994:2):

1. Homogenität innerhalb der Cluster: Die Objekte desselben Clusters zeichnen sich durch eine hohe Ähnlichkeit zueinander aus.
2. Heterogenität zwischen den Clustern: Objekte unterschiedlicher Cluster zeichnen sich durch eine hohe Unähnlichkeit zueinander aus.

Eine Bewertung der Ähnlichkeit beziehungsweise Unähnlichkeit zweier Cluster kann nur auf der Grundlage von Merkmalen erfolgen. Sie werden vor einer Clusteranalyse klar definiert und müssen, um maschinell interpretiert werden zu können, numerisch ausdrückbar sein. Sind diese Voraussetzungen erfüllt, kann jedes Objekt in eine geeignete und allein auf die definierten Merkmale reduzierte Repräsentation überführt werden. Jedes Objekt wird dafür in einen numerischen Vektor überführt, dessen Dimension durch die Anzahl der Merkmale festgelegt ist. Jeder Eintrag eines Objektvektors steht dann für die Ausprägung des entsprechenden Merkmals.

Folgendes Beispiel soll diesen Vorgang verdeutlichen: Um Bauklötze zu clustern, können die Merkmale *Form* und *Gewicht* verwendet werden. Um das Merkmal *Form* eines Bauklotzes numerisch auszudrücken, kann man es mit der Anzahl seiner Ecken übersetzen. Um für einen Clustering-Algorithmus lesbar zu sein, kann dann jeder Bauklotz durch einen zweidimensionalen Vektor repräsentiert werden, dessen erster Eintrag der Anzahl seiner Ecken und zweiter Eintrag seinem Gewicht in Gramm entspricht. Mit Hilfe eines Ähnlichkeitsmaßes für Vektoren, wie sie die Mathematik in größerer Zahl bereitstellt, kann die Ähnlichkeit zweier Bauklötze – bezüglich der beiden Merkmale – maschinell berechnet werden. Sie entspricht der mathematischen Nähe ihrer Vektorrepräsentationen.

Die Speicherung der Objekte als Vektorrepräsentationen ist Voraussetzung zur Anwendung eines Clustering-Algorithmus. Auf Basis eines Ähnlichkeitsmaßes wird in einem iterativen Verfahren die finale Clusterstruktur erstellt und aus einer heterogenen Menge von Vektoren homogene Teilmengen generiert. Welche Objekte diese Vektoren repräsentieren spielt für den Algorithmus keine Rolle, sondern wird erst bei der Interpretation der Ergebnisse relevant.

Ein realitätsnäherer Anwendungsfall der Clusteranalyse ist ihr Gebrauch in der Computerlinguistik. Das Clustern großer Textkorpora hinsichtlich ihrer thematischen Inhalte wird auch Document Clustering genannt.

2.1 Document Clustering

Das Ziel von Document Clustering ist es, inhaltliche Beziehungen von Einzeltexten innerhalb großer Textsammlungen aufzudecken. Als optimales Ergebnis wird eine Clustereinteilung erreicht, bei der sich Texte, die demselben Cluster zugeteilt wurden, durch eine enge thematische Verwandtschaft auszeichnen. Gleichzeitig unterscheidet sie ihre inhaltliche Gemeinsamkeit von den Texten anderer Cluster. Im Gegensatz zur Textklassifizierung, wo Texte bereits zuvor bekannten Kategorien zugeordnet werden, erfordert eine Clusteranalyse kein Vorwissen über die Texte. Vielmehr werden bestehende Kategorien erst durch eine Clusteranalyse aufgedeckt. Für die Beurteilung der thematischen Nähe von Texten, dient dem menschlichen Leser seine Intuition. Um diese Aufgabe maschinell und in großer Zahl zu bewältigen, steht der Computerlinguist vor der Herausforderung eine verarbeitbare Form des Merkmals *thematischer Inhalt* zu formulieren. Auch Textdokumente müssen, um für einen Clustering-Algorithmus verwertbar zu sein, in Vektoren überführt werden. Entscheidend ist, dass durch diesen Vorgang die thematischen Beziehungen der Dokumente erhalten bleiben, so dass die mathematische Nähe zweier Vektoren mit der semantischen Nähe ihrer Dokumente übersetzt werden kann. Wie solche Dokumentvektoren generiert werden, wird im folgenden Abschnitt erläutert.

2.1.1 Dokumentvektoren und tf-idf-Werte

Da Dokumentvektoren Texte auf ihren thematischen Inhalt reduzieren sollen, werden sie auf Grundlage der im Text enthaltenen Wörter generiert(vgl. zu den folgenden Ausführungen auch Heyer et al.2008: 201ff).Dieser Ansatz baut auf der Voraussetzung auf, dass sich die Thematik eines Textes aus seinen Wörtern ergibt und von jedem Wort zu einem bestimmten Teil mit geformt wird. [1] Wörter werden als Merkmale aufgefasst, die in verschiedenen Texten unterschiedlich stark, oder gar nicht, ausgeprägt sind. Jedes Wort hat dann eine bestimmte Aussagekraft beziehungsweise Wichtigkeit für ein Dokument. Diese werden durch einen numerischen Wert, den tf-idf-Wert ausgedrückt und bilden die Einträge der Dokumentvektoren. Den Ausgangspunkt für die Berechnung der tf-idf-Werte bilden die folgenden Annahmen:

[1] Dem entgegen steht das Kompositionalitätsprinzip des Logikers Gottlob Frege. Es betont, dass sich die Bedeutung eines Satzes nicht nur aus seinen Wörtern, sondern auch aus der Art ihrer Zusammensetzung ergibt. Da Document Clustering jedoch nicht den Anspruch hat, zwischen der genauen Semantik der Texte zu differenzieren, sondern an übergeordneten Themen interessiert ist, entsteht hier kein Widerspruch zum Kompositionalitätsprinzip.

1. Die Termfrequenz (tf) eines Wortes zu einen bestimmten Dokument beschreibt seine Häufigkeit in diesem Dokument. Je höher die Termfrequenz eines Wortes ist (je öfter es im Dokument verwendet wird), desto mehr Aussagekraft besitzt es über diesen Text.
2. Die Dokumentfrequenz (df) eines Wortes entspricht der Anzahl der Dokumente, die es mindestens einmal enthalten. Je niedriger die Dokumentfrequenz eines Wortes ist, desto aussagekräftiger ist es für die Texte die es enthalten.

Aus einer Kombination aus Term- und Dokumentfrequenz wird jedem Wort ein tf-idf-Wert zugewiesen, der seine Wichtigkeit für ein Dokument beschreibt. Er berechnet sich wie folgt:

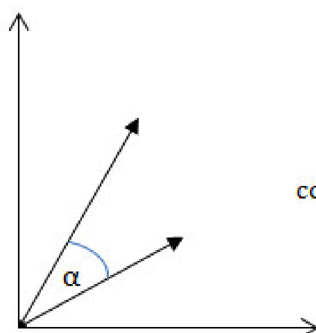
$$tfidf(t,d) = \frac{tf}{maxtf} \, log \frac{|d|}{df}$$

maxtf bezeichnet die Termfrequenz des im Dokument am stärksten vertretenen Wortes. |d| steht für die Anzahl aller Dokumente im Korpus. Ein Wort gilt als umso aussagekräftiger für den thematischen Inhalt eines Textes, je höher sein tf-idf-Wert ist. Ist ein Wort gar nicht in einem Dokument vertreten, erhält es der Formel entsprechend den tf-idf-Wert 0. Dokumentvektoren setzen sich nun wie folgt zusammen: Jedes im Gesamtvokabular des Korpus enthaltene Wort steht für eine Dimension des Vektorraumes. Die einzelnen Vektoreinträge entsprechen dann dem tf-idf-Wert dieses Wortes für das entsprechende Dokument.

2.1.2 Ähnlichkeitsmaße für Vektoren

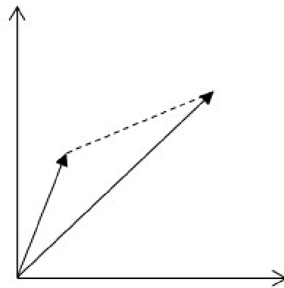
Grund für die Überführung der Clusterobjekte in numerische Vektoren ist, dass die Ähnlichkeit von Vektoren mathematisch exakt berechnet werden kann und somit indirekt auch die thematische Ähnlichkeit der Dokumente. Es existieren eine Reihe verschiedener Ähnlichkeitsmaße für Vektoren, die unterschiedliche Kriterien berücksichtigen. Ein für das Clustern von Textdokumenten sehr bewährtes Ähnlichkeitsmaß ist die Kosinus-Ähnlichkeit von Vektoren. (vgl. hierzu auch Huang 2008 und Strehl et al. 2000)

Das Cosinus Maß definiert die Ähnlichkeit zweier Vektoren durch die Größe des zwischen ihnen eingeschlossenen Winkels:



$$\cos(\vec{a},\vec{b}) = \cos(\alpha) = \frac{\sum_{i=1}^{n} a_i b_i}{\sqrt{\sum_{i=1}^{n} a_i^2}\sqrt{\sum_{i=1}^{n} b_i^2}}$$

Entscheidend für die Ähnlichkeit zweier Vektoren ist also nur ihre Richtung. Für Dokumentvektoren bedeutet dies, dass nur die Wortverhältnisse berücksichtigt werden und nicht ihre tatsächlichen Häufigkeiten. Im Gegensatz dazu steht die euklidische Distanz, ein Distanzmaß, das nicht nur die Richtung, sondern auch die Länge der Vektoren berücksichtigt. Im zweidimensionalen Fall entspricht der euklidische Abstand zweier Vektoren genau dem Abstand der zugehörigen Punkte im Koordinatensystem.



$$dist(\vec{a}, \vec{b}) = \sqrt{\sum_{i=1}^{n} (a_i - b_i)^2}$$

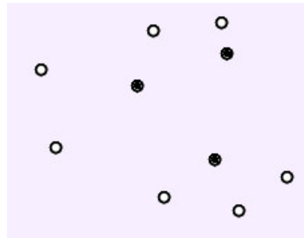
2.2 Der K-Means-Algorithmus

Nachdem alle Dokumente in Dokumentvektoren überführt wurden, erfolgt die eigentliche Clusteranalyse durch einen Clustering-Algorithmus. Ein sehr populärer Clustering-Algorithmus ist der K-Means Algorithmus. Er zeichnet sich unter anderem dadurch aus, dass die Anzahl der zu bildenden Cluster im Voraus manuell festgelegt werden muss. Diese Eigenschaft widerspricht zwar in gewisser Weise dem Anspruch, dass kein Vorwissen über den Datensatz erforderlich ist, rechtfertigt sich aber durch den großen Performance- und Speicherplatzvorteil gegenüber anderen Verfahren.[2] Des Weiteren zählt der K-Means Algorithmus zu den harten Clustering Verfahren. Im Gegensatz zu weichen Verfahren wird jeder Vektor genau einem Cluster zugeordnet. Weiche Verfahren erlauben hingegen, durch die Angabe von Wahrscheinlichkeiten, eine Zuordnung zu mehreren Clustern. Dies ermöglicht mehr Flexibilität im Umgang mit schwer einzuordnenden Vektoren. Gleichzeitig erschwert es aber auch den Umgang mit den wesentlich komplexeren und nicht eindeutigen Ergebnissen.

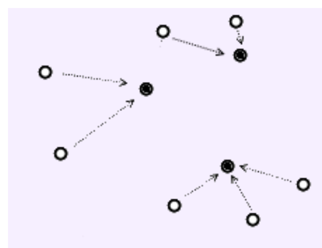
Die Einteilung der Vektoren in eine Clusterstruktur erfolgt beim K-Means-Algorithmus iterativ und baut auf einer zufälligen initialen Clustereinteilung auf, die sukzessive verbessert wird. Die Bestimmung der Cluster erfolgt in drei Schritten (vgl. zu den folgenden Ausführungen auch McQueen 1967):

[2] In der Regel werden verschiedene Konfigurationen getestet und die beste durch Clusterevaluationen ermittelt. (Zur Clusterevaluation siehe z.B. Manning et al. 2008: Kap. 16.3)

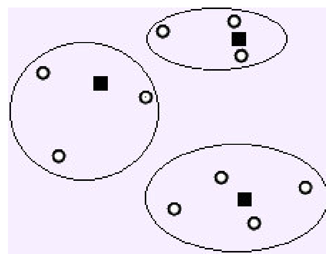
1. Initialisierungsphase: Zu Beginn werden zufällig k Vektoren ausgewählt, die jeweils zum Zentrum eines der k Cluster bestimmt werden.



2. Zuordnungsphase: Auf der Basis eines Ähnlichkeitsmaßes wird jeder Vektor dem Cluster zugeordnet, dessen Zentrum er am nächsten liegt.



3. Aktualisierungsphase: Aus den Mittelwerten aller Vektoren eines Clusters werden neue Clusterzentren berechnet. (Die Clusterzentren bilden somit nicht mehr konkrete Dokumente des Korpus, sondern hypothetische Dokumentvektoren, die als Prototypen der Cluster beschrieben werden können)



Die Schritte 2 und 3 werden so oft wiederholt, bis sich die Clustereinteilung nicht mehr verändert. Auf diese Weise wird die Clustereinteilung mit jedem Schritt optimiert.

Ein Nachteil bei diesem Verfahren ist, dass das endgültige Ergebnis in Abhängigkeit von den Startzentren variieren kann. In der Praxis werden daher oft mehrere Durchläufe des Algorithmus mit verschiedenen Startzentren ausgeführt. Anhand von Clusterevaluationsverfahren wird dann das Ergebnis ausgewählt, das die Anforderungen an eine Clusteranalyse(Homogenität innerhalb

und Heterogenität zwischen den Clustern) am besten erfüllt.[3] Dies bedeutet aber auch, dass sich der Zeitfaktor der Clusteranalyse vervielfacht. Welche Konsequenz dies hat, zeigt sich im folgenden Abschnitt, in dem das Laufzeitverhalten von einem einzelnen Durchlauf des K-Means-Algorithmus analysiert wird.

2.3 Das Laufzeitproblem

Die Laufzeit des K-Means-Algorithmus hängt stark von der Anzahl der zu clusternden Vektoren ab. Der Initialisierungsschritt wird nur einmalig ausgeführt und hat daher kaum Einfluss auf die Laufzeit. Für die Zuordnung der Vektoren zu ihrem nächsten Clusterzentrum müssen jedoch alle paarweisen Ähnlichkeiten berechnet werden. Erst wenn der Abstand zu allen Clusterzentren bekannt ist, kann ein Vektor mit Sicherheit einem Cluster zugeordnet werden. Diese Phase hat somit eine Laufzeit von $n*k*t_{dist}$, wobei n die Anzahl der Vektoren, k die Clusteranzahl und t_{dist} die benötigte Zeit für eine Ähnlichkeitsberechnung angibt. Für die Neuberechnung der Clusterzentren müssen alle Vektoren eines Clusters aufsummiert werden, damit ihr Mittelwert gebildet werden kann. Auch hier ist folglich eine Schleife über sämtliche Vektoren notwendig Die Anzahl der Vektoren hat folglich erheblichen Einfluss auf die Laufzeit beider Iterationsphasen unabhängig davon, ob Textdokumente oder andere Objekte geclustert werden. Dass aber auch die Art der Objekte große Auswirkungen hat, zeigt sich, wenn man den Faktor t_{dist} näher betrachtet. Am Beispiel des Cosinus Maß wird dies deutlich:

$$\mathrm{sim(v,c)} = \sum_{i=1}^{d} v_i c_i \frac{1}{\sqrt{\sum_{i=1}^{d} v_i^2}\sqrt{\sum_{i=1}^{d} c_i^2}}$$

Jeder Dimensionseintrag muss einzeln betrachtet und summiert werden. Die Laufzeigt steigt also ebenfalls mit der Dimension der Objektvektoren. Auch die Zeit für die Neuberechnung der Zentren muss auf den zweiten Blick durch eine Schleife über alle Dimensionen ergänzt werden. Zur Addition zweier Vektoren muss auf jeder Dimension einzeln summiert werden.

$$\vec{v} + \vec{c} = \begin{matrix} v[1] + c[1] \\ \vdots \\ v[n] + c[n] \end{matrix}$$

Die tatsächliche Laufzeit einer Clusteranalyse hängt also stark vom konkreten Anwendungsfall ab und kann bei sonst gleichbleibenden Werten stark variieren. Im Fall des Document Clusterings zeigt sich nun auch der Grund für die Zeitintensität des Verfahrens: Hier steht jedes Wort des

[3] Gemessen wird die Erfüllung dieser Anforderungen zum Beispiel mit dem Dunn-Index oder dem Davies-Bouldin-Index. (Beide z. B. angewandt in Maulik 2002)

gesamten Vokabulars für eine weitere Dimension der Dokumentvektoren. Zwar kann das Vokabular durch die Entfernung von Stoppwörtern[4] und der Durchführung eines Stemmings[5] im Voraus deutlich reduziert werden; dennoch sind fünf- oder auch sechsstellige Vektordimensionen nicht ungewöhnlich. Zur Anschauung wurde eine Merkmalsreduzierung an einem Korpus von 89 Nachrichtenartikeln der Webseite Spiegel-Online [6] getestet. Aus ursprünglich 83.887 verschiedenen Worttypen verblieben nach Entfernung von Stoppwörter und Stemming noch 11.502 Terme.

2.4 Zusammenfassung und Überleitung

Durch eine Clusteranalyse können Ähnlichkeitsstrukturen innerhalb großer manuell nicht überschaubarer Datensätze unbekannter Ordnung aufgedeckt werden. Durch die Überführung von Textdokumenten in Dokumentvektoren eröffnet sich die Möglichkeit, auch natürlich sprachliche Texte zu clustern. Die Vorrausetzung für eine am thematischen Inhalt der Dokumente orientierten Clusteranalyse ist, dass die thematische Nähe zweier Dokumente auch nach der Überführung in Dokumentvektoren rekonstruierbar bleibt. Auf Basis der Term- und Dokumentfrequenzen einzelner Wörter werden Dokumentvektoren so generiert, dass ihre mathematisch berechenbare Nähe mit der inhaltlichen Nähe ihrer Dokumente gleichgesetzt werden kann. Unter Verwendung eines Distanzmaßes für Vektoren ermittelt der K-Means-Algorithmus Gruppen ähnlicher Vektoren/Dokumente. Da die Dimension von Dokumentvektoren sehr groß werden kann (sie entspricht der Größe des Gesamtvokabulars), ist Document Clustering in der Praxis ein äußerst zeitintensives Verfahren.

Im folgenden Kapitel wird ein Programmierverfahren vorgestellt, das zur Beschleunigung rechenintensiver Programmabläufe genutzt wird und möglicherweise auch die Laufzeit des K-Means Algorithmus reduzieren kann. Dieses Verfahren nennt sich General Purpose Computation on Graphics Processing Units (GPGPU) und hat das Ziel Programmabläufe durch die Einbeziehung von Grafikprozessoren zu beschleunigen.

---

[4] Inhaltslose Funktionswörter; z.B. Konjunktionen und Artikel

[5] Abbildung von Wortformen auf ihre Wortstämme

[6] Link der Webseite: http://www.spiegel.de, zuletzt aufgerufen: 17.06.2014

3 General Purpose Computation on Graphics Processing Units

GPGPU bezeichnet die Verwendung von Grafikprozessoren für allgemeine nicht-grafische Berechnungen und somit für Zwecke, die über ihren ursprünglichen Aufgabenbereich hinausgehen. Die eigentliche Funktion einer Grafikkarte besteht in der Berechnung der Bildschirmausgabe, um die CPU von aufwändigen Grafikberechnungen zu entlasten. Ihre Leistung ist in der Hauptsache für grafisch anspruchsvolle Computerspiele und 3D-Animationen relevant. In den letzten Jahren entwickelte sich jedoch ein Trend, die Grafikkarte universeller einzusetzen und auch für allgemeine rechenintensive Aufgaben zu gebrauchen. Im Vordergrund steht hierbei stets das Ziel einer Geschwindigkeitssteigerung gegenüber der alleinigen Verwendung der CPU.

Moderne GPUs erreichen heutzutage in bestimmten Aufgabenbereichen deutlich höhere Rechenleistungen als CPUs. Ihre Stärke liegt in ihrer hochgradig parallelen Verarbeitungsweise. Die parallele Rechenleistung von Grafikprozessoren resultiert aus ihrer zielgerichteten Optimierung zur Verarbeitung von Grafikdaten. Während die CPU für einen universellen Gebrauch konzipiert ist, sind Grafikprozessoren hoch spezialisiert und können im Zuge dessen effektiver an ihre Aufgaben angepasst und optimiert werden:

> Driven by the demand for real-time three-dimensional graphics rendering, a highly parallel problem, GPUs have evolved rapidly as very powerful, filly programmable, task and data parallel architectures. (Gaster et al. 2013: 4)

Die Entwicklungsarbeit in die Beschleunigung von GPU-Berechnungen ist vor allem der Computerspielindustrie und der stetigen Nachfrage nach grafisch immer aufwändigeren Computerspielen zu verdanken. In der Folge wurden GPUs zu immer leistungsfähigeren Parallelprozessoren und hatten die CPUs bald in ihrer Rechenpower überholt. Erst im Nachhinein entstand die Idee, dieses Potential auch für allgemeinere Berechnungen zu nutzen. Zu diesem Zwecke wurden spezielle Schnittstellen entwickelt, die das freie Programmieren von GPUs möglich machten. 2006 stellte einer der führenden Grafikkartenhersteller Nvidia[7] die Programmierplattform CUDA (Compute Unified Device Architecture)[8] vor. Diese ermöglicht eine freie Programmierung von Nvidia Grafikchips und war die erste bedeutende Bibliothek für GPGPU. Auch andere Hersteller entwickelten proprietäre Plattformen zur Programmierung der

---

[7] http://www.nvidia.de/, zuletzt aufgerufen: 17.06.2014

[8] http://www.nvidia.com/object/cuda_home_new.html, zuletzt aufgerufen: 17.06.2014

eigenen Grafikkarten. 2008 wurde durch den Zusammenschluss einiger Firmen mit OpenCL (Open Compute Unit)[9] der erste plattformunabhängige Standard eingeführt.

3.1 Motivation – *The Concurrency Revolution*

Ein Grund für das wachsende Interesse an der Programmierung von GPUs zeigt sich bei einem direkten Vergleich der Rechenleistungsentwicklung von GPUs und CPUs in den vergangenen Jahren. (Abbildung 3.1)

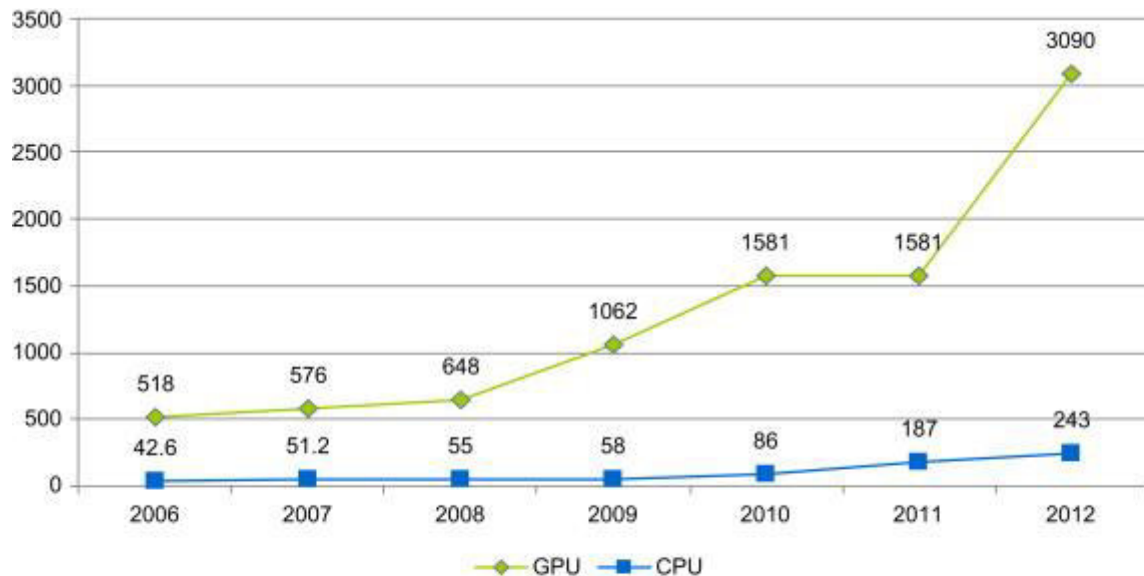


Abbildung 3.1: *Leistungsentwicklung von CPU und GPU in Gigaflops.*(Quelle: Cook 2013:14)

Gemessen in der Anzahl der Gleitkommaoperationen pro Sekunde (Gigaflops), ist die GPU der CPU klar überlegen. Während die Rechenpower der GPUs in den letzten Jahren rasant anstieg, blieb die Rechenleistung der CPUs deutlich zurück. In den 80er und 90er Jahren dominierten Ein-Kern-Prozessoren den PC-Bereich, dessen Rechenleistung durch immer schnellere Taktfrequenzen kontinuierlich erhöht wurde. Softwareentwickler konnten sich darauf verlassen, dass ihre sequentiell ausgerichteten Programme mit jeder neuen Prozessorgeneration automatisch an Performance gewannen. Schwierigkeiten mit der Wärmeableitung sind jedoch ein Grund dafür, dass höhere Prozessortaktungen heute nicht mehr profitabel realisierbar sind. Stattdessen wurde paralleles Rechnen zu einem immer wichtigeren Faktor für performanceorientierte Softwareentwicklung. Mit den Worten „the free lunch is over" betitelt der Softwareentwickler Herb Sutter im Jahr 2005 die Folgen dieser Entwicklung (Sutter 2005) und prägt den Begriff der *Concurrency Revolution* (Sutter & Larus 2005). Hardwareseitig ist durch den Übergang zu mehrkernigen Architekturen auch weiterhin Leistungszuwachs möglich. Dass dieser nicht nur

[9], http://khronos.org/opencl/, zuletzt aufgerufen: 17.06.2014

theoretisch bleibt, sondern effektiv genutzt wird, liegt von nun an aber in der Verantwortung der Entwickler, die ihre Programme explizit an die Änderungen der Hardware anpassen müssen.

> The architectural changes in multicore processors benefit only concurrent applications and therefore have little value for most existing mainstream software. (Sutter & Larus 2005)

Vor dem Hintergrund dieser Entwicklungen zeigt sich GPGPU als ein zukunftsorientiertes und lohnenswertes Entwicklungsfeld. Darüber hinaus sprechen auch andere Gründe für die Entwicklung GPU kompatibler Anwendungen. Leistungsfähige Grafikchips sind nicht nur generell verfügbar (sie gehören heute zur Standardausstattung jedes PCs), sondern auch vergleichsweise preiswert. „Commodity computer graphics are probably today's most powerful computational hardware for the dollar. "(Owens et al. 2007: 21)

3.2 CPU und GPU: Rechnerarchitekturen im Vergleich

Die hohe parallele Rechenleistung moderner GPUs ergibt sich aus ihrer speziellen Architektur. Im Gegensatz zu modernen Multicore-Prozessoren mit bis zu 8 Kernen besitzt eine GPU mehrere hundert Kerne. (Abbildung 3.2) Grafikberechnungen sind von einer hohen arithmetischen Dichte, erfordern aber nur wenig Speicherzugriff oder Steuerlogik. Durch den Verzicht auf Steuerungs- und Cachebausteine ist mehr Chipfläche für zusätzliche Kerne verfügbar. Diese werden in Gruppen zusammengefasst, die sich jeweils einen kleinen Cache und Befehlssatz teilen. Dieses Architekturmodell ermöglicht es, viele grafische Berechnungen zeitgleich statt sequentiell auszuführen. Dies ist jedoch nur möglich, weil die Berechnungen der einzelnen Bildpunkte unabhängig voneinander ausführbar sind. Jeder Kern kann die Berechnung eines einzelnen Bildpunktes übernehmen, ohne mit anderen Kernen kommunizieren zu müssen.

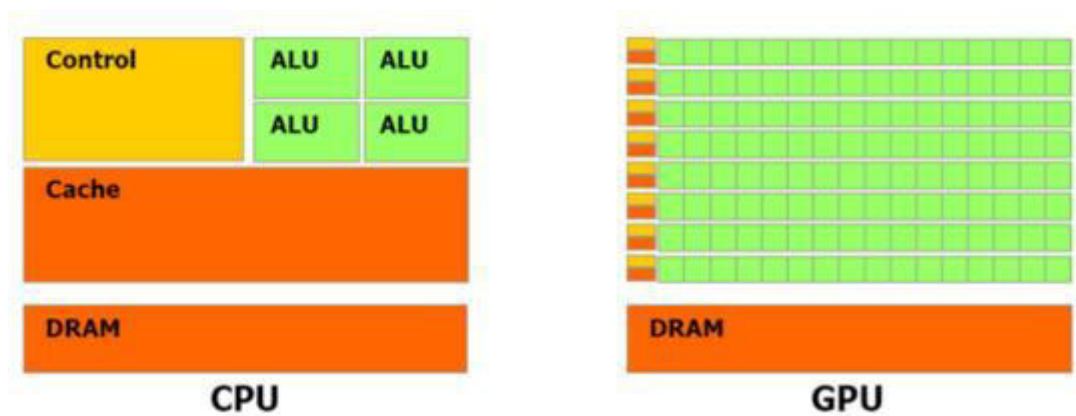


Abbildung 3.2: *Schematische Darstellung von CPU und GPU* (Quelle: Kirk & Hwu 2005: 4)

Die CPU verfügt im Gegensatz zur GPU über einen sehr großen Cache und Befehlssatz. Dies ist auch notwendig, um die vielfältigen, meist sehr komplexen und viel Speicherzugriff erfordernden Aufgaben der CPU schnellstmöglich zu bewältigen. Im Gegensatz dazu besitzt sie nur wenige ALUs (Arithmetisch Logische Einheiten). Diese Architektur ist für die sehr universell angelegte CPU am besten geeignet. Eine Umverteilung der Chipfläche zu Gunsten mehrerer Kerne wäre im Falle der CPU von Nachteil. Diese würden bei dem Großteil ihrer Aufgaben, auf Grund mangelnder Parallelisierbarkeit, ungenutzt bleiben.

3.3 GPGPU in der Praxis

Die vielkernige Architektur der GPU ist optimiert für **datenparalleles Rechnen**. Datenparallel bedeutet, dass derselbe Befehlssatz zeitgleich auf unterschiedlichen Daten ausgeführt wird. Ein anschauliches Beispiel für eine datenparallelisierbare Anwendung ist die Addition zweier Vektoren:

$$\vec{a} + \vec{b} = \begin{matrix} a[1] + b[1] \\ \vdots \\ a[n] + b[n] \end{matrix}$$

In einer herkömmlichen sequentiellen CPU-Anwendung erfordert diese Aufgabe eine Schleife über alle Vektordimensionen:

```
for each d
    sum[d] = a[d]+b[d]
end
```

Die Berechnung der einzelnen Summen stehen jedoch in keiner Abhängigkeit zueinander und könnten theoretisch auch zeitgleich berechnet werden, ohne dass das Ergebnis verfälscht würde. Auf der GPU ist genau dies möglich. Jeder Dimensionseintrag kann zeitgleich von einem der GPU Kerne bestimmt werden. Hierfür führt jeder Kern denselben Befehl (a+b) aus, jedoch jeweils auf unterschiedlichen Datenstellen.

Ein Programm, das für die Ausführung auf einer GPU bestimmt ist, nennt sich **Kernel**. Um einen Kernel von einer GPU ausführen zu lassen, muss jede GPGPU-Anwendung aus einer **Host-Applikation**, die auf der CPU läuft, gestartet werden. Die CPU zusammen mit dem Hauptspeicher wird in diesem Zusammenhang auch **Host-System** genannt. Die Host-Applikation initiiert zunächst die GPGPU-Schnittstelle, welche die Kommunikation mit der GPU ermöglicht. Über diese Schnittstelle können, gesteuert aus der Host-Applikation, Daten

vom Host-Speicher an den Speicher der GPU und anders herum transferiert werden. Die Host-Applikation leitet außerdem den Befehlssatz des Kernels an die GPU weiter und erteilt ihr den Befehl zur Ausführung des Kernels. Abbildung 3.3 skizziert den schematischen Aufbau einer GPGPU-Anwendung.

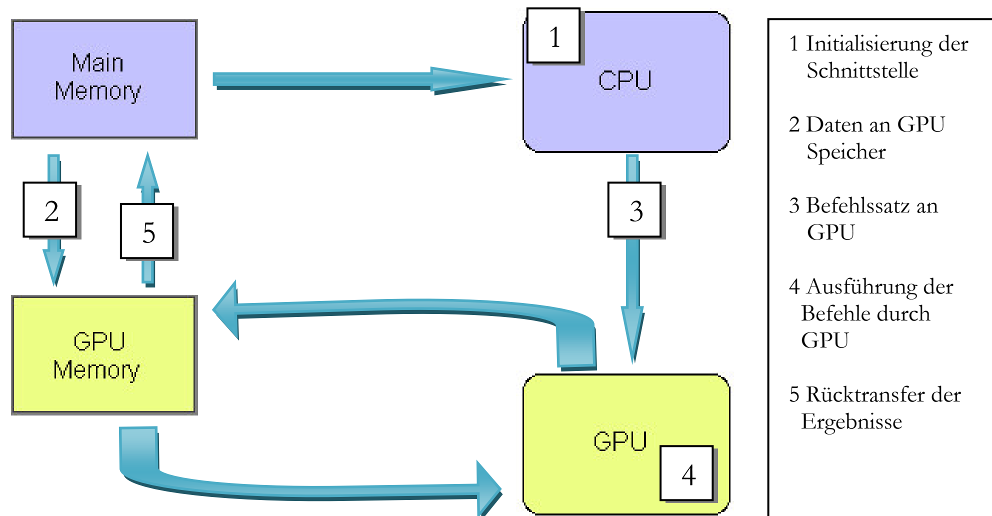


Abbildung 3.3: *Allgemeine Architektur einer GPGPU-Anwendung*

Wird ein Kernel zur Ausführung an die GPU gesendet, wird gleichzeitig eine **Worksize** festgelegt. Sie gibt den Grad der Parallelität an, also wie viele Instanzen des Kernels gebildet werden sollen. Jede Kernelinstanz wird dann auf einem der GPU Kerne ausgeführt. Die verschiedenen Kernelinstanzen sind durch eine KernelID eindeutig identifizierbar. Durch eine Bezugnahme auf diese ID kann der ansonsten identische Code variieren und beispielsweise auf eine bestimmte Datenstelle im Speicherobjekt verweisen. Ein Kernel für die datenparallele Addition zweier Vektoren könnte wie folgt aussehen:

```
vectorSum(float[] A, float[] B, float[] C){
        int id = getKernelID();
        C[id] = A[id] + B[id];
}
```

`getKernelID()` gibt die ID der jeweiligen Kernelinstanz zurück. Sie reicht von 0 bis zur im Host-Programm festgelegten Worksize. Um jeden Eintrag der Vektorsumme A+B zu

berechnen, wird im Beispiel folglich eine Worksize benötigt, die mit der Dimension der Vektoren übereinstimmt. Jede Kernelinstanz ersetzt dann genau einen Schleifendurchlauf der sequentiellen Vektoraddition.

3.4 Die GPGPU Schnittstelle OpenCL

OpenCL ist eine plattformunabhängige Schnittstelle für GPGPU-Anwendungen. Der in OpenCL C[10] geschriebene Kernel kann auf GPUs aller gängigen Grafikkartenhersteller ausgeführt werden. Die ursprüngliche Idee zu OpenCL stammt von Apple und wurde schließlich in einer Zusammenarbeit der Firmen Nvidia, AMD, IBM und Intel entwickelt und 2008 in einer ersten Version vorgelegt (Munshi 2014). Die fortlaufende Entwicklung von OpenCL wird unter der Leitung der Khronos-Group, einem Industriekonsortium von über 100 Firmen, durchgeführt.[11]

Zur Nutzung von OpenCL müssen folgende Voraussetzungen erfüllt werden: Zunächst ist eine Implementierung des Standards im Betriebssystem notwendig. Außerdem wird das Vorhandensein OpenCL kompatibler Hardware vorausgesetzt. Die Grafikkarten der beiden größten Hersteller AMD und Nvidia sind inzwischen alle zu OpenCL kompatibel. Dafür müssen allerdings Treiber für die jeweilige Hardware installiert werden.

Der Vorteil von OpenCL gegenüber proprietären Frameworks wie CUDA von Nvidia liegt in seiner universellen Einsetzbarkeit. Gerade im wissenschaftlichen Bereich und unter dem Aspekt der Reproduzierbarkeit ist dies ein entscheidender Vorteil. Die Plattformunabhängigkeit wird dadurch erreicht, dass kein spezielles Hardwaremodell zu Grunde gelegt wird, sondern ein abstrahiertes Prozessormodell. Dieses Modell schließt auch die Verwendung anderer Prozessorarten nicht aus. So kann zum Beispiel auch eine zweite CPU mit dem Host-System verknüpft werden.

Die Architektur von OpenCL spaltet sich in vier Modelle auf: Das Plattformmodell, das Ausführungsmodell, das Speichermodell und das Programmiermodell.[12]

3.4.1 Das Plattformmodell

Das in Abbildung 3.4 dargestellte Plattformmodell beschreibt die zu Grunde liegende Hardware einer OpenCL Anwendung. Sie besteht aus einem Host (CPU) welcher mit einem oder mehreren Devices (z.B. der GPU) verbunden ist. Ein Device entspricht dem abstrahierten Prozessormodell von OpenCL. Es besteht aus mehreren **Compute Units**, welche wiederum mehrere **Processing Elements** beinhalten. Die Processing Elements sind für die Ausführung der einzelnen

[10] http://www.khronos.org/registry/cl/specs/opencl-2.0-openclc.pdf, zuletzt aufgerufen: 17.06.2014
[11] http://www.khronos.org/, zuletzt aufgerufen am 17.06.2014
[12] Die folgenden Ausführungen wurden der OpenCL 2.0 Spezifikation entnommen. (vgl. Munshi 2014)

Kernelinstanzen zuständig, können also mit den Kernen einer GPU assoziiert werden. Processing Elements derselben Compute Unit teilen sich, wie die Kerne einer GPU, einen gemeinsamen Speicher und Befehlssatz.

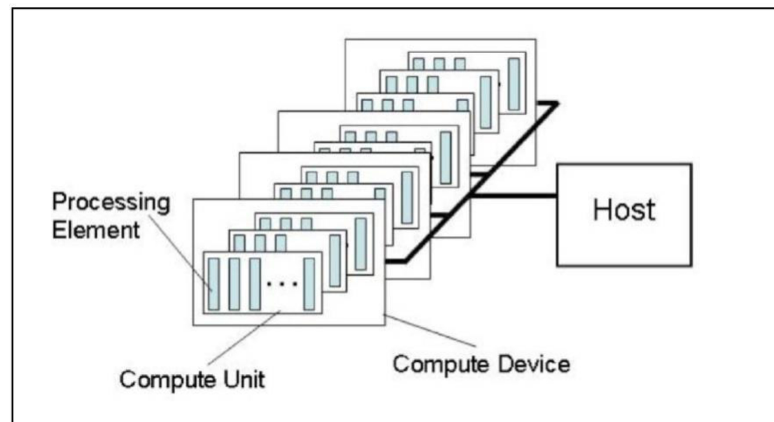


Abbildung 3.4: *Plattformmodell – Ein Host verbunden mit einem oder mehreren Compute Devices* (Quelle: Munshi 2014: 24)

3.4.2 Das Ausführungsmodell

Das Ausführungsmodell beschreibt, wie die datenparallele Ausführung eines Kernels auf einem Device umgesetzt wird. Wird ein Kernel zur Ausführung an ein Device gesendet, wird ein Indexraum angelegt. Jedem Punkt des Indexraumes wird dann ein **Work Item,** also eine Instanz des Kernels zusammen mit einer globalen ID, zugeordnet. Jedes Work Item wird mit einem Processing Element assoziiert, welches die Kernelbefehle ausführt. Work Items, die auf diese Weise derselben Compute Unit angehören, werden zu **Work Groups** zusammengefast. Work Items derselben Work Group teilen sich also denselben Speicher.

3.4.3 Das Speichermodell

Das Speichermodell beschreibt die Speicherhierarchie auf einem Device. Work Items haben Zugriff auf vier verschiedenen Speicherregionen:

- **Globaler Speicher:** Auf diesen Speicher haben alle Work Items Lese- und Schreibzugriff. Schreibzugriffe verschiedener Work Items auf dieselbe Speicherstelle sind jedoch problematisch, da keine Konsistenz garantiert wird und Zugriffskonflikte entstehen können.
- **Konstanter Speicher:** Der konstante Speicher ist eine Region des globalen Speichers, der nur Lesezugriffe erlaubt.

- **Lokaler Speicher:** Jede Compute Unit verfügt über einen eigenen lokalen Speicher. Er dient dem Datenaustausch zwischen Work Items derselben Work Group.
- **Privater Speicher:** Ein einzelnes Work Item verfügt über einen kleinen eigenen Speicherbereich. Hier werden im Code definierte Variablen, zum Beispiel die KernelID gespeichert.

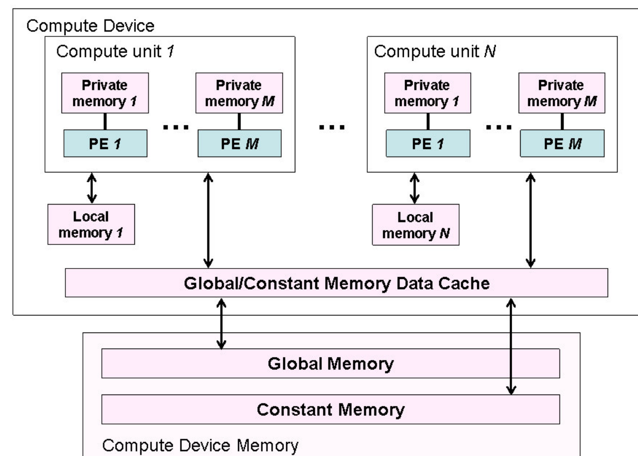


Abbildung 3.5: *Device Architektur mit Processing Elements (PE), Compute Devices und Speicherregionen* (Quelle: Munshi 2014: 24)

Die Unterscheidung der verschiedenen Speicherregionen wird in erster Linie durch den Geschwindigkeitsaspekt gerechtfertigt. Hierbei gibt es große Unterschiede zwischen den Speichertypen. Ausgehend vom globalen Speicher mit der geringsten Zugriffsgeschwindigkeit nimmt diese mit jeder Stufe auf der Speicherhierarchie zu. In dieselbe Richtung nimmt aber auch die Speicherkapazität ab. Eine durchdachte Speicherorganisation ist folglich von entscheidender Bedeutung für die Performance einer OpenCL Anwendung. Daten, die sehr häufig von verschiedenen Work Items benötigt werden, sollten, wenn es die Speichergröße zulässt, möglichst im lokalen Speicher abgelegt werden.

3.4.4 Das Programmiermodell

Dieses Modell beschreibt die unterschiedlichen Programmiermodelle, die mit OpenCL realisiert werden können. Es umfasst das datenparallele Modell und das taskparallele Modell. Ausschlaggebend für das Design von OpenCL und den weiteren Verlauf dieser Arbeit ist das datenparallele Modell. Datenparalleles Rechnen wurde bereits am Beispiel der Vektoraddition erläutert und die Umsetzung im Ausführungsmodell näher beschrieben.

Das taskparallele Modell ist vergleichbar mit der Benutzung mehrerer Threads auf einer CPU. Taskparallelität bedeutet, dass parallel verschiedene Befehlsfolgenausgeführt werden. OpenCL bietet hierfür die Möglichkeit, verschiedene Kernel von unterschiedlichen Devices ausführen zu lassen.

3.5 Zusammenfassung und Überleitung

Dank der Entwicklung von Schnittstellen zur freien Programmierung von Grafikchips, ist es heute möglich, von der hohen Rechenpower moderner Grafikprozessoren auch für nicht-grafische Berechnungen zu profitieren. Durch ihre auf Grafikberechnungen spezialisierte und vielkernige Architektur ermöglicht die GPU hochgradig paralleles Rechnen. Das parallele Rechnen ist gegenwärtig, wo höhere Prozessortaktungen nicht realisiert werden können, zu einem sehr wichtigen Faktor für performanceorientierte Softwareentwicklung geworden. Dennoch ist hervorzuheben, dass GPGPU keinesfalls als Universallösung für schnelleres Rechnen verstanden werden darf und die CPU überflüssig macht. Ein Blick auf die unterschiedlichen Prozessorarchitekturen hat gezeigt, dass die theoretisch mögliche Rechenpower der GPU nur unter bestimmten Voraussetzungen erreicht werden kann, nämlich dann, wenn alle Rechenwerke zeitgleich ausgelastet werden. In der Praxis bedeutet dies, dass die GPU die CPU nicht ersetzt, sondern im geeigneten Fall als mathematischer Co-Prozessor fungieren kann, und zwar für die Berechnungsschritte, die sich auf das datenparallele Rechenmodell übertragen lassen.

> It should be clear now, that GPUs are designed as numeric computing engines, and they will not perform well on some tasks on which CPUs are designed to perform well; therefore one should expect, that most applications will use both CPUs and GPUs, executing the sequential parts on the CPU and numerically intensive parts on the GPUs. (Kirk &Hwu 2010:5)

Doch auch wenn ein Algorithmus oder Teile dieses auf die Hardwarestruktur der GPU übertragbar sind, ist noch kein Performancegewinn gewährleistet. Dies liegt unter anderem daran, dass der Transfer von Daten zwischen CPU und GPU Speicher über den PCI-Bus Zeit kostet, die berücksichtigt und ausgeglichen werden muss. Darüber hinaus kommt aber auch der Speicherorganisation eine wesentlich größere Bedeutung hinzu, die für den Erfolg einer GPGPU-Anwendung ausschlaggebend ist. Bei der folgenden Planung und Implementierung einer eigenen OpenCL-Anwendung werden diese Anforderungen im Einzelnen deutlich werden. Darüber hinaus werden Strategien vorgestellt, wie die Leistung der GPU bestmöglich ausgeschöpft werden kann.

4 Document Clustering auf der GPU

In Kapitel 2 dieser Arbeit wurde deutlich, dass das Clustern von Textdokumenten ein sehr zeit- und rechenintensives Verfahren ist. Grund dafür ist die sehr hohe Dimension der Dokumentvektoren, welche starken Einfluss auf die Laufzeit des K-Means Clustering Algorithmus hat. Nachdem in Kapitel 3 die Grundlagen und Vorteile der GPU-Programmierung vorgestellt wurden, soll nun überprüft werden, ob auch das Clustern von Dokumenten von diesem Verfahren profitieren kann. Begleitend zu dieser Arbeit wurde eine GPU-basierte Variante des K-Means-Algorithmus implementiert. Dafür wurde die GPGPU-Schnittstelle OpenCL und die damit gegebenen Möglichkeiten des datenparallelen Rechnens genutzt. Im Vergleich mit einer sequentiellen Variante, die auf herkömmliche Weise von der CPU ausgeführt wird, soll schließlich getestet werden, ob und unter welchen Bedingungen ein Performancegewinn durch den Einsatz von GPGPU möglich ist.

Die Idee, den K-Means Algorithmus mittels GPGPU zu beschleunigen ist nicht neu. Es existieren bereits Studien, die sich mit dieser Problemstellung erfolgreich befasst haben. Zu nennen sind in diesem Zusammenhang insbesondere die GPU-basierten K-Means Algorithmen GPUMiner (Fang et al. 2008), UV_k-Means (Che et al. 2008) und HP_k-Means (Wu et al. 2009). Die Studien dokumentieren jeweils beeindruckende Performanceleistungen im Vergleich mit CPU-basierten Vergleichsprogrammen. Außer dass die bereits existierenden Anwendungen mit der an NVIDIA gebundenen GPGPU-Schnittstelle CUDA realisiert wurden, unterscheiden sie sich von der hier vorgestellten Anwendung in einem weiteren sehr entscheidenden Punkt: Bei der Implementierung des Algorithmus wurden jeweils Vektordimensionen angenommen, die weit unter einer realistischen Dimension für einen Dokumentvektor liegen. Die anschließenden Performancetests konnten daher auch nur auf Vektoren von meist ein- bis maximal zweistelliger Dimension durchgeführt werden. Zu den neueren Arbeiten über GPU-basiertes Clustern gehört auch ein 2012 im Journal of Computer ans System Sciences veröffentlichter Artikel mit dem Titel „Speeding up k-Means algorithm by GPUs“. Einleitend wird hier zwar festgestellt, dass die Dimension der zu Grunde liegenden Vektoren für die Implementierung eine wichtige Rolle spielt:

> Different from existing GPU-based k-Means Algorithms, we observe, that data dimensionality is an important factor that should be taken into consideration when parallelizing k-Means on GPUs. In particular we use two different strategies for low-dimensional data sets and high-dimensional data sets respectively in order to make best use of GPU computing horsepower. (Li et al. 2013)

Im weiteren Verlauf zeigt sich jedoch, was die Autoren unter hochdimensionale Daten verstehen: Die maximale Vektordimension in den durchgeführten Performancetests beträgt 160. Auf den Hintergrund des Document Clusterings sind die bisherigen Ergebnisse somit nicht übertragbar und die zentrale Fragestellung bleibt weiterhin unbeantwortet.

Im Folgenden wird erläutert, wie die Implementierung eines datenparallelen K-Means Algorithmus für hochdimensionale Vektoren umgesetzt wurde. Hierbei wird auch deutlich werden, dass gerade der Umgang mit hochdimensionalen Vektoren Einschränkungen und Schwierigkeiten bereitet, die bisher nicht berücksichtigt wurden.

4.1 Implementation

Bei der Planung einer GPGPU-Anwendung steht zunächst die Frage im Mittelpunkt, welche Berechnungsschritte für eine Ausführung auf der GPU überhaupt in Frage kommen. Die Parallelisierbarkeit ist hierfür das ausschlaggebende Kriterium. Aber auch die Anteile eines Berechnungsschritts an der Gesamtlaufzeit sind zu berücksichtigen. Die erste Phase des K-Means-Algorithmus, die Auswahl k zufälliger Clusterzentren, hat kaum Einfluss auf die Laufzeit einer Clusteranalyse, da sie lediglich einmal zu Beginn ausgeführt wird. Eine Ausführung auf der GPU würde auch bei guter Parallelisierbarkeit keinen merklichen Geschwindigkeitsvorteil erzielen. Entscheidend sind stattdessen die Iterationsschritte, also die Zuordnung der Vektoren zu einem Clusterzentrum und die anschließende Neuberechnung der Zentrumsvektoren. Im Folgenden werden beide Phasen daher gesondert auf ihre GPU-Kompatibilität geprüft. Da die Zuordnung der Vektoren zu ihrem nächsten Clusterzentrum mit Abstand am zeitintensivsten ist, eine gute Parallelisierung also den größten Geschwindigkeitsvorteil verspricht, wird mit der Analyse dieser Phase begonnen.

4.1.1 Parallelisierung der Ähnlichkeitsberechnungen

Um einen Dokumentvektor eindeutig einem Cluster zuteilen zu können, muss sein Abstand zu allen Zentrumsvektoren berechnet werden. Die Zuordnungsphase besteht somit aus n*k Berechnungen der Cosinus Ähnlichkeit und kommt auf eine sequentielle Laufzeit von n*k*d. Der Schritt vom sequentiellen zum datenparallelen Rechnen ist immer dann möglich, wenn eine

sequentielle Schleife durch die Verteilung der einzelnen Schleifeniterationen auf verschiedene GPU-Kerne aufgelöst werden kann. Zur sequentiellen Berechnung der Cosinus Ähnlichkeit ist eine Schleife über sämtliche Vektordimensionen notwendig:

```
floata,b,c = 0
for each dim
      a += vector[dim] ²
      b +=  center[dim]²
)     d += vector[d]*center[d]
 end
similarity = ab/c
```

Wie man dem Code entnehmen kann, stehen die einzelnen Iterationen in Abhängigkeit zueinander, da sie auf den jeweils zuvor berechneten Variablen aufbauen. Sie sind damit für die Ausführung auf der GPU nicht geeignet und eine vollständige Parallelisierung auf Dimensionsebene nicht möglich.

Da jedoch jede Ähnlichkeitsberechnung für sich genommen ein unabhängiger und in sich abgeschlossener Vorgang ist, ergeben sich andere Möglichkeiten zur Parallelisierung. Es sind zwei Strategien möglich, die insgesamt n*k Distanzberechnungen auf GPU-Kerne zu verteilen. Jeder Dokumentvektor kann mit einem anderen GPU-Kern assoziiert werden, der jeweils die Distanz dieses Vektors zu allen Zentrumsvektoren bestimmt. Umgekehrt kann auch jedes Clusterzentrum mit einem GPU-Kern assoziiert werden. Jeder Kern bestimmt dann die Distanz des ihm zugeteilten Clusterzentrums zu allen Dokumentvektoren.

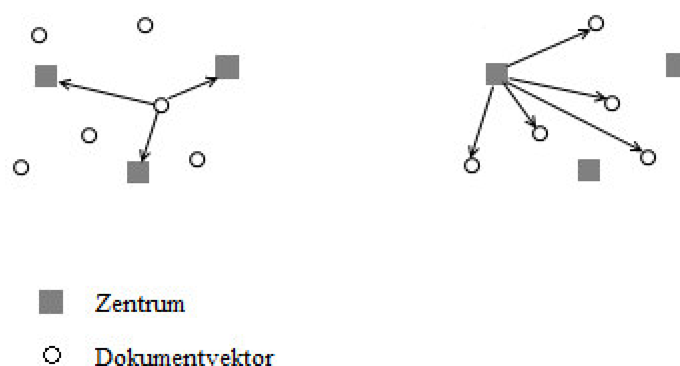


Abbildung 4.1: *Mögliche Strategien zur Parallelisierung der Ähnlichkeitsberechnungen. Vektororientiert (links) und Zentrumsorientiert (rechts)*

Da mit der ersten Strategie ein höherer Grad an Parallelität erreicht werden kann, wird diese Variante ausgewählt. Die Zahl der Clusterzentren ist im Regelfall wesentlich geringer, als GPU-Kerne verfügbar sind. Die volle Rechenpower einer GPU würde also mit der zweiten Variante nicht vollständig ausgeschöpft werden können.

Nachdem die bestmögliche Parallelisierungsstrategie bestimmt ist, kommt es auf eine effiziente Organisation der Datenspeicherung an. Die Speicherorganisation ist ein sehr wichtiger Faktor für die Performance einer GPU-Anwendung und sollte daher gut an die veränderten Anforderungen der GPU Hardware angepasst werden.

4.1.2 Speicherorganisation

Da der Datentransfer zwischen dem Hauptspeicher und dem Speicher der GPU sehr zeitspielig ist, ist zunächst zu bestimmen, welche Daten die GPU für ihre Ausführungen benötigt, um unnötigen Datenaustausch zu vermeiden. In diesem Fall benötigt jeder GPU-Kern sämtliche Zentrumsvektoren, sowie den Dokumentvektor, für dessen Ähnlichkeitsberechnungen er zuständig ist. Der einmalige Transfer aller Clusterzentren und Dokumentvektoren in den GPU Speicher lässt sich damit nicht umgehen. Weiter kann durch den geschickten Gebrauch der verschiedenen Speicherregionen Einfluss auf die Performance genommen werden. Der Zugriff eines Kernels auf den globalen Speicher ist kostspieliger als der Zugriff auf den lokalen Speicher. In den bereits bestehenden GPU-basierten K-Means Varianten wurden deshalb sämtliche Zentrumsvektoren in den lokalen Speicherregionen abgelegt. Dies ist jedoch bei hochdimensionalen Vektoren, wie sie fürs Document Clustering benötigt werden, nicht möglich. Das lokale Speichervolumen reicht nicht aus, um alle Clusterzentren aufzunehmen. Sämtliche Vektoren müssen somit im globalen Speicher abgelegt werden und von dort aus gelesen werden.

Einfluss auf die Geschwindigkeit des Datenzugriffs kann somit nur durch die Gewährleistung eines **coalescen Speicherzugriffs** genommen werden. Coalescer Speicherzugriff bedeutet, dass die Daten möglichst so im Speicher verteilt liegen, dass die Zugriffe aus den verschiedenen Kernelinstanzen möglichst nah beieinanderliegen. Um dies zu gewährleisten, muss darauf Rücksicht genommen werden, welche Kernelinstanz zu welchem Zeitpunkt auf welche Datenstellen zugreift. Jede Kernelinstanz berechnet zunächst die Ähnlichkeit des ihm zugeteilten Dokumentvektors zum ersten Clusterzentrum. Zeitgleich werden somit alle ersten Einträge der Dokumentvektoren und der erste Eintrag des ersten Clusterzentrums benötigt. Daraufhin wird auf alle zweiten Eintrage der Dokumentvektoren und des ersten Clusterzentrums zugegriffen, bis jede Kernelinstanz schließlich beim letzten Dimensionseintrag angekommen ist. Daraufhin geht es weiter mit dem zweiten Clusterzentrum bis schließlich alle Ähnlichkeiten berechnet wurden. In

Tabelle 4.1 sind die Speicherzugriffe (Z) der Kernelinstanzen auf die Dokumentvektoren dargestellt. Da als Datenstruktur auf der GPU nur eindimensionale Arrays zugelassen sind, sind alle Vektoren und Zentren jeweils in einem großen Array zusammengefasst.

	V_{11}	V_{12}	V_{13}	$\ldots V_{1m}$	V_{21}	V_{22}	V_{23}	$\ldots V_{2m}$	V_{31}	V_{32}	V_{33}	$\ldots V_{3m}$	…	V_{n1}	V_{n2}	V_{n3}	$\ldots v_{nm}$
Z	1	2	3	m	1	2	3	m	1	2	3	m		1	2	3	m

Tabelle4.1: *Speicherzugriffe auf das Vektorarray zur Berechnung der paarweisen Vektorähnlichkeiten*

Wie man in Tabelle 4.1 sehen kann, sind die Lesezugriffe auf die Dokumentvektoren sehr ungünstig verteilt. Aus jeder Kernelinstanz wird zeitgleich auf sehr weit auseinanderliegende Datenstellen zugegriffen, was die Zugriffsgeschwindigkeit deutlich verlangsamt. Bei der Implementierung wird die Struktur deshalb zu Gunsten der Lesezugriffe umsortiert. Statt Vektorweise, werden die Einträge deshalb Dimensionsweise gespeichert. Tabelle 4.2 zeigt die neue Speicherordnung der Dokumentvektoren.

	V_{11}	V_{21}	V_{31}	$\ldots V_{n1}$	V_{12}	V_{22}	V_{32}	$\ldots V_{m2}$	V_{13}	V_{23}	V_{33}	$\ldots V_{n3}$	…	V_{1m}	V_{2m}	V_{3m}	$\ldots v_{nm}$
Z	1	1	1	1	2	2	2	2	3	3	3	3		m	m	m	m

Tabelle 4.2: *Coalesce Speicherorganisation der Dokumentvektoren zur Berechnung der paarweisen Vektorähnlichkeiten*

Die parallelen Lesezugriffe liegen nun dicht beieinander, so dass jede Kernelinstanz dieselbe Cachezeile nutzt. Damit ist die Speicherorganisation nun bestmöglich an die zuvor gewählte Parallelisierungsvariante angepasst. Im nächsten Abschnitt wird analysiert, ob auch der nächste Schritt des K-Means Algorithmus, die Neuberechnung der Clusterzentren, von einer Ausführung auf der GPU profitieren kann.

4.1.3 Parallelisierung der Zentrumsberechnungen

Die neuen Zentren berechnen sich aus dem Mittelwert aller einem Cluster zugeteilten Vektoren. Hierfür müssen zunächst alle Vektoren eines Clusterzentrum auf das jeweilige alte

Clusterzentrum aufsummiert werden. Auch hier sind theoretisch drei Möglichkeiten der Parallelisierung denkbar, die nun der Reihe nach betrachtet werden.

Behält man die Assoziierung jedes GPU-Kerns mit einem anderen Dokumentvektor bei, so müsste jede Kernelinstanz den entsprechenden Vektor auf den ihm zugeteilten Zentrumsvektor addieren. Hierbei würden jedoch Schreibkonflikte entstehen, da zeitgleich verschiedene Dokumentvektoren auf denselben Zentrumsvektor addiert würden. Zwar bietet OpenCL die Möglichkeit verschiedene Kernelinstanzen miteinander zu synchronisieren, so dass bestimmte Instanzen erst ausgeführt werden, wenn andere bereits abgearbeitet sind, doch würde damit eine parallele Arbeitsweise so gut wie aufgehoben werden.

Eine weitere Möglichkeit ist auch hier, jeden GPU-Kern mit der Aufsummierung eines anderen Zentrumsvektors zu beauftragen. In diesem Fall würde jede Kernelinstanz alle Dokumentvektoren auf Zugehörigkeit zu ihrem Zentrum abfragen und bei positiver Antwort auf das Clusterzentrum addieren. Zwei Argumente sprechen jedoch gegen diese Variante: Erstens können Kontrollstrukturen (wie if- oder switch-Statements) negativen Einfluss auf die Performanceleistung haben. Dies geschieht, wenn Kernelinstanzen derselben Work Group verschiedenen Programmflüssen folgen (vgl. Che et al. 2008: 1374). Zweitens würde ohnehin nur ein geringer Parallelitätsgrad erreicht werden, da die Zahl der Cluster und somit der parallel ausgelasteten GPU-Kerne im Regelfall gering ist.

Eine dritte Strategie ist die Parallelisierung auf Dimensionsebene, wie sie im vorherigen Algorithmus-Schritt nicht möglich war. Der entsprechende Kernel für diese Strategie ist nachfolgend skizziert:

```
calcNewCenters(){

    int dim = getKernelID();
    for each v
        c = centerOfV;
        c[dim] += v[dim];
    end
}
```

Jede Kernelinstanz kann ohne Schreibkonflikte und unabhängig von den anderen Instanzen ausgeführt werden. Eine Parallelisierung auf Dimensionsebene scheint damit die überzeugendste der drei Strategien zu sein. Ein Nachteil dieser Strategie zeigt sich jedoch im Hinblick auf die Speicherorganisation, die zu Gunsten der ersten Algorithmus-Phase festgelegt wurde (Tabelle

4.2). Für die parallele Neuberechnung der Zentren ist diese Speicherstruktur genau widersprüchlich, wie Tabelle 4.3 zeigt.

	V_{11}	V_{21}	V_{31}	$\ldots V_{n1}$	V_{12}	V_{22}	V_{32}	$\ldots V_{m2}$	V_{13}	V_{23}	V_{33}	$\ldots V_{n3}$	…	V_{1m}	V_{2m}	V_{3m}	$\ldots v_{nm}$
Z	1	2	3	m	1	2	3	m	1	2	3	m		1	2	3	m

Tabelle 4.3: *Speicherzugriffe auf das Vektorarray zur Neuberechnung der Clusterzentren*

Um mit Sicherheit die perfomanteste aller Strategien benennen zu können, wurden in einer Testphase alle drei Parallelisierungsmöglichkeiten umgesetzt und miteinander verglichen. Außerdem wurde zum Vergleich eine Variante implementiert, die die gesamte Phase der Zentrumsberechnung sequentiell auf der CPU ausführt. Die Ergebnisse haben gezeigt, dass jede der drei Parallelisierungsstrategien mehr Verzögerungen verursacht, als Zeit gewinnt. Mit Vektoren von hoher mindestens vierstelliger Dimension, ist eine sequentielle Zentrumsberechnung auf der CPU am performantesten. Für die Fragestellung dieser Arbeit bedeutet dies, dass eine Performancesteigerung durch GPGPU nur dann möglich ist, wenn bereits die Parallelisierung des ersten Iterationsschritts ausreicht, um genügend Zeitvorsprung gegenüber der sequentiellen Variante zu gewinnen.

4.1.4 Zusammenfassung

Abbildung 4.2 zeigt das Ergebnis der Überlegungen zur bestmöglichen Parallelisierbarkeit des K-Means Algorithmus für hochdimensionale Vektoren. Die Anwendung besteht aus drei grundlegenden Arbeitsschritten. Aus der Host-Applikation heraus wird zunächst die OpenCL Schnittstelle initialisiert (1). Hier wird unter anderem ermittelt, welche Devices auf dem jeweiligen Rechner zur Verfügung stehen und bevorzugt eine GPU als ausführendes Device ausgewählt. Die initialen Clusterzentren und Dokumentvektoren werden daraufhin an den globalen Speicher der GPU übermittelt. Darüber hinaus wird Speicherplatz für ein weiteres Array alloziert. Es dient der Speicherung der Kernelergebnisse, dokumentiert also die Clusterzugehörigkeiten der Dokumentvektoren. Die Anwendung blockiert so lange, bis sämtliche Daten vollständig an den GPU-Speicher übermittelt wurden. Erst dann erfolgt der Befehl zur Ausführung des Kernels auf dem Device. Für jeden Dokumentvektor wird eine eigene Kernelinstanz gebildet, welche das nächstgelegenste Clusterzentrum ermittelt und das Ergebnis im dafür vorgesehenen Array festhält (2). Auf Basis der so ermittelten Clusterzugehörigkeiten werden im nächsten Schritt die

Clusterzentren aktualisiert. Da sich eine sequentielle Ausführung auf der CPU für diesen Arbeitsschritt als die schnellste Variante herausgestellt hat, wird dieser Schritt auch in der GPU-basierten Variante von der CPU übernommen. Um die neuen Zentren auf der CPU berechnen zu können, müssen die Ergebnisse der Kernelausführung zurück in den Host-Speicher geladen werden. Die Host-Applikation prüft, ob sich die Clusterzugehörigkeiten im Vergleich zur vorherigen Runde geändert haben und berechnet im positiven Fall die neuen Clusterzentren (3). Die aktualisierten Clusterzentren werden daraufhin wieder in den GPU Speicher geladen und ersetzen die alten Zentren aus der vorangegangenen Runde. Mit der erneuten Ausführung des Kernels beginnt der Ablauf von vorn.

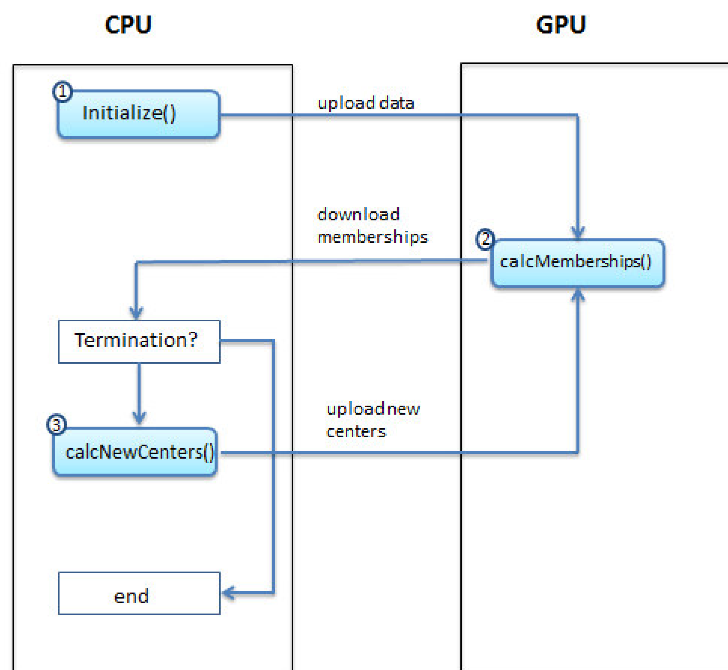


Abbildung 4.2: *schematischer Ablauf des GPU-basieren K-Means-Algorithmus*

Die Antwort auf die zentrale Fragestellung dieser Arbeit kann nun durch einen direkten Laufzeitvergleich des datenparallelen und sequentiellen K-Means-Algorithmus beantwortet werden. Der folgende Abschnitt fasst die Ergebnisse dieser Laufzeitmessungen zusammen und klärt, ob die Verwendung der GPU zum Clustern von Dokumenten lohnenswert ist.

4.2 Performancevergleich

Für den Performancevergleich zwischen der GPU-basierten K-Means Variante und der herkömmlichen sequentiellen K-Means Variante wurde die Gesamtlaufzeit beider Algorithmen unter jeweils gleichen Voraussetzungen und identischen Ausgangsvektoren gemessen. Da das

primäre Interesse der Untersuchung dem Laufzeitverhalten gilt und weniger den tatsächlichen Ergebnissen des Clusterings, werden für das Experiment künstlich generierte Vektoren erzeugt, die in Umfang und Wertebereich realen Dokumentvektoren entsprechen könnten. Ein Vergleichstest stellt sicher, dass beide Algorithmen unter gleichen Voraussetzungen identische Clusterergebnisse liefern. Um datenunabhängige Ergebnisse zu erhalten, wurde in beiden Algorithmen die Zahl der Iterationen auf 50 begrenzt. Die Laufzeitmessung umfasst jeweils a) die Gesamtlaufzeit der 50 Iterationsschritte, b) die Initialisierungszeit, sowie c) die von der GPU-basierten Variante benötigte Zeit zum Datentransfer zwischen CPU und GPU. Ausgehend von einer Basiskonfiguration wurde jeweils das Laufzeitverhalten bei Veränderung einer der Konfigurationsvariablen (Vektordimension, Vektorzahl und Clusterzahl) verglichen. Die Basiskonfiguration bestand aus 4096 Dokumentvektoren mit jeweils 2048 Dimensionen und 128 Clusterzentren.[13]

Als erstes wurde das Laufzeitverhalten der beiden K-Means Varianten bei Veränderung der Dimensionsvariablen getestet. Abbildung 4.3 stellt das Ergebnis der Laufzeitmessungen bei Dokumentvektoren von 512 bis maximal 16384 Dimensionen dar. (Noch größere Dimensionen lässt die Speicherkapazität der GPU nicht zu) Es lässt sich unschwer erkennen, dass das datenparallele Rechnen auf der GPU der CPU hinsichtlich des Laufzeitverhaltens deutlich überlegen ist. Allerdings bleibt der relative Performancevorteil nahezu konstant. Mit dem datenparallelen Algorithmus ist durchgehend mehr als acht Mal schnelleres Rechnen möglich. (Abbildung 4.4)

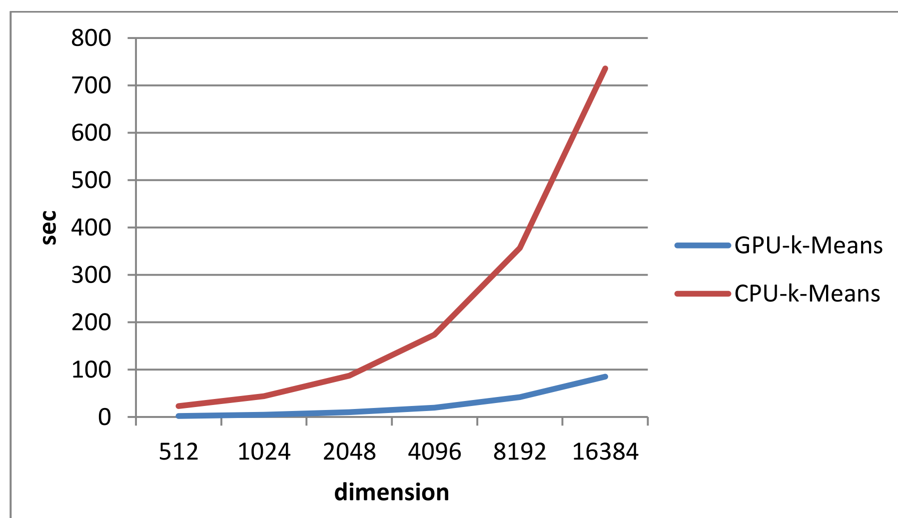


Abbildung 4.3: *Laufzeit bei steigender Vektordimension*

[13] Für die Messungen wurde ein Arbeitsplatzrechner mit Intel Core Duo E6750 mit 2,66GHz und 4GB Arbeitsplatzspeicher verwendet. Grafikkarte war eine Nvidia GeForce GT630 mit 2048MB on board device memory.

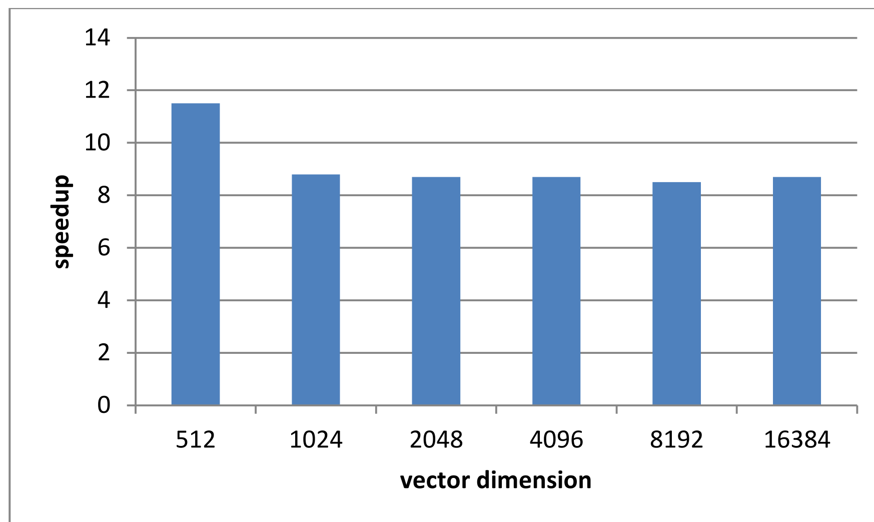


Abbildung 4.4: *Performancevorteil des daten-parallelen K-Means-Algorithmus bei steigender Vektordimension*

Ähnliche Beobachtungen lassen sich auch bei Veränderung der Clusterzahlen machen. Mit steigender Clusterzahl nimmt die Laufzeit beider K-Means Varianten zu. Unabhängig davon ist die GPU-basierte Variante jedoch auch hier deutlich überlegen und mehr als acht mal schneller als die herkömmliche Variante. (vgl. Abbildung 4.5)

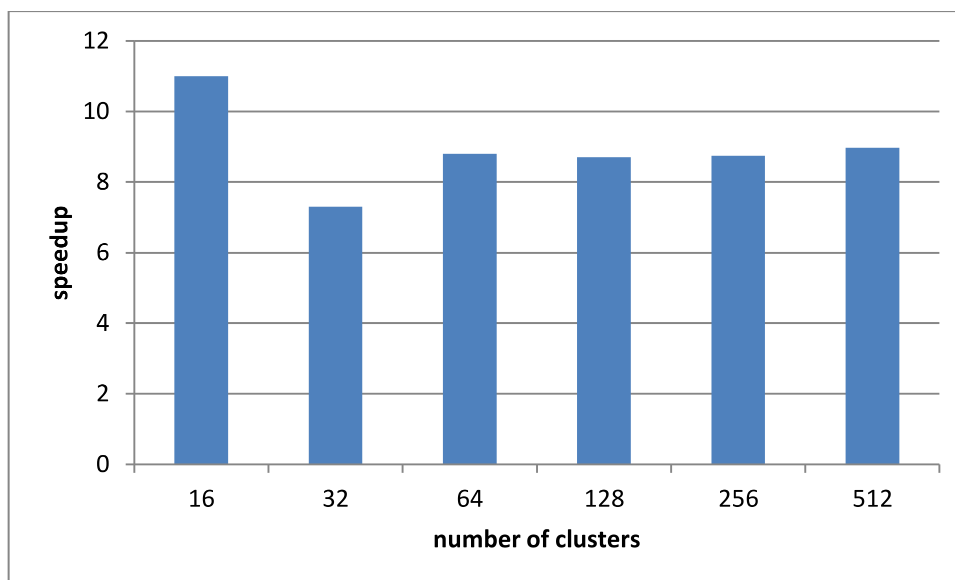


Abbildung 4.5: *Performancevorteil des datenparallelen K-Means-Algorithmus bei steigender Clusterzahl*

Als letztes wurde getestet, welches Laufzeitverhalten bei wachsender Dokumentvektormenge zu beobachten ist, also bei Veränderung der Variablen, auf dessen Ebene die GPU-basierte K-

Means Variante parallelisiert wurde. Tabelle 4.4 zeigt die Messergebnisse bei 128 bis maximal 16384 Dokumentvektoren. Neben der Gesamtlaufzeit beider Algorithmen wird auch die durchschnittliche Zeit für einen Iterationsschritt angegeben. Auffällig bei der GPU-basierten K-Means Variante ist, dass die durchschnittliche Laufzeit einer Iteration bei niedriger Vektorzahl trotz einer Verdopplung unverändert bleibt. Beim sequentiellen Algorithmus verdoppelt sich stattdessen mit der Zahl der Vektoren auch die Bearbeitungszeit. Hieran lässt sich sehr gut und direkt der Vorteil des datenparallelen Rechnens erkennen. 1024 Vektoren können genauso schnell abgearbeitet werden wie 128 Vektoren, da offenbar bis zu 1024 Kernelinstanzen parallel von jeweils einem GPU-Kern abgearbeitet werden können. Erst bei einer noch größeren Vektormenge sind nicht mehr ausreichend Kerne verfügbar, so dass einige Vektoren erst auf freie Kerne warten müssen.

Dataset			**GPU-K-Means**		**CPU-K-Means**	
V	D	K	time	per Iteration	time	per Iteration
128	2048	128	3,25	0,07	2,78	0,06
256	2048	128	3,28	0,07	5,5	0,11
512	2048	128	3,4	0,07	10,98	0,22
1024	2048	128	3,73	0,07	21,94	0,44
2048	2048	128	5,13	0,1	43,91	0,88
4096	2048	128	10	0,2	87,76	1,76
8192	2048	128	10,1	0,4	174,78	3,5
16384	2048	128	41,57	0,83	349,4	6,99

Tabelle 4.4: *Laufzeitmessungen bei steigender Vektorzahl*

Insgesamt lässt sich also auch bei Veränderung der Vektorzahl feststellen, dass der daten-parallele Algorithmus durchgehend performanter ist. Zusätzlich wird der Performancevorteil durch größere Vektormengen sogar begünstigt, da der relative Laufzeitzuwachs durch mehr Vektoren geringer ist als beim sequentiellen K-Means. Während die Iterationszeit des sequentiellen Algorithmus mit jedem Vektor stetig ansteigt, bleibt sie beim GPU-basierten Algorithmus bis zu einem gewissen Zuwachs nahezu konstant. Abbildung 4.6 zeigt den relativen Performancegewinn bei schrittweiser Vergrößerung der Vektormenge.

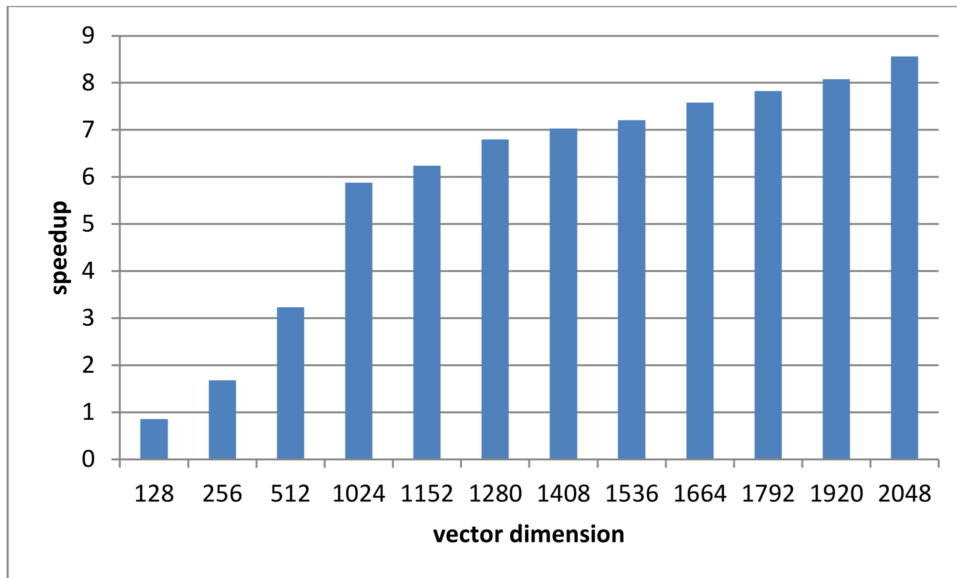


Abbildung 4.6: *Performancevorteil des datenparallelen K-Means-Algorithmus bei steigender Vektorzahl*

5 Fazit und Ausblick

Das Ziel dieser Arbeit war, die Motivation für die Auseinandersetzung mit der GPU-Programmierung am Beispiel des Document Clustering Verfahrens herzuleiten, die grundlegenden Konzepte und Vorzüge dieses relativ neuen Programmierverfahrens in kompakter Form darzustellen und schließlich seine Anwendung zu demonstrieren, indem es auf eines der gebräuchlichsten Clustering-Verfahren, den K-Means-Algorithmus angewandt wurde. Als Hauptmotivation stand hierbei im Vordergrund den Prozess des K-Means Clusterings durch den Gebrauch von GPGPU-Techniken zu beschleunigen.

Diese Ziele wurden im Verlauf der Arbeit wie folgt umgesetzt: In Kapitel 2 „Clusteranalyse" wurde nach einer Einführung in die Grundlagen der Clusteranalyse herausgestellt, dass das K-Means-Clustern im Allgemeinen, insbesondere aber im Falle des Clustern von Textdokumenten mit sehr hohen Laufzeiten verbunden ist. Vor dem Hintergrund dieser Feststellung wurde in Kapitel 3 „General Purpose Computation on Graphics Processing Units" ein Programmierverfahren vorgestellt, das gezielt die vielkernige Architektur von Grafikprozessoren nutzt, um rechenintensive Programmabläufe, wie sie auch im K-Means Algorithmus zu finden sind, parallel auszuführen. In Kapitel 4 „Document Clustering auf der GPU" wurden die zuvor behandelten Themen schließlich zusammengeführt und die Entwicklung eines GPU-gestützten K-Means Algorithmus erläutert. Hierbei wurde auf Anforderungen eingegangen, die insbesondere durch das Clustern von Textdokumenten entstehen und in bereits existierenden GPU-gestützten Clustering-Algorithmen unberücksichtigt bleiben: Der Umgang mit sehr hohen Vektordimensionen.

Die Beantwortung der zentralen Fragestellung, ob der Prozess der Clusteranalyse von Textdokumenten, durch die Verwendung von GPGPU-Techniken beschleunigt werden kann, konnte schließlich im letzten Kapitel durch den direkten Laufzeitvergleich einer herkömmlichen K-Means Variante mit der in Kapitel 4 entwickelten GPU-basierten Variante beantwortet werden. Verglichen wurde das Laufzeitverhalten bei verschiedenen, aber jeweils identischen Konfigurationen, die unterschiedliche Clusterszenarien darstellen sollten. Als Resultat können die folgenden Aussagen gemacht werden: Insgesamt kann der Versuch, den Clustering-Prozess durch GPGPU zu Beschleunigen, als gelungen bewertet werden. Bei allen getesteten Szenarien war die Laufzeit des GPU-basierten K-Means Algorithmus geringer als die der herkömmlichen CPU-Variante. In fast allen Fällen rechnete der GPU-gestützte Algorithmus mehr als acht mal

schneller als sein Konkurrent. Bedenkt man, dass in der Praxis meist erst mehrere Clustering-Durchläufe zum endgültigen Ergebnis führen, ist dies ein durchaus vielversprechendes Ergebnis.

Mit der Erhöhung der Vektordimension und der Anzahl der zu bildenden Cluster stieg der Laufzeitbedarf beider Versionen in etwa gleich stark an, so dass der Performancevorteil konstant blieb. Die Erhöhung der zu clusternden Vektormenge stelle sich darüber hinaus sogar als begünstigend für die daten-parallele Verarbeitungsweise des GPU-basierten Algorithmus heraus. Der Anstieg der Laufzeit durch größere Vektormengen konnte im Vergleich zur sequentiellen Variante verringert werden, sodass der Performancevorteil mit wachsender Vektormenge zunahm. Interessant wäre gewesen, ob sich diese Entwicklung auch bei noch größeren Datenmengen fortgesetzt hätte, doch war dies durch die begrenzte Speicherkapazität der Grafikkarte leider nicht möglich. Für die zukünftige Beschäftigung mit diesem Thema wäre daher eine Möglichkeit, mit der Verteilung der Daten auf verschiedene Prozessoren zu experimentieren. Eine weitere Anregung könnte außerdem sein, große Vektormengen in Blöcke aufzuteilen und nacheinander von der GPU bearbeiten zu lassen. In jedem Fall konnte in dieser Arbeit gezeigt werden, dass die GPU-Programmierung ein auch für die Computerlinguistik interessantes und zukunftsorientiertes Forschungsfeld darstellt und sich eine Auseinandersetzung damit trotz aufwändiger Einarbeitungszeit lohnen kann. Die Computerlinguistik bietet mit Sicherheit auch andere Ansatzpunkte, die von der GPU-Programmierung profitieren könnten.

Literaturverzeichnis

BACHER, J.: 1994, *Clusteranalyse. Anwendungsorientierte Einführung,* Oldenbourg Verlag, München.

CHE, S., M. BOYER, J. MENG, D. TARJAN, J. W. SHEAFFER & K. SKADRON: 2008, 'A performance study of general-purpose applications on graphics processors using CUDA'*, Journal of Parallel and Distributed Computing* 68, S. 1370 – 1380.

COOK, S.: 2013, *CUDA Programming. A developer's guide to parallel computing with GPUs,* Morgan Kaufmann, Amsterdam; Boston.

FANG, W., K. K. LAU, M. LU, X. XIAO, C.K. LAM, P.Y. YANG, B. HE, Q. LUO, P.V. SANDER & K.YANG: 2008, 'Parallel Data Mining on Graphics Processors', Tech. rep., Hong Kong University of Science and Technology.

GASTER, B., L. HOWES, D.R. KAELI, P. MISTRY & D. SCHAA: 2012, *Heterogeneous Computing with OpenCL,* Morgan Kaufmann, Waltham.

HEYER, G., U. QUASTFOFF & T. WITTIG: 2006, *Text Mining: Wissensrohstoff Text. Konzepte, Algorithmen, Ergebnisse,* W3L GmbH, Witten Herdecke.

HUANG, A.: 2008, 'Similarity Measures for Text Document Clustering', *Proceedings of the New Zealand Computer Science Research Student Conference*, S. 49-56.

KIRK, D.B. & W.W. HWU: 2010, *Programming massively parallel processors: a hands-on approach,* Morgan Kaufmann, Amsterdam u.a.

LI, Y., K. ZHAO, X. CHU & J. LIU: 2013, 'Speeding up k-Means algorithm by GPUs', *Journal of Computer and System Sciences* 79, S. 216-229.

MACQUEEN, J.B.: 1967, 'Some methods for classification and analysis of multivariate observations', Proceedings *of the 5th Berkeley Symposium on mathematical statistics and probability* 1, S. 281-297.

MANNING, C. D, RAGHAVAN & SCHÜTZE: 2008, *Introduction to Information Retrieval,* Cambridge University Press, New York.

MAULIK, U: 2002, 'Performance evaluation of some clustering algorithms and validity indices', *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24, S. 1650-1654.

MUNSHI, A. (ed.): 2014, 'The OpenCL Specification. Version 2.0', https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf, zuletzt aufgerufen: 17.06.2014.

OWENS, J.D., D. LUEBKE, N. GOWINDARAJU, M. HARRIS, J. KRÜGER, A. E. LEFOHN & T.J. PURCELL: 2007, 'A Survey of General-Purpose Computation on Graphics Hardware', *Computer Graphics Forum* 26, S. 80-113.

STREHL, A., J. GHOSH & R. MOONEY: 2000, 'Impact of similarity measures on web-page clustering', Tech. rep., American Association for Artificial Intelligence, S. 58-64.

SUTTER, H& J. LARUS: 2005, 'Software and the Concurrency Revolution', *Queue* 3, S. 54 – 62.

SUTTER, H.: 2005, 'the free lunch is over: A fundamental turn toward concurrency in software', *Dr. Dobb's Journal* 30, S. 202-210.

WU, R.., B. ZHANG & M. HSU: 2009, 'Clustering Billions of Data Points Using GPUs', *Proceedings of the Combined Workshops: Unconventional High Performance Computing Workshop Plus Memory Access Workshop,* S. 1-6.