Bachelorarbeit

Implementierung des K-Means- und Needleman-Wunsch- Algorithmus mit der SkelCL-Bibliothek

Implementing the K-Means and Needleman-Wunsch Algorithm with the SkelCL Library

vorgelegt am: 27. Juni 2013

an der Westfälischen Wilhelms-Universität Münster

Autor:
Kai Kientopf

Matrikelnummer:
355503

Betreuer:
Prof. Dr. Sergei Gorlatch
Michel Steuwer

Inhaltsverzeichnis

1. Einführung

In dieser Bachelorarbeit werden zwei Algorithmen, für die es bereits eine Implementierung in OpenCL gibt, nach SkelCL portiert. SkelCL baut als Bibliothek auf OpenCL auf und soll das Programmieren für Grafikprozessoren (GPUs) vereinfachen. Mit GPUs lassen sich parallelisierbare Probleme oft weitaus schneller lösen, als auf normalen Prozessoren. Jedoch benötigen Grafikprozessoren immer einen normalen Prozessor, von dem sie gesteuert werden. Durch diesen Systemaufbau ergeben sich neben den normalen Anforderungen beim Erstellen von parallelen Programmen, wie Synchronisation der Prozesse bei Speicherzugriffen, weitere Herausforderungen. Ein populärer Ansatz für das Programmieren von GPUs ist OpenCL. Dieses bietet allerdings nur eine sehr hardwarenahe Programmierung an, die für Anwendungsentwickler als nicht intuitiv und umständlich empfunden werden kann. SkelCL wird deshalb mit dem Ziel entwickelt das Programmieren von GPU-Systemen zu vereinfachen. Deshalb soll mit der Portierung der beiden Programme untersucht werden, wie sich die Komplexität des Quelltextes und die Laufzeit der SkelCL- zu der OpenCL-Implementierung verhält. Zum Portieren wurden der K-Means-Algorithmus aus dem Bereich des Data-Mining und der Needleman-Wunsch-Algorithmus aus der Bioinformatik gewählt. Für beide Algorithmen gibt es OpenCL Referenzimplementierungen in der Rodinia Benchmark Suite[4]. Als Grundlage für die Programmierung werden zunächst OpenCL und SkelCL genauer betrachtet.

1.1. OpenCL

OpenCL (Open Computing Language)[1, 2] definiert eine standardisierte Schnittstelle, über die verschiedene Arten von Parallelprozessoren programmiert werden können. Dies können z.B. Grafikprozessoren oder Mehrkernprozessoren sein. Für die zur Schnittstelle gehörende Implementierung müssen die Hardware-Hersteller drei theoretische Modelle auf ihre eigene Hardware abbilden: Das Plattform-Modell, das Ausführungs-Modell und das Speicher-Modell.

1.1.1. Plattform-Modell

Durch das Plattform-Modell wird der Aufbau eines OpenCL-Systems abstrakt beschrieben. Dieses kann grundsätzlich in einem Host und ein oder mehrere OpenCL-Devices unterteilt werden. In Abbildung 1.1 wird dies beispielhaft mit einem Host und drei OpenCL-Devices dargestellt. Der Host führt das Host-Programm aus, welches die OpenCL-Devices steuert und sequenzielle Berechnungen ausführt. Ein OpenCL-Device selbst wird in Compute Units aufgeteilt. Über eine solche Compute Unit werden Processing Elements ver-

waltet, welche die Rechenoperationen über sogenannte Kernel parallel ausführen können. In Abbildung 1.1 wird zur Verdeutlichung ein OpenCL-Device mit drei Compute Units gezeigt, in denen jeweils drei Processing Elements verwaltet werden.

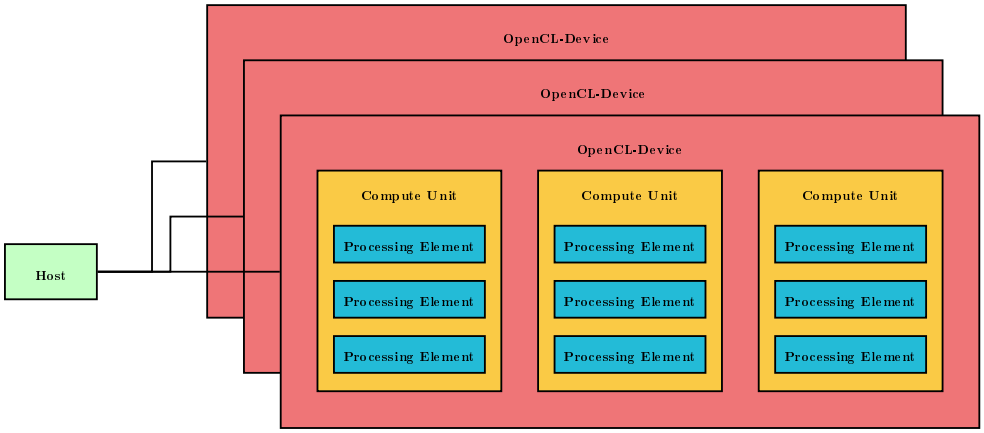

Abbildung 1.1.: OpenCL Plattform-Modell.

1.1.2. Ausführungs-Modell

Ein OpenCL-Programm ist aufgeteilt in ein Host-Programm und ein oder mehrere sogenannte Kernel. Das Host-Programm wird auf dem Host ausgeführt, auf dem es sequenzielle Berechnungen vornimmt, die sich nicht parallelisieren lassen, und die Ausführung der Kernel steuert. Ein Kernel definiert eine Funktion, die auf verschiedenen Daten parallel ausgeführt wird. Zu diesem Zweck werden Instanzen vom Kernel erzeugt - die sogenannten Work-Items. Diese werden in Work-Groups organisiert. Dabei werden in jeder Work-Group gleich viele Work-Items verwaltet, für die garantiert wird, dass sie immer zusammen ausgeführt werden. Im Bezug zum Plattform-Modell werden die Work-Items einer Work-Group auf den Processing Elements einer Compute Unit ausgeführt. Im Host-Programm wird die Anzahl der Work-Items festgelegt und bestimmt, in wie vielen Work-Groups sie organisiert werden. Für die Identifikation auf einem OpenCL-Device gibt es drei IDs. Die Global ID bestimmt jedes Work-Item eindeutig. Um auch die Work-Items einer Work-Group untereinander identifizieren zu können, gibt es die Local ID. Mit der Work-Group ID wird ersichtlich, auf welcher Work-Group ein Work-Item läuft. In der Abbildung 1.2 sollen diese Zusammenhänge beispielhaft verdeutlicht werden. Dort sind auf einem OpenCL-Device drei Compute Units mit je drei Processing Elements vorhanden und es werden neun Work-Items ausgeführt, die in drei Dreiergruppen in Work-Groups aufgeteilt sind. Die Global IDs der Work-Items laufen von 0 bis 8. Über die Work-Group IDs von 0 bis 2 lässt sich bestimmen, in welcher Work-Group ein Work-Item ausgeführt wird und über die Local IDs von 0 bis 2 werden die Work-Items innerhalb der Work-Group identifiziert.

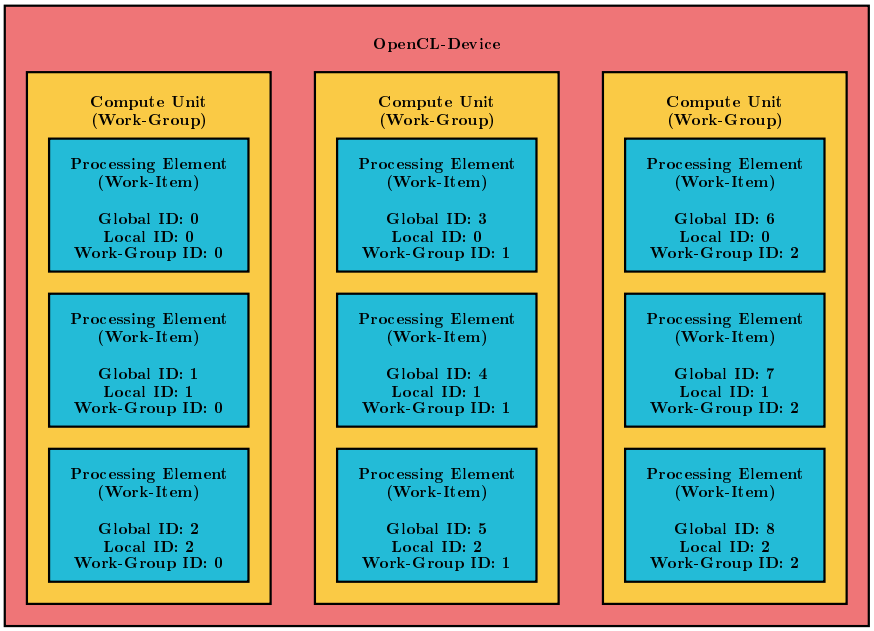

Abbildung 1.2.: Identifizierung der Work-Items und Work-Groups auf einem OpenCL-Device.

1.1.3. Speicher-Modell

Wie beim Plattform- oder Ausführungs-Modell ist auch das Speicher-Modell in unterschiedliche Bereiche aufgeteilt. Der Speicher vom Host ist logisch vom Speicher der OpenCL-Devices getrennt. Bedingt durch diese Trennung müssen die Daten, mit denen die Work-Items rechnen, vom Host auf das OpenCL-Device und anschließend die Ergebnisse zurück in den Speicher des Hosts kopiert werden.
Auf einem OpenCL-Device selbst stehen vier unterschiedliche Arten von Speicher zur Verfügung: Der globale Speicher, der konstante Speicher, der lokale Speicher und der private Speicher.

Globaler Speicher

In dem globalen Speicher können alle Work-Items eines OpenCL-Devices lesend und schreibend zugreifen. Außerdem können Daten vom Host zum globalen Speicher und zurück kopiert werden. Im Bezug zum Plattform-Modell ist der globale Speicher einmal für das komplette OpenCL-Device vorhanden (siehe Abbildung 1.3). Der globale Speicher ist in der Praxis meist um ein vielfaches langsamer als der lokale oder der private Speicher, da dieser am weitesten vom eigentlichen Processing Element entfernt ist. Dafür ist dieser Speicher in der Regel größer als der lokale oder der private Speicher.

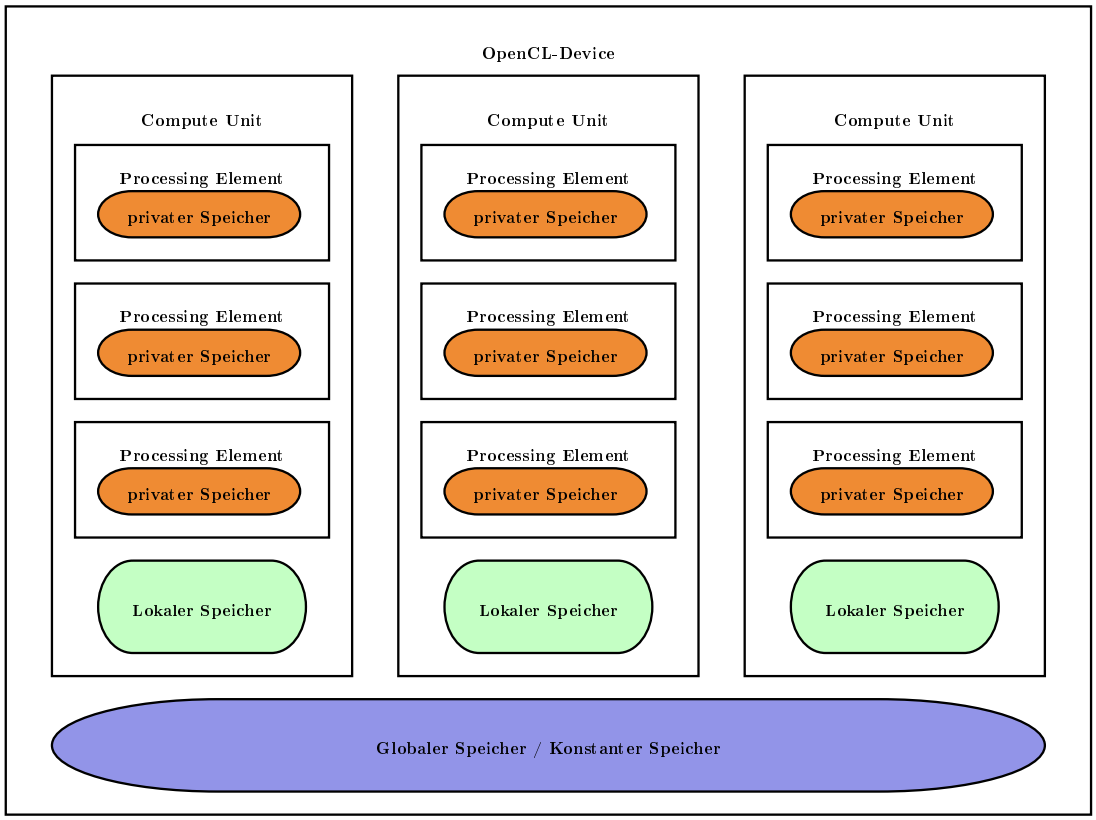

Abbildung 1.3.: OpenCL Speicher-Modell eines OpenCL-Devices.

Konstanter Speicher

Der konstante Speicher entspricht weitgehend dem globalen Speicher. Allerdings haben die Work-Items nur lesenden Zugriff auf den konstanten Speicher. Deshalb ist es nicht nötig, Daten vom konstanten Speicher zurück auf den Speicher des Hosts zu kopieren.

Lokaler Speicher

Auf dem lokalen Speicher können alle Work-Items einer Work-Group lesend und schreibend zugreifen. Allerdings ist es nicht möglich, Daten vom Host direkt in den lokalen Speicher zu kopieren. Hierzu müssen die Daten vom Host-Programm zunächst auf den globalen Speicher kopiert werden und anschließend von den Work-Items aus weiter in den lokalen Speicher. Im Bezug zum Plattform-Modell ist der lokale Speicher einmal für jede Compute Unit vorhanden (siehe Abbildung 1.3). Dieser Speicher ist in der Regel kleiner als der globale Speicher, aber auch schneller.

Privater Speicher

Der private Speicher steht jeweils einem Work-Item exklusiv zur Verfügung. Wie beim lokalen Speicher ist es nicht möglich, Daten direkt vom Host in den privaten Speicher zu kopieren. Im Bezug zum Plattform-Modell ist der lokale Speicher einmal für jedes Processing Element vorhanden (siehe Abbildung 1.3). In der Praxis ist dies der schnellste und kleinste Speicher.

1.1.4. OpenCL C

Als Programmiersprache für Kernel wird OpenCL C benutzt, welche auf dem C99 Standard aufbaut. Gegenüber diesem gibt es sowohl einige Einschränkungen, als auch OpenCL spezifische Erweiterungen in OpenCL C. Es ist z.B. nicht möglich, Rekursion zu nutzen oder Arrays variabler Länge anzulegen. Auch Standardfunktionen wie `printf` stehen nicht zur Verfügung. In OpenCL C werden aber eigene Standardfunktionen eingeführt, wie `barrier()`, um über Barrieren zu synchronisieren oder `get_global_id()`, um die Global ID auszulesen. Des Weiteren werden sogenannte Qualifier eingeführt, mit denen sich beispielsweise bestimmen lässt, welchen Speichertyp eine Variable haben soll.

1.1.5. Plattformunabhängigkeit

Das Host-Programm eines OpenCL-Programms lässt sich wie normale C oder C++ Programme kompilieren und ausführen. Da OpenCL-Devices auf sehr unterschiedlicher Hardware basieren können, muss der Kernel speziell für diese kompiliert werden. Damit ein kompiliertes OpenCL-Programm aber auf möglichst vielen Geräten lauffähig ist, wird der Kernel vom Host-Programm während der Laufzeit kompiliert. Dies bringt völlige Plattformunabhängigkeit gegenüber den OpenCL-Devices. Das Host-Programm unterliegt hingegen den üblichen Plattformabhängigkeiten von C bzw. C++ Programmen.

1.2. SkelCL

SkelCL[3] ist eine auf OpenCL aufbauende Bibliothek, die an der Westfälischen Wilhelms-Universität Münster entwickelt wird, um das Programmieren von parallelen Prozessoren zu erleichtern. Insbesondere werden Multi-GPU Systeme unterstützt, für die es mit herkömmlichen Ansätzen, wie OpenCL, keine direkte Unterstützung gibt. SkelCL benutzt algorithmische Skelette, die wiederkehrende Vorgehensmuster beim parallelen Programmieren beschreiben. Für Probleme, die auf diese Skelette übertragen werden, übernimmt die Bibliothek viele OpenCL spezifische Aufgaben automatisch. Außerdem stellt SkelCL Datentypen, wie z.B. Vektor oder Matrix, für die parallele Verarbeitung der Daten zur Verfügung.

1.2.1. Datentypen

Die SkelCL-Bibliothek führt Datentypen ein, welche sowohl für die Eingabe als auch für die Ausgabe der Skelette genutzt werden. Dazu gehören sowohl der Datentyp Vektor als auch Matrix. SkelCL erledigt das Kopieren der Daten zwischen Host und OpenCL-Device automatisch. Eine besondere Bedeutung fällt dem Index-Vektor und der Index-Matrix zu. Sie sind für die Eingabe vorgesehen und speichern keine Daten, sondern geben lediglich Indices aus, die Daten in einem normalen SkelCL-Vector oder einer SkelCL-Matrix hätten.

1.2.2. Algorithmische Skelette

Skelette beschreiben wie eine vom Anwendungs-Entwickler angegebene Funktion auf die Eingabedaten angewendet wird. Diese Skelett-Funktion muss je nach Skelett bestimmte Parameter übergeben bekommen, kann aber beliebig mit zusätzlichen Parametern erweitert werden. Intern wird sie durch eine Funktion eines OpenCL Kernels repräsentiert. Es stehen momentan vier Skelette in SkelCL zur Verfügung: Das Map-Skelett, das Zip-Skelett, das Reduce-Skelett und das Scan-Skelett.
Die folgenden Beschreibungen der Skelette und die verwendeten Notation orientieren sich an [3].

Map-Skelett

Das Map-Skelett wendet eine unäre Funktion f auf einzelne Elemente eines Vektors $[x_1, x_2, \ldots, x_n]$ an und gibt die Ergebnisse in einem Vektor aus:

$$map(f)([x_1, x_2, \ldots, x_n]) = [f(x_1), f(x_2), \ldots, f(x_n)]$$

Dieses Skelett lässt sich analog auch auf Matrizen anwenden.

Zip-Skelett

Das Zip-Skelett wendet einen assoziativen binären Operator $\oplus$ auf die einzelnen Elemente zweier Vektoren $[x_1, x_2, \ldots, x_n]$ und $[y_1, y_2, \ldots, y_n]$ an und gibt die Ergebnisse in einem Vektor aus:

$$zip(\oplus)([x_1, x_2, \ldots, x_n], [y_1, y_2, \ldots, y_n]) = [(x_1 \oplus y_1), (x_2 \oplus y_2), \ldots, (x_n \oplus y_n)]$$

Dieses Skelett lässt sich analog auch auf Matrizen anwenden.

Reduce-Skelett

Das Reduce-Skelett wendet einen assoziativen binären Operator $\oplus$ auf alle Elemente eines Vektors $[x_1, x_2, \ldots, x_n]$ an und gibt das skalare Ergebnis aus:

$$reduce(\oplus)([x_1, x_2, \ldots, x_n]) = x_1 \oplus x_2 \oplus \cdots \oplus x_n$$

Scan-Skelett

Das Scan-Skelett wendet einen assoziativen binären Operator $\oplus$ auf einen Vektor $[x_1, x_2, \ldots, x_n]$ an und gibt die Ergebnisse in einem Vektor aus, wobei an Stelle i das Ergebnis von $x_1 \oplus x_2 \oplus \cdots \oplus x_i$ steht:

$$scan(\oplus)([x_1, x_2, \ldots, x_n]) = [x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \oplus \cdots \oplus x_n]$$

2. K-Means-Algorithmus

Der K-Means-Algorithmus[5] gehört zu den Clusteranalysen, welche beim Data-Mining genutzt werden. Er wird benutzt, um große Mengen von Daten zu strukturieren. Die Art der Daten spielt dabei keine Rolle, solange eine Distanzfunktion auf sie anwendbar ist und ein Mittel gebildet werden kann.

2.1. Algorithmus

Ziel des Algorithmus ist es, große Mengen von Datensätzen (Objekte) in eine festgelegte Anzahl von k Gruppen (Clustern) zu ordnen. Der K-Means-Algorithmus benötigt als Eingabe eine Menge von Objekten und die Anzahl k der Cluster-Zentren, mit denen der Algorithmus starten soll. Die Objekte können eine beliebige Menge von Daten enthalten (z.B. für ein Objekt Person die Angaben über Alter, Größe und Gewicht), dessen Anzahl im Folgenden mit Dimension bezeichnet wird. Allerdings wird vorausgesetzt, dass es eine Distanzfunktion und eine Möglichkeit zum Berechnen des Mittels für die Objekte gibt. Als Ausgabe liefert der K-Means-Algorithmus eine Zuordnung jedes Objektes zu einem Cluster. Um dies zu erreichen werden zunächst die k Cluster-Zentren zufällig festgelegt. Danach wird über die Distanzfunktion jedes Objekt dem nähesten Cluster zugeordnet. Über das Mittel aller zugeordneten Objekte wird anschließend für jedes Cluster ein neues Zentrum bestimmt. Die Zuordnung der Objekte über die Distanzfunktion und das Neuberechnen der Cluster-Zentren wird so lange wiederholt, bis eine festgelegte Anzahl von Iterationen stattgefunden hat oder die Cluster-Zentren sich nicht mehr ändern, alle Objekte also immer wieder den gleichen Zentren zugeordnet werden.

2.1.1. Beispiel

Zur Verdeutlichung des Algorithmus betrachten wir als Beispiel Punkte im zweidimensionalen Raum als Objekte:
$a = (1;1)$, $b = (2;1)$, $c = (1;2)$, $d = (2;2)$, $e = (3;3)$, $f = (4;3)$, $g = (3;4)$ und $h = (4;4)$
Es sollen zwei Cluster X und Y gebildet werden. Die Cluster-Zentren werden am Anfang auf $x = (3;2)$ für das Cluster X und $y = (4;2)$ für das Cluster Y festgelegt. Diese Ausgangssituation wird in Abbildung 2.1 dargestellt. Der Punkt x wird durch ein ausgefülltes Dreieck dargestellt und in den folgenden Schritten werden die zu X zugeordneten Punkte mit einem leeren Dreieck gekennzeichnet. Der Punkt y und die zu Y gehörenden Punkte werden analog mit einem Viereck dargestellt.

Zum Berechnen der Distanz wird der euklidische Abstand (Formel 2.1) benutzt und zum Bestimmen des Mittels das arithmetisches Mittel (Formel 2.2) verwendet.

$$d(p_1, p_2) = \sqrt{(u_1 - u_2)^2 + (v_1 - v_2)^2} \tag{2.1}$$
mit $p_1 = (u_1; v_1)$ und $p_2 = (u_2; v_2)$

$$m(p_1, \ldots, p_n) = (\frac{1}{n} \sum_{i=1}^{n} u_i; \frac{1}{n} \sum_{i=1}^{n} v_i) \tag{2.2}$$
wobei $p_1, \ldots, p_n$ Punkte mit $p_i = (u_i; v_i)$ für $1 \leq i \leq n$

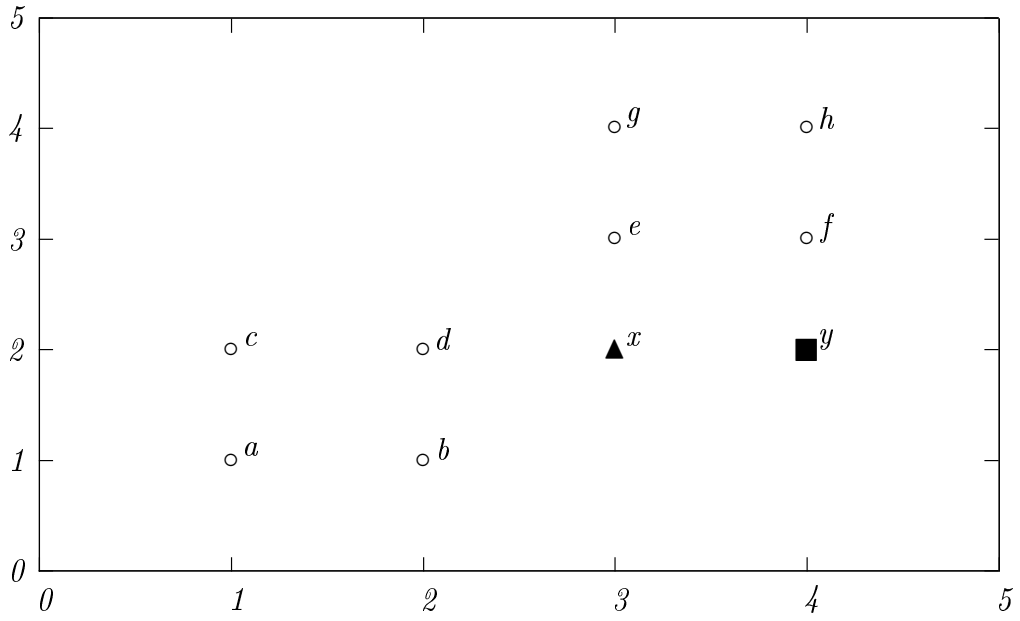

Abbildung 2.1.: Anfangsverteilung

Zunächst müssen alle Abstände zu den Cluster-Zentren bestimmt werden:

Punkt	Abstand zu x	Abstand zu y
a	$\sqrt{5}$	$\sqrt{10}$
b	$\sqrt{2}$	$\sqrt{5}$
c	$\sqrt{4}$	$\sqrt{9}$
d	$\sqrt{1}$	$\sqrt{4}$
e	$\sqrt{1}$	$\sqrt{2}$
f	$\sqrt{2}$	$\sqrt{1}$
g	$\sqrt{4}$	$\sqrt{5}$
h	$\sqrt{5}$	$\sqrt{2}$

Mithilfe dieser Abstände können nun a, b, c, d, e und g dem Cluster X zugeordnet werden. f und h gehören dem nach zu Cluster Y. Mit dem arithmetischen Mittel über die zugeordneten Punkte können nun neue Cluster-Zentren errechnet werden. Diese sind $x = (2; 2\frac{1}{6})$ und $y = (4; 3\frac{1}{2})$. In Abbildung 2.2 ist die Situation nach der ersten Iteration dargestellt.

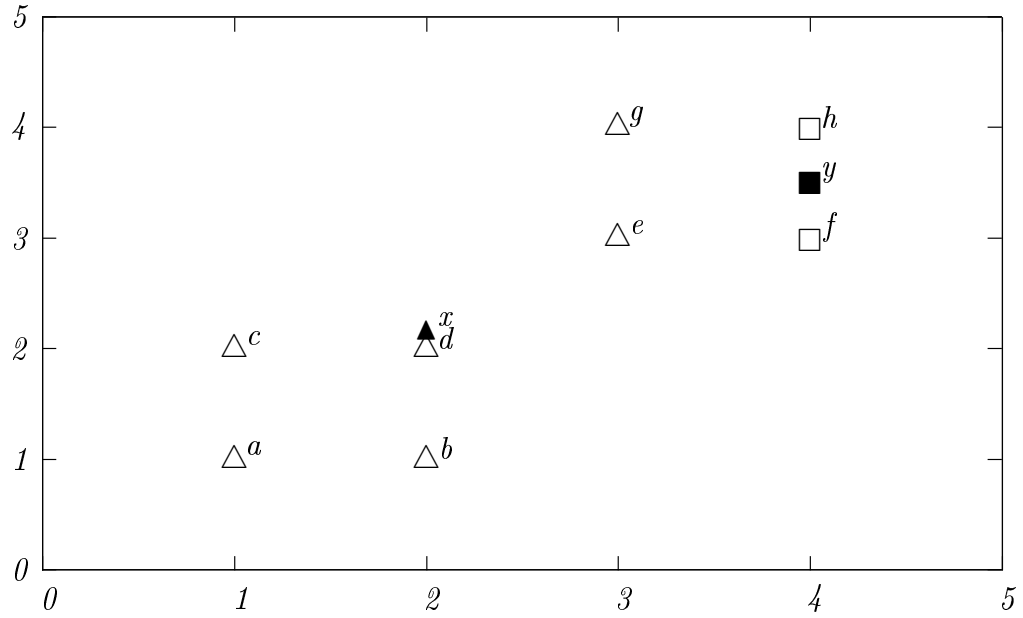

Abbildung 2.2.: Verteilung nach der ersten Iteration

Anschließend werden die Schritte wiederholt. Für die Abstände zu den neu errechneten Cluster-Zentren ergibt sich:

Punkt	Abstand zu x	Abstand zu y
a	$\sqrt{2\frac{13}{36}}$	$\sqrt{15\frac{9}{36}}$
b	$\sqrt{1\frac{13}{36}}$	$\sqrt{10\frac{9}{36}}$
c	$\sqrt{1\frac{1}{36}}$	$\sqrt{11\frac{9}{36}}$
d	$\sqrt{\frac{1}{36}}$	$\sqrt{6\frac{9}{36}}$
e	$\sqrt{1\frac{25}{36}}$	$\sqrt{1\frac{9}{36}}$
f	$\sqrt{4\frac{25}{36}}$	$\sqrt{\frac{9}{36}}$
g	$\sqrt{4\frac{13}{36}}$	$\sqrt{1\frac{9}{36}}$
h	$\sqrt{7\frac{13}{36}}$	$\sqrt{\frac{9}{36}}$

Nun können also a, b, c und d dem Cluster X zugeordnet werden. e, f, g und h gehören also zu Cluster Y. Dadurch ergeben sich $x = (1\frac{1}{2}; 1\frac{1}{2})$ und $y = (3\frac{1}{2}; 3\frac{1}{2})$ als neue Cluster-Zentren. Dieser Zustand ist in Abbildung 2.3 abgebildet.

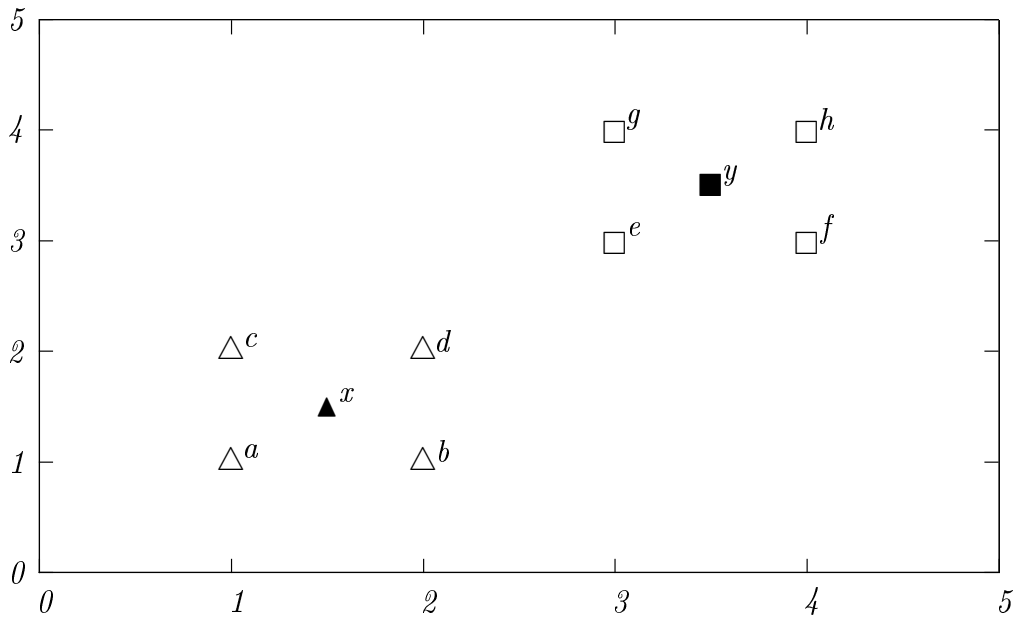

Abbildung 2.3.: Verteilung nach der zweiten und dritten Iteration

Das erneute Berechnen der Distanzen ergibt Folgendes:

Punkt	Abstand zu x	Abstand zu y
a	$\sqrt{\frac{1}{2}}$	$\sqrt{12\frac{1}{2}}$
b	$\sqrt{\frac{1}{2}}$	$\sqrt{8\frac{1}{2}}$
c	$\sqrt{\frac{1}{2}}$	$\sqrt{8\frac{1}{2}}$
d	$\sqrt{\frac{1}{2}}$	$\sqrt{4\frac{1}{2}}$
e	$\sqrt{4\frac{1}{2}}$	$\sqrt{\frac{1}{2}}$
f	$\sqrt{8\frac{1}{2}}$	$\sqrt{\frac{1}{2}}$
g	$\sqrt{8\frac{1}{2}}$	$\sqrt{\frac{1}{2}}$
h	$\sqrt{12\frac{1}{2}}$	$\sqrt{\frac{1}{2}}$

Die Zugehörigkeit der Punkte zu den Cluster-Zentren ändert sich nicht. Der Algorithmus bricht hier also ab und Abbildung 2.3 zeigt das Ergebnis.

2.2. Implementierung

Als Grundlage der angefertigten Implementierung dient der Quelltext des OpenCL-Programms zum K-Means-Algorithmus aus der Rodinia Benchmark Suite[4] in der Version 2.3. Dieses Programm geht wie oben beschrieben vor. Das Zuordnen der Objekte zu den Clustern wird dabei parallel ausgeführt. Für jedes Objekt der Menge wird ein Work-Item gestartet, welches die Daten eines Objektes mit denen aller Cluster-Zentren mit Hilfe des euklidischen Abstandes vergleicht und das nächstgelegene Cluster ausgibt. Es wird dabei keine Work-Group Größe festgelegt und auch kein lokaler Speicher verwendet. Das Neuberechnen der Cluster-Zentren geschieht über das arithmetische Mittel im Host-Programm.

Die Implementierung mit SkelCL ahmt das Verhalten des Programms aus der Rodinia Benchmark Suite nach. Das Programm ist so gestaltet, dass sowohl die Berechnungen mit OpenCL als auch mit SkelCL getätigt werden. Die Ergebnisse der Berechnungen werden miteinander verglichen, um die Korrektheit der Implementierung mit SkelCL sicherzustellen.

Host-Programm

Die Implementierung mit OpenCL kann sowohl Daten aus Textdateien als auch in binärer Form einlesen. Da die Beispieldaten, welche später zur Messung der Laufzeit benutzt wurden in Textdateien vorliegen, wurde die Eingabe bei der Implementierung mit SkelCL auf diese beschränkt. In den Textdateien muss für jedes Objekt eine Zeile mit den Daten existieren. Das Einlesen der Werte geschieht folgendermaßen:

```
skelcl::Matrix<float> *feature_matrix =
  new skelcl::Matrix<float>({npoints, nfeatures});
int k = 0;
while (fgets(line, 1024, infile) != NULL) {
  for (j=0; j<nfeatures; j++) {
    float value = atof(strtok(NULL, " ,\t\n"));
    (*feature_matrix)[k][j] = value;
  }
  k++;
 }
```

Zunächst wird eine SkelCL-Matrix `feature_matrix` von `float` Werten mit der Größe `npoints`×`nfeatures` erstellt. Die Werte `npoints` und `nfeatures` wurden bereits vorher ermittelt. `npoints` entspricht dabei der Anzahl der in der Textdatei enthaltenen Objekte und `nfeatures` der Dimension, also der Anzahl von Daten, die ein Objekt enthält. Anschließend wird in einer `while`-Schleife mit `fgets` zeilenweise die Eingabedatei `infile` ausgelesen. In der `while`-Schleife werden dann mit einer `for`-Schleife die einzelnen Daten aus den ausgelesenen Zeilen mittels `atof` in `float`-Werte umgewandelt und in der SkelCL-Matrix `feature_matrix` abgelegt. Jede Zeile der `feature_matrix` repräsentiert also die Daten eines eingelesenen Objektes.
Als nächstes werden die Cluster-Zentren initialisiert:

```
skelcl::Matrix<float> *cluster_matrix =
    new skelcl::Matrix<float>({nclusters, nfeatures});
for (i=0; i<nclusters; i++) {
  for (j=0; j<nfeatures; j++) {
    (*cluster_matrix)[i][j] = feature[i][j];
  }
}
```

Mit `cluster_matrix` wird eine `nclusters`×`nfeatures` SkelCL-Matrix erstellt, in der die Cluster-Zentren abgespeichert werden. `nclusters` beschreibt dabei die Anzahl k der zu bildenden Cluster. Die Cluster-Zentren werden dabei mit den ersten Objekten aus der `feature_matrix` initialisiert. Analog zur `feature_matrix`, repräsentiert jede Zeile der `cluster_matrix` die Daten eines Cluster-Zentrums.

Nun werden die Zugehörigkeiten der Objekte zu den Clustern bestimmt:

```
skelcl::Map<int(skelcl::Index)>
    clusterix(std::ifstream("kmeans.skelcl"));
skelcl::IndexVector index(npoints);
skelcl::Vector<int> membership_vector_OCL = clusterix(index,
    (*feature_matrix), nclusters, nfeatures, (*cluster_matrix));
```

Um parallel die Zugehörigkeit der Objekte zu den Clustern zu bestimmen wird ein Map-Skelett verwendet. Dies läuft über einen Index-Vektor und gibt einen SkelCL-Vektor mit `int`-Werten aus. Die zugehörige Skelett-Funktion wird aus der Datei kmeans.skelcl ausgelesen. Um für jedes Objekt ein Work-Item laufen zu lassen wird der Index-Vektor mit der Größe `npoints` erstellt. In den SkelCL-Vektor `membership_vector_OCL` von `int`-Werten werden die Zugehörigkeiten der Objekte zu den Clustern über einen Index abgespeichert. Dies geschieht durch Aufrufen des Skelettes, welches `feature_matrix`, `nclusters`, `nfeatures` und `cluster_matrix` als zusätzliche Argumente benötigt. Anschließend werden die neuen Cluster-Zentren wie folgt berechnet:

```
for (int i = 0; i < membership_vector_OCL.size(); i++){
  int cluster_id = membership_vector_OCL[i];
  new_centers_len[cluster_id]++;
  if (cluster_id != membership[i])
  {
    delta++;
    membership[i] = membership_vector_OCL[i];
  }
  for (int j = 0; j < n_features; j++)
  {
    new_centers[cluster_id][j] += feature[i][j];
  }
}
```

Dafür wird über die Ausgabe des Skelettes iteriert und zunächst der Index des nähesten Clusters aus dem Ergebnis-Vektor `membership_vector_OCL` ausgelesen. Im `int`-Array `new_centers_len` wird abgespeichert, wie viele Objekte den einzelnen Clustern zugeordnet sind. Danach wird überprüft, ob die Zuordnung des Objektes zu dem entsprechenden Cluster bereits vorher bestand. Dies geschieht über ein `int`-Array `membership`, in dem für jedes Objekt der Index des vorher zugeordneten Clusters steht (am Anfang mit `-1` initialisiert). Wurde das Objekt neu zugeordnet, wird der Zähler `delta` erhöht, um die Häufigkeit der Neuzuordnungen zu zählen. Außerdem wird der Index des neu zugeordneten Clusters in `membership` abgespeichert. `new_centers` ist ein zweidimensionales `float`-Array, in der Größe von `cluster_matrix` an. In diesem Array werden, in der anschließenden `for`-Schleife, die Daten aus dem jeweiligen Objekt an Stelle des zugeordneten Clusters addiert.

Aus diesen Daten und der Anzahl der zugeordneten Objekte aus dem `new_centers_len`-Array lässt sich anschließend das arithmetische Mittel für die neuen Cluster-Zentren berechnen:

```
for (i=0; i<nclusters; i++) {
  for (j=0; j<nfeatures; j++) {
    if (new_centers_len[i] > 0){
      cluster_matrix[i][j] =
        new_centers[i][j] / new_centers_len[i];
    }
  }
}
```

Dazu wird lediglich über alle Einträge iteriert und die addierten Daten durch die Anzahl der zugeordneten Objekte geteilt.

Die Ausführung des Skelettes und das Neuberechnen der Cluster-Zentren wird in einer Schleife so oft wiederholt, bis entweder 500 Schleifendurchläufe vergangen sind oder der errechnete Wert `delta` geringer ist als ein Referenzwert. Dieser kann per Kommandozeilenparameter `-t` eingestellt werden.

Skelett-Funktion

In der Skelett-Funktion (Quelltextausschnitt 2.1) wird der euklidische Abstand eines durch das Work-Item bearbeiteten Objektes zu allen Cluster-Zentren berechnet und der Index des nähesten Clusters ausgegeben. Dazu werden zwei ineinander verschachtelte Schleifen benutzt. Die äußere Schleife iteriert über alle Indices der Cluster. Mit Hilfe der inneren Schleife wird der euklidische Abstand bestimmt. Die Wurzel wird dabei allerdings nicht berechnet. Dies hat aber keinen Einfluss auf das Ergebnis, da die Wurzelfunktion streng monoton ist und lediglich das nächste Cluster ermittelt werden soll. Auf das Berechnen des euklidischen Abstands in der inneren Schleife folgt eine `if`-Abfrage in der ermittelt wird, ob der errechnete Abstand der geringste ist. Ist dies der Fall wird der Index des Clusters abgespeichert. Der Index des nähesten Clusters wird am Ende zurückgegeben.

Quelltextausschnitt 2.1: Skelett-Funktion zum Bestimmen des nähesten Clusters

```
1  int func(Index index, float_matrix_t feature_matrix,
2      int nclusters, int nfeatures, float_matrix_t cluster_matrix){
3    float min_dist=3.40282347e+38;
4    int ind = 0;
5    float dist;
6    float ans;
7    float hlp;
8
9    for (int i=0; i < nclusters; i++){
10     ans  = 0;
11
12     for (int l=0; l<nfeatures; l++){
13       hlp = get(feature_matrix, index, l) -
14               get(cluster_matrix, i, l);
15       ans += hlp * hlp;
16     }
17     dist = ans;
18     if (dist < min_dist) {
19       min_dist = dist;
20       ind    = i;
21     }
22   }
23   return ind;
24 }
```

2.3. Vergleich der OpenCL- mit der SkelCL-Implementierung

Um einen Vergleich zwischen der OpenCL- mit der SkelCL-Implementierung zu ziehen, wird auf der einen Seite die Laufzeit der Programme betrachtet und auf der anderen Seite die Menge und Komplexität an OpenCL bzw. SkelCL spezifischen Quelltext. Die Laufzeit kann anzeigen, ob und wie viel Zusatzarbeit ein Rechner durch die SkelCL-Bibliothek leisten muss. Der Quelltext lässt hingegen Rückschlüsse darauf ziehen, inwiefern SkelCL das Programmieren von parallelen Programmen vereinfacht.

2.3.1. Laufzeit

Für die Zeitmessungen wurden verschiedene vergleichbare Programmabschnitte mit Zeitmarken versehen. So ist eine Messung der folgenden Programmteile möglich:

- Initialisierung von OpenCL bzw. SkelCL
- Erstellen des Skelettes, bzw. Kompilieren des Kernel
- Ausführen des Skelettes, bzw. Ausführen des Kernel mit Datentransfer und Neuberechnung der Cluster-Zentren (beim Kernel ohne den Datentransfer der Objekte)

Die Messergebnissen (Tabelle 2.1) zeigen, dass die SkelCL- im Vergleich zur OpenCL-Implementierung eine längere Gesamtlaufzeit hat. Die längeren Laufzeiten beim Initialisieren von SkelCL und Erstellen des Skelettes fallen dabei kaum ins Gewicht. Vor allem das wiederholte Ausführen des Skelettes bzw. des Kernels auf großen Datenmengen machen hier die Unterschiede aus. Die SkelCL-Implementierung benötigt beim Ausführen des Skelettes durchschnittlich ca. 4,41% mehr Zeit als die OpenCL-Implementierung beim Ausführen des Kernels.

2.3.2. Quelltext

Um die Unterschiede im Quelltext zu verdeutlichen, werden zunächst die Anzahl der SkelCL zu den OpenCL spezifischen Quelltextzeilen verglichen. Zunächst betrachten wir die Initialisierungen. SkelCL wird mit einer einzigen Zeile initialisiert:

```
skelcl::init(skelcl::nDevices(1));
```

Dafür wird angegeben, wie viele Devices die Berechnungen ausführen sollen. Bei der OpenCL-Implementierung wurde für die Initialisierung eine eigene Funktion mit folgendem Inhalt geschrieben:

6 Zeilen - OpenCL Kontext erzeugen
7 Zeilen - GPU auswählen
2 Zeilen - Command Queue erzeugen

Prog.	Initial.	Skelett/Kernel vorb.	Skelett/Kernel ausf.	Summe
s_1	22 ms	10 ms	29907 ms	29939 ms
s_2	22 ms	9 ms	29869 ms	29900 ms
s_3	21 ms	9 ms	29991 ms	30021 ms
s_4	20 ms	9 ms	29940 ms	29969 ms
s_5	20 ms	10 ms	29945 ms	29975 ms
s_6	22 ms	9 ms	29971 ms	30002 ms
s_7	22 ms	9 ms	29886 ms	29917 ms
s_8	20 ms	9 ms	29852 ms	29881 ms
s_9	22 ms	9 ms	29886 ms	29917 ms
s_{10}	22 ms	9 ms	30092 ms	30123 ms
∅	21,3 ms	9,2 ms	29933,9 ms	29964,4 ms
o_1	18 ms	0 ms	28681 ms	28699 ms
o_2	18 ms	0 ms	28672 ms	28690 ms
o_3	18 ms	0 ms	28676 ms	28694 ms
o_4	18 ms	0 ms	28675 ms	28693 ms
o_5	18 ms	0 ms	28685 ms	28703 ms
o_6	18 ms	0 ms	28675 ms	28693 ms
o_7	18 ms	0 ms	28669 ms	28687 ms
o_8	18 ms	0 ms	28664 ms	28682 ms
o_9	18 ms	0 ms	28625 ms	28643 ms
o_{10}	18 ms	0 ms	28662 ms	28680 ms
∅	18 ms	0,0 ms	28668,4 ms	28686,4 ms

Tabelle 2.1.: Zehn Messungen (mit Kommandozeilenparameter `-i kdd_cup -v -` SkelCL Messungen sind mit s_i und OpenCL Messungen mit o_i gekennzeichnet) auf einem Rechner mit einem Intel Core 2 Duo CPU E8400 und einer Nvidia GeForce 9500 GT. `kdd_cup` ist ein Datensatz mit 494.020 Objekten der Dimension 35.

Das Erstellen des Skelettes geschieht bei SkelCL mit einer Zeile. Bei OpenCL muss kein Skelett erstellt werden, allerdings muss auch der Kernel eingeladen und kompiliert werden. Dafür werden 19 Zeilen Quelltext benötigt. Zum eigentlichen Berechnen in SkelCL wird nur noch ein Index-Vektor, der die Anzahl der ausführenden Work-Items bestimmt, erstellt und dieser zusammen mit anderen Parametern dem Skelett übergeben:

```
skelcl::IndexVector index((*feature_matrix).size().rowCount());
skelcl::Vector<int> membership_vector_OCL = clusterix(index,
   feature_matrix, n_clusters, n_features, cluster_matrix);
```

Die Parameter werden dabei automatisch auf das OpenCL-Device übertragen und die Ergebnisse in den SkelCL-Vektor `membership_vector_OCL` gespeichert. Um den Kernel in OpenCL auszuführen, müssen zunächst die Daten auf das OpenCL-Device und später die Ergebnisse wieder herunter geladen werden. Außerdem muss die Anzahl der Work-Items gesetzt werden. Für diese Schritte benötigt die OpenCL-Implementierung 32 Quelltextzeilen. Zusammengenommen benötigt die SkelCL-Implementierung nur vier Zeilen SkelCL spezifischen Quelltext. Dagegen umfasst die OpenCL-Implementierung mit 66 OpenCL spezifischen Quelltextzeilen über 16 mal so viel Quelltext.

Komplexität

Je nach Erfahrungsschatz kann die Einschätzung zur Komplexität eines Programms bei Anwendungs-Entwicklern sehr voneinander abweichen. Im Folgendem wird, trotz dieses subjektiven Empfindens, versucht die Komplexität der SkelCL- im Gegensatz zur OpenCL-Implementierung aufzuzeigen. Die weit größere Anzahl von OpenCL spezifischen Code macht bereits deutlich, dass dieser schwerer zu verstehen ist. Um die hohe Komplexität des OpenCL Quelltextes zu demonstrieren soll hier exemplarisch das Einladen und Kompilieren des Kernels mit dem Erstellen des Skelettes verglichen werden. Mit folgender Quelltextzeile wird mit SkelCL die Skelett-Funktion aus `kmeans.skelcl` eingelesen und ein Map-Skelett erstellt, welches mit einem Index-Vektor als Eingabe arbeitet und einen Vektor mit `int`-Werten ausgibt:

```
skelcl::Map<int(skelcl::Index)>
    clusterix(std::ifstream("kmeans.skelcl"));
```

Um mit OpenCL den Kernel Quelltext einzulesen wird zunächst Speicherplatz reserviert:

```
int sourcesize = 1024*1024;
char * source = (char *)calloc(sourcesize, sizeof(char));
if(!source) { printf("ERROR: calloc(%d) failed\n", sourcesize);}
```

Anschließend wird die Datei eingelesen:

```
char * tempchar = "./kmeans.cl";
FILE * fp = fopen(tempchar, "rb");
if(!fp) { printf("ERROR: unable to open '%s'\n", tempchar);}
fread(source + strlen(source), sourcesize, 1, fp);
fclose(fp);
```

Um den Kernel dann zu kompilieren, müssen noch folgende Befehle ausgeführt werden:

```
cl_int err = 0;
const char * slist[2] = {source, 0};
cl_program prog = clCreateProgramWithSource(context, 1, slist,
   NULL, &err);
if(err != CL_SUCCESS){
    printf("ERROR: clCreateProgramWithSource() => %d\n", err);}
err = clBuildProgram(prog, 0, NULL, NULL, NULL, NULL);
if(err != CL_SUCCESS){
    printf("ERROR: clBuildProgram() => %d\n", err);}
char * kernel_kmeans_c  = "kmeans_kernel_c";
kernel_s = clCreateKernel(prog, kernel_kmeans_c, &err);
if(err != CL_SUCCESS){
    printf("ERROR: clCreateKernel() 0 => %d\n", err);}
clReleaseProgram(prog);
```

Abgesehen von der Menge des Quelltextes fällt auf, dass sehr viele Fehlerbehandlungen vorkommen. Bei SkelCL-Implementierungen wird diese automatisch ausgeführt. Außerdem müssen bei einigen Befehlen viele Parameter angegeben werden. Dieses Beispiel zeigt, wie SkelCL das Programmieren vereinfacht, indem viele Schritte aus OpenCL stark vereinfacht in einem SkelCL Befehl zusammengefasst werden.

2.3.3. Fazit

Die Implementierung mit SkelCL ist zwar mit einer ca. 4,41%[1] längeren Laufzeit etwas langsamer als die OpenCL-Implementierung, allerdings nur geringfügig. Der Quelltext wurde dagegen mit SkelCL stark vereinfacht, indem die OpenCL spezifischen Befehle mit denen von SkelCL ersetzt wurden, die dabei nur $\frac{1}{16}$ der Quelltextzeilen benötigen. Falls die geringfügig längere Laufzeit hingenommen werden kann, bietet SkelCL beim K-Means-Algorithmus damit eine simple Alternative zu OpenCL, die trotzdem die Geschwindigkeit von Grafikprozessoren ausnutzt.

[1] Bezogen auf die Ausführung des Kernels bzw. des Skelettes, wie in 2.3.1.

3. Needleman-Wunsch-Algorithmus

Der Needleman-Wunsch-Algorithmus[6] ist ein Algorithmus aus der Bio-Informatik und dient dazu, Ähnlichkeiten zwischen verwandten Aminosäuresequenzen zu bestimmen, die im Laufe der Evolution durch Mutationen verändert wurden. Dies hilft dabei die Verwandtschaftsverhältnisse von verschiedenen Arten zu bestimmen.

3.1. Algorithmus

Der Algorithmus erstellt ein sogenanntes globales Sequenzalignment von zwei Aminosäuresequenzen, in dem jede Aminosäure einer Sequenz einer Lücke oder einer Aminosäure der anderen Sequenz zugeordnet wird. Ein solches Sequenzalignment gibt eine Möglichkeit an, wie es im Laufe der Evolution zu Unterschieden in den Sequenzen kam.

3.1.1. Substitutionsmatrizen

Für den Algorithmus werden Substitutionsmatrizen genutzt, welche angeben wie wahrscheinlich ein Austausch einer Aminosäure durch eine andere ist bzw. wie wahrscheinlich es ist, dass eine Aminosäure in einem Genom erhalten bleibt. Hohe positive Werte in der Matrix bedeuten, dass ein Austausch bzw. das Erhalten der Aminosäure wahrscheinlich ist, niedrige negative Werte, dass es unwahrscheinlich ist. Ein Beispiel für eine häufig verwendete Substitutionsmatrix, ist die BLOSUM62 (siehe Anhang A), welche auf Grundlage von Aminosäuresequenzen erstellt wurden, die zu maximal 62% identisch waren.

3.1.2. Funktionsweise

Um zwei Sequenzen s und t mit dem Needleman-Wunsch-Algorithmus zu vergleichen, werden die beiden Sequenzen in einer Matrix Z gegeneinander aufgetragen (s über die x-Achse, t über die y-Achse), so dass Z die Dimension $n \times m$ mit $n = |s|$ und $m = |t|$ hat. Anschließend werden die Zellen der Matrix Z mit den Werten einer Substitutionsmatrix SUB gefüllt:

$$Z_{i,j} = SUB_{s_i,t_j} \; \forall \; 0 < i \leq n, 0 < j \leq m$$

Ziel ist es, einen Weg durch die Matrix von oben links ($Z_{1,1}$) nach unten rechts ($Z_{n,m}$) zu finden, der einen maximalen Wert ergibt, also evolutionär am wahrscheinlichsten ist.

Dazu gibt es zwei Möglichkeiten den Wert einer Zelle zu aktualisieren:

- diagonal nach unten rechts $\searrow$ ($Z_{i-1,j-1} \rightarrow Z_{i,j}$ für $0 < i \leq n$ und $0 < j \leq m$)
  - Bedeutet, die Aminosäuren stimmen überein oder wurden im Laufe der Evolution ausgetauscht.
  - Der bisherige Wert wird mit dem Wert der Substitutionsmatrix addiert ($Z_{i,j} := Z_{i-1,j-1} + Z_{i,j}$).
- nach unten $\downarrow$ oder nach rechts $\rightarrow$ ($Z_{i,j-1} \rightarrow Z_{i,j}$ oder $Z_{i-1,j} \rightarrow Z_{i,j}$ für $0 < i \leq n$ und $0 < j \leq m$)
  - Eine Lücke wird in einer der Sequenzen eingefügt.
  - Bedeutet eine Aminosäure wurde im Laufe der Evolution entfernt oder hinzugefügt.
  - Ein negativer konstanter Strafwert p beschreibt wie wahrscheinlich ein solches Ereignis ist.
  - p wird zum bisherigen Wert addiert ($Z_{i,j} := Z_{i,j-1} + p$ oder $Z_{i,j} := Z_{i-1,j} + p$).

Eine Zelle $Z_{i,j}$ ist also von ihren Nachbarzellen $Z_{i,j-1}$, $Z_{i-1,j-1}$ und $Z_{i-1,j}$ abhängig. Das heißt, diese Zellen müssen aktualisiert worden sein, bevor die Zelle $Z_{i,j}$ aktualisiert wird. Außerdem ergibt sich das Problem, das für die erste Zeile $Z_{\bullet,1}$ und die erste Spalte $Z_{1,\bullet}$ mindestens zwei der erforderlichen Werte nicht gegeben sind.
Es bietet sich allerdings an, für die Zelle $Z_{0,0}$ den Wert 0 anzunehmen, da noch keine Vergleiche vorgenommen wurden. Außerdem wird für jeden Schritt nach unten oder rechts der Strafwert p berechnet, so dass man $Z_{i,0} = i \cdot p$ und $Z_{0,j} = j \cdot p$ annehmen kann.
Der neue Zellenwert einer Zelle $Z_{i,j}$ ergibt sich dann aus dem Maximum von $Z_{i-1,j-1} + Z_{i,j}$, $Z_{i,j-1} + p$ und $Z_{i-1,j} + p$. Werden die Zellen, aus denen die Maxima stammen, mit abgespeichert, kann anschließend das Alignment rückwärts aus der Tabelle abgelesen werden, in dem von Zelle $Z_{n,m}$ aus den abgespeicherten Zellen bis zur Zelle $Z_{1,1}$ gefolgt wird. Es ist allerdings auch möglich, den Weg zurück zu berechnen. Dafür wird bei jeder Zelle rechnerisch bestimmt, aus welcher Zelle das abgespeicherte Maximum errechnet wurde.

3.1.3. Beispiel

Zur Verdeutlichung des Algorithmus betrachten wir als Beispiel die Sequenzen $s = (Ser, Tyr, Glu, Asp, Val, Val)$ und $t = (Asn, Tyr, Cys, Asp, Val)$. Als Strafwert nehmen wir $p = -3$ und BLOSUM62 als Substitutionsmatrix. Es ergibt sich die folgende initiale Matrix:

	$j \cdot p$	Ser	Tyr	Glu	Asp	Val	Val
$i \cdot p$	0	-3	-6	-9	-12	-15	-18
Asn	-3	1	-4	0	1	-3	-3
Tyr	-6	-2	7	-2	-3	-1	-1
Cys	-9	-1	-2	-4	-3	-1	-1
Asp	-12	0	-3	2	6	-3	-3
Val	-15	-2	-1	-2	-3	4	4

Es ist zunächst nur möglich den neuen Wert von Zelle $Z_{1,1}$ zu berechnen:

$$Z_{1,1} = \max\{0 + 1, (-3) + (-3), (-3) + (-3)\} = \max\{1, -6, -6\} = 1$$

An der Matrix verändert sich also in diesem Schritt nichts. Nun lassen sich $Z_{1,2}$ und $Z_{2,1}$ berechnen:

$$Z_{1,2} = \max\{(-3) + (-2), 1 + (-3), (-6) + (-3)\} = \max\{-5, -2, -9\} = -2$$

$$Z_{2,1} = \max\{(-3) + (-4), (-6) + (-3), 1 + (-3)\} = \max\{-7, -9, -2\} = -2$$

Anschließend können die Zellen $Z_{1,3}$, $Z_{2,2}$ und $Z_{3,1}$ berechnet werden und so nach und nach jede weitere Diagonale (siehe Abbildung 3.1). Die vollständig aktualisierte Tabelle sieht folgendermaßen aus:

	$j \cdot p$		Ser		Tyr		Glu		Asp		Val		Val
$i \cdot p$	0	$\leftarrow$	-3	$\leftarrow$	-6	$\leftarrow$	-9	$\leftarrow$	-12	$\leftarrow$	-15	$\leftarrow$	-18
	$\uparrow$	$\nwarrow$						$\nwarrow$					
Asn	-3		1	$\leftarrow$	-2	$\leftarrow$	-5	$\leftarrow$	-8	$\leftarrow$	-11	$\leftarrow$	-14
	$\uparrow$		$\uparrow$	$\nwarrow$									
Tyr	-6		-2		8	$\leftarrow$	5	$\leftarrow$	2	$\leftarrow$	-1	$\leftarrow$	-4
	$\uparrow$		$\uparrow$		$\uparrow$	$\nwarrow$		$\nwarrow$		$\nwarrow$		$\nwarrow$	
Cys	-9		-5		5		4		2		1		-2
	$\uparrow$		$\uparrow$		$\uparrow$	$\nwarrow$		$\nwarrow$					
Asp	-12		-8		2		7		10	$\leftarrow$	7	$\leftarrow$	4
	$\uparrow$		$\uparrow$		$\uparrow$		$\uparrow$		$\uparrow$	$\nwarrow$		$\nwarrow$	
Val	-15		-11		-1		4		7		14	$\leftarrow$	11

Die Pfeile $\leftarrow$, $\nwarrow$ und $\uparrow$ zeigen an, aus welcher Zelle der Wert stammt, aus dem das Maximum berechnet wurde.

Ausgehend von der Zelle unten rechts, kann anhand der Pfeile das Sequenzalignment rückwärts abgelesen werden. Dies geschieht folgendermaßen:

- $\leftarrow$ in der Zelle $Z_{i,j}$ bedeutet, eine Lücke wird in die Sequenz t eingefügt: Der Wert s_i wird einer Lücke _ zugeordnet.
- $\uparrow$ in der Zelle $Z_{i,j}$ bedeutet, eine Lücke wird in die Sequenz s eingefügt: Eine Lücke _ wird dem Wert t_j zugeordnet.
- $\nwarrow$ in der Zelle $Z_{i,j}$ bedeutet, der Wert s_i wird dem Wert t_j zugeordnet.

Gibt es mehrere Pfeile in einer Zelle, sind nach der vorgenommenen Rechnung, alle durch die Pfeile angezeigten Zuordnungen gleich wahrscheinlich. Wird z.B. dem Pfeil $\leftarrow$ in der Zelle unten rechts gefolgt, so ergibt sich folgendes Sequenzalignment:

```
Ser Try Glu Asp Val Val
Asn Try Cys Asp Val  _
```

Wird hingegen dem Pfeil $\nwarrow$ gefolgt, so ergibt sich ein anderes Sequenzalignment:

```
Ser Try Glu Asp Val Val
Asn Try Cys Asp  _  Val
```

3.2. Implementierung

Als Grundlage der angefertigten Implementierung dient die OpenCL-Implementierung des Needleman-Wunsch-Algorithmus aus der Rodinia Benchmark Suite[4] in der Version 2.3. Diese Implementierung teilt die zu berechnende Matrix in 16×16-Blöcke ein, welche von Work-Groups mit 16 Work-Items neu berechnet werden. Es wird zunächst der obere linke Block berechnet und anschließend immer die Blöcke, welche direkt an den zuletzt berechneten Blöcken angrenzen. Es werden also nacheinander die Diagonalen neu berechnet. Zur Verdeutlichung ist dies in Abbildung 3.1 mit Hilfe eines kleinen Beispieles dargestellt. Die Diagonalen werden in jedem Schritt um eine Einheit größer, bis die Hauptdiagonale in der Mitte erreicht ist (im Beispiel die Diagonale mit Länge 3). Dies ist die längste Diagonale und alle Nachfolgenden werden ab diesem Schritt jeweils eine Einheit kleiner als die vorherige Diagonale. Genauso wird innerhalb der Blöcke mit den Zellen verfahren. Durch die unterschiedlichen Größen der Diagonalen errechnen durchschnittlich nur ca. die Hälfte der Work-Items neue Werte, allerdings können durch die Nutzung des lokalen Speichers viele Zugriffe auf den globalen Speicher eingespart werden. Bei der Berechnung der Matrix werden die Vorgängerzellen, aus denen die Maxima berechnet wurden, nicht mit abgespeichert. Das Host-Programm rechnet den Rückweg daher nachträglich aus. Dafür wird berechnet, aus welcher Zelle das abgespeicherte Maximum der unteren rechten Zelle stammt. Dieses Verfahren wird für die errechnete Zelle so lange wiederholt, bis die Zelle oben links erreicht ist. Die Zellen, die in dieser Rechnung vorkommen, bilden den Rückweg.

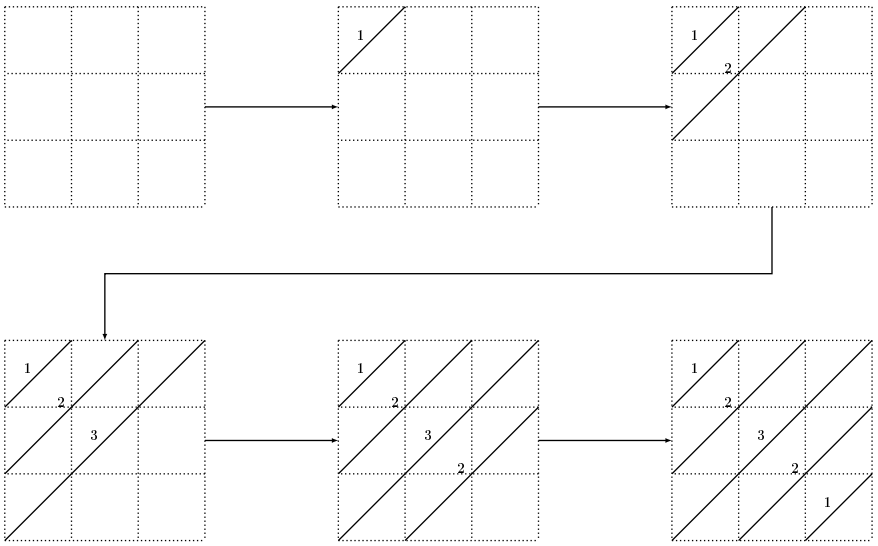

Abbildung 3.1.: Ablauf der Berechnungen der Diagonalen bei einer 3×3-Matrix oder einem 3×3-Block

3.2.1. Erster Ansatz

Bei den ersten Überlegungen, wie der Needleman-Wunsch-Algorithmus mit den Skeletten der SkelCL-Bibliothek umgesetzt werden kann, wurde ein zeilenweises Vorgehen in Betracht gezogen. Dazu sollte wie folgt verfahren werden:
Jede Zeile der Matrix wird durch einen Vektor repräsentiert und diese in der Reihenfolge von oben nach unten neu berechnet. Die Abhängigkeiten einer Zelle $Z_{i,j}$ zu den Zellen $Z_{i-1,j-1}$ und $Z_{i,j-1}$ stellen bei diesem Vorgehen kein Problem dar. Die Berechnungen der Maxima aus den beiden Zellen kann mit dem Map-Skelett realisiert werden. Dabei wird mit einem Index-Vektor *index* gearbeitet und der Vektor der zu berechnenden Zeile *cur*, sowie der Vorgängerzeile *pre* zusammen mit dem negativen Strafwert p als zusätzliche Parameter übergeben. In *pre* befinden sich dabei bereits berechnete Maxima und in *cur* die Werte aus der Substitutionsmatrix. Die Funktion mit dem das Map-Skelett arbeitet sieht dabei wie folgt aus:

$$f(index, cur, pre, p) = \max\{(pre[index-1] + cur[index]), (pre[index] + p)\}$$

Die Abhängigkeit zur Zelle $Z_{i-1,j}$ stellt ein Problem dar, weil alle Zellen einer Zeile parallel bearbeitet werden. Eine Zelle $Z_{i,j}$ ist indirekt aber von allen Zellen $Z_{0,j}, Z_{1,j}, \ldots, Z_{i-1,j}$ abhängig. Dieses Problem kann scheinbar durch Verwendung des Scan-Skelettes gelöst werden. Der entsprechende Operator dafür arbeitet auf dem Ergebnis der oben erläuterten Map-Funktion und sieht folgendermaßen aus:

$$\oplus(a, b, p) = \max\{a + p, b\}$$

p ist hierbei ein zusätzlicher Parameter. Dieser Operator ist allerdings nicht assoziativ. Folgendes Beispiel verdeutlicht dies:

$$\oplus(\oplus(1, 2, 10), 3, 10) = \max\{[\max\{[1 + 10], 2\} + 10], 3\} = \max\{[11 + 10], 3\} = 21$$

$$\neq$$

$$\oplus(1, \oplus(2, 3, 10), 10) = \max\{[1 + 10], \max\{[2 + 10], 3\}\} = \max\{[1 + 10], 12\} = 12$$

Aufgrund des Operators funktioniert dieser Ansatz nicht, da das Scan-Skelett (siehe 1.2.2) einen assoziativen Operator voraussetzt.

3.2.2. Zweiter Ansatz mit Implementierung

Für eine korrekte Implementierung wurde das Verhalten des original OpenCL-Programmes der Rodinia Benchmark Suite mit dem Map-Skelett der SkelCL-Bibliothek nachempfunden. Dabei wird mit Index-Vektoren gearbeitet, deren Länge der Anzahl von Zellen auf den jeweiligen Diagonalen entsprechen. Das Programm wurde so geschrieben, dass das originale OpenCL-Programm möglichst ohne große Veränderungen weiterhin ausgeführt werden kann (Kommandozeilenparameter: `-o` oder `-original`).

Host-Programm

Zunächst werden wie im originalen Programm zwei zufällige Sequenzen erzeugt.

```
srand(7); //set seed for random generator

//generate random sequence
for(int j = 1; j <= dimension; j++){
  seqy[j] = rand() % 10 + 1;
  seqx[j] = rand() % 10 + 1;
}
```

Diese Sequenzen entsprechen bei gleicher Dimension denen des Ausgangs-Programms. Wie in diesem werden nur zufällige Werte von 1 bis 10 benutzt, also nur die zehn entsprechenden Aminosäuren aus der BLOSUM62-Matrix (siehe Anhang A). Anschließend wird die SkelCL-Matrix `reference` vorbereitet, in dem die Werte der Substitutionsmatrix und die Strafwerte am oberen und linken Rand eingetragen werden.

```
//set the values from the Substitution matrix
for (int i = 1 ; i < dimension; i++){
  for (int j = 1 ; j < dimension; j++){
  reference[i][j] = blosum62[seqx[i]][seqy[j]];
  }
}
//set border values
for(int i = 1; i < dimension; i++){
  reference[i][0] = i * (-penalty);
  reference[0][i] = i * (-penalty);
}
```

Nun werden die Map-Skelette durch Einlesen der Skelett-Funktion erstellt und die Work-Group Größe `wgs` gesetzt.

```
SkelCL::Map<void(SkelCL::Index)>
    upper_left(std::ifstream("upper_left.cl"));
upper_left.setWorkGroupSize(wgs);
SkelCL::Map<void(SkelCL::Index)>
    lower_right(std::ifstream("lower_right.cl"));
lower_right.setWorkGroupSize(wgs);
```

Das eigentliche Ausführen der Skelette geschieht in den folgenden Schleifen. In der ersten Schleife werden die oberen linken Diagonalen und in der zweiten Schleife die unteren rechten Diagonalen berechnet.

```
//perform calculation for upper left matrix
for(int i = 1; i <= dimension /wgs; i++){
  SkelCL::IndexVector index = SkelCL::IndexVector(i *wgs);

  upper_left(index, SkelCL::out(reference),
    SkelCL::Local(sizeof(int) * (wgs +1) * (wgs +1)),
    i, penalty, dimension, wgs);
}

//perform calculation for lower right matrix
for(int i = (dimension /wgs) -1; i > 0 ; i--){
  SkelCL::IndexVector index = SkelCL::IndexVector(i *wgs);

  lower_right(index, SkelCL::out(reference),
    SkelCL::Local(sizeof(int) * (wgs +1) * (wgs +1)),
    i, penalty, dimension, wgs);
}
```

Erläuterung:

- `wgs` gibt die Work-Group Größe an, welche auch die Blockgröße bestimmt (kann per Kommandozeilenparameter `-w` oder `-work_ group_ size` angegeben werden).

- Die erste Schleife läuft ab `1`, da mit dem Block oben links angefangen wird. Auf der längsten Diagonalen (von oben rechts nach unten links) stehen genau `dimension` viele Einträge, welche in `dimension/wgs` viele Blöcke aufgeteilt sind, weshalb die Schleife bis `dimension/wgs` läuft.

- Die zweite Schleife läuft von `(dimension/wgs) -1`, da die längste Diagonale bereits in der vorherigen Schleife berechnet wurde und endet beim Laufindex 1 (der Block unten rechts).

- Der Index-Vektor erhält die Größe `i *wgs`, da in jedem Block `wgs` viele Work-Items arbeiten.

- `SkelCL::out(reference)` gibt an, dass die Werte der `reference` Matrix verändert werden und diese bei Zugriff zurück auf den Host kopiert werden müssen.

- `SkelCL::Local(sizeof(int) * (wgs +1) * (wgs +1))` reserviert lokalen Speicher in der Größe `wgs` $+1^2$ für `int` Werte.

In Abbildung 3.2 wird verdeutlicht welche Blöcke der Matrix, bei welchem Skelett-Aufruf bearbeitet werden. u_i bezeichnet dabei die Blöcke, welche in der ersten Schleife mittels `upper_left` berechnet werden und l_i entsprechend die Blöcke, welche in der zweiten Schleife mittels `lower_right` bearbeitet werden. Der Index i entspricht dabei jeweils der Laufvariable `i` aus den Schleifen. Zu beachten ist, dass `upper_left` die Hauptdiagonale(die längste Diagonale) berechnet. Deshalb muss `lower_right` einmal weniger aufgerufen werden.

u_1	u_2	u_3	u_4	u_5	u_6	u_7	u_8
u_2	u_3	u_4	u_5	u_6	u_7	u_8	l_7
u_3	u_4	u_5	u_6	u_7	u_8	l_7	l_6
u_4	u_5	u_6	u_7	u_8	l_7	l_6	l_5
u_5	u_6	u_7	u_8	l_7	l_6	l_5	l_4
u_6	u_7	u_8	l_7	l_6	l_5	l_4	l_3
u_7	u_8	l_7	l_6	l_5	l_4	l_3	l_2
u_8	l_7	l_6	l_5	l_4	l_3	l_2	l_1

Abbildung 3.2.: Kennzeichnung der Skelett-Aufrufe

Skelett-Funktion

Für die Berechnung eines Blockes der Größe `wgs`×`wgs` muss immer die oberste Zeile und die linke Spalte gegeben sein. Dies wird dadurch gewährleistet, dass die Work-Groups über die Grenzen des eigentlichen Blockes hinaus, eine weitere Zeile und eine weitere Spalte berechnen. Da die oberste Zeile und die linke Spalte bereits gegeben waren, kann dies mit `wgs` vielen Work-Items geschehen.
Für die Berechnung müssen also zuerst die oberste Zeile und linke Spalte geladen werden, damit anschließend zunächst die obere linke Hälfte und darauf die untere rechte Hälfte berechnet werden kann.

Quelltextausschnitt 3.1: Skelett-Funktion zur Berechnung der Blöcke in der oberen linken Hälfte der Matrix

```
1  void func(Index index, int_matrix_t reference, __local int* l_ref,
2      int i, int penalty, int n, int wgs){
3    int lid = get_local_id(0) +1;
4    //calculate start block-index
5    int index_x = (i - ((index + wgs) / wgs)) * wgs;
6    int index_y = (index / wgs) * wgs;
7    if(get_local_id(0) == 0){
8      l_ref[0] = get(reference, index_y, index_x);
9    }
10   l_ref[lid] =            get(reference, index_y, index_x + lid);
11   l_ref[(wgs +1) * lid] =   get(reference, index_y +lid, index_x);
12   barrier(CLK_LOCAL_MEM_FENCE);
13   // calculate upper left block
14   for(int k = 1; k <= wgs; k++){
15     if(lid <= k){
16       //read in data
17       int d = l_ref[(wgs +1) * (lid -1) + k -lid] +
18                 get(reference, index_y + lid, index_x +k -lid +1);
19       int l = l_ref[(wgs +1) * lid + k -lid] - penalty;
20       int t = l_ref[(wgs +1) * (lid -1) + k -lid +1] - penalty;
21       //calculate maximim
22       if(t > l){
23         if(t > d){d = t;}
24       }else if(l > d){d = l;}
25       //safe data
26       if((index_x +k -lid +1 < n) && (index_y + lid < n)){
27         l_ref[(wgs +1) * lid + k -lid +1] = d;
28         set(reference, index_y + lid, index_x +k -lid +1, d);
29       }
30     }
31     barrier(CLK_LOCAL_MEM_FENCE);
32   }
33   // calculate lower right block analog
34   return;
35 }
```

Erläuterung der Zeilen aus Quelltextausschnitt 3.1

5,6 Es wird der Index der oberen linken Zelle des Blocks berechnet.

7-9 Der Wert dieser Zelle wird vom Work-Item mit der Local ID 0 in den lokalen Speicher geladen.

10,11 Jedes Work-Item läd einen Wert der obersten Zeile und einen der linken Spalte in den lokalen Speicher.

12 Eine lokale Barriere wird gesetzt, damit sichergestellt wird, dass die oberste Zeile und die linke Spalte des Blocks komplett in den lokalen Speicher geladen werden.

14 Innerhalb dieser Schleife wird die obere linke Hälfte des Blocks berechnet.

15 Es werden wie im Host-Programm jeweils die Diagonalen (von oben rechts nach unten links) berechnet. Es muss also zunächst nur eine Zelle berechnet werden und dann immer mehr, bis zur vollen Diagonalen. Durch die `if`-Abfrage wird sichergestellt, das immer nur entsprechend viele Work-Items Werte berechnen.

17-20 Es werden die Werte bestimmt, die die zu berechnende Zelle annehmen kann.

22-24 Das Maximum dieser Werte wird bestimmt.

62-29 Das Maximum wird abgespeichert. Dies geschieht nur, wenn die berechnete Zelle sich noch innerhalb der Grenzen der Matrix befindet.

31 Eine lokale Barriere wird gesetzt, damit sichergestellt ist, dass alle Ergebnisse gespeichert werden.

Anschließend wird in einer weiteren Schleife, die sich analog zur oben aufgeführten verhält, die untere rechte Hälfte der Matrix berechnet. Es wird allerdings eine Iteration weniger durchgeführt, da die vorherige Schleife bereits die längste Diagonale berechnet hat und die Indices werden anders berechnet.

Die Skelett-Funktion zur Berechnung der Blöcke in der unteren rechten Hälfte der Matrix unterscheidet sich von dieser Skelett-Funktion durch die Berechnung des Indices der oberen linken Zelle des Blocks.

```
//calculate start block-index
int index_x = n -((i - (index / wgs)) * wgs);
int index_y = n -(((index / wgs) +1) * wgs);
```

3.3. Vergleich der OpenCL- mit der SkelCL-Implementierung

Für den Vergleich der Implementierungen mit OpenCL und SkelCL wird zum Einen die Laufzeit herangezogen, die Aufschluss darüber gibt, ob und wie viel Zusatzarbeit die SkelCL-Bibliothek für das ausführende System bedeutet. Zum Anderen wird die Menge und Komplexität der Quelltexte verglichen, um Rückschlüsse auf eine mögliche Vereinfachung der Programmierung mit SkelCL zu ziehen.

3.3.1. Laufzeit

Um die Programme möglichst gut miteinander vergleichen zu können, wurden jeweils im OpenCL- und im SkelCL-Programm vergleichbare Programmabschnitte mit Zeitmarken versehen. So sind Messungen der einzelnen Teilbereiche der Programme möglich. Diese sind:

- Initialisierung von OpenCL bzw. SkelCL
- Erstellen und vorbereiten der benötigten Daten
- Erstellen der Skelette, bzw. Kompilieren der Kernel
- Ausführen der Skelette, bzw. Ausführen der Kernel inklusive Datentransfer

Aus den Messergebnissen (Tabelle 3.1) wird ersichtlich, dass die SkelCL-Implementierung eine längere Gesamtlaufzeit hat. Diese kommt allerdings im Wesentlichen durch das Erstellen der Skelette zu Stande. Da dies unabhängig von den Eingabedaten geschieht, ist zu erwarten, dass sich die Laufzeit hierfür bei steigender Problemgröße nicht ändert. Auch die anderen Programmabschnitte brauchen (durchschnittlich) länger. Während die Initialisierung genau wie das Erstellen der Skelette unabhängig von der Problemgröße ist, ist nicht zu erwarten, dass sich die Laufzeit bei größeren Eingabesequenzen ändert. Dies ist beim Vorbereiten der Daten und Ausführen der Skelette anders. Bei großen Problemgrößen ist daher zu erwarten, dass diese Laufzeiten weiter ansteigen wird.

3.3.2. Quelltext

Im Folgenden wird die Anzahl der einzelnen SkelCL-Befehle aus der SkelCL-Implementierung mit denen der OpenCL-Befehle aus der OpenCL-Implementierung verglichen. Sowohl SkelCL als auch OpenCL müssen zunächst initialisiert werden. In SkelCL funktioniert dies mit einem einzigen Befehl:

```
skelcl::init(skelcl::nDevices(1));
```

Es wird lediglich angegeben, wie viele Devices benutzt werden sollen.

Prog.	Initial.	Daten vorb.	Skelett/Kernel vorb.	Skelett/Kernel ausf.	Summe
s_1	22 ms	3 ms	23 ms	3 ms	51 ms
s_2	22 ms	3 ms	23 ms	3 ms	51 ms
s_3	22 ms	3 ms	18 ms	3 ms	46 ms
s_4	20 ms	3 ms	19 ms	3 ms	45 ms
s_5	20 ms	3 ms	19 ms	2 ms	44 ms
s_6	20 ms	3 ms	18 ms	2 ms	43 ms
s_7	20 ms	3 ms	18 ms	2 ms	43 ms
s_8	20 ms	3 ms	18 ms	2 ms	43 ms
s_9	22 ms	3 ms	18 ms	2 ms	45 ms
s_{10}	22 ms	3 ms	18 ms	3 ms	16 ms
∅	21,0 ms	3,0 ms	19,2 ms	2,5 ms	45,7 ms
o_1	19 ms	1 ms	0 ms	2 ms	22 ms
o_2	20 ms	1 ms	0 ms	2 ms	23 ms
o_3	20 ms	1 ms	0 ms	2 ms	23 ms
o_4	21 ms	1 ms	0 ms	2 ms	24 ms
o_5	21 ms	1 ms	0 ms	2 ms	24 ms
o_6	19 ms	1 ms	0 ms	2 ms	22 ms
o_7	20 ms	1 ms	0 ms	2 ms	23 ms
o_8	20 ms	1 ms	0 ms	2 ms	23 ms
o_9	22 ms	1 ms	0 ms	2 ms	25 ms
o_{10}	21 ms	1 ms	0 ms	2 ms	24 ms
∅	20,3 ms	1 ms	0 ms	2 ms	23,3 ms

Tabelle 3.1.: Jeweils zehn Messungen mit SkelCL (mit Kommandozeilenparameter `-n 256 -v` - mit s_i gekennzeichnet) und zehn Messungen mit OpenCL (mit Kommandozeilenparameter `-n 256 -v -o` - mit o_i gekennzeichnet) auf einem Rechner mit einem Intel Core 2 Duo CPU E8400 und einer Nvidia GeForce 9500 GT.

Für die OpenCL-Implementierung wurde eine eigene Funktion für die Initialisierung geschrieben, in der folgende Schritte ausgeführt werden:

6 Zeilen - OpenCL Kontext erzeugen
7 Zeilen - GPU auswählen
2 Zeilen - Command Queue erzeugen

Bei SkelCL kann das Einlesen der beiden Skelett-Funktion mit dem Erstellen des Skelettes in je einer Quelltextzeile pro Skelett zusammengefasst werden:

```
skelcl::Map<void(skelcl::Index)>
    skelett(std::ifstream("skelett.cl"));
```

So werden beide Kernel eingeladen und dazu das Skelett erstellt.

Bei OpenCL entfällt das Erstellen des Skelettes, allerdings werden nur für das Laden der beiden Kernel schon 23 Zeilen Quelltext benötigt. Um festzulegen, wie viele Work-Items gestartet und wie viele von diesen in einer Work-Group organisiert werden sollen, werden in beiden Implementierungen nur 2 Zeilen Quelltext pro Skelett bzw. Kernel benötigt. In SkelCL geschieht dies mit einem expliziten Setzen der Work-Group Größe und dem Erstellen des Index-Vektors:

```
skelett.setWorkGroupSize(wgs);
skelcl::IndexVector index = skelcl::IndexVector(i *wgs);
```

Der Index-Vektor wird später dem Skelett übergeben. Ist ein Index-Vektor nicht nötig, geschieht das Festlegen der Anzahl zu startender Work-Items implizit durch die Größe, des vom Skelett benötigten Vektors. Bei der OpenCL-Implementierung werden die Größen als Variablen angegeben, die bei der Ausführung mit angegeben werden:

```
global_work[0] = BLOCK_SIZE * blk;
local_work[0]  = BLOCK_SIZE;
```

In der SkelCL-Implementierung werden die beiden Skelette je innerhalb einer Quelltextzeile aufgerufen:

```
skelett(index, skelcl::out(reference), skelcl::Local(sizeof(int) *
    (wgs +1) * (wgs +1)), i, penalty, dimension, wgs);
```

Durch die Angabe der Parameter werden die Daten automatisch auf das OpenCL-Device kopiert. Bei der OpenCL-Implementierung werden für das Übertragen der Daten beider Kernel zusammen 38 Zeilen Quelltext benötigt. Für das Ausführen kommen noch einmal 2 Quelltextzeilen pro Kernel hinzu. Auch das Freigeben der Buffer und der Comand Queue benötigen weitere 5 Zeilen Quelltext. Insgesamt benötigt die SkelCL-Implementierung nur 9 Quelltextzeilen für die angegebenen Funktionalitäten. Die OpenCL-Implementierung braucht mit 89 Quelltextzeilen fast zehn mal so viele.

Komplexität

Schon allein die Menge an Quelltextzeilen der Implementierungen lässt darauf schließen, dass die SkelCL-Implementierung leichter umzusetzen ist als die OpenCL-Implementierung. Anhand der Übertragung einer Matrix auf das OpenCL-Device soll weiter gezeigt werden, dass Programmieren mit OpenCL komplexer ist als mit SkelCL. Dafür betrachten wir die SkelCL Matrix `reference` bzw. das zweidimensionale Array `input_itemsets` der OpenCL-Implementierung. In SkelCL wird `reference` wie folgt als Parameter dem Skelett übergeben:

```
skelett(..., skelcl::out(reference),...);
```

Der Befehl `skelcl::out` muss angegeben werden, da die Daten auf dem OpenCL-Device verändert werden. Die Zeile sorgt dafür, dass die Daten aus `reference` auf das OpenCL-Device kopiert werden, dort verarbeitet und bei erneuten Zugriff auf `reference` im Host-Programm zurück auf den Host kopiert werden.

In der OpenCL-Implementierung wird hierzu zunächst ein Buffer angelegt:

```
cl_mem input_itemsets_d;
input_itemsets_d = clCreateBuffer(context, CL_MEM_READ_WRITE,
   max_cols * max_rows * sizeof(int), NULL, &err );
if(err != CL_SUCCESS) { printf("ERROR: clCreateBuffer
   input_item_set (size:%d) => %d\n", max_cols * max_rows, err);
   return -1;}
```

Anschließend werden die Daten aus `input_itemsets` auf das OpenCL-Device geladen:

```
err = clEnqueueWriteBuffer(cmd_queue, input_itemsets_d, 1, 0,
   max_cols * max_rows * sizeof(int), input_itemsets, 0, 0, 0);
if(err != CL_SUCCESS) { printf("ERROR: clEnqueueWriteBuffer bufIn1
   (size:%d) => %d\n", max_cols * max_rows, err); return -1; }
```

Danach wird es als Argument für den Kernel gesetzt:

```
clSetKernelArg(kernel1, 1, sizeof(void *), (void*) &
   input_itemsets_d);
```

Nachdem der Kernel ausgeführt wurde, werden die Daten zurück auf den Host in das Array `output_itemsets` kopiert:

```
err = clEnqueueReadBuffer(cmd_queue, input_itemsets_d, 1, 0,
   max_cols * max_rows * sizeof(int), output_itemsets, 0, 0, 0);
```

Dieses Beispiel zeigt exemplarisch, wie komplex das Übertragen von Daten auf ein OpenCL-Device und zurück in OpenCL im Gegensatz zu SkelCL ist.

3.3.3. Fazit

Die Implementierung des Needleman-Wunsch-Algorithmus mit SkelCL läuft zwar 25%[1] langsamer, ist aber mit 9 SkelCL spezifischen Quelltextzeilen nicht so komplex wie die OpenCL-Implementierung, die 89 OpenCL spezifische Quelltextzeilen benötigt. Wie beim K-Means-Algorithmus hat sich also gezeigt, dass sich die Implementierung mit SkelCL im Bezug zur Komplexität des Quelltextes lohnt, solange die leichten Unterschiede in der Laufzeit vernachlässigt werden können. Insbesondere hat sich gezeigt, dass auch komplexe Berechnungsmuster wie beim Needleman-Wunsch-Algorithmus auf die Skelette übertragbar sind.

[1] Bezogen auf die Ausführung der Kernel, bzw. der Skelette wie in 3.3.1.

4. Zusammenfassung

Ziel dieser Bachelorarbeit war das Portieren des K-Means- und des Needleman-Wunsch-Algorithmus von OpenCL zu SkelCL, um zu analysieren, inwiefern sich die Komplexität der Quelltexte und die Laufzeit der Programme unterscheiden. Dazu wurden zunächst das Plattform-, Ausführungs- und Speicher-Modell von OpenCL vorgestellt. Danach wurde kurz auf OpenCL C eingegangen und die Plattformunabhängigkeit der OpenCL-Programme beschrieben. Anschließend wurde in die Datentypen und algorithmischen Skelette von SkelCL eingeführt.

Für die Portierung des K-Means-Algorithmus wurde zunächst der Algorithmus theoretisch beschrieben und daraufhin mit einem Beispiel verdeutlicht. Daraufhin wurden die für SkelCL relevanten Stellen der Portierung beschrieben. Für die Implementierung mit SkelCL bot sich direkt die Verwendung des Map-Skelettes an, da die Objekte alle unabhängig von den anderen dem nächsten Cluster zugeordnet werden müssen. Als Datenstruktur für die Objekte hätte sich ein Vektor von Vektoren angeboten. Da ein solches Konstrukt aber nicht über die Skelett-Funktion auf das OpenCL-Device übertragbar ist, wurde auf eine Matrix und den Index-Vektor ausgewichen. Der Vergleich der OpenCL Implementierung mit der in SkelCL erstellten Implementierung hat gezeigt, dass das SkelCL Programm durchschnittlich ca 4,41%[1] mehr Laufzeit benötigt als das OpenCL Programm. Im Quelltext konnten dafür aber 66 Zeilen OpenCL spezifischen Quelltext durch 4 Zeilen SkelCL Befehle ersetzt werden. Die Komplexität der Portierung hat dementsprechend abgenommen.

Auch der Needleman-Wunsch-Algorithmus wurde zunächst theoretisch beschrieben und mit einem Beispiel verdeutlicht. Anschließend wurde ein erster Ansatz aufgezeigt, der allerdings nicht funktioniert. Dabei sollte das Scan-Skelett, neben dem Map-Skelett, benutzt werden, welches einen assoziativen Operator benötigt. Bei dem Operator, der für die Berechnung benötigt wurde, war Assoziativität aber nicht gegeben. Daraufhin wird die Portierung mittels zweier Map-Skeletten beschrieben, die sich an die OpenCL-Implementierung anlehnen. Dabei werden unterschiedlich große Index-Vektoren als Eingabe benutzt und die Ausgabe nicht direkt über das Skelett realisiert, sondern über eine Matrix, die als zusätzlicher Parameter angegeben wurde. Die Ausführung der Skelette hat bei den Tests durchschnittlich 25%[2] mehr Laufzeit als das OpenCL-Programm benötigt. Hierbei konnten 89 OpenCL spezifische Quelltextzeilen durch 9 SkelCL spezifische ersetzt werden, was auf eine Vereinfachung der Programmierung mit SkelCL gegenüber OpenCL hinweist.

[1] Bezogen auf die Ausführung des Kernels bzw. des Skelettes, wie in 2.3.1.
[2] Bezogen auf die Ausführung der Kernel, bzw. der Skelette wie in 3.3.1.

Die fehlende Dokumentation hat das Programmieren mit SkelCL am Anfang erschwert. Waren die Konzepte und Befehle einmal klar, stellte es sich aber als sehr intuitiv raus. Das Arbeiten mit Skeletten erwies sich durch den Index-Vektor und der Möglichkeit, Daten nicht nur über das Ergebnis der Skelett-Funktion auszugeben, als flexibler als gedacht. Durch die Messungen hat sich gezeigt, dass sich mit SkelCL, im Vergleich zu OpenCL, einfacher Programme für GPU-Systeme schreiben lassen, welche die Rechenleistung von GPUs ähnlich gut wie OpenCL ausnutzen können. In dieser Bachelorarbeit wurde nur mit einer GPU gearbeitet. Die Unterstützung von Mulit-GPU Systemen durch SkelCL bietet einen weiteren interessanten Aspekt, der im Vergleich zu entsprechenden OpenCL-Programmen, welche Multi-GPU Systeme nutzen, untersucht werden könnte.

Literaturverzeichnis

[1] OpenCL bei Khronos Group `https://www.khronos.org/opencl/`

[2] Folien zur Vorlesung „Multi-core und GPU: Parallele Programmierung“ aus dem Sommersemester 2012 von Prof. Sergei Gorlatch und Michel Steuwer

[3] Michel Steuwer, Philipp Kegel und Sergei Gorlatch „Towards High-Level Programming of Multi-GPU Systems Using the SkelCL Library“ erschienen 2012 auf dem *IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDPSW)* in Shanghai

[4] Die Rodinia Benchmark Suite `http://lava.cs.virginia.edu/Rodinia/`

[5] Christopher M. Bishop „Pattern Recognition and Machine Learning“ erschienen 2006 im Springer Verlag, ISBN 978-0-387-31073-2

[6] Waterman, Smith und Beyer „Some Biological Sequence Metrics“ erschienen 1976

A. Substitutionsmatrix BLOSUM62

	Ala	Arg	Asn	Asp	Cys	Gln	Glu	Gly	His	Ile	Leu	Lys	Met	Phe	Pro	Ser	Thr	Trp	Tyr	Val
Ala	4	−1	−2	−2	0	−1	−1	0	−2	−1	−1	−1	−1	−2	−1	1	0	−3	−2	0
Arg	−1	5	0	−2	−3	1	0	−2	0	−3	−2	2	−1	−3	−2	−1	−1	−3	−2	−3
Asn	−2	0	6	1	−3	0	0	0	1	−3	−3	0	−2	−3	−2	1	0	−4	−2	−3
Asp	−2	−2	1	6	−3	0	2	−1	−1	−3	−4	−1	−3	−3	−1	0	−1	−4	−3	−3
Cys	0	−3	−3	−3	9	−3	−4	−3	−3	−1	−1	−3	−1	−2	−3	−1	−1	−2	−2	−1
Gln	−1	1	0	0	−3	5	2	−2	0	−3	−2	1	0	−3	−1	0	−1	−2	−1	−2
Glu	−1	0	0	2	−4	2	5	−2	0	−3	−3	1	−2	−3	−1	0	−1	−3	−2	−2
Gly	0	−2	0	−1	−3	−2	−2	6	−2	−4	−4	−2	−3	−3	−2	0	−2	−2	−3	−3
His	−2	0	1	−1	−3	0	0	−2	8	−3	−3	−1	−2	−1	−2	−1	−2	−2	2	−3
Ile	−1	−3	−3	−3	−1	−3	−3	−4	−3	4	2	−3	1	0	−3	−2	−1	−3	−1	3
Leu	−1	−2	−3	−4	−1	−2	−3	−4	−3	2	4	−2	2	0	−3	−2	−1	−2	−1	1
Lys	−1	2	0	−1	−3	1	1	−2	−1	−3	−2	5	−1	−3	−1	0	−1	−3	−2	−2
Met	−1	−1	−2	−3	−1	0	−2	−3	−2	1	2	−1	5	0	−2	−1	−1	−1	−1	1
Phe	−2	−3	−3	−3	−2	−3	−3	−3	−1	0	0	−3	0	6	−4	−2	−2	1	3	−1
Pro	−1	−2	−2	−1	−3	−1	−1	−2	−2	−3	−3	−1	−2	−4	7	−1	−1	−4	−3	−2
Ser	1	−1	1	0	−1	0	0	0	−1	−2	−2	0	−1	−2	−1	4	1	−3	−2	−2
Thr	0	−1	0	−1	−1	−1	−1	−2	−2	−1	−1	−1	−1	−2	−1	1	5	−2	−2	0
Trp	−3	−3	−4	−4	−2	−2	−3	−2	−2	−3	−2	−3	−1	1	−4	−3	−2	11	2	−3
Tyr	−2	−2	−2	−3	−2	−1	−2	−3	2	−1	−1	−2	−1	3	−3	−2	−2	2	7	−1
Val	0	−3	−3	−3	−1	−2	−2	−3	−3	3	1	−2	1	−1	−2	−2	0	−3	−1	4

Plagiatserklärung

Hiermit versichere ich, dass die vorliegende Arbeit über Implementierung des K-Means- und Needleman-Wunsch- Algorithmus mit der SkelCL-Bibliothek selbstständig verfasst worden ist, dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt worden sind und dass die Stellen der Arbeit, die anderen Werken – auch elektronischen Medien – dem Wortlaut oder Sinn nach entnommenen wurden, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht worden sind.

Münster, 27. Juni 2013

Ich erkläre mich mit einem Abgleich der Arbeit mit anderen Texten zwecks Auffindung von Übereinstimmungen sowie mit einer zu diesem Zweck vorzunehmenden Speicherung der Arbeit in eine Datenbank einverstanden.

Münster, 27. Juni 2013