Entwicklung algorithmischer Skelette für CUDA am Beispiel von Affintiy Propagation

Christoph Winter

Fakultät für Informatik und Mathematik
Ostbayerische Technische Hochschule Regensburg
Prüfeninger Str. 58
93049 Regensburg
christoph.winter@st.oth-regensburg.de

Abstract: In diesem Artikel wird anhand des Clusteralgorithmus Affinity Propagation (AP) eine Bibliothek bestehend aus algorithmischen Skeletten vorgestellt, mit deren Hilfe Berechnungen auf die GPU verlagert werden können. Nach einer kurzen Beschreibung von AP wird eingehend auf die Implementierung der Bibliothek und deren Skelette eingegangen: Sowohl die abstrakten Bausteine als auch die konkrete Umsetzung und Verwendung für AP werden dargestellt. Durch das hohe Abstraktionsniveau der Bibliothek und durch Nutzung etablierter CUDA/C++ Konzepte wie Iteratoren und Funktionsobjekte entsteht eine Sammlung nützlicher Funktionen, die einerseits als Ergänzung zu bestehenden Bibliotheken, andererseits als Basis für weitere Entwicklungen dient. Die Wiederverwendbarkeit, Wartbarkeit und Übersichtlichkeit der Anwendungen werden durch deren Verwendung gesteigert. Eine kurze Analyse über das Laufzeitverhalten im Vergleich zu anderen Funktionssammlungen zeigt, dass die entwickelten Funktionen bessere Laufzeiten erzielen. Dadurch lässt sich AP unter Nutzung der Skelette im Vergleich zu einer sequenziellen Version um Faktor 40 - 50 beschleunigen.

1 Einführung

Im Rahmen eines studentischen Projektes an der Ostbayerischen Technischen Hochschule Regensburg wurde über zwei Semester der Clusteralgorithmus Affinity Propagation [FD07] mithilfe von Graphics Processing Units (GPUs) parallelisiert. Bei AP handelt es sich um ein Verfahren der Clusteranalyse, bei dem aus einer heterogenen Gesamtheit von Objekten homogene Teilmengen (*Cluster*) identifiziert werden sollen. Um die GPUs zu programmieren, wird Nvidia's Compute Unified Device Architecture (CUDA) verwendet. Diese ermöglicht es Entwicklern auf niedrigem Abstraktionsniveau die Abbildung von Berechnungen und Daten auf die Hardware genauestens zu kontrollieren. In CUDA werden Gruppen von Threads in einzelnen logischen Blöcken zusammengefasst. Während die Threads innerhalb eines Blockes synchronisiert werden können, können Blöcke untereinander nicht synchronisiert werden und müssen in jeder beliebigen Reihenfolge ausführbar sein. Blöcke sind somit ein essenzieller Bestandteil, um Parallelität zu fördern. Zur Inter-Thread Kommunikation innerhalb eines Blockes kann schneller Shared-Memory

alloziert werden, wohingegen Threads verschiedener Blöcke ausschließlich über den globalen Gerätespeicher kommunizieren können. Der Programmcode, der auf der GPU ausgeführt werden soll, wird in sog. Kernel-Funktionen beschrieben, welche auf der GPU n-mal ausgeführt werden.

Das Hauptaugenmerk bei der Implementierung von AP lag auf der Abstraktion von Algorithmus und dem CUDA Programmiermodell durch Nutzung von high-level Bibliotheken und Konzepten. Dadurch wird die Wiederverwendbarkeit und Einfachheit der implementierten Kernel und Bibliotheksaufrufe garantiert. Für viele parallele Berechnungen existieren bereits Bibliotheken wie cuBLAS[1], welche aber lediglich eine C Schnittstelle besitzen. Dadurch können Features der objektorientierten Programmierung in C++ wie Datenkapselung, Vererbung, Polymorphie und Templates nicht genutzt werden, obwohl sie von CUDA unterstützt werden. Können einzelne Teile eines Algorithmus nicht direkt mit Funktionen einer Bibliothek realisiert werden, wird häufig ein monolithischer Kernel implementiert, der weder auf ein sich veränderndes Umfeld (z. B. Einsatz auf verschiedenartigen GPUs) angepasst werden kann, noch Wiederverwendbarkeit durch Abstraktion der Berechnung vom Kernel garantiert. Anpassungen an ein anderes Umfeld müssen bei monolithischen Kerneln direkt im Algorithmus verankert werden, wobei das Grundprinzip der Trennung der Verantwortlichkeiten (»Seperation of Concerns«) verletzt wird: Kenntnis von Hardwarespezifika (z. B. Anzahl Threads pro Block, Anzahl Blöcke) wird mit Wissen um den Algorithmus, also *was* berechnet werden soll, vermischt. Hier können algorithmische Skelette nach [Col89] verwendet werden, um mittels Funktionen höherer Ordnung (HOFs) die algorithmische Struktur eines Problems von konkreten Berechnungen zu abstrahieren.

Für die Realisierung von AP ist es ausreichend, Transformationen und Reduktionen als Funktionen höherer Ordnung zu implementieren. Die Thrust-Bibliothek [BH11] stellt derartige Funktionstemplates zur Verfügung, bei welchen der Entwickler kein detailliertes Wissen über die Zielarchitektur benötigt. Er muss lediglich die Berechnungsvorschrift, die parallel abgearbeitet werden soll, spezifizieren und die Bibliothek entscheidet dann anhand der Hardware mit wie vielen Threads und Blöcken die Berechnung ausgeführt wird. Gerade bei den für AP besonders wichtigen zeilen- und spaltenweisen Reduktionen stößt Thrust jedoch an seine Grenzen. Auch hardwarespezifische Besonderheiten wie zum Beispiel CUDA-Streams können durch die Unterstützung mehrerer Backends (CUDA, OpenMP, TBB) nicht genutzt werden. Durch die verschiedenen Backends steigt auch die Komplexität des Thrust-Quellcodes, wodurch sich Erweiterungen und eigene Entwicklungen schlecht in die bestehende Codebasis einbringen lassen. Aus diesen Gründen wurde eine auf Templates basierende high-level Bibliothek für Funktionen höherer Ordnung implementiert, die auf CUDA spezialisiert, mit Thrust vollständig kompatibel und vor allem leicht erweiterbar ist.

Im Folgenden werden verwandte Arbeiten und der Clusteralgorithmus vorgestellt. Anschließend wird auf die Implementierung von AP und der dort verwendeten HOFs eingegangen. Ein Kapitel über die experimentellen Resultate zur Leistungsfähigkeit der Entwicklungen schließt die Arbeit ab.

[1] https://developer.nvidia.com/cublas [Abruf: 18.04.2014]

2 Einordnung der Arbeit

Es wurde bereits viel Forschung in Richtung des *hierarchischen Clusterings* betrieben. Anstelle fester Zuordnungen von Datenpunkten zu Clustern generiert die Methode des hierarchischen Clusterings Bäume aus sukzessive zusammengeführten Clustern oder eines sukzessive geteilten Clusters. Abhängig vom verwendeten Distanzmaß erreicht [CKO09] Speedups von bis zu 48, wohingegen eine neuere Arbeit [SD13] die sequenzielle Ausführung um Faktor 90 übertrifft.

Auch das klassische *k-Means Clustering* wurde mit CUDA in [ZG09] bereits erfolgreich auf eine GPU portiert, wobei nur die Berechnung der Distanzen auf der GPU ausgeführt wurde. Die Cluster-Centroids wurden aufgrund der GPU-Ergebnisse sequenziell von der CPU aktualisiert. [HtLlDt+09] generalisieren beide Berechnungen, sodass sowohl die Zuordnung der Datenpunkte zum Cluster als auch die iterative k-Centroids Berechnung parallel auf der GPU ausgeführt werden. Sie erzielen dabei Speedups von bis zu 40 verglichen mit der sequenziellen k-Means Variante. [ALK07] entwickelt seine vorherigen Forschungsergebnisse zum *Fuzzy c-Means Clusterings* weiter und zeigt, wie die Implementierung angepasst werden muss, um nicht nur auf Basis von euklidischen Distanzen, sondern auch auf Basis der Mahalanobis und Gustafson-Kessel Metrik zu clustern. Da der globale Speicher der GPUs beim Clustering häufig den limitierenden Faktor darstellt, ist die Arbeit von [WZH09] hervorzuheben, welche sehr große Datensätze, die in ihrer Gesamtheit nicht in den Speicher der GPU passen, mittels k-Means clustern und trotz der beschränkten Bandbreite zur GPU einen Speedup von 11 im Vergleich zur sequenziellen Variante erzielen.

In der Arbeit von [KB13] wird eine Umsetzung von Affinity Propagation auf Clustern von GPUs mit OpenCL beschrieben. Während ihr Ziel die Implementierung des Algorithmus war, legt dieses Paper Wert auf die Programmiermethodik, indem die von CUDA unterstützten Features einer Hochsprache für die Implementierung eigener HOFs in Verbindung mit der Thrust Bibliothek genutzt werden. Dadurch soll die Wiederverwendbarkeit der Kernel sichergestellt werden. Mit Muesli [CK10], SkePu [EK10] und SkelCL [SKG12] existieren bereits objektorientierte HOF-Frameworks für GPUs, welche allerdings keine Skelette für spalten- oder zeilenweise Reduktionen von Matrizen anbieten.

3 Affinity Propagation

Bei einer Eingabe von n Datenpunkten benötigt AP im Allgemeinen drei reelle, dicht besetzte $n \times n$ Matrizen, die im Folgenden mit **S**, **A** und **R** bezeichnet werden. Dabei ist die Matrix **S** die sog. Similarity-Matrix, welche die konstanten paarweisen Distanzen der Datenpunkte speichert. Der Wert $s(i,k)$ mit $i,k \in [0,n)$ gibt dabei an, wie passend Punkt i als Exemplar für Datenpunkt k wäre. Ein Exemplar stellt dabei den Mittelpunkt eines Clusters dar, dem alle zum Cluster zugehörigen Datenpunkte zugeordnet werden. Die sog. Preferences $s(k,k) \in \mathbb{R}_{\leq 0}$ geben an, wie wahrscheinlich Punkt k zum Exemplar gewählt wird: je größer $s(k,k)$ desto eher wird k zum Exemplar eines Clusters bestimmt. Aus-

gehend von $\mathbf{S}$ werden nun in einem iterativen Verfahren pro Wiederholung zwei Arten von Nachrichten zwischen allen Datenpunkten ausgetauscht. Diese werden *Responsibilities* bzw. *Availabilities* genannt und werden durch die Werte in den Matrizen $\mathbf{R}$ bzw. $\mathbf{A}$ repräsentiert.

Die Responsibility $r(i,k) \in \mathbf{R}$ bildet dabei das Verlangen von Datenpunkt i ab, dem Cluster anzugehören, dessen Exemplar Punkt k ist:

$$r(i,k) \leftarrow s(i,k) - \max_{l \neq k} \left\{ a^{t-1}(i,l) + s(i,l) \right\}, \tag{1}$$

wobei $l \in [0,n)$, $t \in \mathbb{N}$ und $\mathbf{A}$ mit 0_{nn} initialisiert wird. Der hochgestellte Index $t-1$ bezeichnet dabei den Wert der vergangenen Iteration. Die Berechnung wird realisiert, indem alle Elemente $r(i,\bullet)$ mithilfe des größten Elements der Zeile i aktualisiert werden, außer das größte Element selbst, welches mit dem zweitgrößten Element aktualisiert wird. Da sich die Rechnung jeweils nur auf eine Zeile der Matrix bezieht, können alle Zeilen parallel berechnet werden.

Nach der Berechnung der Responsibilities werden die Availabilities ermittelt. Die Availability $a(i,k) \in \mathbf{A}$ spiegelt die Bereitschaft des Punktes k wider, als Exemplar für i zu dienen. Dabei wird auch die Unterstützung all jener Knoten berücksichtigt, für die ebenfalls der Knoten k als Exemplar vorteilhaft wäre.

$$a(i,k) \leftarrow \begin{cases} \min\left\{0,\ r(k,k) + \sum_{j \neq i,k} \max\{0, r(j,k)\}\right\} & \text{falls } i \neq k \\ \sum_{j \neq k} \max\{0,\ r(j,k)\} & \text{sonst} \end{cases}. \tag{2}$$

Damit die Berechnung der Availabilities zu einem späteren Zeitpunkt effizient auf mehrere GPUs verteilt werden kann, werden zuerst die Spaltensummen ermittelt.

$$c(k) \leftarrow r(k,k) + \sum_{i \neq k} \max\{0,\ r(i,k)\}. \tag{3}$$

Da keine Datenabhängigkeiten zu anderen Spalten bestehen, können die Summen erneut vollständig parallel bestimmt und im Vektor c gespeichert werden. Im Anschluss werden die Availabilities nach Gleichung (4) aktualisiert, wobei jedes Element $a(i,k)$ anhand der Spaltensumme $c(k)$ und dem Element $r(i,k)$ transformiert wird.

$$a(i,k) \leftarrow \begin{cases} \min\{0,\ c(k) - \max\{0,\ r(i,k)\}\} & \text{falls } i \neq k \\ c(k) - r(k,k) & \textit{sonst} \end{cases}. \tag{4}$$

Die obigen Gleichungen werden wiederholt durchlaufen; dabei terminiert AP entweder nachdem die Exemplare über mehrere Iterationen hinweg stabil geblieben sind, oder nach einer a priori festgelegten Anzahl an Iterationen. Dabei ist ein Datenpunkt d_k genau dann ein Exemplar, wenn gilt $r(k,k)+a(k,k) > 0$. Um ein Oszillieren der Nachrichtenwerte zu verhindern, werden zudem alle neu berechneten Werte für $\mathbf{R}$ und $\mathbf{A}$ mit einem $\lambda \in [0.5, 1)$ gedämpft:

$$\mathbf{R^t} = \lambda \mathbf{R^{t-1}} + (\mathbf{1} - \lambda)\mathbf{R}\ , \qquad \mathbf{A^t} = \lambda \mathbf{A^{t-1}} + (\mathbf{1} - \lambda)\mathbf{A}. \tag{5}$$

4 Implementierung

Bevor auf die konkrete Umsetzung von AP eingegangen werden kann, muss zunächst die Struktur der erstellten Bibliothek erläutert werden. Anstelle eines monolithischen Kernels wird jede Berechnung an ein datenparalleles Skelett der Bibliothek weitergeleitet, welches die Berechnung auf der GPU durchführt. Abbildung 1 zeigt den grundlegenden Aufbau eines generischen Transformations- und Reduktions-Skeletts. Ähnlich dem Programmiermodell von CUDA setzt sich die Bibliothek aus drei Bausteinen zusammen: CUDA unterscheidet zwischen Kernel-Funktionen, Host-Code und Device-Code. Während Kernel-Funktionen mit der Anzahl der Threads, Anzahl der Blöcke und der benötigten Größe des Shared-Memory parametrisiert und von der GPU ausgeführt werden können, müssen Unterfunktionen eines Kernels für den Nvidia-Compiler als Device-Code markiert werden. Der Host-Code hingegen ist nur auf der CPU ausführbar und ist deshalb der Standard für alle nicht markierten Funktionen. Diese Struktur findet sich auch in der Bibliothek wieder: Der Aufruf an die Bibliothek ① läuft komplett als Host-Code auf der CPU ab. Ein einfaches Funktionsobjekt (*Closure*) in ② kapselt den Device-Code innerhalb des ( )-Operators und ein Dispatcher ③ führt das Closure aus, indem ein Kernel den ( )-Operator des Funktionsobjektes aufruft. Da der Dispatcher lediglich ein Closure ausführt, ist der Device-Code

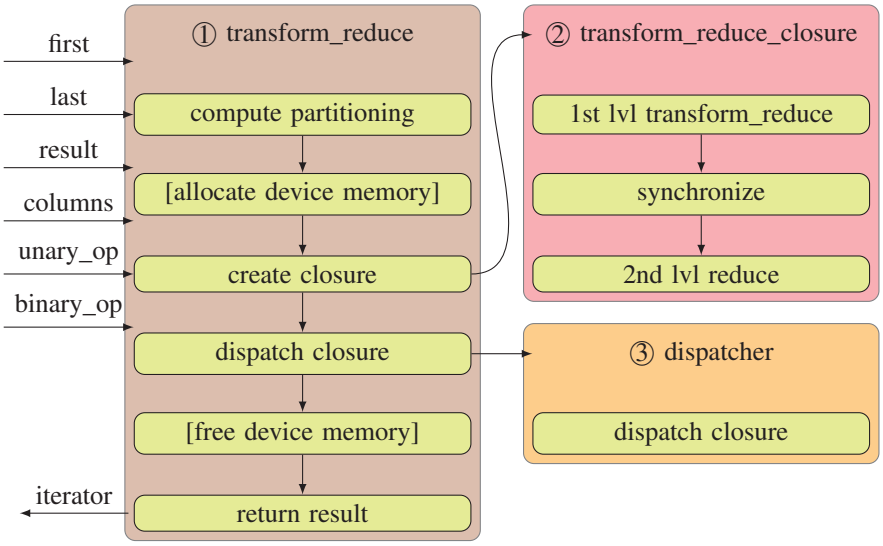

Abbildung 1: Das Blockdiagramm veranschaulicht ein datenparalleles Skelett, bei dem zuerst alle Elemente transformiert und anschließend reduziert werden. Der Aufruf gliedert sich in drei Teile, dem Host- bzw. Device-Code und dem Kernelaufruf. Im Host-Code wird sowohl die optimale Anzahl an Threads und Blöcken berechnet als auch die korrekte Menge an Shared-Memory alloziert und ein Closure, das den Device-Code enthält, erstellt. Anschließend wird das Closure vom Dispatcher in einem Kernelaufruf ausgeführt.

von der Kernel-Funktion, d. h. dem Einsprungpunkt der GPU, getrennt. Dadurch lässt sich

die Logik des Kernels komplett in einem Closure beschreiben. Der ( )-Operator des Closures enthält somit das gesamte Wissen, *wie* die Berechnung auf der GPU ausgeführt wird. Somit können verschiedene Implementierungen, z. B. unterschiedliche Zugriffsmuster auf den Speicher, getestet und iterativ verbessert werden, ohne Änderungen im Anwendungscode vornehmen zu müssen. Von den Verbesserungen profitieren alle Applikationen, die die Bibliothek nutzen, da die Bibliothek als zentrale Anlaufstelle für Berechnungen auf der GPU dient. Dadurch wird sowohl die Wartbarkeit als auch die Übersichtlichkeit aller Anwendungen gesteigert. Die eigentliche Berechnungsvorschrift, d.h. *was* berechnet werden soll, wird ebenfalls über Funktionsobjekte im Aufruf der Bibliotheksfunktion in ① übergeben. Exemplarisch wird im Beispiel sowohl eine unäre Transformation (unary_op) als auch eine binäre Operation zur Reduktion (binary_op) verwendet. In diesen Funktionsobjekten formuliert der Endanwender die Anweisungen der durchzuführenden Berechnung, also welche Daten mit welchen Operationen verknüpft werden sollen. Die Berechnung an sich ist domänenspezifisches Wissen und sollte somit von der Bibliothek strikt getrennt werden. Die Bibliothek übernimmt durch Kenntnis der Hardware lediglich die Partitionierung der Eingabedaten und die Abbildung von Threads auf Partitionen der Daten. Um die Idee zu konkretisieren, wird nun anhand der ersten Gleichung zur Berechnung der Responsibilities die Nutzung der Bibliothek demonstriert.

Nach Gleichung (1) muss zunächst die größte und zweitgrößte Summe aus a^{t-1} und s jeder Zeile bestimmt werden. Da durch die Verwendung von Templates und Iteratoren die Bibliothek von Datentypen und Zeigern entkoppelt wurde, kann die Berechnung auf das folgende Codefragment reduziert werden.

Listing 1: Berechnung des größten und zweitgrößten Elements einer Zeile

```
ap::transform_reduce_row(
    thrust::make_zip_iterator(
            make_tuple( similarities.begin(), availabilities.begin() )),
    thrust::make_zip_iterator(
            make_tuple( similarities.end(), availabilities.end() )),
    thrust::make_zip_iterator(
            make_tuple( mx1.begin(), mx2.begin() )),
    columns,
    convert_to_tuple< thrust::tuple<value_type,value_type> >(),
    reduce_tuple< thrust::tuple<value_type,value_type> >() );
```

Die Bibliothek ist vollständig kompatibel zu allen Iteratoren von Thrust. So wird hier der Zip-Iterator benutzt, um aus den einzelnen Device-Vektoren ein Structure of Arrays (SoA) zu erzeugen [BH11]. Das unäre Funktionsobjekt (convert_to_tuple) addiert die einzelnen Elemente des SoA und transformiert die Summe in ein Tupel, welches mit dem binären Funktionsobjekt (reduce_tuple) reduziert wird. Die Resultate werden pro Zeile in die Ergebnisvektoren mx1 bzw. mx2 zurückgeschrieben und alle Elemente in **R** in einem nachfolgenden, hier nicht weiter aufgeführten Aufruf, entsprechend aktualisiert.

Die Funktionsobjekte sind dabei sehr einfach aufgebaut und enthalten lediglich domänenspezifisches Wissen darüber, *was* berechnet werden soll:

Listing 2: Funktionsobjekte

```
template<class T>
struct convert_to_tuple : public std::unary_function<T,T> {

    __host__ __device__
    T operator()(const T& value) const {
        return thrust::make_tuple(
            thrust::get<0>(value)+thrust::get<1>(value),-MAX_VALUE );
    }
};

template<class T>
struct reduce_tuple : public std::binary_function< T, T, T > {
__host__ __device__
T operator()(const T& a, const T& b) const {
        typedef typename thrust::tuple_element<0,T>::type value_type;

        value_type max1 = thrust::max(
            thrust::get<0>(a), thrust::get<0>(b) );

        value_type max2 = thrust::max(
            thrust::min(
                thrust::get<0>(a), thrust::get<0>(b)),
            thrust::max(
                thrust::get<1>(a), thrust::get<1>(b)));

        return thrust::make_tuple(max1,max2);
    }
};
```

Der Aufruf der Bibliothek sorgt dafür, dass die Reduktion zeilenweise durchgeführt wird. Da alle Matrizen zeilenweise im Speicher abgelegt sind, wird jede Zeile von einem Thread-Block bearbeitet. Da nur ein Block pro Zeile verwendet wird, entfallen sowohl Synchronisationspunkte zwischen den Blöcken als auch das Allozieren globalen Device-Memories zum Speichern der Zwischenergebnisse jedes einzelnen Blocks. Stattdessen schreibt jeder Block seine Zwischenergebnisse der ersten Reduktion in den weitaus schnelleren Shared-Memory [Nvi12]. Nach der Transformation werden in einer ersten Reduktionsphase alle n Eingabedaten auf die Anzahl der Threads pro Block tpb zurückgeführt. In einer zweiten Reduktion werden die Zwischenergebnisse auf ein Endergebnis pro Zeile verdichtet. Eine einfache Heuristik übernimmt die Berechnung der Anzahl der Threads pro Block. Durch den übersichtlichen Aufbau der Bibliothek könnte die Heuristik leicht über einen optionalen Funktionsparameter durch eigene Implementierungen ausgetauscht werden.

Um die Availabilities zu berechnen, werden zuerst die Spaltensummen nach Gleichung (3) gebildet. Damit sichergestellt ist, dass zusammen ausgeführte Threads eines Blocks (*Warp*) von zusammenhängenden Speicherbereichen lesen (*memory coalescing*), bearbeitet ein Thread genau eine Spalte. Ein Block bearbeitet somit eine Gruppe zusammenhängender Spalten, wobei bei einer Eingabe von n Datenpunkten mit tpb Threads pro Block insgesamt $\lceil n/tpb \rceil$ Blöcke benötigt werden. Um die Anzahl der Blöcke und dadurch die Parallelität zu erhöhen, wird die Matrix zusätzlich zeilenweise in m gleichgroße Teile zerlegt, die jeweils parallel transformiert und reduziert werden. Die $m \times n$ vielen Zwischenergebnisse werden in einem temporär allozierten Device-Speicher abgelegt und anschließend in einer letzten Reduktionsphase zu den n Endergebnissen zusammengefasst. Ein letzter Kernelaufruf traversiert die Diagonale $diag(\mathbf{R})$ und addiert alle Elemente den

entsprechenden Spaltensummen hinzu. Nachdem die Spaltensummen nach (3) ermittelt wurden, werden alle Elemente in **A** entsprechend Gleichung (4) transformiert. Während der Transformation werden die Exemplare anhand der Ergebnisse der Transformation bestimmt, wodurch weitere Zugriffe auf den globalen Speicher der GPU entfallen.

5 Experimentelle Ergebnisse

Die Test-Plattform ist mit einer Intel Xeon E5-2667 Hexacore CPU, 6×16 GB RAM und einer Nvidia Tesla K20 mit 5 GB Speicher ausgestattet. Es wurden ausschließlich die Werkzeuge des CUDA Toolkits 6.0 verwendet, sodass der GPU-Code mit dem Nvidia Cuda Kompiler Treiber (nvcc) der Version 6.0.1, der CPU-Code hingegen mit dem GNU C Kompiler 4.7.2 übersetzt wurde. Bei der Kompilierung der Programme wurden keine Optimierungsoptionen angegeben. Die Gerätetreiber lagen in der Version 331.62 vor.

Der limitierende Faktor dicht besetzter Affinity Propagation ist der Speicherverbrauch. Dieser wird von den drei quadratischen Matrizen **S**, **R** und **A** dominiert. Hinzu kommen die Vektoren zum Speichern der Spaltensummen (c), der größten und zweitgrößten Werte pro Zeile ($mx1$, $mx2$), sowie zwei Vektoren (e^t, e^{t-1}), um AP auf Konvergenz zu überwachen. Insgesamt beläuft sich der Speicherverbrauch also auf $3n^2 + 5n$. Verzichtet man auf die Überwachung des Konvergenzkriteriums, kann der benötigte Speicher auf $3n^2 + 3n$ gesenkt werden. Bei einfacher Genauigkeit lassen sich auf einer K20 GPU somit etwa 20 k Punkte, bei doppelter Genauigkeit 14 k Datenpunkte clustern.

In Abbildung 2 werden zur Veranschaulichung die für AP entwickelten HOFs mit anderen Bibliotheken und einer sequenziellen CPU Variante verglichen. In der Abbildung werden die Messpunkte aus Gründen der Übersichtlichkeit auf einer logarithmischen Skala aufgetragen. Um das Skalierungsverhalten besser zu veranschaulichen, wurde jeweils der erste Messpunkt mit der kleinsten Problemgröße und der letzte Messpunkt mit der größten Problemgröße mit der exakten Laufzeit versehen. Ein Algorithmus skaliert dabei umso schlechter, je höher die einzelnen Graphen auf der y-Achse verzeichnet sind. Bei der sequenziellen Variante wird die Zeilensumme klassisch über zwei for-Schleifen berechnet. Die äußere Schleife iteriert dabei über die einzelnen Zeilen, die innere Schleife über die Spalten. Die sequenzielle Laufzeit ist mit bis zu 4.1 s bei der größten Problemgröße mit Abstand am langsamsten; zudem skaliert dieser einfache Algorithmus schlecht. Thrust hingegen weist jedem Wert der Eingabe einen *Key* zu, über den alle zugehörigen Werte reduziert werden. Dadurch werden sehr individuelle Reduktionen möglich, die Verallgemeinerung schlägt sich jedoch auch in einer langsamen Ausführungsgeschwindigkeit nieder. Auch wenn dieser Aufruf mit Speedups von bis zu Faktor 16 im Vergleich zum sequenziellen Aufruf weitaus besser skaliert, können die eigens für AP entwickelten HOFs und die cuBLAS Bibliothek die Laufzeit nochmals um Faktor $9 - 11$ beschleunigen. Bei dem cuBLAS Aufruf wird die Matrix mit einem Einsvektor multipliziert, was der Berechnung der Zeilensummen entspricht. Dabei müssen sowohl für Thrust als auch für cuBLAS zusätzliche Hilfsvektoren der Größe n alloziert werden: Thrust erfasst darin die *Keys* während cuBLAS den Einsvektor speichert. Des Weiteren kann bei keiner der beiden Bibliotheken vor der eigentlichen Reduktion eine Transformation ausgeführt werden.

Stattdessen müsste die Transformation in einem gesonderten Kernel der Reduktion vorangestellt werden. Für das Speichern der Ergebnisse der Transformation muss dann ggf. erneut Speicher alloziert werden. Für das Ermitteln der Spaltensummen in Gleichung (3) würde der Speicherbedarf demnach auf $4n^2+5n$ anwachsen, da eine von negativen Werten bereinigte Matrix $\mathbf{R}^+$ im Speicher alloziert werden müsste. Hinzu kommen die unnötigen Zugriffe auf den globalen Hauptspeicher der GPU, da jede Operation die Daten wieder aus dem globalen Hauptspeicher laden muss. Durch die Verschmelzung von Transformation und Reduktion innerhalb der spezialisierten algorithmischen Skelette lässt sich dieser Mehraufwand vermeiden. Für eine einfache zeilen- oder spaltenweise Summation von Ma-

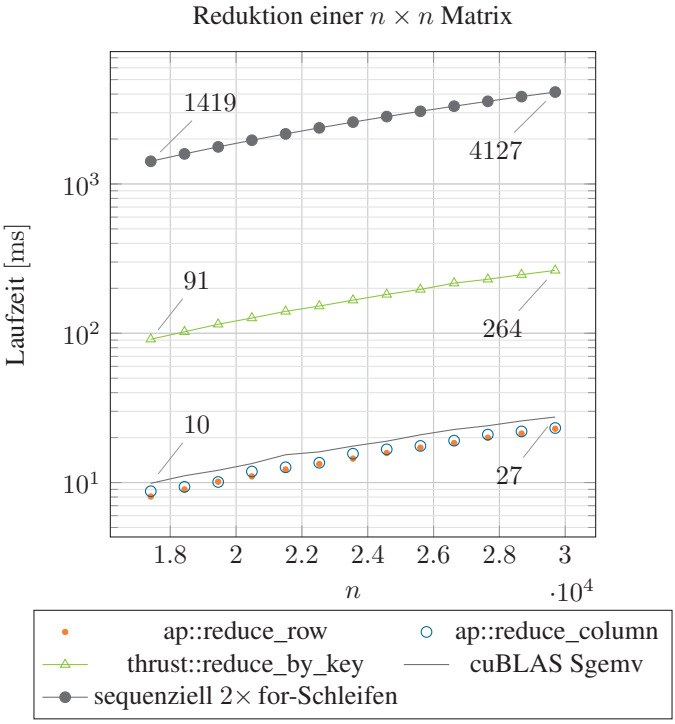

Abbildung 2: Vergleich verschiedener Bibliotheken bei der Berechnung von zeilen- und spaltenbasierten Reduktionen einer $n \times n$ Matrix bei einfacher Genauigkeit.

trixelementen lässt sich mit den neu entwickelten HOFs im Vergleich zur sequenziellen Variante ein maximaler Speedup von bis zu 185 erzielen. Der Speedup relativiert sich jedoch im Zusammenhang mit komplizierteren Berechnungsvorschriften im AP-Algorithmus: Da die Berechnungen oft Datenabhängigkeiten zu anderen Eingaben aufweisen (Gleichungen (1), (2) bzw. (4)), sind weitere Zugriffe auf den globalen Device-Speicher nötig, welche sich negativ auf die Laufzeit auswirken. Um den Speedup von AP zu messen, wurde die GPU Variante mit der originalen sequenziellen Referenzimplementierung in C von Frey & Dueck verglichen[2]. Pro Testlauf wurden 1000 Iterationen jeweils mit einfacher und

[2]http://www.psi.toronto.edu/affinitypropagation/apcluster.txt [Abruf: 18.04.2014]

Problemgröße	Sequenzielle Ausführung [s]	GPU Laufzeit [s]	Speedup
	Doppelte Genauigkeit		
1.000×1.000	21.8	0.8	27.3
2.000×2.000	88.5	2.3	38.3
4.000×4.000	342.0	8.7	39.3
6.000×6.000	759.4	19.1	39.8
8.000×8.000	1335.3	33.6	39.7
10.000×10.000	2050.0	51.6	39.7
12.000×12.000	2972.3	74.4	40
	Einfache Genauigkeit		
1.000×1.000	21	0.7	30
2.000×2.000	84.3	1.8	46.8
4.000×4.000	324.8	6.2	52.4
6.000×6.000	721	14.1	51.1
8.000×8.000	1273.8	23.8	53.5
10.000×10.000	1982	38.9	51
12.000×12.000	2857.8	55.2	51.8

Tabelle 1: Mittlere Laufzeiten von 1000 Iterationen der GPU Variante (ECC aktiviert) im Vergleich zu sequenziellem AP.

doppelter Genauigkeit durchgeführt. Jeder Durchlauf wurde fünfmal wiederholt und der Mittelwert gebildet.

Tabelle 1 zeigt, dass im Vergleich zur sequenziellen Variante Speedups von Faktor $40 - 50$ möglich sind. Rechnet man statt einfacher mit doppelter Genauigkeit, hat man mit Einbußen von bis zu $30\,\%$ zu rechnen, was in etwa dem Verhältnis der Recheneinheiten für einfache und doppelte Genauigkeit einer K20 GPU entspricht. Die Anzahl der gefundenen Cluster ist dabei sowohl bei der GPU- als auch bei der CPU-Variante gleich und bei den verwendeten synthetischen Eingabedaten unabhängig von der Genauigkeit. Auch die Zuordnung von Datenpunkten zu deren Exemplaren stimmt bei allen Varianten überein. Sind in den Ausgangsdaten exakt gleiche Punkte enthalten, kann es lediglich zu einer anderen Auswahl des Exemplars unter allen identischen Datenpunkten kommen, was jedoch keinen Einfluss auf die Güte des Ergebnisses hat. Insgesamt konnte AP durch die Nutzung der entwickelten Bibliotheksaufrufe im Vergleich zur sequenziellen Version deutlich beschleunigt werden. Durch die Zusammenführung von Transformation und Reduktion konnten sowohl Zugriffe auf das globale Device-Memory eingespart als auch der Speicherverbrauch auf das für dicht besetzte AP nötige Minimum reduziert werden.

6 Zusammenfassung und Ausblick

Im Gegensatz zu [KB13] wurde in dieser Arbeit C++/CUDA zusammen mit der Thrust Bibliothek [BH11], vor allem aber eigene high-level Erweiterungen zur Parallelisierung von Affinity Propagation verwendet. Die entwickelte Bibliothek mit ihren hochperformanten algorithmischen Skeletten ermöglicht mit geringem Programmieraufwand den Clusteralgorithmus zu implementieren, wobei die Laufzeiten der Operationen im Vergleich zu Thrust und einer sequenziellen Variante enorm verbessert werden konnten. Verglichen mit einer sequenziellen Version von AP konnte schlussendlich ein Speedup von bis zu Faktor 50 erzielt werden. Die gesamte Bibliothek basiert dabei auf dem Designprinzip der Trennung der Verantwortlichkeiten, sodass das Wissen *wie* eine Berechnung auf der Hardware optimal ausgeführt wird von der Berechnungsvorschrift, d.h. *was* gerechnet werden soll, isoliert wird. Durch diese klare Softwarearchitektur wird sowohl die Wiederverwendbarkeit der Bibliothek als auch die Wartbarkeit und Einfachheit der Endanwendung gesteigert ohne dabei auf Leistung verzichten zu müssen. Eine Masterarbeit wird die Bibliothek um ein CUDA-aware Message Passing Interface (MPI) erweitern. Dadurch wird die Bibliothek auch auf Clustern von GPUs einsetzbar sein. Affinity Propagation wird dabei weiterhin als Beispielanwendung dienen, sodass mehr Datenpunkte geclustert werden können, als der Speicher einer einzelnen GPU fassen kann. Da die Anwendung keine Kenntnis über die tatsächliche Hardware hat, muss lediglich die Bibliothek um das Wissen mehrerer GPUs erweitert werden. Die parallele Implementierung des Clusteralgorithmus für eine GPU wird bereits vom Institut für Funktionelle Genomik der Universität Regensburg[3] für ihre Forschungen verwendet. Dort clustert AP biologische Proben, um Unterschiede im Metabolismus in Abhängigkeit von biologischen Randbedingungen zu erkennen. In einer der vielen Fragestellungen wird untersucht, inwiefern sich der Stoffwechsel von Krebszellen von dem normaler Zellen unterscheidet [DVR+13].

7 Danksagung

Nvidia hat dieses Forschungsvorhaben der Hochschule mit zwei K20 Tesla GPUs im Zuge einer akademischen Partnerschaft unterstützt.

Literatur

[ALK07] Derek Anderson, RobertH. Luke und JamesM. Keller. Incorporation of Non-euclidean Distance Metrics into Fuzzy Clustering on Graphics Processing Units. In *Analysis and Design of Intelligent Systems using Soft Computing Techniques*, Advances in Soft Computing, Seiten 128–139. Springer Berlin Heidelberg, 2007.

[BH11] Nathan Bell und Jared Hoberock. Thrust: A Productivity-Oriented Library for CUDA. *GPU Computing Gems: Jade Edition*, Seiten 359–373, 2011.

[3] https://genomics.uni-regensburg.de [Abruf: 18.04.2014]

[CK10] P Ciechanowicz und H Kuchen. Enhancing Muesli's Data Parallel Skeletons for Multi-core Computer Architectures. *IEEE 12th International Conference on High Performance Computing and Communications (HPCC)*, Seiten 108–113, September 2010.

[CKO09] DJ Chang, MM Kantardzic und Ming Ouyang. Hierarchical Clustering with CUDA/GPU. *ISCA PDCCS*, Seiten 7–12, 2009.

[Col89] Murray I. Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989.

[DVR+13] Katja Dettmer, Franziska C Vogl, Axel P Ritter, Wentao Zhu, Nadine Nürnberger, Marina Kreutz, Peter J Oefner, Wolfram Gronwald und Eva Gottfried. Distinct metabolic differences between various human cancer and primary cells. *Electrophoresis*, 34(19):2836–2847, Oktober 2013.

[EK10] Johan Enmyren und Christoph W. Kessler. SkePU: A multi-backend skeleton programming library for multi-GPU systems. *Proc. 4th Int. Workshop on High-Level Parallel Programming and Applications (HLPP-2010)*, 2010.

[FD07] Brendan J Frey und Delbert Dueck. Clustering by passing messages between data points. *Science (New York, N.Y.)*, 315(5814):972–6, Februar 2007.

[HtLlDt+09] Bai Hong-tao, He Li-li, Ouyang Dan-tong, Li Zhan-shan und Li He. K-Means on Commodity GPUs with CUDA. In *Computer Science and Information Engineering, 2009 WRI World Congress on*, Jgg. 3, Seiten 651–655, 2009.

[KB13] Marcin Kurdziel und Krzysztof Boryczko. Finding exemplars in dense data with affinity propagation on clusters of GPUs. *Concurrency and Computation: Practice and Experience*, 25(8):1137–1152, 2013.

[Nvi12] Nvidia. Cuda C Programming Guide. Oktober 2012.

[SD13] S A Arul Shalom und Manoranjan Dash. Parallel Computations for Hierarchical Agglomerative Clustering using CUDA. *Parallel & Cloud Computing*, 2(4):90–100, 2013.

[SKG12] Michel Steuwer, Philipp Kegel und Sergei Gorlatch. Towards High-Level Programming of Multi-GPU Systems Using the SkelCL Library. *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, Seiten 1858–1865, Mai 2012.

[WZH09] Ren Wu, Bin Zhang und Meichun Hsu. Clustering billions of data points using GPUs. *Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop (UCHPC-MAW)*, Seiten 1–6, 2009.

[ZG09] M Zechner und M Granitzer. Accelerating K-Means on the Graphics Processor via CUDA. In *First International Conference on Intensive Applications and Services, INTENSIVE '09*, Seiten 7–15, April 2009.