Institut für Informatik
Lehr- und Forschungseinheit
für Datenbanksysteme

Ludwig
Maximilians
Universität
München
LMU

Diplomarbeit

Clustering mittels Grafikprozessor

Robert O. Noll

Aufgabensteller:	Prof. Dr. Christian Böhm
Betreuer:	Bianca Wackersreuther
Abgabetermin:	10.02.2009

Erklärung

Hiermit versichere ich, daß ich diese Diplomarbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 10.02.2009

. . . . . .
Robert O. Noll

Zusammenfassung

Diese Arbeit untersucht die Verwendung von CUDA-fähigen Grafikprozessoren zur Beschleunigung der Ausführung von Clustering-Verfahren.

Für KMeans und DBScan werden Implementierungen mit CUDA vorgeschlagen, die massiv parallel arbeiten um die Rechenpower der Grafikkarte ausnutzen zu können.

Einige der verwendeten Techniken, die bei der Programmierung auf der Grafikkarte erst mit CUDA nutzbar werden, werden genauer erläutert.

Ausserdem wird eine einfache Indexstruktur vorgeschlagen, die für DBScan auf der GPU genutzt werden kann.

Bei Experimenten werden für beide Verfahren erhebliche Performance-Verbesserungen im Vergleich zur Ausführung auf der CPU aufgezeigt.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Grafikkarten werden verwendet um Echtzeit-3D Grafik in hoher Qualität zu ermöglichen, wie z.B. in Computerspielen, Simulationen und 3D-Modelling bzw. CAD Anwendungen.

Es findet ein regelrechtes Wettrennen der Hersteller statt, um immer mehr Leistung zu niedrigen Preisen anzubieten, vorallem für Computerspiele. Die Programmierbarkeit dieser Grafikkarten wird ständig verbessert, unter anderem für Spezialeffekte, und inzwischen auch für Physik-Simulation, wie z.B. Fahrzeuge und bewegliche Objekte in Spielen [1].

Wegen der kostengünstigen Rechenpower der Grafikkarten besteht bereits seit einiger Zeit ein Interesse daran, diese auch für nicht-grafische Anwendungen nutzbar zu machen (näheres hierzu in Abschnitt 1.3).

Die Programmierbarkeit hat nun mit dem Erscheinen von SDKs wie CUDA von NVIDIA und CTM/Stream von ATI ganz neue Dimensionen erreicht, da diese Zugriff auf Low-Level-Funktionalität der Grafikkarten erlauben, die bisher nicht nutzbar war.

Hier setzt diese Arbeit an. Es werden Verfahren vorgeschlagen und untersucht, um die Rechenpower von CUDA-fähigen Grafikkarten für die Clustering-Algorithmen KMeans und DBScan nutzbar zu machen.

[1] z.B. NVidia PhysX

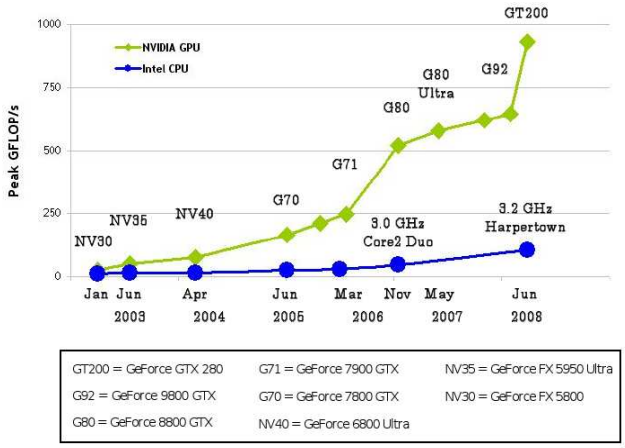

Abbildung 1.1: Floating-Point Operations per Second für CPU und GPU ([NVI08], Figure 1-1)

Da CUDA von allen NVIDIA Grafikkarten ab der GeForce8 Serie unterstützt wird, und auch Desktop PCs die nicht speziell für Spiele ausgelegt sind inzwischen serienmässig eine Grafikkarte haben, z.B. für grafische Effekte in der Benutzeroberfläche, ist diese Rechenpower in manchen Fällen also sogar ohne Neuanschaffungen und zusätzliche Kosten verfügbar.

1.2 Grafikkarten-Leistung

Durch den massiv parallelen Aufbau der GPU (Graphics Processing Unit) erreichen Grafikkarten enorme Leistungen im Vergleich zur CPU, wie in den Abbildungen 1.1 und 1.2 veranschaulicht wird.

1.3 GPGPU

General Purpose Computation on Graphics Processing Unit (GPGPU) bezeichnet die Verwendung des Grafikprozessors für Berechnungen die nichts mit der eigentlichen Grafikberechnung zu tun haben, z.B. wirtschaftliche oder technische Simulationen.

Hierbei kann die enormen Geschwindigkeit der GPU genutzt werden, die durch Einschränkung auf bestimmte Probleme möglich ist. Die GPU ist so entworfen, dass der Großteil der Transistoren für Rechenoperationen verwendet

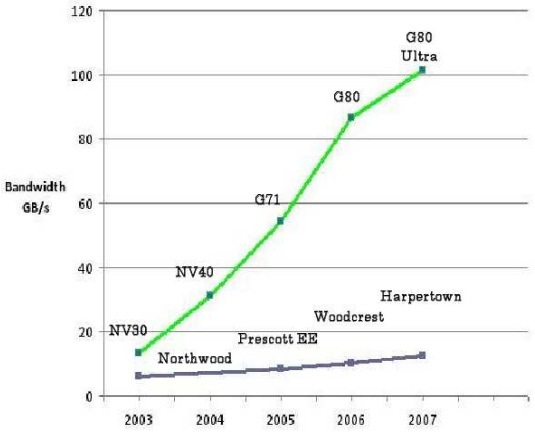

Abbildung 1.2: Memory Bandwidth für CPU und GPU ([NVI08], Figure 1-1)

wird und nicht für Steuerungsaufgaben und Caching, wie es auf CPUs der Fall ist.

Bei GPGPU ohne Verwendung vom CUDA müssen die zu bearbeitenden Daten zunächst entweder als Geometrie-Daten, oder in die Farb- und Transparenz-Kanäle einer Textur kodiert und in den Grafikkarten Speicher geladen werden. Texturen sind Bilddaten die gewissen Grössen-Einschränkungen unterliegen, sie werden bei grafischen Anwendungen verwendet, z.B. um Materialien darzustellen.

Zur Berechnung werden die programmierbaren Teile der Grafik-Pipeline verwendet, Vertex- oder Pixel-Shader. Für diese können Assembler-artige Programme geschrieben werden, die dann auf den Textur-Daten ausgeführt werden. Die Ergebnisse werden mittels Render-To-Texture wieder als Textur-Daten in den Hauptspeicher geschrieben.

1.4 CUDA

CUDA ist eine Programmierschnittstelle die darauf ausgelegt ist, eine CUDA-fähige Grafikkarte bei nicht-grafischen Anwendungen zu verwenden. Im Gegensatz zum bisherigen GPGPU mit Shadern ermöglicht CUDA den Zugriff auf Low-Level-Funktionalität der Grafikkarte, was einige Einschränkungen vom bisherigen Shader-basierten GPGPU aufhebt, und neue Möglichkeiten eröffnet :

Scatter und Gather Threads sind nicht an feste Ein- und Ausgabe-Adressen im Speicher gebunden wie beim Shader-basierten GPGPU, sondern können an beliebige, auch zur Laufzeit berechnete Adressen im Device Memory schreiben und davon lesen.

Branching Codeverzweigungen (If-Anweisungen,Schleifen,...) sind im Gegensatz zum bisherigen GPGPU ohne Performance-Einbussen möglich, solange alle Threads in einer Thread-Gruppen-Einheit (Warp=32 Threads) den selben Pfad nehmen.

Shared Memory auf jedem Multiprozessor steht ein 16 kByte grosser Speicherbereich zur Verfügung, auf den mit Register-Geschwindigkeit zugegriffen werden kann, und der von den Threads einer Thread-Gruppe gemeinsam genutzt werden kann.

ThreadSync es ist möglich alle Threads einer Thread-Gruppe zu synchronisieren, dies erlaubt z.B. Kommunikation und den Austausch von Zwischenergebnissen über Shared Memory.

AtomicOps es stehen verschiedene atomare Speicher-Zugriffs-Operationen zur Verfügung, d.h. mehrere Threads können parallel auf den selben Speicherbereich zugreifen, ohne das es zu Race-Conditions kommt.

Dies ermöglicht unter anderem effiziente Datenstrukturen wie Listen für mehreren Threads gemeinsam zu verwenden. Somit ist es nicht notwendig für jeden Thread einen eigenen Speicherbereich zu reservieren, man kann einen gemeinsam genutzten Bereich reservieren.

Dies ist besonders praktisch wenn die Anzahl der Ergebnisse die ein Thread einträgt nicht im voraus bekannt ist, z.B. wenn nur Treffer eingetragen werden. Somit braucht man deutlich weniger Speicher als für Ergebnis-Matrizen im GPGPU ohne CUDA.

Daten Man kann Speicherbereiche auf der Grafikkarte direkt reservieren und beschreiben, und muss nicht wie beim bisherigen GPGPU die Daten erst als Bilder bzw. Texturen kodieren. Dadurch entfallen bei CUDA auch einige Beschränkungen bezüglich Struktur und Aufbau der Daten.

Weniger umständliche Programmierung Die Programmierung ist in einigen Punkten wesentlich angenehmer, man programmiert bei der Verwendung von CUDA in einer erweiterten C Syntax, und nicht in den assemblerartigen Shader Programmiersprachen. Über einen CUDA-Emulator kann man die Ausführung der Programmabschnitte für die GPU auf der CPU simulieren und dabei Debug-Informationen ausgeben.

Einschränkungen Im Vergleich zur CPU-Programmierung unterliegt man bei Programmen für Grafikkarten (sowohl mit als auch ohne CUDA) gewissen Einschränkungen, unter anderem :

- Rekursion ist nicht möglich
- es ist nicht möglich während eines Programmabschnitts auf der GPU neue Speicherblöcke zu reservieren
- während eines Programmabschnitts auf der GPU können keine weiteren Threads gestartet werden
- während eines Programmabschnitts auf der GPU kann nicht auf das (Mainboard-)RAM oder auf die Festplatte zugegriffen werden
- SIMD (Single Instruction, Multiple Data) : alle Threads führen den selben Code aus
- das Device Memory auf der Grafikkarte hat zwar eine hohe Bandbreite, aber auch eine hohe Latenz-Zeit
- die Anzahl der Register für lokale Variablen ist beschränkt, und wird von parallel laufenden Threads gemeinsam genutzt, also muss man hier sehr sparsam sein

1.5 Clustering-Verfahren

In dieser Arbeit werden Verfahren vorgeschlagen und untersucht, um die Rechenpower von CUDA-fähigen Grafikkarten für die Clustering-Algorithmen KMeans und DBScan nutzbar zu machen.

KMeans KMeans ist ein Daten-partitionierender Clustering-Algorithmus, der wiederholt Punkte zu Clustern neu zugeordnet, und die Cluster-Zentren neu berechnet. Hierfür wird eine massive Parallelisierung der Cluster-Zuordnung vorgeschlagen, die bei Ausführung auf der Grafikkarte einen beträchtlichen Leistungsschub im Vergleich zu einer KMeans Umsetzung auf der CPU bringt.

DBScan DBScan ist ein Dichte-basierter Clustering-Algorithmus, bei dem Cluster über Kernpunkten ausgebreitet werden, die eine bestimmte Mindestanzahl an Nachbarpunkten in einer Epsilon-Umgebung haben. Hierfür wird eine massiv-parallele Variante vorgeschlagen, bei der mehrere Kernpunkte gleichzeitig ausgebreitet werden. Ausserdem wird eine einfache Indexstruktur vorgeschlagen, die trotz der Einschränkungen auf der Grafikkarte verwendet werden kann. Dies bringt gegenüber einer DBScan Umsetzung auf der CPU mit Index einen erheblichen Leistungsschub.

Kapitel 2

Related Work

[SDT08] : Eine parallele Implementierung des traditionellen K-Means Algorithmus wird beschrieben, bei dem der Datenaustausch zwischen GPU und CPU zwischen den Iterationen vermieden wird. Dabei wird noch kein CUDA, sondern noch Shader-basiertes GPGPU verwendet.

[LR08] Es wird ein Überblick über CUDA präsentiert, die Architektur der Grafikkarte erklärt, und es wird auf Missverständnisse und Vorurteile gegen GPU Programmierung eingegangen. Ausserdem werden zahlreiche Anwendungsgebiete vorgestellt.

[AC01] : Es wird eine Parallelisierung vom Data-Mining Algorithmus DBSCAN vorgeschlagen, die dazu geeignet ist, um die Geschwindigkeit bei hochdimensionalen Daten zu beschleunigen. Es wird eine generische, räumliche Indexstruktur verwendet.

[GGKM06] : Der Sortier-Algorithmus GPUTeraSort wird präsentiert. Dieser verwendet die GPU für Speicher- und Rechen-intensive Aufgaben, während die CPU verwendet wird um I/O und Ressourcen-Management durchzuführen. Der Algorithmus ist auch für Datenbanken mit Milliarden von Schlüsseln geeignet. Hierbei wurde kein CUDA, sondern Shader-basiertes GPGPU verwendet.

[SEKX98] : Eine generalisierte Variante von DBSCAN, genannt GDBSCAN, wird vorgeschlagen, die sowohl Punkte, als auch ausgedehnte räumliche Objekte clustern kann. Ausserdem werden verschiedene Anwendungen auf real-world Probleme demonstriert.

[BKP06] : Eine parallele Variante von DBSCAN wird vorgeschlagen, dabei wird der hierarchische Clustering Algorithmus OPTICS zur Partitionierung der Datenmenge verwendet, und es wird ausgenutzt, dass es in vielen Fällen lower-bounding Distanz-Funktionen gibt, die effizient berechnet werden können.

[GLW+04] : Es werden Algorithmen für die schnelle Berechnung einiger weit verbreiteter Datenbank Operationen präsentiert. Die Parallelität der GPUs wird verwendet, um die Datenbank Operationen effizient auszuführen. Hierbei wurde kein CUDA, sondern Shader-basiertes GPGPU verwendet.

[XJK99] : Eine parallele Variante von DBSCAN, genannt PDBSCAN, wird vorgeschlagen. PDBSCAN ist für mehrere, über ein Netzwerk verbundene Computer ausgelegt. Eine verteilte räumliche Indexstruktur, genannt dR-Tree, wird verwendet, um die Daten auf mehrere Rechner zu verteilen.

[LSS08] : Es wird ein neuer Similarity Join Algorithmus präsentiert, genannt LSS, der auf der GPU ausgeführt wird. Hierbei wurde eine frühe Version des CUDA SDKs verwendet.

[BPZN09] : Es wird eine parallele Umsetzung vom Similarity Join vorgestellt, die auf der GPU ausgeführt wird, und eine Indexstruktur verwendet. Hierbei wurde CUDA verwendet.

[CTZ06] : Es wird ein Algorithmus für scalable Clustering präsentiert, der den Grafikprozessor verwendet. Der Grundlegende Ansatz basiert auf KMeans, wobei Distanzberechnungen und Vergleiche auf der GPU ausgeführt werden. Hierbei wurde kein CUDA, sondern Shader-basiertes GPGPU verwendet.

[HH04] Es werden Methoden beschrieben um viele Varianten von iterativem Clustering Algorithmen, basierend auf Lloyds Algorithmus (oft als KMeans Algorithmus bezeichnet), mit Hilfe von programmierbaren Grafikkarten zu beschleunigen. Hierbei wurde kein CUDA, sondern Shader-basiertes GPGPU verwendet.

[OJL+07] : In diesem Bericht werden Forschungsergebnisse bezüglich General Purpose Computation auf Grafikarten (GPGPU) beschrieben und analysiert, und die dahinter liegenden Entwicklungen in Hard- und Soft-Ware werden erläutert.

[SAEA03] : Hier wird vorgeschlagen, den Verfeinerungs-Schritt bei Suchanfragen in räumlichen Datenbanken, insbesondere für Polygone, mit Hilfe der Grafikkarte zu beschleunigen. Hierbei wurde kein CUDA, sondern Shader-basiertes GPGPU verwendet.

[BSAEA04] : Komplexe Daten, wie räumliche Geometrie oder Protein-Strukturen, stellen vom Berechnungs-Aufwand her neue Herausforderungen für Datenbank-Systeme dar, die bisher hauptsächlich darauf ausgelegt waren, I/O Kosten zu minimieren. Es wird eine Architektur vorgestellt, um zu zeigen wie herkömmliche Grafikkarten verwendet werden können, um räumliche Anfragen zu beschleunigen. Bei ausgiebigen Experimenten mit real-world Datensätzen werden erhebliche Performance-Verbesserungen aufgezeigt. Hierbei wurde kein CUDA, sondern Shader-basiertes GPGPU verwendet.

Kapitel 3

Grundlagen

3.1 Architektur der Grafikkarte

Die GPU (Graphic Processing Unit) einer CUDA-fähigen Grafikkarte besteht aus einer Reihe von Multiprozessoren.

Multiprozessor Jeder Multiprozessor besteht aus 8 Prozessoren, so dass ein Multiprozessor in der Lage ist, 32 Threads in 4 Clock-Cycles zu bearbeiten.

SIMD Die 8 Prozessoren eines Multiprozessors arbeiten mit SIMD (Single Instruction, Multiple Data) parallel, d.h. sie führen zur gleichen Zeit die selbe arithmetische oder logische Anweisung aus, aber möglicherweise mit unterschiedlichen Daten.

Beispielsweise die NVIDIA GTX280 verfügt über 30 Multiprozessoren mit je 8 Prozessoren, also insgesamt 128 Prozessoren auf einer Grafikkarte. (Siehe auch [NVI08] Appendix A 1)

massiv parallel Grafikkarten arbeiten massiv parallel, bei 3D-Anwendungen wird oft ein Thread pro Pixel verwendet, was bei Auflösungen von 1024x768 und höher schon mehrere hundertausend Threads sind. Wegen SIMD, Thread-Gruppierung, und einer grossen Anzahl an Registern haben Threadwechsel einen weitaus geringeren Overhead im Vergleich zu Threads auf der CPU.

3.1.1 Multiprozessor

Die Architektur der GPU wird in Abbildung 3.1 verdeutlicht.

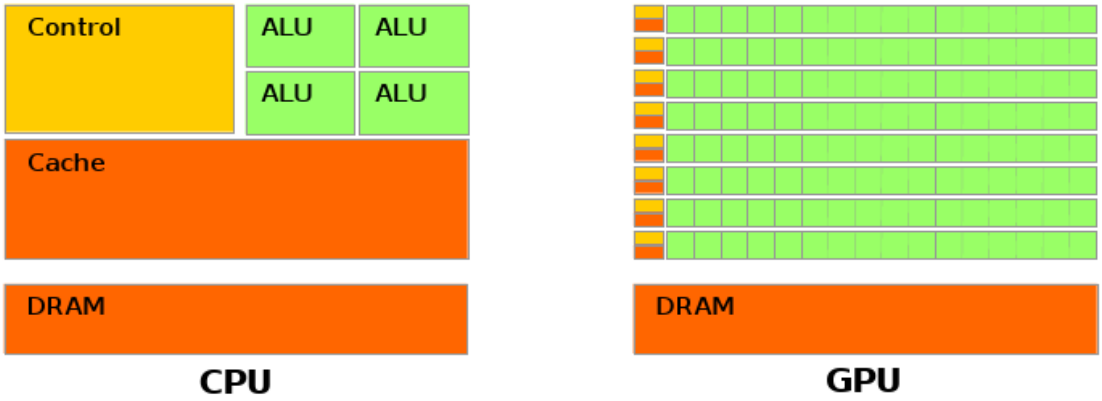

Abbildung 3.1: Die GPU verwendet mehr Transistoren für Daten-Bearbeitung ([NVI08], Figure 1-2)

Spezialisierung Der Ursprung der Rechenpower von Grafikkarten liegt darin, dass die GPU spezialisiert ist auf rechenintensive, hoch parallele Berechnung. Daher ist der Aufbau der GPU so gestaltet, dass mehr Transistoren für Daten-Bearbeitung statt für Caching und Kontrollstrukturen verwendet werden.

Genauer gesagt ist die GPU besonders gut für Probleme geeignet, die als daten-parallele Berechnung ausgedrückt werden können : Das selbe Programm wird parallel auf vielen Daten-Elementen ausgeführt, mit hoher arithmetischer Intensität (das Verhältnis von Arithmetischen Operationen zu Speicher-Zugriffs-Operationen).

Weil das selbe Programm für alle Daten-Elemente ausgeführt wird, ist der Bedarf nach anspruchsvollen Kontrollstrukturen (Schleifen, Verzweigungen...) wesentlich geringer. Wegen der hohen arithmetischen Intensität können Latenz-Zeiten bei Speicherzugriffen mit Berechnungen statt mit grossen Caches versteckt werden.

Die Präzision bei Floating-Point-Operationen entspricht dabei in etwa der von Single-Precision Floating-Point-Operationen auf der CPU. Siehe hierzu auch [NVI08] Appendix B für genaue Angaben von Abweichungen und [LR08] für einen Vergleich mit der Genauigkeit der CPU. Double-Precision ist ab Grafikkarten mit Computer-Ability 1.3 verfügbar.

3.1.2 Device Memory

Der grösste, aber auch langsamste Speicher auf der Grafikkarte ist das Device Memory. Es erreicht zwar im Vergleich zum Hauptspeicher auf dem Mainboard deutlich höhere Bandbreiten , jedoch hat es auch hohe Latenz-Zeiten (400-600 Clock-Cycles). [1]

Die Latenz-Zeiten können jedoch vom Thread-Scheduler auf der GPU versteckt werden, wenn es ausreichend unabhängige arithmetische Anweisungen gibt, die währenddessen ausgeführt werden können. Näheres zu Strategien hierzu wird in Abschnitt 3.3.9 erläutert.

Bei aktuellen Modellen stehen Speicher-Grössen im GigaByte-Bereich zur Verfügung, z.B. 1GB bei GTX280.

3.1.3 Shared Memory

Innerhalb jedes Multiprozessors stehen insgesamt 16KByte Shared Memory zur Verfügung.

Shared Memory Zugriff ist so schnell wie Register-Zugriffe.

Shared Memory kann von allen Prozessoren auf dem selben Multiprozessor gemeinsam genutzt werden, dies ermöglicht z.B. vor dem Rechnen das Einlesen von gemeinsam genutzten Daten ins Shared Memory, oder sogar den Datenaustausch zwischen den Prozessoren, hierzu mehr in Abschnitt 3.3.7.

3.1.4 Constant Memory

Weiterhin steht ein gecachter Constant Memory Bereich zur Verfügung, bei aktuellen Grafikkarten 64kb, auf den mit Register-Geschwindigkeit zugegriffen werden kann, solange es keinen Cache-Miss gibt.

Der Constant Memory kann während der Ausführung eines Programm-Abschnitts auf der GPU nicht beschrieben werden.

[1] [NVI08] Section 5.1.1.3

3.2 GPGPU

General Purpose Computation on Graphics Processing Unit (GPGPU) bezeichnet die Verwendung des Grafikprozessors für Berechnungen die nichts mit der eigentlichen Grafikberechnung zu tun haben, z.B. wirtschaftliche oder technische Simulationen.

3.2.1 Shader-basiertes GPGPU

Daten als Texturen codiert Bei GPGPU ohne Verwendung vom CUDA müssen die zu bearbeitenden Daten zunächst als Textur oder Geometrie-Daten kodiert werden. Texturen sind Bilddaten die bei grafischen Anwendungen z.B. verwendet werden um Materialien darzustellen.

Texturen unterliegen gewissen Grössen-Einschränkungen :

- Die Seitenlänge muss eine 2er Potenz sein
- es gibt ein Limit für die maximale Seitenlänge, abhängig von dem Modell der Grafikkarte, bei aktuellen Karten 4096 Pixel

Die Daten werden in die Farb- und Transparenz-Kanäle kodiert.

Shader Teile der Grafikpipeline auf Grafikkarten sind seit einiger Zeit programmierbar :

- Vertex-Shader um Positionen der Eckpunkte von Polygonen zu verändern, z.B. verzerren. Dabei können jedoch keine Eckpunkte gelöscht oder neu hinzugefügt werden.
- Fragment- bzw. Pixel-Shader können die gezeichneten Pixel verändern, um realistischere Material- oder Oberflächen-Eigenschaften zu simulieren.
- neuere Grafikkarten unterstützen ausserdem Geometrie-Shader, die es ermöglichen auf der Grafikkarte neue Geometrie zu erzeugen, z.B. Objekte zu vervielfältigen, oder Polygone für Schatten zu erzeugen.

Für diese Shader können Assembler-artige Programme geschrieben werden, die oft auch Shader genannt werden. Diese Programme werden für die Berechnung bei GPGPU verwendet.

Ausgabe als Render-To-Texture Um die Ergebnisse zurück an die CPU zu geben, werden diese nicht auf dem Bildschirm dargestellt, sondern mittels Render-To-Texture in einen Pixel-Buffer im Device-Memory geschrieben.

Dieser Pixel-Buffer zählt als Textur, und unterliegt den gleichen Grössen-Einschränkungen. Er kann aus dem Device-Memory zurück in den Hauptspeicher transferiert und ausgewertet werden.

Einschränkungen von Shader-basiertem GPGPU

- kein direktes Scatter-Write, jedem Thread ist eine feste Ausgabe-Adresse zugeordnet. (Dies kann zwar in einem gewissen Rahmen Umgangen werden, was jedoch zusätzliche Rechenzeit kostet)
- Branching (Verzweigungen des Ausführungs-Pfades, z.B. IF-Anweisungen oder Schleifen) ist nur begrenzt und ineffizient möglich, z.B. in dem alle Pfade berechnet werden, und dann mittels Z-Buffer die nicht benötigten verworfen werden.
- Threads können nicht miteinander kommunizieren

3.2.2 Einschränkungen gegenüber Programmierung auf der CPU

Da Grafikkarten hochspezialisiert sind, unterliegt die Programmierung darauf gewissen Einschränkungen.

Diese Einschränkungen betreffen sowohl Shader-basiertes GPGPU als auch CUDA.

keine Rekursion Alle Funktionsaufrufe werden ge-inlined, d.h. der Code in der Funktion wird beim Compilen an die Stelle kopiert an der die Funktion aufgerufen wird. Dies macht u.a. Rekursion unmöglich, was bei der Umsetzung einiger Datenbank-Algorithmen Probleme bereiten kann, insbesondere wenn Index-Strukturen verwendet werden.

keine neuen Threads Während eines Programmabschnitts auf der GPU können keine weiteren Threads gestartet werden, die Anzahl der Threads wird beim Start eines Programmabschnitts vorgegeben, und kann nicht mehr verändert werden bis der Programmabschnitt beendet ist.

keine dynamische Speicher-Allokation Es ist nicht möglich neue Speicherbereiche zu reservieren während auf der GPU ein Programmabschnitt ausgeführt wird. Bei Verwendung von CUDA kann man allerdings mit Hilfe von einem Zähler der via Atomic Ops inkrementiert wird zuvor reservierten Speicher-

bereich, wie etwa eine Liste, nach und nach füllen, und muss nicht vorher festlegen welcher Thread wie viel davon verwenden darf.

kein Zugriff auf Hauptspeicher oder Festplatte Während eines Programmabschnitts auf der GPU kann nicht auf das (Mainboard-)RAM oder auf die Festplatte zugegriffen werden, auch Netzwerk Kommunikation oder Vergleichbares ist nicht möglich, die Grafikkarte ist während eines Programmabschnitts von der Aussenwelt abgeschnitten.

Branching ist nur begrenzt möglich Branching (Verzweigungen des Ausführungs-Pfades, z.B. IF-Anweisungen oder Schleifen) ist bei Shader-basiertem GPGPU nur begrenzt und ineffizient möglich, z.B. in dem alle Pfade berechnet werden, und dann mittels Z-Buffer die nicht benötigten verworfen werden. Bei CUDA ist effizientes Branching möglich ohne alle Pfade zu berechnen, solange alle Threads eines Warps (32 Threads) dem selben Ausführungs-Pfad folgen. Sonst müssen auch hier die verschiedenen Pfade nacheinander ausgeführt werden.

SIMD Wegen der SIMD (Single Instruction, Multiple Data) Architektur und dem teuren Branching führen meistens alle Threads den selben Code aus.

begrenzte Registeranzahl für lokale Variablen Die Anzahl der für lokale Variablen zur Verfügung stehenden Register ist zwar sehr gross, aber dennoch begrenzt. Da viele Threads gleichzeitig laufen, und jeder Thread seine eigenen Register hat, werden die Register schnell knapp. Je weniger Lokale Variablen von jedem Thread verwendet werden, desto mehr Threads können parallel ausgeführt werden. Hier kann man durch Verwendung von zur Compile-Zeit bekannten Konstanten und Preprozessor-Macros einiges einsparen.

Device Memory : hohe Bandbreite aber hohe Latenz-Zeit Das Device Memory auf der Grafikkarte hat zwar eine sehr hohe Bandbreite, jedoch auch eine hohe Latenz-Zeit, was beim Entwurf der Programme berücksichtigen werden muss. Hohe arithmetischer Intensität (d.h. viele Arithmetischen Operationen im Vergleich zu Speicher-Zugriffs-Operationen) ist von Vorteil, und die Latenz-Zeiten können vom Thread-Scheduler auf der GPU versteckt werden, wenn es ausreichend unabhängige arithmetische Anweisungen gibt, die während der Wartezeiten ausgeführt werden können.

3.3 CUDA

3.3.1 Überblick

CUDA ist eine Programmierschnittstelle, die dafür ausgelegt ist, eine CUDA-fähige Grafikkarte bei nicht-grafischen Anwendungen zu verwenden. Das SDK ist kostenlos auf der NVIDIA Webseite verfügbar [2]. Es lässt sich unter Windows(tm) und unter Linux verwenden. Die Programmierung findet hauptsächlich in einer erweiterten C Syntax statt, es lässt sich jedoch auch mit C++ verbinden, ausserdem steht u.a. ein Binding für die Scriptsprache Python zur Verfügung [3] .

3.3.2 Aufbau eines CUDA Programms

Bei einem CUDA Programm steuert die CPU den generellen Ablauf, und lagert die rechenintensiven Programm-Abschnitte (sog. Kernel) auf die Grafikkarte aus.

Haupt-Programm Das Haupt-Programm wird auf der CPU ausgeführt, legt mit Funktionen aus der CUDA API (Application Programming Interface) Speicherbereiche im Device Memory an, und befüllt diese mit Daten. Anschliessend wird der Kernel auf der Grafikkarte gestartet, und auf dieser parallel ausgeführt. Die CPU wartet bis dieser vollständig abgearbeitet wurde. Danach können die Ergebnisse aus dem Device Memory zurück gelesen werden, und noch weiter-verarbeitet oder ausgegeben werden. Pseudocode 1 veranschaulicht dies.

Algorithm 1 MAIN()

```
1: repeat
2:     Speicherbereiche im Device Memory anlegen
3:     Daten ins Device Memory hochladen
4:     Kernel starten
5:     Ergebnisse aus dem Device Memory zurück lesen
6:     Ergebnisse auswerten bzw. ausgeben
7: until fertig
```

[2] `http://www.nvidia.com/object/cuda_get.html`
[3] `http://mathema.tician.de/software/pycuda`

Oft wird auch eine Schleife verwendet um den Kernel mehrfach auszuführen und Zwischenergebnisse auf der CPU zu verarbeiten bzw. zusammenzufassen.

Es können auch nacheinander mehrere verschiedene Kernel ausgeführt werden.

Konstanten und Preprozessor Makros Die Compile-Zeit für die Programme ist im allegemeinen Recht gering, so dass es durchaus vertretbar ist, für zusätzliche Performance durch Compiler-Optimierung (z.B. Loop-Unrolling oder das Einsparen von Lokalen Variablen bzw. Registern) gewisse Parameter und Werte wie z.B. Epsilon oder die Anzahl der Datenpunkte nicht dynamisch einzulesen, sondern als zur Compilezeit bekannte Konstanten oder Preprozessor-Makros zu hardcoden. Bei einer Benutzer-freundlichen Umsetzung des Programms mit einer Grafischen Oberfläche ist es durchaus möglich vor Programmstart die Werte in den Code einzufügen und das eigentlichen CUDA Programm per Commandozeilen-Befehl neu zu compilen.

3.3.3 Emulator

Es ist ein Emulator verfügbar mit dem man die Ausführung eines CUDA Programms auf der CPU simulieren kann.

Es wird keine CUDA-fähige Grafikkarte benötigt um Programme mit dem Emulator auszuführen, so kann man z.B. auch das Programme entwickeln und die grundlegende Funktionalität testen, wenn keine solche Karte zur Verfügung steht.

Geschwindigkeit Allerdings ist der Emulator nur brauchbar um die Funktionalität zu überprüfen, für Geschwindigkeits-Messungen ist er nicht geeignet, da viele Threads gestartet werden, und der Thread-Wechsel auf der CPU wesentlich teurer ist als auf der Grafikkarte.

Debug-Ausgaben Man hat bei Ausführung über den Emulator auch die Möglichkeit Debug-Informationen auszugeben, oder den Code mit einem Debugger schrittweise auszuführen.

3.3.4 Thread-Gruppen und Warps

Threads werden in CUDA zu Gruppen zusammengefasst.

Warp Die kleinste Gruppeneinheit ist ein sogenannter Warp von 32 Threads. Alle Threads in einem Warp werden auf den 8 Prozessoren eines Multiprozessor mit SIMD quasi gleichzeitig ausgeführt, mittels Pipelining in 4 Prozessorzyklen.

Thread-Gruppen Mehrere dieser Warps werden zu grösseren Thread-Gruppen zusammengefasst, alle Threads in einer Thread-Gruppe nutzen einen Shared Memory Bereich gemeinsam, und können synchronisiert werden, d.h. es wird gewartet bis alle Threads in der Thread-Gruppe einen bestimmten Arbeitsschritt erreicht haben.

Jede Thread-Gruppe wird einem Multiprozessor zugeordnet, also sollte es mindestens soviele Thread-Gruppen geben, wie es Multiprozessoren auf der verwendeten Grafikkarte gibt.

3.3.5 Kernel

Die Programm-Abschnitte, die parallel auf der GPU ausgeführt werden, werden Kernel genannt.

- Die CPU steuert die Kernel-Aufrufe.
- Vor dem Kernel-Aufruf werden die Daten vom CPU-RAM ins Device Memory auf der GPU transferiert.
- Ein Programm kann aus einem, oder auch aus mehreren Kernel-Aufrufen bestehen.
- Datenblöcke im Device Memory bleiben zwischen Kernel-Aufrufen erhalten, Daten im Shared Memory gehen jedoch verloren.
- Die Ausführung eines Kernels ist Asynchron, d.h. die CPU kann andere Aufgaben bearbeiten während der Kernel ausgeführt wird.

WatchDog Unter Windows(tm) ist die Ausführungszeit eines Kernels beschränkt wenn er auf der Primären Grafikkarte ausgeführt wird, da ein sog. WatchDog den Kernel abbricht, wenn dieser nach ein paar Sekunden nicht beendet ist. Dem kann man entgegenwirken in dem man das Programm entweder in mehrere kleine Schritte unterteilt, oder eine zweite Grafikkarte verwendet. Unter Linux kann man den Watchdog zusätzlich umgehen in dem man ohne grafische Oberfläche startet, oder ihn in den Systemeinstellungen deaktiviert.

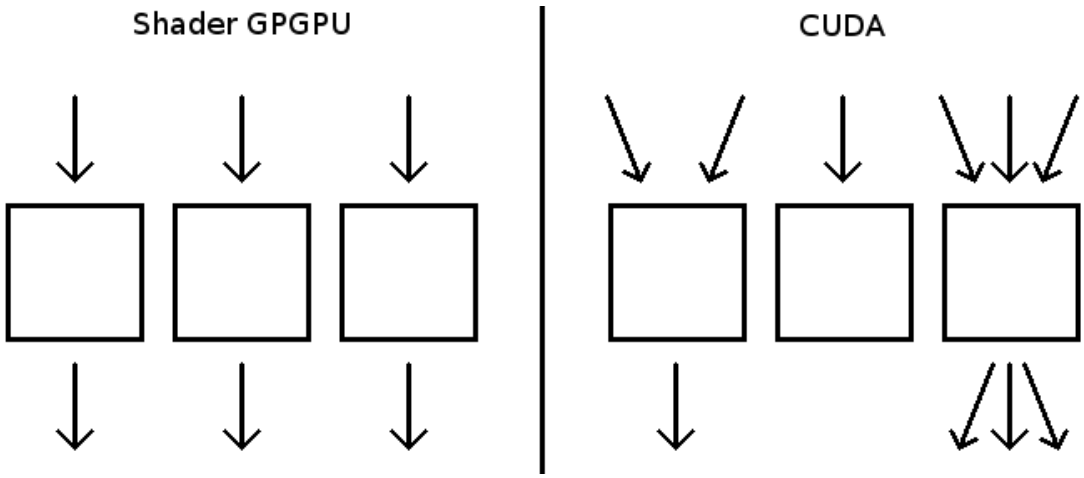

Abbildung 3.2: Scatter and Gather

3.3.6 Vorteile von CUDA gegenüber Shader-basiertem GPGPU

Durch die Verwendung von CUDA werden einige der Einschränkungen beim Shader-basiertem GPGPU auflöst.

Scatter und Gather Threads sind nicht mehr an feste Ein- und Ausgabe-Adressen im Speicher gebunden wie beim bisherigen GPGPU.

- Jeder Thread kann auf beliebige Speicherbereiche im Device Memory schreiben und davon lesen.
- Jeder Thread kann mehrere Schreibzugriffe und Lesezugriffe ausführen.
- Die Adressen können zur Laufzeit berechnet werden, z.B. für Array zugriff.

Siehe Abbildung 3.2.

Schreibkonflikte Wenn mehrere Threads eines Warps auf die gleiche Speicher-Adresse im Device Memory oder im Shared Memory schreiben, ist zwar die Anzahl und Reihenfolge der tatsächlich ausgeführten Schreibzugriffe nicht definiert, aber es ist garantiert, dass mindestens einer erfolgreich ist.

Thread Sync und Kommunikation zwischen Threads Es ist möglich die Threads in einer Gruppe zu synchronisieren, d.h. zu warten bis alle Threads der Gruppe einen gewissen Punkt im auszuführenden Code erreicht haben, bevor das Programm weiter ausgeführt wird.

Shared Memory Threads können lesend und schreibend auf Speicherbereiche im Shared Memory zugreifen, die von allen Threads in einem Threadblock gemeinsam genutzt werden. Dies ermöglicht z.B. Kommunikation und die Verwendung von gemeinsam errechnete Zwischenergebnissen von allen anderen Threads.

Näheres hierzu in Abschnitt 3.3.7.

Branching Verzweigungen (Branching) sind besser als bei Shader-basiertem GPGPU möglich. Was bei Shader-GPGPU umständlich mit Culling oder Z-Buffer gelöst werden musste, und mit zusätzlichen Kosten verbunden war, kann unter CUDA direkt mit IF-Anweisungen gelöst werden, was einige unnötige Berechnungen und Unannehmlichkeiten bei der Programmierung erspart.

Die Ausführung ist effizient solange alle Threads eines Warps dem gleichen Ausführungs-Pfad folgen, ansonsten muss die Verzweigung sequentiell ausgeführt werden, d.h. erst der eine Pfad, dann der andere.

Siehe Abbildung 3.3.

Atomare Operationen Es stehen spezielle atomare Operationen für Zugriffe auf das Device-Memory zur Verfügung um Race-Conditions zu vermeiden, wenn mehrere Threads auf die selbe Adresse schreiben. Diese ermöglichen effizientere Datenstrukturen, wie z.B. Listen in die nur positive Ergebnisse bzw. Treffer eingetragen werden. So ist es auch möglich das ein Thread mehrere Ergebnisse einträgt. Bei neueren Modellen können diese Operationen auch für Speicherbereiche im Shared Memory verwendet werden.

Beim Shader-basierten GPGPU war es oft nötig eine Ergebnis-Matrix zu verwenden, mit einem Eintrag pro Thread, oder zwischendurch von der CPU die Listenstrukturen erweitern zu lassen, was Wartezeiten mit sich bringt, und es schwierig macht für einen Thread mehr als ein Ergebnis auszugeben.

Siehe Abbildung 3.4.

Näheres zu Atomic Ops in Abschnitt 3.3.8.

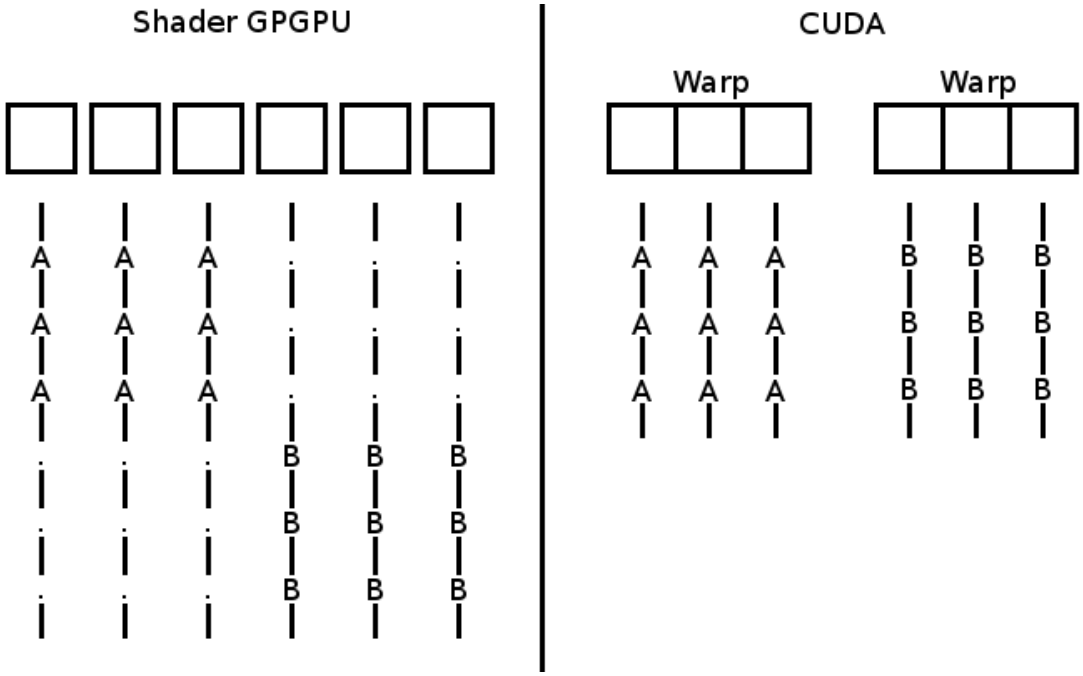

Abbildung 3.3: effizientes Branching mit Warps

Weniger Umständliche Programmierung Die Programmierung ist in einigen Punkten wesentlich angenehmer, so kann man statt in den assemblerartigen Shader Programmiersprachen in CUDA eine Erweiterte C Syntax verwenden.

Emulator Über den Emulator kann man die Ausführung der Programmabschnitte für die GPU simulieren und dabei Debug-Informationen ausgeben.

keine Daten-Kodierung als Bilder Man muss Daten nicht als Bilder bzw. Texturen kodieren, und ist nicht an 2er Potenzen bei der Textur-Grösse und an Farb-Kodierungen gebunden, sondern kann Speicherbereiche frei zuweisen.

Kontrolle über Optimierungen Man hat Kontrolle über einige Optimierungen, so kann man z.B. direkt angeben das auf eine Schleife mit zur Compile-Zeit bekannter Länge Loop-unrolling angewendet werden soll. D.h. beim Compilen wird der Schleifen-Inhalt kopiert und die Werte der Lauf-Variable direkt als Konstante eingesetzt.

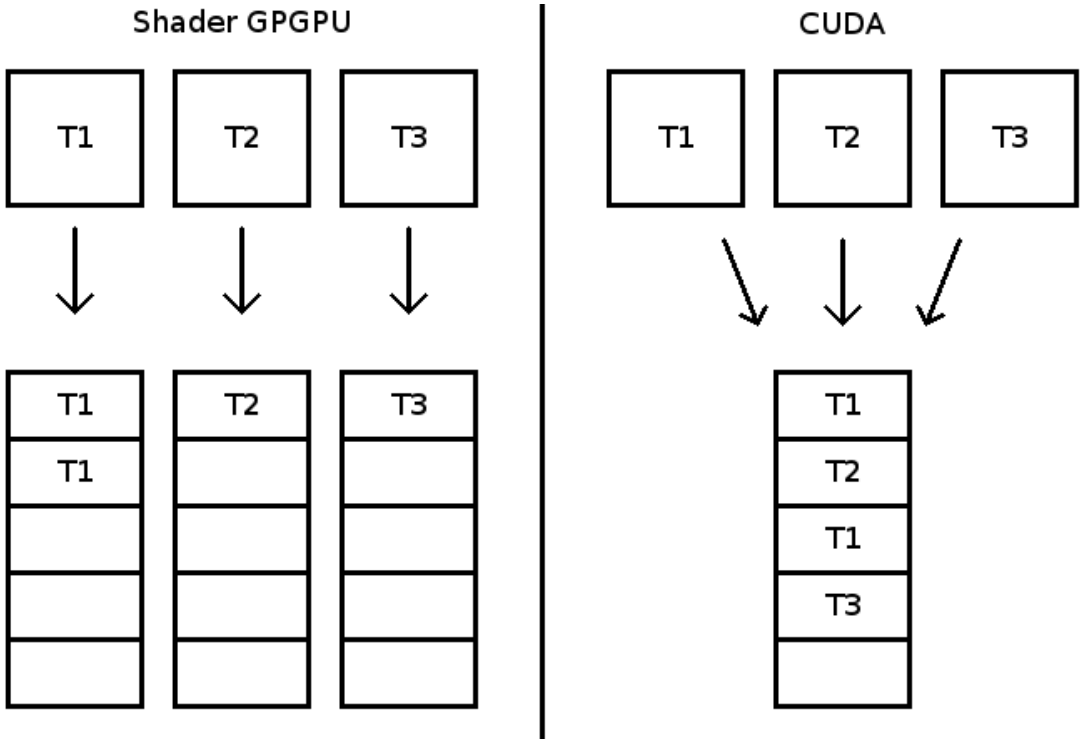

Abbildung 3.4: Atomic-Ops für Listen

Das erhöht zwar die Grösse des compilierten Codes, dieser ist jedoch schneller, weil keine Schleifen-Variable angelegt und zur Laufzeit erhöht werden muss.

3.3.7 Kommunikation zwischen Threads

Es ist möglich Threads in eine Gruppe zu synchronisieren, d.h. zu warten bis alle Threads der Gruppe einen gewissen Punkt erreicht haben, bevor das Programm weiter ausgeführt wird.

Zusammen mit dem Shared Memory, der von allen Threads einer Gruppe gemeinsam genutzt, ermöglicht dies die Kommunikation zwischen Threads.

Daten einlesen Ein häufig verwendetes Muster ist hier das Einlesen von gemeinsam genutzten Daten ins Shared Memory vor den Berechnungen, dies ist wichtig, da das Device Memory hohe Latenz-Zeiten hat. Pseudocode 2 verdeutlicht dies. (Siehe auch [NVI08] Kapitel 5.1.2)

Algorithm 2 KERNEL()

```
1: repeat
2:     Daten vom Device Memory ins Shared Memory laden
3:     Alle Threads im Block synchronisieren
4:     Daten im Shared Memory verarbeiten
5:     Ergebnisse zurück ins Device Memory schreiben
6:     Alle Threads im Block synchronisieren
7: until fertig
```

Nach dem Laden der Daten werden die Threads synchronisiert, damit sichergestellt ist, dass alle Daten fertig ins Shared Memory geladen wurden, bevor die Threads beginnen diese zu verarbeiten.

Am Ende eines Schleifen-Durchlaufs wird nochmals synchronisiert, um sicherzustellen, dass kein Thread mehr auf die Daten im Shared Memory zugreift, bevor dort andere Daten geladen werden.

Zwischenergebnisse Auf diese Art ist es auch möglich Zwischenergebnisse auszutauschen, so dass nicht alles mehrfach berechnet werden muss.
Siehe Abbildung 3.5.

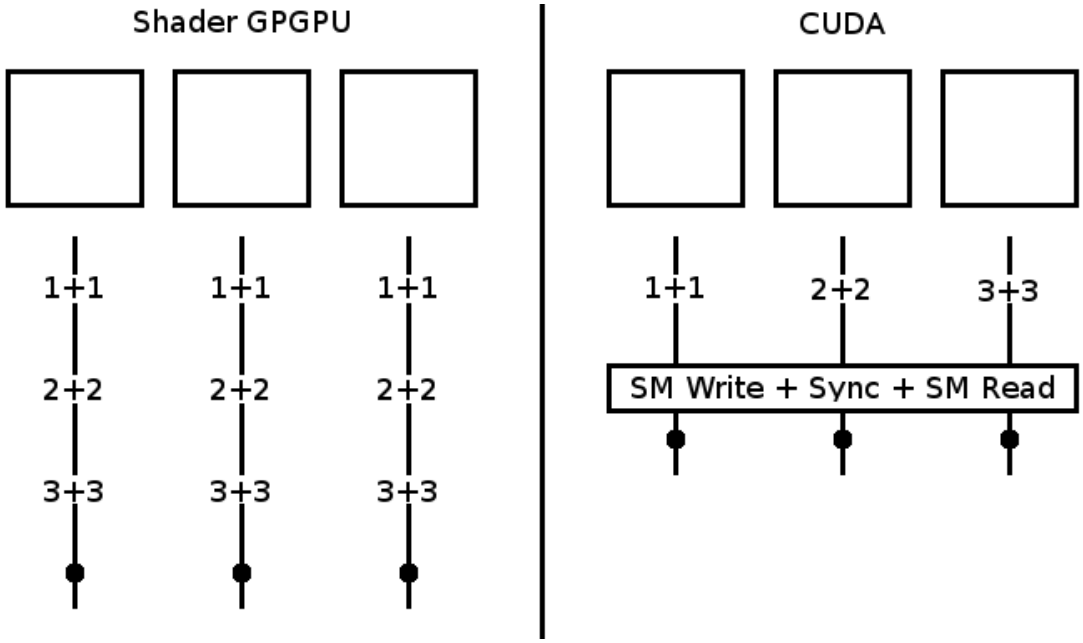

Abbildung 3.5: Thread-Kommunikation : Zwischenergebnisse

Signale Wenn mehrere Threads eines Warps auf die gleiche Speicher-Adresse im Device Memory oder im Shared Memory schreiben, ist zwar die Anzahl und Reihenfolge der tatsächlich ausgeführten Schreibzugriffe nicht definiert, aber es ist garantiert, dass mindestens einer erfolgreich ist.

Dies kann man verwenden um anderen Threads ein bestimmtes Ereignis zu signalisieren, z.B. wenn mehrere Threads eine Datenmenge durchsuchen, kann man den anderen Threads signalisieren dass das Gesuchte gefunden wurde, so dass diese aufhören können zu suchen.

Hierzu wird die Speicherstelle im Shared Memory zuerst mit 0 initialisiert, und alle Threads synchronisiert, bei einem Treffer wird eine 1 an die Stelle geschrieben, und alle Threads überprüfen regelmässig den Wert an der Stelle. Da auf Shared Memory mit Registergeschwindigkeit zugegriffen werden kann ist dieser Check sehr schnell.
Siehe Abbildung 3.6.

Atomic Ops Ausserdem stehen atomare Operationen zur Verfügung, mit denen gemeinsamen Berechnung, wie z.B. die Gesamt-Anzahl der Treffer, durchgeführt werden können. Näheres hierzu in Abschnitt 3.3.8

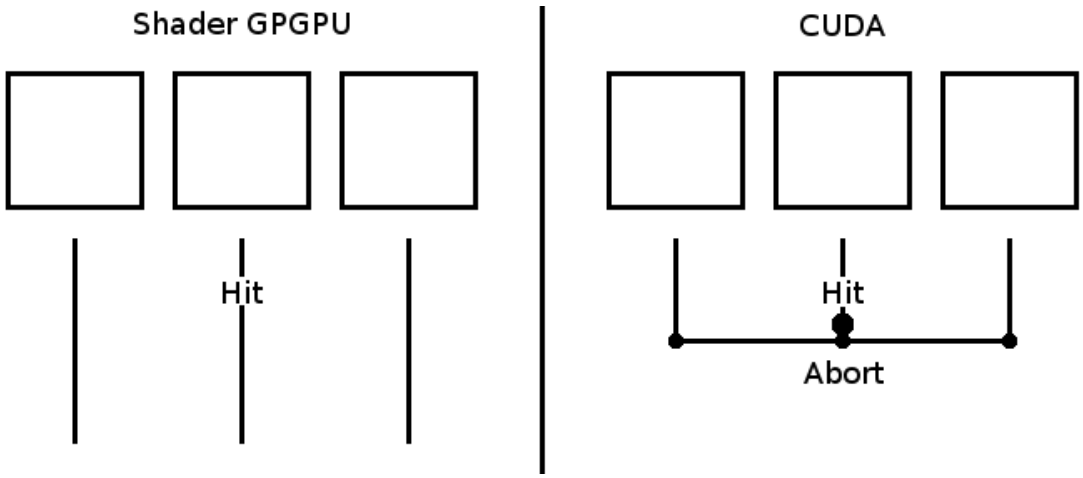

Abbildung 3.6: Thread-Kommunikation : Signale

3.3.8 Atomic Ops

Alle schritte einer atomaren Operationen werden als atomare Transaktion durchgeführt.

Einige der atomic Ops unterstützen nur signed und unsigned Int als Typen, keine Floats.

atomicAdd T atomicAdd (T* address, T val);

liest old von address, berechnet (old+val) und schreibt das Ergebnis zurück nach address, und liefert old als Funktionsergebnis zurück.

atomicSub T atomicSub (T* address, T val);

liest old von address, berechnet (old-val) und schreibt das Ergebnis zurück nach address, und liefert old als Funktionsergebnis zurück.

atomicExch T atomicExch (T* address, T val);

liest old von address, schreibt val nach address, und liefert old als Funktionsergebnis zurück. (unterstützt float)

atomicMin T atomicMin (T* address, T val);

liest old von address, berechnet Min(old-val) und schreibt das Ergebnis zurück nach address, und liefert old als Funktionsergebnis zurück.

atomicMax T atomicMax (T* address, T val);

liest old von address, berechnet Max(old-val) und schreibt das Ergebnis zurück nach address, und liefert old als Funktionsergebnis zurück.

atomicInc T atomicInc (T* address, T val);

liest old von address, berechnet ((old $\geq$ val) ? 0 : (old+1)), schreibt das Ergebnis zurück nach address, und liefert old als Funktionsergebnis zurück. (unterstützt nur unsigned int) praktisch für Zähler und Listen.

atomicDec T atomicDec (T* address, T val);

liest old von address, berechnet (((old == 0) or (old > val)) ? val : (old-1)) und schreibt das Ergebnis zurück nach address, und liefert old als Funktionsergebnis zurück. (unterstützt nur unsigned int)

atomicCAS T atomicCAS (T* address, T compare, T val);

liest old von address, berechnet ((old == compare) ? val : old) und schreibt das Ergebnis zurück nach address, und liefert old als Funktionsergebnis zurück. (CompareAndSwap)

atomicAnd T atomicAnd (T* address, T val);

liest old von address, berechnet BitwiseAnd(old,val) und schreibt das Ergebnis zurück nach address, und liefert old als Funktionsergebnis zurück.

atomicOr T atomicOr (T* address, T val);

liest old von address, berechnet BitwiseOr(old,val) und schreibt das Ergebnis zurück nach address, und liefert old als Funktionsergebnis zurück.

atomicXor T atomicXor (T* address, T val);

liest old von address, berechnet BitwiseXor(old,val) und schreibt das Ergebnis zurück nach address, und liefert old als Funktionsergebnis zurück.

Atomic Ops auf Shared Memory Auf neueren Grafikkarten, die über Compute Capability 1.3 (z.B. GTX280 und GTX260) verfügen, ist es sogar möglich die Atomic Ops für Speicherbereiche im Shared Memory zu verwenden.

3.3.9 Device Memory

Da das Device Memory zwar hohe Bandbreite, aber auch hohe Latenz-Zeiten hat, muss man dies beim Entwurf der Algorithmen berücksichtigen.

Latenz-Zeiten Man sollte beim Entwurf des Programms beachten, dass On-Chip Memory (insbes. Shared Mem) deutlich geringere Latenz-Zeiten und noch höhere Bandbreite als Device Memory ermöglicht.

Eine mögliche Strategie hierzu ist es die verwendeten Daten vor der Berechnung erst ins Shared Memory zu laden. Dies wird in Abschnitt 3.3.7 näher beschrieben. Dies ist insbesondere wenn man auf Teile davon mehrfach zugreift von Vorteil.

Die Latenz-Zeiten können vom Thread-Scheduler auf der GPU versteckt werden, wenn es ausreichend unabhängige arithmetische Anweisungen gibt, die währenddessen ausgeführt werden können. Es ist auch von Vorteil wenn möglichst viele unabhängige Thread-Gruppen aktiv sind, so dass die GPU während eine Thread-Gruppe auf den Abschluss einer Lese-Operation wartet bereits eine andere Thread-Gruppe ausführen kann.

Kapitel 4

KMeans

4.1 Überblick

K-Means ist ein partitionierendes Clustering Verfahren, das eine Menge von Objekten in eine vorgegebene Anzahl (k) Cluster einteilt. (Siehe auch [Ste56])

- Im ersten Schritt wird für jeden Cluster ein zufälliges Zentrum gewählt.
- Jedes Objekt wird dem nächstgelegenen Cluster zugeordnet.
- Für jeden Cluster wird das Zentrum neu berechnet.
- Schritt 2 und 3 werden solange wiederholt, bis sich nichts mehr ändert.

4.2 Parallelisierung

Um die Rechenpower von CUDA-fähigen Grafikkarten für den K-Means-Algorithmus nutzen zu können, führen wir die Abstands-Berechnung zu den Cluster-Zentren und die Auswahl des nächsten Clusters parallel auf der Grafikkarte aus. Dabei wird für jeden Datenpunkt ein eigener Thread gestartet.

Der Abstand zum nächstgelegenen Cluster, der dabei als Zwischenergebnis anfällt, wird gespeichert und später bei der Kostenberechnung verwendet.

Auf der CPU werden in jedem Schritt nach der Ausführung des Kernels die Neuen Cluster-Zentren und die aktuellen Kosten berechnet.

Die Kosten werden als Summe der quadrierten Abstände aller Punkte zum jeweils ausgewählten Cluster-Zentrum berechnet.

Der Algorithmus wird solange wiederholt bis sich die Kosten nicht mehr verändern.

Pseudocode 3 und 4 veranschaulichen dies.

Algorithm 3 KMeansMain()

```
1: gPunktKoordinaten = DatenEinlesen()
2: gClusterZentrumKoordinaten = ZufaelligeZentrenWaehlen()
3: UploadToDeviceMemory(gPunktKoordinaten)
4: UploadToDeviceMemory(gClusterZentrumKoordinaten)
5: letzteKosten = +∞
6: repeat
7:     StarteKernel(KMeansKernel,ANZAHLDATENPUNKTE)        {ein
       Thread pro Datenpunkt}
8:     gBesterAbstand = DownloadFromDeviceMemory()
9:     gBesterCluster = DownloadFromDeviceMemory()
10:    kosten = BerechneKosten()
11:    kostenUnterschied = abs(letzteKosten - kosten)
12:    letzteKosten = kosten
13:    gClusterZentrumKoordinaten = BerechneNeueZentren()
14:    UploadToDeviceMemory(gClusterZentrumKoordinaten)
15: until kostenUnterschied < kCostDiffEpsilon
```

Algorithm 4 KMeansKernel

```
1: p = gPunktKoordinaten[threadIndex] {eigene Punkt-Koordinaten einlesen}
2: besterAbstand = +∞
3: besterCluster = nil
4: for i = 1 to K do
5:     q = gClusterZentrumKoordinaten[i] {Cluster-Zentrum Koordinaten einlesen}
6:     d = Abstand(p,q)
7:     if d < besterAbstand then
8:         besterAbstand = d
9:         besterCluster = i
10:    end if
11: end for
    {Ergebnis rausschreiben}
12: gBesterAbstand[threadIndex] = besterAbstand
13: gBesterCluster[threadIndex] = besterCluster
```

Kapitel 5

KMeans Evaluation

5.1 Messaufbau

Alle hier aufgeführten Messungen wurden auf dem selben Rechner durchgeführt, dessen Spezifikationen sind wie folgt :

- OS : Windows XP(tm) Service Pack 2
- CPU : Intel(R) Core(tm)2 Duo CPU E4500 2.20 Ghz
- Grafikkarte : Gainward GeForce GTX280 1GB GDDR3 PCIe 2.0
- Mainboard : Asus P5N-D nForce 750i SLI S775
- RAM : 2GB (2 x 1GB Corsair CM2X1024-6400 5-5-5-12 800Mhz)
- Festplatte : Western Digital SATA 250GB WD2500KS

5.2 Datensatz

Die Datensätze auf denen getestet wurde sind zufalls-generierte Cluster von Punkten, dabei ist die Selektivität der Dimensionen gleich.

Am Anfang werden die Cluster Zentren mit zufälligen Koordinaten im Bereich 0.2 bis 0.8, und einem Radius im Bereich 0.02 bis 0.15 erzeugt. Für jeden zu erzeugenden Punkt wird zufällig einer der Cluster ausgewählt, und zufällige Koordinaten innerhalb des Radius gewählt.

5.3 CPU-KMeans

Die CPU Implementierung von KMeans die zum Vergleich verwendet wurde ist recht einfach gehalten und verwendet am Anfang die selben zufällig ausgewählten Cluster-Zentren wie die Messung mit GPU-KMeans.

Sie wird in Pseudeocode 5 und 6 veranschaulicht.

Die CPU Implementierung von KMeans ist single-threaded, d.h. sie nutzt nur einen der beiden zur Verfügung stehenden Prozessor-Cores aus.

Beide Cores auszunutzen würde jedoch höchstens zu einer Halbierung der Rechenzeit führen, und auch das nur wenn die Bandbreite des Hauptspeichers es zulässt.

Ein Vergleich von GPU-KMeans mit der Ausführung auf der CPU über den CUDA Emulator wäre zwar technisch möglich, ist jedoch nicht sehr sinnvoll, da der Emulator für Funktionalität und Debug-Zwecke entworfen wurde, und für Geschwindigkeits-Messungen ungeeignet ist, u.a. da der Overhead bei Thread-Erstellung und Thread-Wechsel auf der CPU deutlich höher ist.

5.4 Messungen

In Abbildung 5.1 A1,A2 bis D1,D2 werden die Mess-Ergebnisse für verschieden grosse Datenmengen mit verschiedener Anzahl Clustern dargestellt.

Es wurden Datensätze mit bis knapp über 2 Millionen 8-dimensionaler Punkte getestet.

Da die Anzahl der Iterationen bis zum Stillstand stark variiert und von den am Anfang zufällig gewählten Cluster-Zentren abhängt, wurde die durchschnittliche Zeit pro Iteration für die Messungen berechnet, und in den Diagrammen für 50 Iterationen dargestellt.

Für 256 Cluster wird ein Speedup von Faktor 676.4 bei 32 768 Punkten bis zu Faktor 1108.2 bei 2 097 152 Punkten erreicht. Hier braucht die GPU ca. 10 Sekunden, während die CPU fast 3 Stunden rechnet.

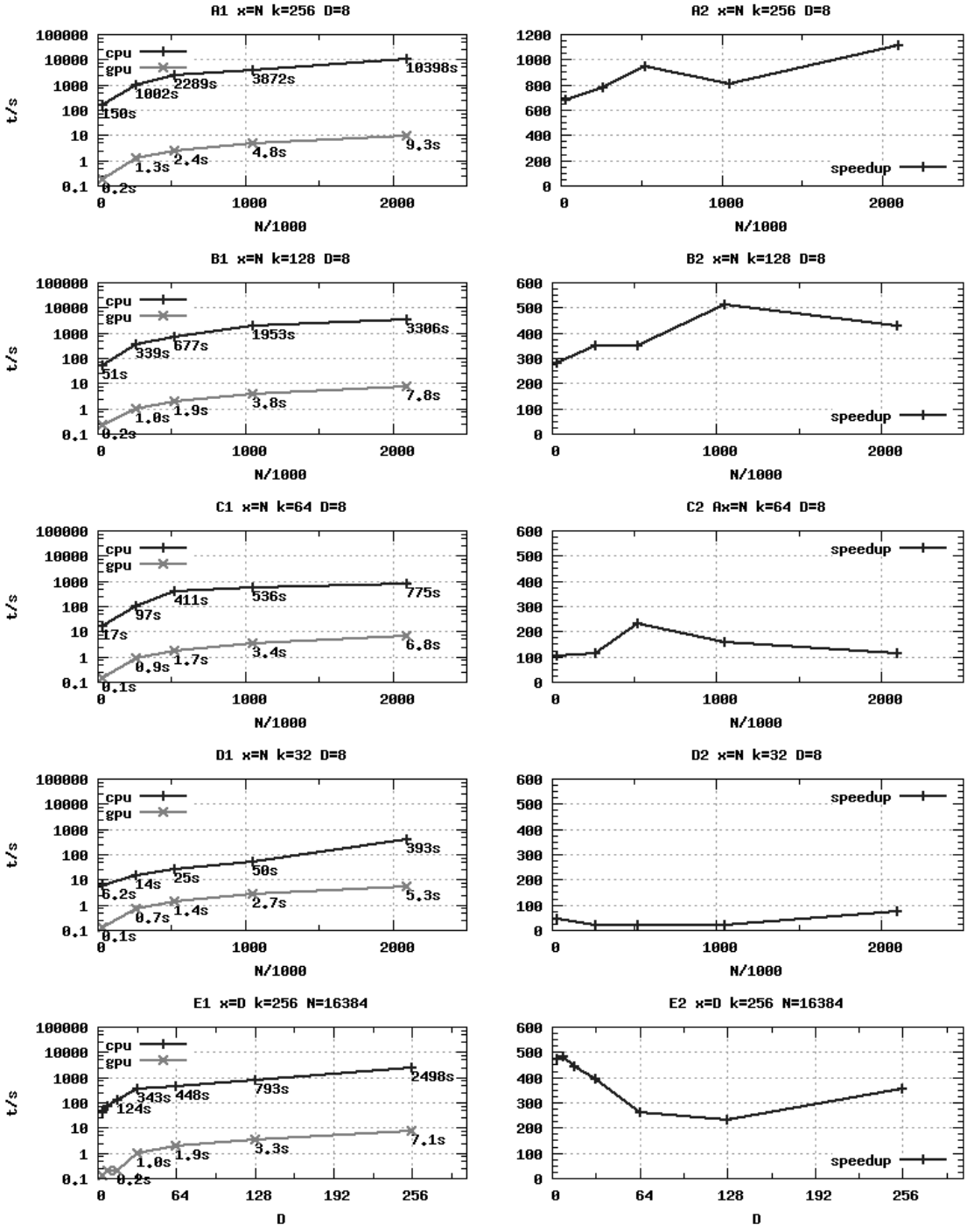

Abbildung 5.1: KMeans Messungen

Algorithm 5 KMeansCPU

```
 1: fLastCost = +∞;
 2: for k = 0 to K-1 do
 3:     for d = 0 to D-1 do
 4:         pMedoids[k][d] = pInitialMedoids[k][d];
 5:     end for
 6: end for
 7: for i = 1 to KMEANSMAXITER do
 8:     for n = 0 to N-1 do
 9:         float fClosestMedoidIndexSqDist = +∞;
10:         int iClosestMedoidIndex = 0;
11:         for k = 0 to K-1 do
12:             float fSqDist = 0.0;
13:             for d = 0 to D-1 do
14:                 fSqDist += square(pMedoids[k][d] - pPoints[n][d]);
15:             end for
16:             if fSqDist < fClosestMedoidIndexSqDist then
17:                 fClosestMedoidIndexSqDist = fSqDist;
18:                 iClosestMedoidIndex = k;
19:             end if
20:         end for
21:         pClosestMedoidSqDist[n] = fClosestMedoidIndexSqDist;
22:         pClosestMedoidIndex[n] = iClosestMedoidIndex;
23:     end for
24:     fCost = CalcCostAndMedoids();
25:     fCostDiff = fabs(fLastCost - fCost);
26:     fLastCost = fCost;
27:     if i > 1 AND fCostDiff < kCostDiffEpsilon then
28:         break;
29:     end if
30: end for
```

Für 128 Cluster wird ein Speedup von Faktor 279.5 bei 32 768 Punkten bis zu Faktor 511.6 bei 1 048 576 Punkten erreicht. Bei 2 097 152 Punkten braucht die GPU ca. 8 Sekunden, während die CPU fast eine Stunde rechnet.

Für 64 Cluster wird ein Speedup von Faktor 104.5 bei 32 768 Punkten bis zu Faktor 233.6 bei 524 288 Punkten erreicht. Bei 2 097 152 Punkten braucht die GPU ca. 7 Sekunden, während die CPU über 12 Minuten rechnet.

Algorithm 6 CalcCostAndMedoids

```
 1: for k = 0 to K-1 do
 2:     pMedoidCounts[k] = 0.0;
 3:     for d = 0 to D-1 do
 4:         pMedoids[k*D+d] = 0.0;
 5:     end for
 6: end for
 7: fCost = 0.0;
 8: for n = 0 to N-1 do
 9:     k = pClosestMedoidIndex[n];
10:     pMedoidCounts[k] += 1;
11:     fCost += pClosestMedoidSqDist[n];
12:     for d = 0 to D-1 do
13:         pMedoids[k][d] += pPoints[n][d];
14:     end for
15: end for
16: for k = 0 to K-1 do
17:     for d = 0 to D-1 do
18:         pMedoids[k][d] /= pMedoidCounts[k];
19:     end for
20: end for
21: return fCost;
```

Für 32 Cluster wird ein Speedup von Faktor 17.7 bei 524 288 Punkten bis zu Faktor 74.6 bei 2 097 152 Punkten erreicht. Hier braucht die GPU ca. 6 Sekunden, während die CPU fast 7 Minuten rechnet.

Der Speedup ist also bei Datenmengen mit vielen Clustern deutlich höher als bei Daten mit nur wenigen Clustern. Die genauen Ursachen hierfür sind leider schwer zu ergründen.

Compiler Optimierungen für den GPU Code könnten einen Einfluss haben, z.B. Loop-Unrolling der Schleife über alle Cluster.

Die GPU profitiert auch davon wenn viele Threads gleichzeitig dieselben Koordinaten der Cluster-Zentren anfragen, was wegen SIMD oft der Fall ist.

Für mehrere parallele Anfragen auf dieselbe Speicher-Adresse muss diese nur einmal ausgelesen werden, während sie auf der CPU immer wieder neu

ausgelesen werden muss wenn bei hoher Anzahl von Clustern nicht ausreichend Platz im Cache ist.

Insgesamt liegt der Speedup jedoch bei allen Messungen deutlich über Faktor 15.

5.5 Auswirkung der Dimensionalität

In Abbildung 5.1 E1 und E2 werden die Mess-Ergebnisse für eine variierende Anzahl an Dimensionen dargestellt.

Es wurden Datensätze mit 16 384 Punkten getestet. Die Dimensionalität wird von 4 bis 256 getestet.

Der Speedup variiert zwischen 230 und 500.

Auf CPU und GPU beeinflusst die Dimensionalität eventuelle Compiler-Optimierungen, wie z.B. Loop-Unrolling, da sie zur Compile-Zeit fest vorgegeben wird.

Auf der GPU bestimmt die Dimensionalität die Grösse der Variable für die aktuellen Punkt, dies hat einen Einfluss darauf wie viele Threads parallel ausgeführt werden können, da die Anzahl der Register für solche lokalen Variablen begrenzt ist.

Da das Device Memory auf der GPU in Speicher-Bänke aufgeteilt ist, und eine Speicherbank nur eine Speicherzelle gleichzeitig auslesen kann, könnte das Layout der Punkte im Speicher, das ja auch durch die Dimensionalität bestimmt wird, auch einen Einfluss auf die Laufzeit, und somit auf den Speedup haben.

Kapitel 6

DBScan

6.1 DBScan Algorithmus

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) ist ein Dichte-basiertes Clustering Verfahren das 1996 von Martin Ester, Hans-Peter Kriegel, Jörg Sander und Xiaowei Xu in [EKJX96] vorgeschlagen wurde.

Parameter DBScan benötigt 2 Parameter: Epsilon und MinPts.

Algorithmus

- Der Algorithmus beginnt an einem beliebigen Punkt der noch nicht besucht wurde.
- Alle Nachbar-Punkte in der Epsilon-Umgebung des Startpunkts werden bestimmt.
- Wenn die Anzahl der Nachbar-Punkte kleiner als MinPts ist, wird der Punkt als Noise markiert.
- Wenn die Anzahl der Nachbar-Punkte grösser oder gleich MinPts ist, wird ein Cluster gebildet.
- Der Startpunkt und seine Nachbar-Punkte werde zu diesem Cluster hinzugefügt, und der Startpunkt wird als besucht markiert.
- Der Algorithmus wiederholt die Auswertung rekursiv bei den Nachbar-Punkten um den Cluster auszubreiten.
- Wenn der Cluster vollständig ausgebreitet ist (alle Punkte in Reichweite wurden besucht), wird der Algorithmus mit dem nächsten, noch nicht besuchten Punkt fortgesetzt.

6.2 Überblick

Diese Arbeit schlägt eine DBScan Variante (GPU-DBScan) für die parallele Ausführung auf CUDA-Fähigen Grafikkarten vor.

Anstatt einem Startpunkt werden mehrere Seed-Punkte gleichzeitig bearbeitet. Die dabei gefundenen Nachbar-Punkte werden der gleichen ID wie der Seed-Punkt zugeordnet. Im nächsten Schritt wird einer davon als neuer Seed-Punkt ausgewählt, so entsteht aus jedem Seed-Punkt eine Kette.

Ein Cluster kann aus mehreren solcher Ketten bestehen, aber eine Kette gehört immer zu nur einem bestimmten Cluster. Die Verbindung von mehreren Ketten zu einem Cluster wird durch Konflikte beim Zuweisen der IDs zu den Nachbar-Punkten erkannt.

Der Algorithmus läuft in mehreren Schritten ab, wobei bei jedem Schritt eine bestimmte Anzahl von Seed-Punkten abgearbeitet wird, und im nächsten Schritt dann neue gewählt werden.

6.3 Seed-Punkte

Eine bestimmte Anzahl (Seednum, in den Versuchen 128) von Seed-Punkten wird gleichzeitig bearbeitet. Während eines Arbeitsschritts hat jeder dieser Seed-Punkte eine eigene, von den anderen Seed-Punkten verschiedene Chain-ID.

Zunächst werden die Nachbar-Punkte in der Epsilon-Umgebung des Seed-Punktes bestimmt.

Wurden weniger als MinPts Nachbar-Punkte gefunden, so wird der Seed-Punkt als Noise markiert.

Wurden mindestens MinPts Nachbar-Punkte gefunden, so wird der Seed-Punkt als Kern-Punkt, und die Nachbar-Punkte als Kandidaten markiert.

Beim markieren wird versucht jedem Nachbar-Punkt mittels Atomic Op die Chain-ID des Seed-Punktes zuzuweisen, aber nur, wenn der Punkt nicht bereits eine Chain-ID zugewiesen hatte. In diesem Fall wird die zugewiesene Chain-ID nicht verändert, d.h. der Nachbar-Punkt behält die erste Chain-ID

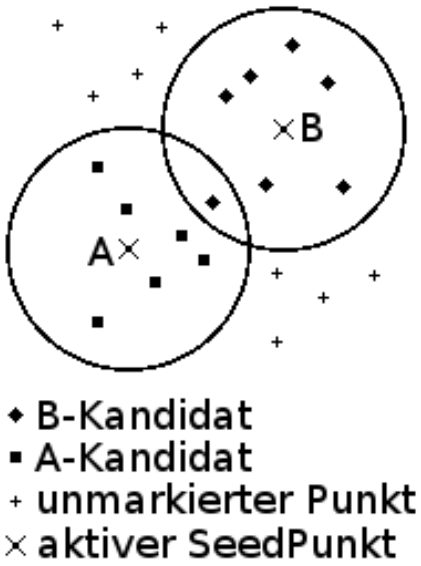

Abbildung 6.1: Konflikt beim markieren der Nachbar-Punkte

die ihm zugewiesen wurde. Durch den Rückgabe-Wert der Atomic Op mit der die Zuweisung versucht wird, ist die Chain-ID des Punktes bekannt.

Die andere Chain-ID kann entweder in einem früheren Arbeitsschritt zugewiesen worden sein, oder wegen der parallele Ausführung sogar während dem aktuellen Arbeitsschritt.

Hatte der Punkt bereits eine Chain-ID, und stimmt diese mit der Chain-ID vom Seed-Punkt überein, so wird nichts weiter unternommen. Dies geschieht z.B. wenn viele Punkte nahe beieinander liegen, und bereits in einem früheren Arbeitsschritt von einem Seed-Punkt der selben Chain-ID markiert wurde.

Hatte der Punkt eine andere Chain-ID, so wurde ein Konflikt erkannt. Dies geschieht wenn verschiedene Chains innerhalb eines Clusters beim Ausbreiten aufeinander stossen. Jeder Punkt ist maximal einer Chain-ID zugeordnet. Wenn der Punkt bereits markiert war, wird ihm keine neue Chain-ID zugewiesen.
Dieser Fall wird in Abbildung 6.1 veranschaulicht.

Bei einem solchen Konflikt ist durch den Rückgabe-Wert der Atomic Op bekannt, welche beiden Chain-IDs zusammengestossen sind, dies wird in der

Connection-Matrix aufgezeichnet, und später verwendet um mehrere Chains zu Clustern zusammenzufassen.

6.4 Chains

Die beim Markieren und Abarbeiten der Nachbar-Punkte entstehenden Ketten von Punkten bezeichnen wir als Chains.

Alle Punkte innerhalb einer Chain haben dieselbe, global eindeutige Chain-ID, und diese wird über mehrere Arbeitsschritte hinweg weitergereicht, wobei für jede noch nicht abgeschlossene Chain (d.h. es sind noch markierte Kandidaten-Punkte übrig) in einem Arbeitsschritt immer genau ein Seed-Punkt aktiv ist.

Pro Chain ist nur ein Seed-Punkt aktiv, und alle aktiven Seed-Punkte haben unterschiedliche Chain-IDs.

Ein Seed-Punkt weist allen gefundenen Nachbar-Punkten seine eigene Chain-ID zu, wenn diese nicht bereits eine Chain-ID haben. Nachdem ein Seed-Punkt abgearbeitet ist wird im nächsten Arbeitsschritt ein neuer Seed-Punkt ausgewählt. Wenn es noch nicht abgearbeitete Punkte mit der Chain-ID gibt, so wird einer von diesen ausgewählt.

Alle Punkte in einer Chain sind dichte-verbunden über eine Reihe von Kern-Punkten die jeweils höchstens Epsilon als Abstand zueinander haben.

Alle Punkte in einer Chain werden dem selben Cluster zugeordnet, jedoch kann ein Cluster aus mehreren Chains bestehen.

Sind alle Punkte mit einer Chain-ID abgearbeitet, und wurden keine neuen Nachbar-Punkte mehr markiert, so ist die Chain beendet.

Wenn es noch Punkte gibt die weder markiert noch als Seed-Punkt ausgewählt wurden, so wird eine neue Chain gestartet, mit einem dieser Punkte als Seed-Punkt.

Ist dies nicht der Fall, und gibt es noch Punkte in anderen Chains die zwar markiert aber noch nicht abgearbeitet sind, so wird eine dieser Chains gesplittet.

6.5 Split

Wenn eine Chain abgeschlossen ist, und es keine unmarkierten Punkte mehr gibt mit denen eine neue Chain gestartet werden könnte, wird eine der noch nicht abgeschlossenen Chains gesplittet.

Da immer alle Nachbar-Punkte in der Epsilon-Umgebung der Seeds markiert werden, aber in jedem Arbeitsschritt von jeder Chain nur ein Kandidaten-Punkt abgearbeitet wird, tritt dieser Fall oft recht früh ein. Es existiert dann noch eine grosse Menge bereits markierter Punkte die noch abgearbeitet werden müssen.

Bei einem Split wird eine neue Chain erzeugt, in dem einer der Kandidaten der zu splittenden Liste um-markiert wird.

Als zu splittende Chain wird hier diejenige mit den meisten noch nicht abgearbeiteten Kandidaten gewählt.

Da die neue Chain mit der gesplitteten verbunden ist und im selben Cluster liegt, wird die Verbindung beim Split in die Connection-Matrix eingetragen.

6.6 Connection-Matrix

Wir zeichnen Zusammenstösse zwischen verschiedenen Chains in einer $K \times K$ Matrix auf, deren Einträge anfangs auf 0 gesetzt werden.

Bei einem Zusammenstoss wird in eine Zelle eine 1 eingetragen, die Zeile der Zelle wird durch die eine der beiden zusammenstossenden Chain-IDs bestimmt, und die Spalte durch die andere Chain-ID.

Da zu jedem Zeitpunkt nur eine begrenzte Anzahl an Seeds aktiv sind (SeedNum, bei den Experimenten 128), genügt uns eine recht kleine Matrix : $SeedNum \times SeedNum$

Wenn eine Chain abgeschlossen ist, und an ihrer Stelle eine neue gestartet, oder von einer existierenden abgesplittet wird, so nimmt die neue den Platz der alten in der Connection-Matrix ein.

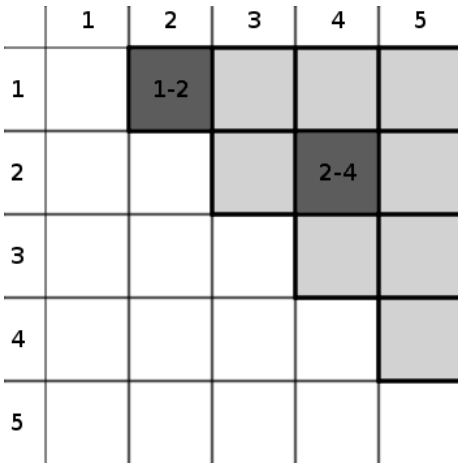

Abbildung 6.2: Connection-Matrix

Dazu werden beim Abschluss der Chain zuerst die aufgezeichneten Verbindungen ausgewertet, und anschliessend in der Connection-Matrix die Zellen für die Verbindungen zur neuen Chain wieder auf 0 gesetzt.

Wenn die neue Chain durch einen Split erzeugt wurde, wird gleich die Verbindung zu der Chain von der sie abgesplittet wurde eingetragen.

Wenn eine Chain abgeschlossen ist, entstehen keine neuen Verbindungen mehr zu dieser, da es in ihrer Epsilon-Umgebung keine Punkte mehr gibt die noch nicht markiert sind. Zu den von anderen Chains markierten, aber noch nicht abgearbeiteten Punkten in ihrer Epsilon-Umgebung wurde die Verbindunge bereits aufgezeichnet.

Ein Beispiel hierzu wird in Abbildung 6.2 veranschaulicht, hier ist eine Verbindung zwischen Chain 1 und 2, und zwischen Chain 2 und 4 eingetragen.

6.7 Cluster

Ein Cluster kann mehrere verschiedene Chains enthalten.

Beim markieren der Nachbar-Punkte von Seed-Punkten wird den Nachbar-Punkten eine Chain-ID zugewiesen. Um zu diesen Chain-IDs dann später

Cluster-IDs zuweisen zu können, werden Zusammenstösse zwischen verschiedenen Chains in der Connection-Matrix aufgezeichnet, denn diese müssen sich dann im selben Cluster befinden.

Wenn eine Chain beendet ist, d.h. es gibt keine noch nicht abgearbeiteten Kandidaten mehr, wird erst geprüft ob die Chain bereits eine Cluster-ID zugewiesen hat.

Anfangs ist noch keiner Chain-ID eine Cluster-ID zugewiesen.

Ist beim Ende einer Chain noch keine Cluster-ID zugewiesen, so wird eine neue Cluster-ID erzeugt.

Über die Connection-Matrix wird bestimmt zu welchen anderen Chains eine Verbindung besteht.

Es kann dabei auch indirekte Verbindungen geben, z.B. Chain A wird gerade abgeschlossen, A ist verbunden mit B, B ist verbunden mit C, also gibt es eine indirekte Verbindung zwischen A und C (über B), und A und C müssen im selben Cluster sein, auch wenn es nie zu einem direkten Zusammenstoss zwischen A und C kam.

Um auch indirekte Verbindungen zu erkennen, werden die Einträge in der Connection-Matrix rekursiv ausgebreitet. Da die Connection-Matrix wegen der begrenzten Anzahl an gleichzeitig aktiven Seeds recht klein ist (Seednum $\times$ Seednum, bei den Experimenten 128×128) , ist der Zeitaufwand hierfür relativ gering, und unabhängig von der Grösse der Datenmenge.

Beim Ausbreiten der Connection-Matrix werden auch bereits zugewiesene Cluster-IDs auf alle verbundenen Chains ausgebreitet.

Beim Check ob die gerade abgeschlossene Chain schon eine Cluster-ID zugewiesen hat, wird auch überprüft ob eine der mit ihr (direkt oder indirekt) verbundenen Chains schon eine Chain-ID zugewiesen hat, ist das der Fall, so wird diese verwendet anstatt eine neue Cluster-ID zu erzeugen.

Ist noch keine Cluster-ID zugewiesen, so wird eine neue erzeugt, und auf alle mit der abgeschlossene Chain (direkt oder indirekt) verbundenen Chains ausgebreitet.

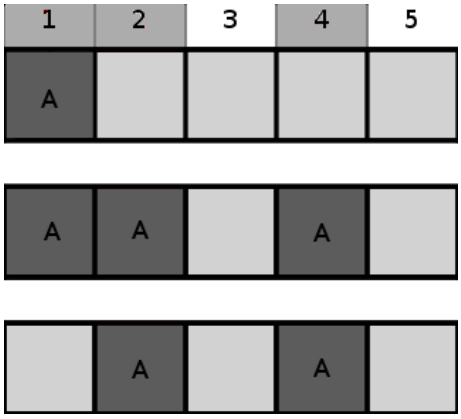

Abbildung 6.3: Ausbreiten der Cluster-IDs

Ein Beispiel hierzu wird in Abbildung 6.3 veranschaulicht, hier besteht eine direkte Verbindung zwischen Chain 1 und 2, und zwischen Chain 2 und 4. Wenn Chain 1 beendet wird, erhält sie die Cluster-ID A, diese wird auf Chain 2 und 4 ausgebreitet. Anschliessend werden die Einträge für Chain 1 gelöscht, da an dieser Stelle eine neue Chain gestartet wird.

Am Ende des gesamten GPU-DBScan Algorithmus wird in einem Nachbearbeitungs-Schritt ($O(N)$) dann allen Punkten die zu ihrer Chain-ID zugehörige Cluster-ID zugewiesen.

In dem Fall dass zwei Chains im selben Cluster abgeschlossen werden bevor es eine (direkte oder indirekte) Verbindung zwischen ihnen gibt, werden für den selben Cluster zwei verschiedene Cluster-IDs erzeugt, und es kommt später beim Ausbreiten der Cluster-IDs zum Konflikt.

Der Konflikt wird aufgezeichnet und im Nachbearbeitungs-Schritt wird allen derart Verbundenen Cluster-IDs eine gemeinsame ID zugewiesen.

Die Situation wird in Abbildung 6.4 veranschaulicht :

- B und C sind noch aktive Chains, in ihrer Reichweite sind noch nicht markierte Punkte
- B und C sind bisher noch nicht zusammengestossen, es ist noch keine Verbindung eingetragen.

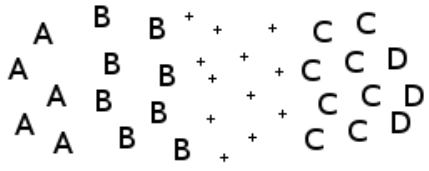

Abbildung 6.4: Cluster-ID Konflikt

- A und D werden jedoch vorher beendet, da es in ihrer Reichweite keine Unmarkierten Punkte mehr gibt, beide erhalten jeweils eine eigene Cluster-ID.
- da es eine Verbindung zwischen A und B gibt, erhält B die Cluster-ID von A
- da es eine Verbindung zwischen D und C gibt, erhält C die Cluster-ID von D
- wenn später B und C zusammenstossen kommt es beim Ausbreiten der Cluster-IDs zum Konflikt.

Dieser Fall tritt jedoch nur sehr selten auf.

Im Vergleich zum normalen DBScan weicht das Verhalten von GPU-DBScan beim zuordnen von Clustern in einem Fall leicht ab :

eine Verbindung zwischen Clustern wird bei GPU-DBScan auch erzeugt, wenn ein markierter Kandidaten-Punkt eines anderen Clusters in der Epsilon-Umgebung eines Seed-Punktes ist, bevor bekannt ist ob der Kandidaten-Punkt ein Kern-Punkt ist (mindestens MinPts Nachbar-Punkte in Epsilon-Umgebung).

Wenn dieser Nachbar-Punkt kein Kern-Punkt ist, würde es beim normalen DBScan keine Verbindung der beiden Cluster geben. Beim normalen DBScan wird ein Cluster nur über Kern-Punkte ausgebreitet, bei GPU-DBScan hingegen genügt es bereits wenn jeder 2te Punkt ein Kern-Punkt ist.

Trotzdem wird das Single-Link-Problem vermieden, da auch bei GPU-DBScan eine Kette von Einzelpunkten nicht ausreicht um zwei Cluster zu verschmelzen, dazu wäre schon eine Kette aus kleinen Clustern notwendig, da mindestens jeder 2te Punkt ein Kern-Punkt sein müsste.

6.8 Kandidaten-Listen

Um nach einem Arbeitsschritt schnell den nächsten Seed-Punkt für eine Chain bestimmen zu können, werden die gefundenen Kandidaten-Punkte nicht nur mit der Chain-ID markiert, sondern auch in eine Liste eingetragen. So kann beim nächsten Arbeitsschritt direkt der letzte Punkt aus der Liste als neuer Seed-Punkt genommen werden.

Die Eintragung in die Liste geschieht auf der GPU mittels Atomic Ops für die Listenzähler. Die Listen müssen nicht zwischen Device Memory und Hauptspeicher hin und her transferiert werden, sondern bleiben im Device Memory, also wird hierdurch Bandbreite gespart.

Es gibt nur eine begrenzte Anzahl von gleichzeitig aktiven Seeds, und für jeden davon nur eine Liste.

In solch einer Liste ist allerdings nur begrenzt Platz, bei unseren Versuchen ca. für 1024 Punkte pro Chain. Ist die Liste voll, so werden weitere Punkte nur mit der Chain-ID markiert, und nicht eingetragen. Mit 2 Zählern wird festgehalten wie viele Punkte in der Liste sind, und wie viele nur markiert wurden ohne in eine Liste eingetragen zu werden.

Wird eine Liste dann später leer, wenn sie nach und nach abgerbeitet wurde, so ist wegen den Zählern bekannt, ob es noch mit der Chain-ID markierte Punkte gibt, die nicht in der Liste stehen.

Gibt es noch solche markierten Punkte, so wird die Liste mit einem speziellen Kernel wieder aufgefüllt, der die Datenpunkte durchgeht, und nach Punkten sucht die nur mit der Chain-ID markiert, aber nicht in die Liste eingetragen wurden, und diese dann in die Liste einträgt.

Das Wiederauffüllen läuft auch auf der GPU ab, so dass keine Markierungs-Daten vom Device Memory in den Hauptspeicher kopiert werden müssen. Dieser Kernel wird in Abschnitt 6.10.3 näher beschrieben.

Wenn es keine solchen markierten Punkte mehr gibt, ist die Chain beendet, und es wird entweder eine neue gestartet, wenn es noch unmarkierte Punkte gibt, oder eine bestehende Chain gesplittet.

6.9 Code Aufbau

Nachdem die Speicherbereiche im Device Memory initialisiert und eine bestimmte Anzahl (SeedNum, bei den Experimenten 128) neuer Chains mit zufälligen Seed-Punkten ausgewählt wurden, wird die Haupt-Schleife gestartet. Diese wird solange ausgeführt, bis alle Punkte abgearbeitet wurden. Nach der Haupt-Schleife wird den Punkten noch in einem Nachbearbeitungsschritt die endgültige Cluster-ID zugeordnet. Dies wird in Pseudocode 7 veranschaulicht.

In der Haupt-Schleife wird zunächst der Main-Kernel ausgeführt, danach wird der Zustand der Chains geprüft und falls nötig beendete Chains abgeschlossen, und neue Chains gestartet bzw. von existierenden abgesplittet. Pseudocode 8 und 9 veranschaulichen dies.

Algorithm 7 DBScanMainCPU()

```
1: AllocateAndInitDeviceMemoryBlocks();
2: CreateNewChainsFromRandomPoints();
3: bFreshSeedsLeft = true;
4: repeat
5:     StartKernel(KernelMain) {ein Thread-Block pro Seed-Punkt}
6:     DownloadFromDeviceMemory(pSeedStates)
7:     DBScanCPUUpdateChains();
8:     DBScanCPUStartNewChains();
9:     if bListRefillNeeded then
10:        StartKernel(KernelRefill) {ein Thread-Block pro Seed-Punkt}
11:    end if
12: until iNumberOfNonRevivableSeeds ≥ kNumberOfSeeds
13: Nachbearbeitungsschritt();
```

6.10 Kernel

Die Programmabschnitte, die parallel auf der GPU ausgeführt werden, werden Kernel genannt. Bei GPU-DBScan verwenden wir 3 verschiedene Kernel, die in einer Hauptschleife abwechselnd aufgerufen werden, bis alle Punkte abgearbeitet sind.

6.10.1 Haupt-Kernel

Im Haupt-Kernel werden die Nachbar-Punkte bestimmt, markiert und Kollisionen in die Connection-Matrix eingetragen.

Algorithm 8 DBScanCPUUpdateChains()

```
 1: bListRefillNeeded = false;
 2: iNewSeedsNeeded = 0;
 3: for iSeedIndex = 0 to kNumberOfSeeds-1 do
 4:     bSeedFinished[iSeedIndex] = false;
 5:     if (pListLen[iSeedIndex] > 0) continue;
 6:     if pNotListedLen[iSeedIndex] > 0 then
 7:         bListRefillNeeded = true;
 8:     else
 9:         bSeedFinished[iSeedIndex] = true; { Chain abgeschlossen }
10:         iNewSeedsNeeded += 1;
11:         if iNewSeedsNeeded = 1 then
12:             DownloadAndSpreadConnectionMatrix(); { nur bei der ersten abgeschlossenen Chain in diesem Arbeitsschritt notwendig. }
13:         end if
14:         if not IsClusterIDAssigned(iSeedIndex) then
15:             {neue clusterid muss erzeugt werden}
16:             SetClusterID(iSeedIndex,GenerateNewClusterID());
17:             SpreadConnectionMatrix(iSeedIndex);
18:         end if
19:         ClearConnections(iSeedIndex);
20:         GenerateAndAssignNewChainID(iSeedIndex);
21:     end if
22: end for
```

Jedem Seed-Punkt wird eine ganze Thread-Gruppe zugeordnet, die Threads der Thread-Gruppe arbeiten dann den relevanten Teil der Datenmenge parallel ab.

Die eigene Chain-ID wird anhand der Thread-Block-ID aus einem Speicherbereich im Constant Memory ausgelesen.

Vom ersten Thread in der Thread-Gruppe wird die Punkt-ID des zur Thread-Gruppe zugeordneten Seed-Punktes bestimmt, die anderen Threads warten währenddessen. Wenn von der CPU keine Punkt-ID vorgegeben wurde, so wird eine aus der Liste der Kandidaten-Punkten entnommen. Die Punkt-ID des Seed-Punktes wird ins Shared Memory geschrieben, so dass sie von allen Threads verwendet werden kann. Danach wird die Thread-Gruppe synchronisiert.

Algorithm 9 DBScanCPUStartNewChains()

```
1: iNumberOfNonRevivableSeeds = 0;
2: if iNewSeedsNeeded > 0 then
3:     iNewSeedsFound = 0;
4:     if bFreshSeedsLeft then
5:         StartKernel(KernelNewSeeds,iNewSeedsNeeded) {wenige Thread-Blöcke}
6:         iNewSeedsFound = DownloadFromDeviceMemory()
7:         pNewSeedsArray = DownloadFromDeviceMemory()
8:         if (iNewSeedsFound < iNewSeedsNeeded) bFreshSeedsLeft = false; {keine unmarkierten Punkte mehr}
9:     end if
10:    for iSeedIndex = 0 to kNumberOfSeeds-1 do
11:        if (not bSeedFinished[iSeedIndex]) continue; {noch aktive Seeds werden hier übersprungen}
12:        InitNewSeed(iSeedIndex);
13:        if iNewSeedsFound > 0 then
14:            iNewSeedsFound -= 1;
15:            n = pNewSeedsArray.Pop();
16:            AssignSeedPointID(iSeedIndex,n);
17:            iNewPointState = kPointStateMarked + GetChainID(iSeedIndex);
18:            SetPointStateInDeviceMemory(n,iNewPointState);
19:        else
20:            iSplitMeSeedIndex = GetSeedIndexWithLongestList() {Split}
21:            if iSplitMeSeedIndex ≠ nil then
22:                n = GetCandidateList(iSplitMeSeedIndex).Pop();
23:                AssignSeedPointID(iSeedIndex,n);
24:                iNewPointState = kPointStateMarked + GetChainID(iSeedIndex);
25:                SetPointStateInDeviceMemory(n,iNewPointState);
26:                MarkConnection(iSeedIndex,iSplitMeSeedIndex)
27:            else
28:                iNumberOfNonRevivableSeeds += 1;
29:            end if
30:        end if
31:    end for
32: end if
```

Die Koordinaten des Seed-Punktes werden eingelesen, dabei wird von jedem Thread eine Koordinate angefragt, und danach wird die Thread-Gruppe synchronisiert.

Vom ersten Thread der Thread-Gruppe wird dann die Bounding-Box (minimales umgebendes Rechteck) der Epsilon-Umgebung des Seed-Punktes berechnet, und im Shared Memory gespeichert, so dass sie später allen Threads zur Verfügung steht. Danach wird die Thread-Gruppe erneut synchronisiert.

Es wird über alle Seiten im Index iteriert, die die Epsilon Bounding-Box schneiden.

Eine Index-Seite wird von mehreren Threads parallel abgearbeitet, dabei werden von jedem Thread die Koordinaten eines Punktes eingelesen, und der Abstand zum Seed-Punkt berechnet.

Ist der Abstand kleiner als Epsilon, so wird der Nachbarzähler erhöht. Hierbei wird via Shared Memory festgehalten ob bereits MinPts oder mehr Punkte erreicht sind.

Wenn bereits vorher mindestens MinPts-1 Nachbar-Punkte gefunden wurden, so wird der Punkt sofort markiert, andernfalls wird er erst zwischengespeichert. So wird vermieden dass Nachbar-Punkte markiert werden, falls der Seed-Punkte insgesamt weniger als MinPts Nachbar-Punkte hat und kein Kern-Punkt ist.

Nachdem alle Index-Seiten abgearbeitet sind, wird die Thread-Gruppe synchronisiert, um zu warten bis alle anderen Threads der Gruppe fertig sind und die Anzahl der Nachbar-Punkte in der Epsilon-Umgebung bekannt ist.

Sind es mindestens MinPts Nachbar-Punkte, so werden die Nachbar-Punkte noch markiert, die bisher nur zwischengespeichert waren. Auch hier wird parallel mit mehreren Threads gearbeitet. Danach wird die Thread-Gruppe synchronisiert.

Abschliessend speichert noch der erste Thread der Thread-Gruppe den nun bekannten Zustand des Seed-Punktes ab, wenn mindestens MinPts Nachbar-Punkte gefunden wurden, ist es ein Kern-Punkt.

Wurden weniger als MinPts Nachbar-Punkte gefunden, so werden die zwischengespeicherten Nachbar-Punkte verworfen, und der Seed-Punkt wird als Noise markiert.

Pseudocode 10 und 11 veranschaulichen dies.

Beim Markieren der Nachbar-Punkte wird atomicCAS (Compare and Swap) verwendet, dadurch wird zunächst überprüft ob an der angegebenen Adresse ein bestimmter Wert gespeichert ist, und nur wenn das der Fall ist, wird dieser mit dem neuen Wert überschrieben, ansonsten bleibt er unverändert. Am Rückgabe-Wert von atomicCAS, dem alten Wert, können wir erkennen welcher Fall eingetreten ist.

So wird sichergestellt, dass der Nachbar-Punkt nur markiert wird, wenn er nicht bereits markiert war. Zu der Anzahl der Nachbar-Punkte wird er in beiden Fällen hinzugerechnet, um festzustellen ob der zugehörige Seed-Punkt ein Kern-Punkt ist.

Wenn der Punkt noch nicht markiert war und erfolgreich markiert werden konnte, so wird er in die Liste der Kandidaten-Punkte eingetragen, sofern in dieser noch Platz ist. Hierzu wird per atomicInc der Listenzähler erhöht, wobei der Rückgabe-Wert den alten Wert des Listenzählers zurückgibt. Ist dieser Wert kleiner als die Listenlänge, so gibt er die Stelle in der Liste für den neuen Eintrag an.

Ist der Rückgabe-Wert aber grösser oder gleich der Listenlänge, so ist die Liste voll. In diesem Fall wird ein zweiter Zähler erhöht, wieder mittels atomicInc, der festhält wie viele Kandidaten-Punkte es gibt die nicht in der Liste vermerkt sind. Dieser zweite Zähler wird dann später im Refill-Kernel verwendet um die Liste wieder aufzufüllen nachdem sie abgearbeitet wurde.

Wenn der zu markierende Punkt bereits als Noise markiert war, so wird nichts weiter unternommen.

War er bereits mit derselben Chain-ID markiert, mit der er jetzt markiert werden soll, so wird auch nichts unternommen.

War der zu markierende Punkt aber bereits mit einer anderen Chain-ID markiert, so wurde dadurch ein Zusammenstoss zweier Chains erkannt.

Algorithm 10 KernelMain()

```
1: shared iNeighborCount = 0;
2: iChainID = gChainIDs[threadBlockIdx];
3: if (IsFirstThreadInThreadGroup()) shared iOwnPointID = SeedPunktID-
   Bestimmen(); {falls nötig aus Kandidaten-Liste holen}
4: syncthreads();
5: shared vOwnPoint = PunktEinlesen(iOwnPointID); { parallele
   Speicherbereichs-Anfrage }
6: syncthreads();
7: if (IsFirstThreadInThreadGroup()) shared vEpsilonUmgebungsBBox =
   CalcEpsilonUmgebungsBBox(vOwnPoint);
8: syncthreads();
9: for all indexPage ∈ IndexAnfrage(vEpsilonUmgebungsBBox) do
10:     { über Index iterieren }
11:     for w = 0 to kIndexPageSize-1 with increment kThreadBlockSize do
12:         { Index Seite parallel abarbeiten }
13:         iCurPointID = GetPointIDFromIndexPa-
            ge(indexPage,w+threadIdx);
14:         PunktBearbeiten(iCurPointID);
15:     end for
16: end for
17: syncthreads(); { warten bis alle Threads fertig sind und die Anzahl der
   Nachbar-Punkte bekannt ist }
18: if iNeighborCount ≥ kMinPts then
19:     {zwischengespeicherte Nachbar-Punkte abarbeiten}
20:     for all iNeighborPointID ∈ pNeighborBuffer do
21:         MarkAsCandidate(iNeighborPointID,iChainID);
22:     end for
23: end if
24: syncthreads();
25: if IsFirstThreadInThreadGroup() then
26:     if iNeighborCount ≥ kMinPts then
27:         pPointState[iOwnPointID] = kPointStateFinished + iChainID;
28:     else
29:         pPointState[iOwnPointID] = kPointStateNoise;
30:     end if
31: end if
```

Zunächst wird dann die Thread-Gruppen-ID der anderen Chain bestimmt, dafür wird die Chain-ID in einer Lookup-Tabelle im Constant Memory nach-

Algorithm 11 PunktBearbeiten($iCurPointID$)

```
1: fSqDist = CalcSquareDistanceToPoint(vOwnPoint,iCurPointID);
2: if fSqDist ≤ kSquareEpsilon then
3:     iNeighborIndex = 0x7fffffff;
4:     if iNeighborCount < kMinPts then
5:         iNeighborIndex = atomicInc(pNeighborCounter);
6:         { zerobased, 0 = erster eintrag }
7:         if iNeighborIndex = kMinPts-1 then
8:             iNeighborCount = kMinPts
9:         end if
10:    end if
11:    if iNeighborIndex+1 < kMinPts then
12:        pNeighborBuffer[iNeighborIndex] = iCurPointID;
13:    else
14:        MarkAsCandidate(iCurPointID,iChainID);
15:    end if
16: end if
```

geschlagen.

Ist der anderen Chain-ID keine Thread-Gruppen-ID mehr zugeordnet, so ist die andere Chain bereits abgeschlossen, und hat ihrerseits bereits alle Verbindungen zu anderen Chains in die Connection-Matrix eingetragen.

Wird aber eine Thread-Gruppen-ID zu der anderen Chain-ID gefunden, so ist die andere Chain noch aktiv, und die Verbindung zu ihr wird in die Connection-Matrix eingetragen.

Pseudocode 12 veranschaulicht dies.

6.10.2 NewSeeds-Kernel

Der NewSeeds-Kernel wird verwendet um nach unmarkierten Punkten zu suchen, wenn neue Chains gestartet werden.

Es können in einem Aufruf mehrere Punkte bestimmt werden, falls nach dem Arbeitsschritt mehrere neue Seeds gestartet werden müssen.

Algorithm 12 MarkAsCandidate($iNeighborPointID, iChainID$)

```
 1: iOldState = atomicCAS(pPointState[iNeighborPointID],kPointStateUnmarked,
    kPointStateMarked + iChainID);
 2: if iOldState = kPointStateInit then
 3:     iListIndex = atomicInc(pListLen);
 4:     if iListIndex < kCandidateListMaxlen then
 5:         pCandidateList[iListIndex] = iNeighborPointID;
 6:     else
 7:         atomicInc(pNotListedLen);
 8:     end if
 9: else if iOldState ≠ kPointStateNoise then
10:     iOtherChainID = ChainIDFromPointState(iOldState)
11:     if iOtherChainID ≠ iChainID then
12:         iOtherThreadGroupID = 0xffffffff;
13:         for k = 0 to kNumberOfThreadGroups-1 do
14:             if iOtherChainID = gChainIDs[k] then
15:                 iOtherThreadGroupID = k;
16:             end if
17:         end for
18:         if iOtherThreadGroupID < kNumberOfThreadGroups then
19:             pConnectionMatrix[iOwnThreadGroupID][iOtherThreadGroupID]
                = 1;
20:         end if
21:     end if
22: end if
```

Mit AtomicOps wird mitgezählt wie viele unmarkierte Punkte bei diesem Aufruf bereits gefunden wurden, und diese werden dann in eine Liste eingetragen.

Ist die gesuchte Anzahl erreicht, so wird dies über das Shared Memory den anderen Threads mitgeteilt, so dass diese die Suche abbrechen können. Hier wird auch ausgenutzt, dass bei mehreren parallelen Schreibzugriffen auf dieselbe Stelle mindestens einer erfolgreich ist.

Die Datenmenge wird parallel von mehreren Threads durchsucht, dabei wird sie so aufgeteilt, dass jeder Thread einen eigenen Teilbereich durchsucht.

Jeder Thread speichert auch über einen Arbeitsschritt hinaus welche Bereiche der Datenmenge er bereits durchsucht hat, so dass die gesamte Datenmenge

nicht mehrfach durchsucht wird.

Die Suche wird also auf der GPU ausgeführt, und die Daten für die Punkt-Markierungen müssen nicht vom Device Memory in den Hauptspeicher transferiert werden.

Pseudocode 13 veranschaulicht dies.

Algorithm 13 KernelNewSeeds(*iNewSeedsNeeded*)

```
1: if (IsFirstThreadInThreadGroup()) shared iNumFoundCache = 0;
2: syncthreads();
3: if (pUnScannedRangeStart ≥ kNewSeedScanSectionSize) return; {Bereich
   bereits abgearbeitet}
4: iBaseID = iGlobalThreadIDIncludingBlockID * kNewSeedScanSectionSi-
   ze;
5: for i = 0 to kNewSeedScanSectionSize-1 do
6:     if (iNumFoundCache ≥ iNewSeedsNeeded) break; {Scan beendet,
       Punkt wird beim nächsten Aufruf nochmals gescannt}
7:     n = iBaseID + pUnScannedRangeStart + i;
8:     if pPointState[n] = kPointStateInit then
9:         iNewSeedIndex = atomicInc(pFoundNewSeeds);
10:        if iNewSeedIndex < iNewSeedsNeeded then
11:            pNewSeeds[iNewSeedIndex] = n;
12:        else
13:            iNumFoundCache = iNewSeedsNeeded; { Abbruch, es wur-
               den genug Punkte gefunden }
14:            break; {Punkt konnte nicht eingetragen werden, muss beim
               nächsten Aufruf nochmals gescannt werden}
15:        end if
16:    end if
17: end for
18: pUnScannedRangeStart = i;
```

6.10.3 Refill-Kernel

Der Refill-Kernel wird verwendet um Kandidaten wieder aufzufüllen nachdem sie abgearbeitet wurden, wenn vorher aufgrund der begrenzten Listenkapazität nicht alle markierten Punkte aufgenommen werden konnten.

Die Liste wird dabei nicht komplett aufgefüllt, sondern höchstens halb, damit noch Platz ist um beim abarbeiten neu markierte Punkte einzutragen.

Er wird eine ganze Thread-Gruppe für jeder aktive Chain gestartet, so dass die Datenmenge mit mehreren Threads parallel durchsucht wird.

Für die Threads deren Kandidaten-Liste nicht leer ist, oder für die wegen dem Listenzähler bekannt ist, dass es keine markierten Punkte mehr ausserhalb der Liste gibt, wird die Thread-Gruppe gleich beendet.

Wird beim durchsuchen der Punktmenge ein markierter Punkt gefunden, so wird mittels atomicInc der Listenzähler erhöht. Der Rückgabe-Wert von atomicInc bestimmt ob in der Liste noch Platz für den neue Eintrag war, und die zugewiesene Listen-Position.

Ist die gewünschte Listenlänge (halb-voll) erreicht, so wird dies den anderen Threads in der Thread-Gruppe via Shared Memory signalisiert, so dass die Suche abgebrochen werden kann.

Pseudocode 14 veranschaulicht dies.

6.11 Vorteile von CUDA gegenüber Shader-basiertem GPGPU

Bei GPU DBScan werden einige durch CUDA ermöglichte Techniken verwendet, die mit reinem Shader-basiertem GPGPU gar nicht, oder nur unter erheblichen Performance-Einbussen möglich gewesen wären.

Atomic Ops An folgenden Stellen werden Atomic Ops verwendet :

Beim Markieren eines Kandidaten-Punktes im Haupt-Kernel wird atomicCAS (CompareAndSwap) verwendet, um Zusammenstösse mit anderen Chains zu erkennen.

Beim Eintragen von Punkten in die Kandidaten-Liste im Haupt-Kernel und beim Zählen der Punkte die nicht mehr in die Liste gepasst haben, wird atomicInc verwendet.

Algorithm 14 KernelRefill()

```
1: if (pListLen > 0) return; {muss nicht aufgefuellt werden}
2: if (pNotListedLen = 0) return; {es gibt keine markierten Punkte mehr
   ausserhalb der Liste}
3: iMaxTake = min(pNotListedLen,kCandidateListRefill);
4: if (IsFirstThreadInThreadGroup()) shared iNumFound = 0;
5: syncthreads();
6: for n = threadIdx to N-1 with increment kThreadBlockSize do
7:     if (iNumFound ≥ iMaxTake) break; {Abbruch, genug Punkte gefun-
       den}
8:     if pPointState[n] = kPointStateMarked + iChainID then
9:         iMyIndex = atomicInc(pListLen);
10:        if iMyIndex < iMaxTake then
11:            atomicDec(pNotListedLen);
12:            pCandidateLists[iMyIndex] = n;
13:            if (iMyIndex + 1 ≥ iMaxTake) iNumFound = iMaxTake;
               {Abbruch, genug Punkte gefunden}
14:        end if
15:    end if
16: end for
17: syncthreads();
18: if IsFirstThreadInThreadGroup() then
19:    if (pListLen > iMaxTake) pListLen = iMaxTake; {over-increment
       durch atomic op abschneiden}
20: end if
```

Beim Eintragen von Verbindungen in die Connection-Matrix im Haupt-Kernel wird ausgenutzt dass bei mehreren Schreibzugriffen gleichzeitig mindestens einer erfolgreich ist.

Der Neighbour-Counter im Haupt-Kernel wird mittels atomicInc erhöht, so können mehrere Threads gleichzeitig nach Nachbar-Punkten für den selben Seed-Punkt suchen.

Beim NewSeeds-Kernel wird atomicInc verwendet, um die gefundenen Punkte in eine Liste einzutragen.

Beim Refill-Kernel wird atomicInc verwendet, um die Punkte in die Kandidaten-Listen einzutragen, und atomicDec um den Zähler für die nicht aufgelisteten Punkte zu verringern.

Listen mit Atomic Ops Atomic Ops ermöglichen es Listen zu verwenden, auf die mehrere Threads gleichzeitig zugreifen können.

Im Haupt-Kernel werden neu gefundene Nachbar-Punkte in einer Liste zwischengespeichert, bis bekannt ist ob der Seed-Punkt ein Kern-Punkt ist.

Im Haupt-Kernel werden markierte Nachbar-Punkte in einer Kandidaten-Liste gespeichert, um später schnell einen neuen Seed-Punkt auswählen zu können.

Im NewSeeds-Kernel werden die gefundenen Punkte als Liste zurückgegeben.

Im Refill-Kernel werden die Kandidaten-Listen wieder aufgefüllt.

Shared Memory An folgenden Stellen wird Shared Memory verwendet :

Im Haupt-Kernel eine Variable um anderen Threads zu signalisieren dass bereits MinPts Nachbar-Punkte gefunden wurden, so dass die anderen Threads die Nachbar-Punkte direkt markieren können, ohne sie zwischenspeichern zu müssen. Somit müssen nur MinPts-1 Nachbar-Punkte zwischengespeichert werden.

Der Speicherbereich für das Zwischenspeichern der Nachbar-Punkte im Haupt-Kernel ist ebenfalls im Shared Memory.

Die Koordinaten des eigenen Seed-Punktes, und die Bounding-Box der Epsilon-Umgebung die beim Nachschlagen im Index verwendet wird, sind im Shared Memory gespeichert, so dass sie als Zwischen-Ergebnisse gemeinsam von allen Threads verwendet werden können, und nicht jeder Thread sie neu berechnen muss.

Im NewSeeds-Kernel wird eine Variable im Shared Memory verwendet, um den anderen Threads zu signalisieren, das die gewünschte Anzahl an neuen Punkten gefunden wurde, und die Suche abgebrochen werden kann.

Im Refill-Kernel wird eine Variable im Shared Memory verwendet, um den anderen Threads zu signalisieren, das die gewünschte Anzahl an markierten Punkten gefunden und in die Kandidaten-Liste eingetragen wurde, so dass die Suche abgebrochen werden kann.

Constant Memory Im Haupt-Kernel und im Refill-Kernel wird Constant Memory verwendet, um den Thread-Gruppen die Chain-ID ihrer zugewiesenen Seed-Punkte mitzuteilen.

Thread Sync

Im Haupt-Kernel wird am Anfang Thread-Sync verwendet, um zu warten bis der erste Thread den eigenen Seed-Punkt bestimmt hat. Dies ermöglicht es den Seed-Punkt an dieser Stelle direkt aus der Kandidaten-Liste zu holen und ihn aus dieser zu entfernen.

Im Haupt-Kernel wird am Anfang Thread-Sync verwendet, um die Koordinaten des Seed-Punktes mit mehreren Threads parallel einzulesen. Die Lese Anfragen für alle Koordinaten wird gleichzeitig gestellt, so kann die Bandbreite der unabhängigen Speicher Bänke voll ausgenutzt werden kann.

Im Haupt-Kernel wird nach dem Berechnen der Bounding-Box der Epsilon-Umgebung Thread-Sync verwendet, um sicherzustellen dass diese fertig berechnet wurde, bevor andere Threads auf sie zugreifen.

Nachdem im Haupt-Kernel alle Punkte abgearbeitet sind, wird Thread-Sync ausgeführt um zu warten bis alle anderen Threads der Thread-Gruppe auch fertig sind. Somit ist bekannt, ob alle Nachbar-Punkte bereits gefunden wurden, bevor entschieden wird ob der Punkt ein Kern-Punkt oder Noise ist. Falls es ein Kern-Punkt ist, werden die zwischengespeicherten Nachbar-Punkte noch markiert.

Scatter und Gather In allen Kernel Abschnitten wird lesend und schreibend auf zur Laufzeit berechnete Speicher-Adressen zugegriffen.

Branching

Beim markieren der Nachbar-Punkte im Hauptkernel wird Branching verwendet um auf Zusammenstösse mit anderen Chains zu reagieren.

Im Haupt-Kernel wird Branching verwendet, um nur die Index-Seiten zu untersuchen, die die Epsilon-Umgebung des Seed-Punktes schneiden. Da die Grösse einer Index-Seite ein vielfaches der Warp-Size ist, nehmen hier immer alle Threads eines Warps den selben Pfad, so dass keine Threads warten müssen.

Beim Bearbeiten der Punkte im Haupt-Kernel wird Branching verwendet, um zu erkennen ob ein Punkt sich in der Epsilon-Umgebung befindet, und ihn in diesem Fall zu markieren, und den Nachbar-Zähler zu erhöhen.

Beim Bearbeiten der Punkte im Haupt-Kernel wird Branching verwendet um zu entscheiden ob ein gefundener Nachbar-Punkt nur zwischengespeichert, oder schon markiert wird, wenn bereits MinPts-1 andere Nachbar-Punkte gefunden wurden.

Am Ende vom Haupt-Kernel wird Branching verwendet um den eigenen Seed-Punkt als Noise oder als Kern-Punkt zu markieren.

Am Ende vom Haupt-Kernel wird Branching verwendet um zwischengespeicherte Nachbar-Punkte nur dann zu markieren, wenn der Seed-Punkt ein Kern-Punkt ist.

Im NewSeeds-Kernel wird Branching verwendet um zu entscheiden welche Punkte in die Ergebnis-Liste eingetragen werden, und um die Suche abzubrechen wenn genügend Punkte gefunden wurden.

Im Refill-Kernel wird Branching verwendet um zu entscheiden welche Punkte in die Kandidaten-Liste eingetragen werden, und um die Suche abzubrechen wenn genügend Punkte gefunden wurden.

Kapitel 7

DBScan Index

7.1 Überblick

Die Einschränkungen auf der GPU (z.B. fehlende Rekursion und teures Branching) machen es schwierig einige Indexstrukturen zu verwenden, die für die CPU entworfen wurden. Dennoch ist es durchaus möglich auf der GPU Indexstrukturen zu verwenden.

Diese Arbeit schlägt eine relativ einfache Indexstruktur vor. Diese Indexstruktur ist keine technische Neuerung, sondern wurde ausgewählt, da sie von den Einschränkungen auf der GPU kaum betroffen ist, und leicht zu implementieren ist. Sie ist z.B. mit einem kD-Tree wie in [Ben75] vergleichbar.

Der Verzweigungs-Grad und die Anzahl der Level sind fest vorgegeben, so dass bei Such-Anfragen keine Rekursion nötigt ist, und einige Compiler-Optimierungen möglich sind.

7.2 Erstellung

- Die gesamte Datenmenge wird nach der ersten Koordinate sortiert.
- Die Datenmenge wird in k (bei den Versuchen k=16) Abschnitte unterteilt.
- Die erste Koordinate von jeder Grenze zwischen den Abschnitten wird gespeichert.
- Jeder Abschnitt wird nach der zweiten Koordinate sortiert.

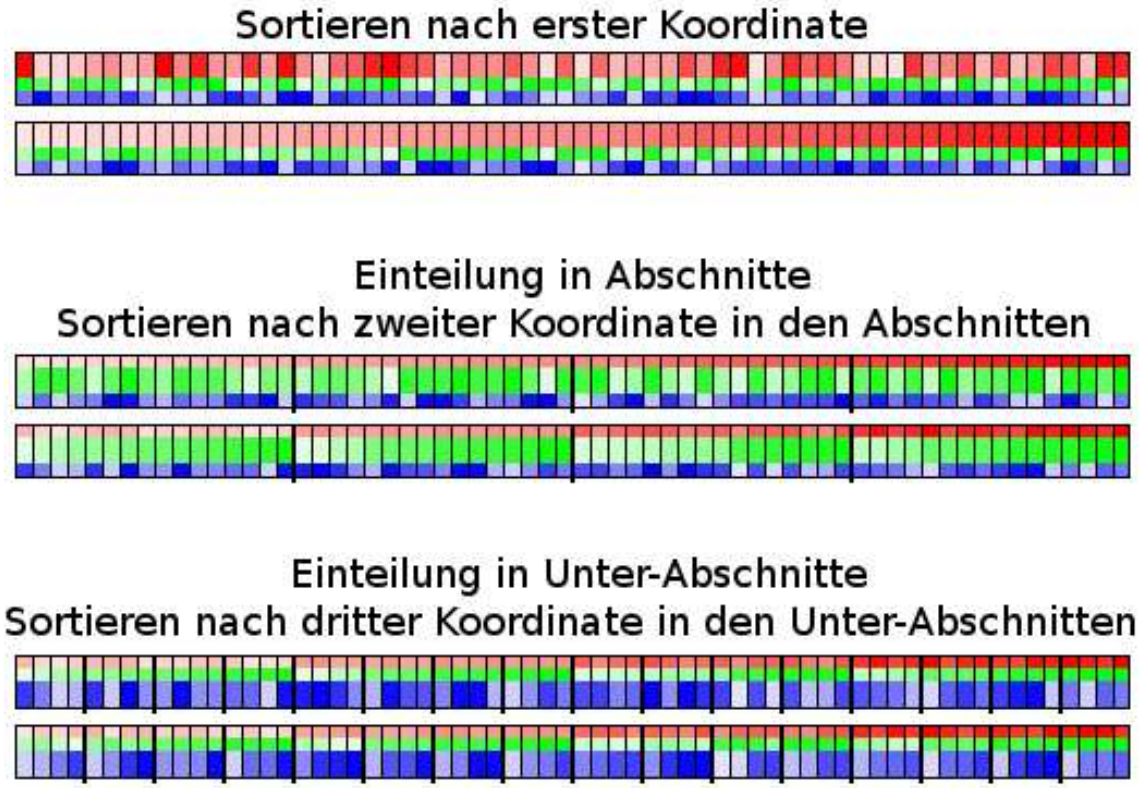

Abbildung 7.1: Index Erstellung

- Jeder Abschnitt wird in k (bei den Versuchen k=16) Unter-Abschnitte unterteilt.
- Die zweiten Koordinate von jeder Grenze zwischen den Unter-Abschnitten wird gespeichert.
- Jeder Unter-Abschnitt wird nach der dritten Koordinate sortiert.
- Jeder Unter-Abschnitt wird in k (bei den Versuchen k=16) Unter-Abschnitte unterteilt.
- Die dritte Koordinate von jeder Grenze zwischen den Unter-Abschnitten wird gespeichert.

Dies wird in Abbildung 7.1 veranschaulicht.

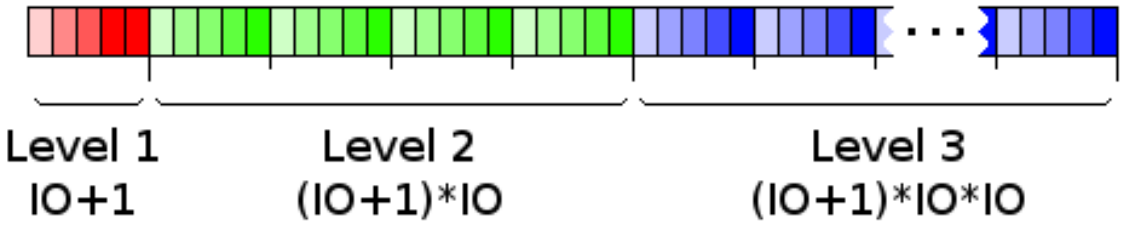

Abbildung 7.2: Index Speicherblock

7.3 Struktur im Speicher

Die Indexstruktur wird in einem zusammenhängenden Speicherblock gehalten, so dass bei Such-Anfragen keine Ketten von Pointern verfolgt werden müssen.

Am Anfang stehen die Grenzen der Abschnitte von Level 1, $IO + 1$ floats, jeweils die erste Koordinate der Eckpunkte.

Danach kommen die Grenzen der Unter-Abschnitte von Level 2, $(IO+1)*IO$ floats, jeweils die zweite Koordinate der Eckpunkte.

Am Ende stehen die Grenzen der Unter-Abschnitte von Level 3, $(IO + 1) * IO * IO$ floats, jeweils die dritte Koordinate der Eckpunkte.

Dies wird in Abbildung 7.2 veranschaulicht.

7.4 Verwendung

Bei GPU DBScan wird die Indexstruktur verwendet, um alle Nachbar-Punkte in der Epsilon-Umgebung eines Seed-Punktes zu finden.

- Es wird über die Abschnitte auf der ersten Ebene iteriert.
- Für alle Abschnitte auf der ersten Ebene die den Anfrage-Bereich schneiden, wird über ihre Unter-Abschnitte auf der zweiten Ebene iteriert.
- Für alle Unter-Abschnitte auf der zweiten Ebene die den Anfrage-Bereich schneiden, wird über ihre Unter-Abschnitte auf der dritten Ebene iteriert.

- Für alle Unter-Abschnitte auf der dritten Ebene die den Anfrage-Bereich schneiden, werden alle dazugehörigen Punkte bearbeitet.

7.5 Eigenschaften

Es handelt sich um eine statische Struktur, Einfügungen und Löschen wäre sehr aufwendig. Die Datenmenge verändert sich aber während des Clusterns nicht, und ist bei Beginn des Clusterings vollständig bekannt, so dass der Index bei Beginn durch Bulk-Loading erstellt werden kann.

Die Indexstruktur ist geclustert, d.h. die Reihenfolge der Elemente im Speicher ist dieselbe wie im Index, dies ermöglicht ein effizientes sortiertes auflisten.

Der Verzweigungs-Grad und die Anzahl der Level werden fest vorgegeben, und sind zur Compile-Zeit bekannt. Es werden weniger lokale Variablen benötigt, so dass mehr Threads gleichzeitig ausgeführt werden können. Es werden auch Compiler Optimierungen ermöglicht, z.B. bei der Berechnung der Speicher-Adressen.

Da die Anzahl der Datenpunkte zur Compile-Zeit bekannt ist, ist auch die Grösse einer Index-Seite bekannt.

Die Anzahl der Datenpunkte wird durch Hinzufügen von Noise-Punkten so erweitert, dass die Grösse einer Index-Seite ein Vielfaches der Thread-Block-Grösse ist. Dies ist vorteilhaft für eine parallele Abarbeitung, um zu vermeiden dass Threads warten müssen.

Der Index verwendet einen zusammenhängenden Speicherbereich, so dass beim Nachschlagen im Index die Adressen direkt berechnet werden können, ohne Ketten von Pointern zu folgen.

Kapitel 8

DBScan Evaluation

8.1 Messaufbau

Alle hier aufgeführten Messungen wurden auf dem selben Rechner durchgeführt, dessen Spezifikationen sind wie folgt :

- OS : Windows XP(tm) Service Pack 2
- CPU : Intel(R) Core(tm)2 Duo CPU E4500 2.20 Ghz
- Grafikkarte : Gainward GeForce GTX280 1GB GDDR3 PCIe 2.0
- Mainboard : Asus P5N-D nForce 750i SLI S775
- RAM : 2GB (2 x 1GB Corsair CM2X1024-6400 5-5-5-12 800Mhz)
- Festplatte : Western Digital SATA 250GB WD2500KS

8.2 Datensatz

Die Datensätze auf denen getestet wurde sind zufalls-generierte Cluster von Punkten, dabei ist die Selektivität der Dimensionen gleich.

Für die Messungen wurden jeweils Datensätze mit 20 Cluster erzeugt.

Am Anfang werden die Cluster Zentren mit zufälligen Koordinaten im Bereich 0.2 bis 0.8, und einem Radius im Bereich 0.02 bis 0.05 erzeugt. Für jeden zu erzeugenden Punkt wird zufällig einer der Cluster ausgewählt, und zufällige Koordinaten innerhalb des Radius gewählt.

8.3 CPU-DBScan

Die DBScan-Implementierung auf der CPU die zum Vergleich verwendet wird ist recht einfach gehalten, und verwendet die gleiche Index-Struktur die für GPU-DBScan verwendet wird.

Die DBScan-Implementierung auf der CPU ist single-threaded, d.h. sie nutzt nur einen der beiden Prozessor-Cores.

Beide Cores auszunutzen würde jedoch höchstens zu einer Halbierung der Rechenzeit führen, und auch das nur wenn die Bandbreite des Hauptspeichers es zulässt.

Ein Vergleich von GPU-DBScan mit der Ausführung auf der CPU über den CUDA Emulator wäre zwar technisch möglich, ist jedoch nicht sehr sinnvoll, da der Emulator für Funktionalität und Debug-Zwecke entworfen wurde, und für Geschwindigkeits-Messungen ungeeignet ist, u.a. da der Overhead bei Thread-Erstellung und Thread-Wechsel auf der CPU deutlich höher ist.

8.4 Messungen mit Index

In Abbildung 8.1 A1 und A2 werden die Mess-Ergebnisse für verschieden grosse Datenmengen dargestellt. Sowohl die CPU- als auch die GPU-Implementierung verwenden hier die Indexstruktur.

Es wurden Datensätze mit bis knapp über 2 Millionen 8-dimensionaler Punkte getestet. Epsilon ist 0.05 und MinPts ist 4.

Der erreichte Speedup der GPU im Vergleich zur CPU steigt mit steigendem N, und reicht von Faktor 3.7 bei 32 768 Punkten bis zu 14.9 bei 2 097 152 Punkten. Hier braucht die GPU ca. 12 Minuten während die CPU fast 3 Stunden rechnet. Ab ca. 250 000 Punkten ist ein Speedup von Faktor 10 erreicht.

8.5 Messungen ohne Index

In Abbildung 8.1 B1 und B2 werden die Mess-Ergebnisse für verschieden grosse Datenmengen dargestellt. Sowohl die CPU- als auch die GPU-Implementierung verwenden hierbei keine Indexstruktur.

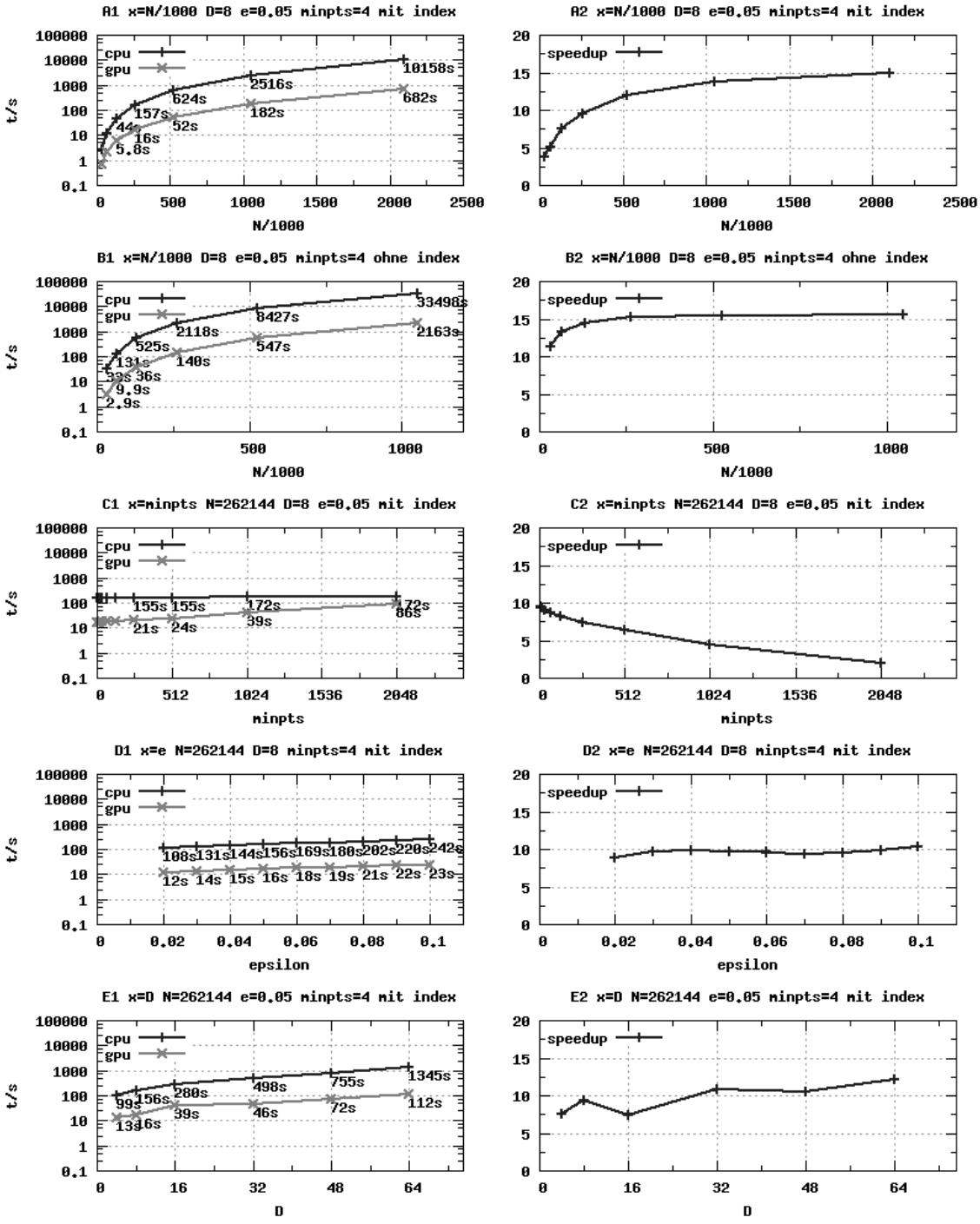

Abbildung 8.1: DBScan Messungen

Es wurden Datensätze mit bis knapp über 1 Million 8-dimensionaler Punkte getestet. Epsilon ist 0.05 und MinPts ist 4.

Der erreichte Speedup der GPU im Vergleich zur CPU steigt mit steigendem N, und reicht von Faktor 11.3 bei 32 768 Punkten bis 15.5 bei 1 048 576 Punkten. Hier braucht die GPU ca. 40 Minuten während die CPU über 9 Stunden rechnet.

8.6 MinPts Auswirkung und Limitierung

In Abbildung 8.1 C1 und C2 werden die Mess-Ergebnisse für verschiedene MinPts-Werte dargestellt. Sowohl die CPU- als auch die GPU-Implementierung verwenden hier die Indexstruktur.

Es wurden Datensätze mit 262 144 Punkten getestet. Epsilon ist 0.05, die Datensätze sind 8-dimensionale. MinPts wird von 4 bis 2048 getestet.

Der erreichte Speedup der GPU im Vergleich zur CPU sinkt mit steigendem MinPts. Von MinPts 4 bis MinPts 512 bleibt der Speedup zwischen Faktor 10 und Faktor 5. Darüber hinaus kommt es zu einem starken Leistungsabfall, jedoch ist GPU-DBScan auch bei MinPts 2048 noch um Faktor 2 schneller als auf der CPU.

Der Leistungsabfall lässt sich dadurch erklären, dass die ersten MinPts-1 gefundenen Nachbarn zwischengespeichert werden müssen, da sie nicht markiert werden dürfen bevor sichergestellt ist, dass der Seed-Punkt ein Kernpunkt ist.

Da sich alle parallel laufenden Thread-Gruppen den knappen Shared Memory teilen müssen, können weniger Thread-Gruppen parallel ausgeführt werden, wenn so viele Punkte zwischengespeichert werden müssen.

Mit einem MinPts von 4096 oder höher kann GPU-DBScan nicht mehr ausgeführt werden, da dafür nicht ausreichen Shared Memory vorhanden ist.

8.7 Epsilon Auswirkung

In Abbildung 8.1 D1 und D2 werden die Mess-Ergebnisse für verschiedene Epsilon Werte dargestellt. Sowohl die CPU- als auch die GPU-Implementierung verwenden hier die Indexstruktur.

Es wurden Datensätze mit 262 144 Punkten getestet. MinPts ist 4, die Datensätze sind 8-dimensionale. Epsilon wird von 0.02 bis 0.10 getestet.

Der Speedup bleibt im Bereich 8.9 bis 10.4, die Wahl von Epsilon hat hier also kaum einen Einfluss.

8.8 Auswirkung der Dimensionalität

In Abbildung 8.1 E1 und E2 werden die Mess-Ergebnisse für eine variierende Anzahl an Dimensionen dargestellt. Sowohl die CPU- als auch die GPU-Implementierung verwenden hier die Indexstruktur.

Es wurden Datensätze mit 262 144 Punkten getestet. MinPts ist 4, Epsilon ist 0.05. Die Dimensionalität wird von 4 bis 64 getestet.

Der Speedup variiert zwischen 7.3 und 12.1 .

Die Dimensionalität bestimmt auf der CPU die Grösse der Variable für den aktiven Punkt, da die Dimensionalität bei der verwendeten Implementierung zur Compilezeit bekannt ist, könnten hier Compiler-Optimierungen, wie z.B. das Auslagern in Register einen Einfluss haben.

Auch auf der GPU bestimmt die Dimensionalität die Grösse der Variable für die aktiven Seed-Punkte, da diese im Shared Memory gespeichert werden, hat dies einen Einfluss darauf wie viele Threads parallel ausgeführt werden können.

Da das Device Memory auf der GPU in Speicher-Bänke aufgeteilt ist, und eine Speicherbank nur eine Speicherzelle gleichzeitig auslesen kann, könnte das Layout der Punkte im Speicher, das ja auch durch die Dimensionalität bestimmt wird, auch einen Einfluss auf die Laufzeit, und somit auf den Speedup haben.

Mit zunehmender Dimensionalität nimmt der Speedup scheinbar ein wenig zu.

8.9 Profiling

Eine Analyse der Rechenzeit verschiedener Programm-Abschnitte beim GPU-DBScan bei einem Datensatz mit 524 288 8-dimensionalen Punkten hat folgendes ergeben :

- ca. 90 % der Zeit wird im Haupt-Kernel verbracht
- ca. 8 % der Zeit wird im Finish-Seeds Bereich verbracht, wo auch der NewSeeds-Kernel enthalten ist.
- ca. 1 % der Zeit wird im Check-Seedstate Bereich verbracht, wo auch die Connection-Matrix ausgewertet wird.
- der Refill-Kernel nimmt unter 1 % der Rechenzeit ein.

Bei einem Datensatz mit 4 194 304 8-dimensionalen Punkten :

- ca. 92 % der Zeit wird im Haupt-Kernel verbracht
- ca. 5 % der Zeit wird im Finish-Seeds Bereich verbracht, wo auch der NewSeeds-Kernel enthalten ist.
- ca. 1 % der Zeit wird im Check-Seedstate Bereich verbracht, wo auch die Connection-Matrix ausgewertet wird.
- der Refill-Kernel nimmt unter 3 % der Rechenzeit ein.

Der Grossteil der Rechenlast liegt also im Hauptkernel, und die Hilfskernel und Berechnungen auf der CPU verbrauchen wie gewünscht nur einen Bruchteil der gesamten Rechenzeit.

Zu bemerken ist hierbei auch, dass die CPU während der Ausführung des Haupt-Kernels, der den Grossteil der Gesamtzeit beansprucht, nur auf dessen Abschluss wartet. Die CPU wird also die meiste Zeit nicht genutzt, und steht für andere Aufgaben zur Verfügung.

8.10 Erzeugung des Index

Bei den vorgestellten Messungen war die Zeit für die Index-Erzeugung weder bei der GPU noch bei der CPU mit eingerechnet. Diese ist jedoch vernachlässigbar, da sie nur einen kleine Bruchteil der Rechenzeit ausmacht. In allen Fällen unter 5% der Rechenzeit von GPU-DBScan, bei den grösseren Datenmengen sogar unter 2%. Der Index ist unabhängig von den Parametern MinPts und Epsilon.

- N=262144 D=64 index:1.8s gpu:111.6s idx/gpu:1.6%
- N=262144 D=32 index:1.0s gpu:45.9s idx/gpu:2.2%
- N=262144 D=16 index:0.6s gpu:38.6s idx/gpu:1.6%
- N=262144 D=8 index:0.4s gpu:16.5s idx/gpu:2.4%
- N=65536 D=8 index:0.1s gpu:2.2s idx/gpu:4.5%
- N=131072 D=8 index:0.2s gpu:5.8s idx/gpu:3.4%
- N=262144 D=8 index:0.4s gpu:16.5s idx/gpu:2.4%
- N=524288 D=8 index:0.8s gpu:52.1s idx/gpu:1.5%
- N=1048576 D=8 index:1.7s gpu:182.4s idx/gpu:0.9%
- N=2097152 D=8 index:3.6s gpu:681.8s idx/gpu:0.5%

Kapitel 9

Zusammenfassung

In dieser Arbeit wurden Verfahren vorgestellt, um die Clustering-Verfahren KMeans und DBScan mit Hilfe einer CUDA-fähigen Grafikkarte erheblich zu beschleunigen.

Die Verfahren wurden für eine massiv parallele Ausführung auf dem Grafikprozessor entworfen, so dass sie die Rechenpower der Grafikkarte ausnutzen können.

Sie wurden an die speziellen Einschränkungen auf dem Grafikprozessor angepasst.

Einige Unterschiede zwischen CUDA und Shader-basiertem GPGPU, und einige der verwendeten Techniken zu denen CUDA den Zugang eröffnet wurden genauer erläutert.

Es wurde eine einfache Indexstruktur vorgeschlagen, die für DBScan auf der GPU genutzt werden kann.

Kapitel 10

Ausblick und Verbesserungen

CUDA hat neue Möglichkeiten für die Verwendung von Grafikkarten eröffnet, die mit bisherigem GPGPU nicht möglich waren.

Dabei wird massiv parallele Berechnung verwendet, die auch bei CPUs immer mehr an Bedeutung gewinnt.

CPUs werden aufgrund physikalischer Grenzen kaum noch schneller, stattdessen geht die Entwicklung auch hier in die Breite, in Form von mehreren Cores. Dual-Core und auch Quad-Core CPUs sind heute schon weit verbreitet, und die Entwicklung in diese Richtung geht ständig weiter.

Die in dieser Arbeit vorgeschlagenen Implementierungen lassen sich noch auf verschiedene Arten verbessern :

Verfahren um Datenmengen, die nicht in das Device Memory passen, zu bearbeiten.

Eine Änderung der Abarbeitungsreihenfolge bei DBScan, so dass sich die Epsilon-Umgebungen der aktiven Seed-Punkte mehr überlappen, würde es ermöglichen, die zur Verfügung stehende Bandbreite auf der Grafikkarte noch um einiges effektiver ausnutzen.

Bei KMeans könnte das Berechnen der Kosten und der neuen Cluster-Zentren auch noch auf der GPU umgesetzt werden.

Die verwendete Indexstruktur für DBScan wurde wegen ihrer Einfachheit ausgewählt, hier lassen sich noch bessere Lösungen finden.

Auch bei KMeans ist es möglich Indexstrukturen zu verwenden, um die Neuberechnungen von Punkten, die ihre Cluster-Zugehörigkeit nur Anfangs ändern, stark zu reduzieren.

Die CPU selbst könnte während den Berechnungen auch noch verwendet werden, bei der bisherigen Umsetzung wartet diese nur untätig während die Programm-Abschnitte auf der Grafikkarte ausgeführt werden.

CUDA ermöglicht es auch mehrere Grafikkarten parallel zu verwenden, um die Rechenleistung noch weiter zu erhöhen, hierfür muss jedoch die Datenmenge und die Arbeit sinnvoll aufgeteilt werden, und die Ergebnisse anschliessend zusammengeführt werden.

Bei Speicherzugriffen und Caching auf der Grafikkarte gibt es unter anderem wegen der Parallelität vieles zu beachten, hier wären bei beiden Verfahren noch Optimierungen möglich.

Die Programmierbarkeit der Grafikkarten hat mit den Zugriffsmöglichkeiten die NVIDIA CUDA und ATI Stream eröffnen eine neue Dimension erreicht, und macht die GPU für eine Vielzahl von nicht-grafischen Anwendungen interessant.

Insgesamt konnte gezeigt werden, dass es durchaus möglich und sinnvoll ist, die GPU für Clustering Algorithmen zu nutzen, und dass dabei auch Indexstrukturen verwendet werden können.

Beim Entwurf von solchen Algorithmen und Indexstrukturen wird parallele Programmierung in Zukunft immer mehr an Bedeutung gewinnen.

Abbildungsverzeichnis

Literaturverzeichnis

[AC01] Domenica Arlia and Massimo Coppola. *Experiments in Parallel Clustering with DBSCAN*. In Euro-Par '01: Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing, pages 326–331, London, UK, 2001. Springer-Verlag.

[Ben75] Jon Louis Bentley. *Multidimensional binary search trees used for associative searching*. Commun. ACM, 18(9):509–517, 1975.

[BKP06] Stefan Brecheisen, Hans-Peter Kriegel, and Martin Pfeifle. *Parallel Density-Based Clustering of Complex Objects*. In Wee Keong Ng, Masaru Kitsuregawa, Jianzhong Li, and Kuiyu Chang, editors, PAKDD, volume 3918 of *Lecture Notes in Computer Science*, pages 179–188. Springer, 2006.

[BPZN09] Christian Böhm, Claudia Plant, Andrew Zherdin, and Robert Noll. *Index-supported Similarity Join on Graphics Processors, BTW 2009*, 2009.

[BSAEA04] Nagender Bandi, Chengyu Sun, Divyakant Agrawal, and Amr El Abbadi. *Hardware acceleration in commercial databases: a case study of spatial operations*. In VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases, pages 1021–1032. VLDB Endowment, 2004.

[CTZ06] Feng Cao, Anthony K. H. Tung, and Aoying Zhou. *Scalable Clustering Using Graphics Processors*. In WAIM, pages 372–384, 2006.

[EKJX96] Martin Ester, Hans-Peter Kriegel, S. Jörg, and Xiaowei Xu. *A density-based algorithm for discovering clusters in large spatial databases with noise*, 1996.

[GGKM06] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. *GPUTeraSort: high performance graphics co-processor sorting for large database management*. In SIGMOD '06: Proceedings

of the 2006 ACM SIGMOD international conference on Management of data, pages 325–336, New York, NY, USA, 2006. ACM.

[GLW+04] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. *Fast computation of database operations using graphics processors.* In SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data, pages 215–226, New York, NY, USA, 2004. ACM.

[HH04] J.D. Hall and J.C. Hart. *GPU Acceleration of Iterative Clustering. in: The ACM Workshop on GPC on GPU and SIGRAPH (2004)*, 2004.

[LR08] David Luebke and NVIDIA Research. *Beyond Programmable Shading: In Action: CUDA Fundamentals, SIGGRAPH 2008*, 2008.

[LSS08] Michael D. Lieberman, Jagan Sankaranarayanan, and Hanan Samet. *A Fast Similarity Join Algorithm Using Graphics Processing Units.* In ICDE, pages 1111–1120, 2008.

[NVI08] NVIDIA. *NVIDIA CUDA Programming Guide 2.0*, 2008.

[OJL+07] Owens, D. John, Luebke, David, Govindaraju, Naga, Harris, Mark, Kruger, Jens, Lefohn, E. Aaron, Purcell, and J. Timothy. *A Survey of General-Purpose Computation on Graphics Hardware.* Computer Graphics Forum, 26(1):80–113, March 2007.

[SAEA03] Chengyu Sun, Divyakant Agrawal, and Amr El Abbadi. *Hardware acceleration for spatial selections and joins.* In SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data, pages 455–466, New York, NY, USA, 2003. ACM.

[SDT08] S. A. Shalom, Manoranjan Dash, and Minh Tue. *Efficient K-Means Clustering Using Accelerated Graphics Processors.* In DaWaK '08: Proceedings of the 10th international conference on Data Warehousing and Knowledge Discovery, pages 166–175, Berlin, Heidelberg, 2008. Springer-Verlag.

[SEKX98] Jörg Sander, Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu. *Density-Based Clustering in Spatial Databases: The Algorithm GDBSCAN and Its Applications.* Data Min. Knowl. Discov., 2(2):169–194, 1998.

[Ste56] H. Steinhaus. *Sur la division des corp materiels en parties.* Bull. Acad. Polon. Sci, 1:801–804, 1956.

[XJK99] Xiaowei Xu, Jochen Jäger, and Hans-Peter Kriegel. *A Fast Parallel Clustering Algorithm for Large Spatial Databases.* Data Min. Knowl. Discov., 3(3):263–290, 1999.