Praktikumsanleitung
GPU-Computing

Manuel Hohmann

20. Juni 2012

Inhaltsverzeichnis

Kapitel 1

Grundlagen

Ziel des Versuchs "GPU-Computing" ist es, die Grundlagen der GPU-Programmierung mittels der Programmiersprache OpenCL zu erlernen. Die vorliegende Anleitung bildet daher eine kurze Einführung in OpenCL und das GPU-Computing. Im Laufe des Versuche sollen einige einfache Aufgaben mittels GPU-Computing gelöst werden.

1.1 GPU-Computing

Dieser Abschnitt erläutert die wichtigsten Begriffe zum Thema GPU-Computing und die grundlegenden Unterschiede zur gewöhnlichen, CPU-basierten Programmierung.

1.1.1 GPUs, Kernels und Arbeitsgruppen

Ähnlich zu einer CPU besteht eine GPU aus Recheneinheiten, die arithmetische und logische Operationen ausführen, sowie aus Einheiten, die dazu dienen, Daten zu transferieren, Maschinenanweisungen zu decodieren und die Ausführung zu kontrollieren. Im Gegensatz zu einer CPU besitzt eine GPU allerdings eine weitaus größere Anzahl an Recheneinheiten, die parallel arbeiten und damit eine größere Zahl von arithmetischen Operationen bei gleicher Taktrate ausführen können. Darüber hinaus verfügen GPUs für gewöhnlich über einen breiteren Datenbus, wodurch ein schnellerer Datentransfer zwischen Speicher und Recheneinheiten möglich ist. Dadurch sind GPUs besonders gut für die parallele Verarbeitung großer Datenmengen geeignet.

Das Programm, das auf einer GPU ausgeführt wird, besteht aus Funktionen, die als Kernels bezeichnet werden. Häufig wird der gleiche Kernel auf vielen, meistens sogar allen Recheneinheiten einer GPU gleichzeitig ausgeführt. Ein Kernel wird also nicht nur einmalig, sondern mit einer großen Anzahl von Instanzen ausgeführt. Diese Kernel-Instanzen sind in einer logischen Struktur eines endlichen Gitters angeordnet, d.h. jede Kernel-Instanz kann durch einen Satz von N Gitterkoordinaten adressiert werden, wobei die Dimension N des Gitters typischerweise im Bereich $N \in \{1, 2, 3\}$ liegt. Diese Koordinaten einer Kernel-Instanz werden als globale ID der Instanz bezeichnet.

Das Gitter der Kernel-Instanzen wird in Blöcke der gleichen Größe unterteilt, die als Arbeitsgruppen bezeichnet werden. Kernel-Instanzen der gleichen Arbeitsgruppe werden auf einer GPU auf Recheneinheiten ausgeführt, die physikalisch zu einer größeren Einheit zusammengefasst sind und z.B. einen gemeinsamen Speicherbereich teilen können. Analog zu den Kernel-Instanzen besitzt jede Arbeitsgruppe über einen Satz von N Koordinaten, der ihre Position im Gitter angibt. Zusätzlich besitzt jede Kernel-Instanz einen Satz von Gitterkoordinaten relativ zum Anfangspunkt der zugehörigen Arbeitsgruppe, die als lokale ID bezeichnet werden. Alternativ zur globalen ID kann eine Kernel-Instanz daher auch über die Arbeitsgruppen-ID und die lokale ID adressiert werden.

1.1.2 Warum GPU-Computing?

Der besondere Aufbau einer GPU, der auf die parallele Verarbeitung großer Datenmengen ausgerichtet ist, prädestiniert ihre Verwendung für grafische Aufgaben wie die Berechnung dreidimensionaler Objekte und Texturen sowie für Effekte wie Beleuchtung, Nebel oder Linseneffekte. Bei diesen Aufgaben muss eine große Datenmenge, wie z.B. die Koordinaten, Farben und Oberflächenstruktur eines Objekts, innerhalb möglichst kurzer Zeit in eine grafische Bildschirmdarstellung umgerechnet werden. Dies trifft insbesondere auf Echtzeit-Anwendungen wie z.B. grafische Computerspiele zu, in denen die Berechnung der Bildschirmdarstellung mit der Framerate getaktet ist. Für eine flüssige Darstellung sind hohe Frameraten und daher eine kurze Rechenzeit erforderlich.
Die hohe Datenverarbeitungsrate ist auch für viele wissenschaftliche Aufgaben der Datenverarbeitung und Datenauswertung nützlich. Ein typisches Beispiel ist die Auswertung der Messdaten von Beschleunigerexperimenten wie dem LHC. Die Detektoren am LHC bestehen auf einer großen Anzahl von einzelnen Messgeräten, die die Spuren von neu erzeugten Teilchen sowie ihre Impulse vermessen. Um diese Daten zu filtern und auszuwerten, muss eine große Zahl von Vektordaten verarbeitet werden. Dies ist auf einer GPU besonders leicht möglich, da diese speziell darauf ausgerichtet sind, die Geometrie dreidimensionaler Objekte zu berechnen.
Eine andere wichtige Aufgabe für GPU-Computing ist die Simulation komplexer physikalischer Systeme. Damit ein physikalisches System unter Benutzung einer GPU simuliert werden kann, muss die Simulation sich für eine parallele Berechnung eignen. Ein Beispiel dafür sind Vielteilchen-Systeme, bei denen der Zustand einer großen Anzahl von Teilchen unter dem Einfluss einer Dynamik berechnet werden soll. Im einfachsten Fall handelt es sich dabei um Teilchen im klassischen Sinne, die durch Positionen und Geschwindigkeiten charakterisiert werden. Eine andere Möglichkeit ist die Simulation von Kontinua, die durch Gitter approximiert werden. Solche Simulationen finden z.B. in der Quantenchromodynamik, der Plasmaphysik oder der Klimaforschung Anwendung.
Auch zahlreiche mathematische Verfahren, die in der Physik Anwendung finden, lassen sich durch GPU-Computing beschleunigen. Ein Beispiel, das auch in diesem Praktikum zur Anwendung kommt, ist die Rechnung mit großen Matrizen und Vektoren. Andere Beispiele sind die Fourier-Transformation und die Radon-Transformation, die z.B. in bildgebenden Verfahren wie der Computer-Tomographie benötigt wird.

1.2 Der Praktikumsversuch

In diesem Abschnitt wird der Ablauf des Praktikumsversuchs GPU-Computing erläutert. Die Dauer des Versuchs beträgt eine Woche.

1.2.1 Voraussetzungen

Da es sich beim Versuch GPU-Computing um einen Versuch zum Thema fortgeschrittene Programmierung handelt, sind für die erfolgreiche Durchführung grundlegende Kenntnisse in Computer-Programmierung notwendig. Insbesondere sind Kenntnisse in der Programmiersprache C erforderlich, auf der die im Versuch benutzte Sprache OpenCL aufbaut. Während des Versuchs werden Programme in C und OpenCL geschrieben. Weiterhin wird in einer Linux-Umgebung und mit der Kommandozeile gearbeitet. Daher sind grundlegende Kenntnisse im Umgang mit Linux und der Kommandozeile erforderlich.

1.2.2 Vorbereitung

Die vorliegende Praktikumsanleitung bildet einen Leitfaden für die Vorbereitung auf den Praktikumsversuch und behandelt die für die Versuchsdurchführung notwendigen Themengebiete. Es handelt sich jedoch nicht um ein vollständiges OpenCL-Lehrbuch. Für die erfolgreiche Vorbereitung auf den Versuch ist daher das Studium weiterer Literatur notwendig.

Eine Auswahl empfohlener Literatur befindet sich am Ende dieser Anleitung. Ziel der Vorbereitung ist es, eine Übersicht über die Programmierung in der Sprache OpenCL sowie über die wichtigsten Funktionen der OpenCL-Laufzeitbibliothek zu gewinnen. Weiterhin sollten zu Beginn des Praktikums Kenntnisse über die in Abschnitt 4 beschriebenen mathematischen Aufgaben vorhanden sein, die im Laufe der Praktikumswoche bearbeitet werden. Im Folgenden werden einige Fragen aufgelistet, die zur Selbstüberprüfung der Vorbereitung dienen können.

1.2.2.1 GPU-Computing

- Was bedeutet GPU?
- Wie ist eine GPU aufgebaut?
- Was ist der Unterschied zwischen einer GPU und einer CPU?
- Warum bieten GPUs einen Geschwindigkeitsvorteil?
- Welche Probleme können durch GPU-Computing bearbeitet werden?

1.2.2.2 OpenCL

- Welche Datentypen sind in OpenCL definiert?
- Was bedeutet `float4`?
- Wie lassen sich Vektoren addieren?
- Was bewirkt die Funktion `get_global_id`?
- Wie wird eine Kernel-Funktion gekennzeichnet?
- Was unterscheidet einen Kernel von anderen Funktionen?
- Welche Speicherbereiche werden in OpenCL unterschieden?
- Wie kennzeichnet man einen Zeiger auf den globalen Speicher?

1.2.2.3 Laufzeitbibliothek

- Was ist eine Plattform?
- Was ist ein Gerät?
- Wie lassen sich die Eigenschaftes eines Geräts abfragen?
- Wie erstellt man einen Kontext?
- Welcher Datentyp beschreibt einen Speicherbereich in OpenCL?
- Was ist ein nicht-blockierender Datentransfer?
- Wie wird ein OpenCL-Quelltext kompiliert?
- Wie kann man Kernel-Argumente mit Werten füllen?
- Welches Objekt ermöglicht die Ausführung eines Kernels?
- Wie kann man die Hintereinanderausführung von Aufgaben gewährleisten?

1.2.2.4 Mathematik und Physik

- Wie werden zwei Matrizen multipliziert?
- Was ist die Mandelbrot-Menge?
- Wie führt man eine Iteration mit Abbruchkriterium durch?
- Was ist ein Vielteilchen-Problem?
- Wie kann man die Dynamik eines Vielteilchen-Systems beschreiben?
- Was ist das Euler-Verfahren?

1.2.3 Versuchsdurchführung

Für die Durchführung des Versuchs steht Ihnen eine Workstation mit einer NVidia Tesla C2075 Grafikkarte zur Verfügung. Ihre Aufgabe wird es sein, sich zunächst mit der Programmierumgebung vertraut zu machen und einige einfache Programme zu kompilieren. Diese Programme werden Ihnen helfen, die GPU-Hardware kennenzulernen und ihre Eigenschaften in Erfahrung zu bringen. Anschließend werden Sie eigene Programme schreiben, um physikalische Simulationen mittels GPU-Computing durchzuführen.

1.2.4 Auswertung und Protokoll

Wichtiger Bestandteil des Versuchs ist die Dokumentation Ihrer Arbeit in einem Versuchsprotokoll. Dieses Protokoll soll so aufgebaut sein, dass es dem Leser ermöglichst, den von Ihnen durchgeführten Versuch nachzuvollziehen und die einzelnen Schritte zu wiederholen. Daher sollten zumindest die folgenden Informationen zu jeder der von Ihnen gelösten Aufgaben vorhanden sein:

- Ziel des Versuchs.
- Quelltext des verwendeten Programms mit Erläuterungen.
- Ausführliche Beschreibung des Programmablaufs.
- Beschreibung der Ein- und Ausgabedaten.
- Diskussion der Ergebnisse.

Beachten Sie insbesondere die Notwendigkeit von Quellenangaben für die von Ihnen benutzte Literatur.

Kapitel 2

Einführung in OpenCL

In diesem Kapitel soll die grundlegende Programmierung in OpenCL erläutert werden.

2.1 Programmaufbau

Jedes OpenCL-Programm besteht aus zwei Komponenten:

- *Device-Code*: Der Device-Code führt die eigentliche Berechnung auf der Rechenhardware (typischerweise eine GPU) durch und ist in der Programmiersprache OpenCL geschrieben. Damit der Code portabel und auf verschiedenen GPU-Architekturen lauffähig ist, wird er dem Programm als ASCII-Datei beigefügt und erst zur Laufzeit kompiliert, wenn feststeht, auf welcher GPU er ausgeführt werden soll. Der dafür notwendige Compiler ist Bestandteil der OpenCL-Laufzeitbibliothek.
- *Host-Code*: Aufgabe des Host-Codes ist es, die Arbeit der GPU zu verwalten, mit dem Benutzer zu interagieren und die Ein- und Ausgabe von Daten zu koordinieren. Im Gegensatz zum Device-Code wird der Host-Code nicht auf der GPU, sondern auf der CPU des Host-Rechners ausgeführt. Es handelt sich dabei um ein gewöhnliches, kompiliertes Programm, das z.B. in C programmiert wird. Damit der Host-Code auf die GPU zugreifen und ihr Aufgaben zuweisen kann, ist die Unterstützung des GPU-Treibers notwendig. Als Interface zwischen dem Host-Code und dem GPU-Treiber dient die OpenCL-Laufzeitbibliothek. Diese enthält u.a. Routinen zum Datentransfer zwischen Hauptspeicher und GPU-Speicher sowie zum Kompilieren und Ausführen des Device-Codes.

Als Interface zwischen dem Device-Code und dem Host-Code dienen die Funktionen der OpenCL-Laufzeitbibliothek. Dieser werden später in diesem Kapitel beschrieben.

2.2 OpenCL-Kernels

Zentraler Bestandteil jedes OpenCL-Programms ist der Kernel, der von der GPU ausgeführt wird. In diesem Abschnitt werden die Grundlagen der Kernel-Programmierung in der Programmiersprache OpenCL erläutert.

2.2.1 Kernel-Funktionen

Die Programmiersprache OpenCL ist von C abgeleitet. Ein OpenCL-Kernel hat daher Ähnlichkeit mit einer C-Funktion. Es werden jedoch keine Daten zurückgegeben, daher handelt es sich um eine Funktion mit Rückgabetyp `void`, wie an folgendem Beispiel deutlich wird:

```
1 __kernel void Calculate(__global float* input, __global float* output)
2 {
```

```
3     // TODO: Führe Berechnung aus.
4 }
```

Durch das Schlüsselwort `__kernel` wird die Funktion als Kernel deklariert, der vom Host-Code aus aufgerufen werden kann. Ohne das Schlüsselwort `__kernel` erhält man eine OpenCL-Funktion, die nicht aus dem Host-Code aufgerufen werden kann und einer gewöhnlichem C-Funktion mit beliebigem Rückgabetyp entspricht, wie im folgenden Beispiel:

```
1 float Sum(float x, float y)
2 {
3     return(x + y);
4 }
5
6 __kernel void Calculate(__global float* input, __global float* output)
7 {
8     // TODO: Führe Berechnung aus und benutze Funktion Sum.
9 }
```

In diesem Fall kann die Kernel-Funktion `Calculate` vom Host-Code aus aufgerufen werden, die Funktion `Sum` dagegen nur innerhalb des Device-Codes.

2.2.2 Datentypen

Neben den aus C bekannten einfachen Datentypen unterstützt OpenCL auch Vektordatentypen und komplexe Datentypen, um z.B. Bilder zu repräsentieren. Die wichtigsten Datentypen sollen im folgenden näher erläutert werden.

2.2.2.1 Einfache Datentypen

Die einfachen Datentypen sind überwiegend, teilweise mit Modifikationen, aus C übernommen. Die folgende Liste erläutert die am häufigsten gebrauchten Datentypen:

- `bool`: Boolescher Wahrheitswert, `true` oder `false`.
- `char`: Vorzeichenbehaftete 8-Bit Ganzzahl.
- `uchar`, `unsigned char`: Vorzeichenlose 8-Bit Ganzzahl.
- `short`: Vorzeichenbehaftete 16-Bit Ganzzahl.
- `ushort`, `unsigned short`: Vorzeichenlose 16-Bit Ganzzahl.
- `int`: Vorzeichenbehaftete 32-Bit Ganzzahl.
- `uint`, `unsigned int`: Vorzeichenlose 32-Bit Ganzzahl.
- `long`: Vorzeichenbehaftete 64-Bit Ganzzahl.
- `ulong`, `unsigned long`: Vorzeichenlose 64-Bit Ganzzahl.
- `float`: 32-Bit Gleitkommazahl.
- `half`: 16-Bit Gleitkommazahl.

Im Gegensatz zum Datentyp `float` ist der Datentyp `double`, der eine 64-Bit Gleitkommazahl darstellt, *nicht* Teil des OpenCL-Standards, sondern eine Erweiterung dazu. Die Verfügbarkeit dieses Datentyps hängt daher von der verwendeten Rechen-Hardware ab.

2.2.2.2 Vektordatentypen

Eine besondere Stärke von OpenCL ist die Möglichkeit, mit Vektoren zu rechnen. Dafür bietet OpenCL eine Reihe von Vektordatentypen an, die aus 2, 3, 4, 8 oder 16 Elementen vom Typ `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `long`, `ulong` oder `float` bestehen (oder `double`, sofern dieser Datentyp verfügbar ist). Die folgenden Beispiele erläutern die Deklaration und Initialisierung von Vektordatentypen:

```
1  // Ein Vektor vom Typ float4 mit 4 Gleitkommazahlen
2  float4 a = (float4)(0.0, 1.0, 2.0, 3.0);
3
4  // Ein Vektor vom Typ float2 mit 2 Gleitkommazahlen
5  float2 b = (float2)(1.0, 2.0);
6
7  // Äquivalent zu (0.0, 0.0)
8  float2 c = (float2)(0.0);
9
10 // Kombination von zwei float2 zu einem float4
11 float4 d = (float4)(b, c);
12
13 // Kombination von gemischten Vektordatentypen
14 float8 e = (float8)(b, a, c);
```

Arithmetische Operationen können auf Vektoren angewandt werden und werden dann komponentenweise ausgeführt. Viele Hardware-Architekturen unterstützen die parallele Ausführung dieser Vektor-Operationen. Dies wird als SIMD (single instruction, multiple data) bezeichnet und führt zu einem deutlichen Geschwindigkeitsvorteil gegenüber der seriellen Ausführung. In OpenCL werden diese Vektoroperationen durch die üblichen arithmetischen Operatoren dargestellt:

```
1  // Deklaration der Variablen
2  float4 a, b, c;
3  int8 d, e, f, g;
4
5  // Komponentenweise Addition von zwei float4 Vektoren
6  c = a + b;
7
8  // Komponentenweise Multiplikation von zwei int8 Vektoren
9  f = d * e;
10
11 // Multiplikation aller Komponenten eines int8 Vektors mit einer Zahl
12 g = 2 * f;
```

Häufig ist es notwendig, auf einzelne Komponenten von Vektoren zuzugreifen. Bei den Datentypen mit 2, 3 oder 4 Elementen erfolgt dieser Datenzugriff wie bei einer in C deklarierten `struct` mit den Elementen `x`, `y`, `z` und `w`. Bei den Datentypen mit 8 und 16 Elementen sind diese Elemente mit `s0` bis `s7` bzw. mit `s0` bis `s0` bis `s9` und `sa` bis `sf` bezeichnet. Die gleiche Notation kann auch für die kürzeren Vektordatentypen benutzt werden, wie in folgendem Beispiel gezeigt:

```
1  // Deklariere und initialisiere eine Variable vom Typ float4.
2  float4 a = (float4)(1.0, 2.0, 3.0, 4.0);
3
4  // b wird mit dem Wert 3.0 initialisiert.
5  float b = a.z;
6
7  // c wird ebenfalls mit dem Wert 3.0 initialisiert.
8  float c = a.s2;
9
```

```
10 // Deklariere und initialisiere eine Variable vom Typ float16.
11 float16 d = (a, a, a, a);
12
13 // e wird mit dem Wert 4.0 initialisiert.
14 float e = d.sf;
```

Im Gegensatz zu C können mehrere Indizes aneinandergefügt werden, um auf mehrere Elemente eines Vektordatentyps gleichzeitig zuzugreifen:

```
1  // Deklaration der Variablen
2  float2 a, b;
3  float4 c;
4  float8 d;
5  float16 e;
6
7  // Umdrehen eines Vektors vom Typ float2.
8  b.xy = a.yx;
9
10 // Die gleiche Operation unter Verwendung von s0 und s1.
11 b.s01 = a.s10;
12
13 // Einlesen eines Teils eines Vektors.
14 c.xyzw = d.s4567;
15
16 // Gleichzeitiges Halbieren und Umdrehen eines Vektors.
17 d.s01234567 = e.sfedcba98;
```

Im Falle der Buchstaben `x`, `y`, `z`, `w` werden diese einfach aneinandergefügt, um mehrere Elemente zu indizieren. Bei der Schreibweise mit `s` werden nur die folgenden Hexadezimalzahlen aneinandergefügt.

2.2.3 Speicherbereiche

OpenCL unterscheidet zwischen vier verschiedenen Speicherbereichen: globaler Speicher, lokaler Speicher, Konstantenspeicher und privater Speicher. Variablen, die innerhalb von Funktionen deklariert sind, befinden sich typischerweise im privaten Speicher. Auf die anderen Speicherbereiche wird üblicherweise mittels Zeigervariablen zugegriffen. Der folgende Abschnitt erläutert die Verwendung dieser Speicherbereiche.

2.2.3.1 Privater Speicher

Lokale Variablen, die innerhalb einer Funktion oder eines Kernels deklariert werden, und Funktionsargumente befinden sich automatisch im privaten Speicher und sind nur innerhalb dieser Funktion zugänglich. Um eine Variable explizit dem privaten Speicher zuzuweisen, dient das Schlüsselwort `__private`:

```
1 // Funktionsargumente sind automatisch __private
2 void Test(int a, float b)
3 {
4     // Lokale Variablen sind automatisch __private
5     float4 c;
6
7     // Explizite (und redundante) __private Deklaration
8     __private int2 d;
9 }
```

Daten im privaten Speicher sind lokal innerhalb einer Kernel-Instanz, d.h. für jede Instanz des Kernels existiert eine eigene Kopie dieser Daten. Wenn eine Kernel-Instanz die Daten im privaten Speicher ändert, sind die anderen, parallel dazu rechnenden Kernel-Instanzen

davon nicht betroffen. OpenCL-Geräte können dafür einen eigenen physikalischen Speicherbereich für jede der Recheneinheiten besitzen. Dieser ist für gewöhnlich deutlich kleiner als der Hauptspeicher. Daher eignet sich der private Speicher nur für kleine Datenmengen.

2.2.3.2 Lokaler Speicher

Lokale Variablen innerhalb einer Funktion können auch dem lokalen Speicher zugewiesen werden. Dafür dient das Schlüsselwort `__local`:

```
1 void Test(void)
2 {
3     // Explizite __local Deklaration
4     __local int8 data;
5 }
```

Daten im lokalen Speicher sind lokal innerhalb einer Arbeitsgruppe, d.h. sie werden von allen Kernel-Instanzen innerhalb dieser Arbeitsgruppe gemeinsam genutzt. Wenn eine Kernel-Instanz die Daten im lokalen Speicher ändert, betrifft dies alle Kernel-Instanzen in der selben Arbeitsgruppe. Für jede Arbeitsgruppe existiert eine eigene Kopie der Daten. Wie auch beim privaten Speicher können OpenCL-Geräte einen eigenen physikalischen Speicher für diesen Zweck besitzen.

2.2.3.3 Globaler Speicher

Daten, die zwischen Host und OpenCL-Gerät übertragen werden sollen, befinden sich im globalen Speicher. Um auf diese Daten zuzugreifen, wird typischerweise ein Zeiger auf den entsprechenden Speicherbereich als Funktionsargument übergeben. Das Schlüsselwort `__global` deklariert einen solchen Zeiger als Zeiger auf den globalen Speicher:

```
1 // data zeigt auf Daten im globalen Speicher.
2 void Test(__global float4* data)
3 {
4     // TODO: Verarbeite Daten.
5 }
```

Von Daten im globalen Speicher existiert nur eine Kopie, auf die sämtliche Kernel-Instanzen zugreifen. Wenn eine Kernel-Instanz diese Daten verändert, sind davon auch alle anderen Kernel-Instanzen betroffen. Der globale Speicher ist üblicherweise der größte Speicherbereich eines OpenCL-Gerätes, allerdings ist der Zugriff darauf auch am langsamsten.

2.2.3.4 Konstantenspeicher

Daten, die von OpenCL-Funktionen nur gelesen, aber nicht geschrieben werden, können statt im globalen Speicher auch im Konstantenspeicher abgelegt werden. Die Deklaration ist ähnlich der des globalen Speichers, hier wird jedoch das Schlüsselwort `__constant` benutzt:

```
1 // data zeigt auf Daten im Konstantenspeicher.
2 void Test(__constant float4* data)
3 {
4     // TODO: Verarbeite Daten.
5 }
```

Da der Datentransfer zwischen Speicher und Recheneinheiten beim Konstantenspeicher nur in eine Richtung stattfindet, können OpenCL-Geräte hierfür einen eigenen physikalischen Speicher bereitstellen, auf den schneller zugegriffen werden kann. Darüber hinaus können häufig abgefragte Daten in einem lokalen Cache zwischengespeichert werden, da garantiert ist, dass diese Daten sich nicht durch die Ausführung anderer Kernel-Instanzen ändern.

2.2.4 Eingebaute Funktionen

Wie in C gibt es auch in OpenCL eine Reihe von vordefinierten, eingebauten Funktionen. In C werden dieser Funktionen üblicherweise in Header-Dateien wie `stdio.h` deklariert, die dann mittels `#include` eingebunden werden. Dieses Einbinden von Header-Dateien für eingebaute Funktionen ist in OpenCL nicht notwendig. Die hier beschriebenen Funktionen sind automatisch global verfügbar.

2.2.4.1 Arbeitsverwaltungsfunktionen

Wie in Abschnitt 1.1.1 beschrieben wird ein GPU-Kernel nicht nur einmal aufgerufen, sondern einmal für jeden Gitterplatz in einem N-dimensionalen, endlichen Gitter. Die hier beschriebenen Funktionen dienen dazu, die Eigenschaften dieses Gitters innerhalb des Device-Codes abzufragen.

- `uint get_work_dim(void)`

  Gibt die Dimension N des Gitters zurück.

- `size_t get_global_size(uint dim)`

  Gibt die Anzahl der Gitterplätze entlang der Dimension `dim` zurück, wobei `dim` im Bereich $0, \ldots, N-1$ liegt.

- `size_t get_global_id(uint dim)`

  Gibt eine Koordinate des Gitterplatzes zurück, an dem die aktuelle Kernel-Instanz ausgeführt wird.

- `size_t get_local_size(uint dim)`

  Gibt die Anzahl der Gitterplätze entlang der Dimension `dim` innerhalb der lokalen Arbeitsgruppe zurück.

- `size_t get_local_id(uint dim)`

  Gibt eine Koordinate des Gitterplatzes innerhalb der lokalen Arbeitsgruppe zurück, an dem die aktuelle Kernel-Instanz ausgeführt wird.

- `size_t get_num_groups(uint dim)`

  Gibt die Anzahl der Arbeitsgruppen entlang der Dimension `dim` zurück.

- `size_t get_group_id(uint dim)`

  Gibt zurück, in welcher Arbeitsgruppe die aktuelle Kernel-Instanz ausgeführt wird.

- `size_t get_global_offset(uint dim)`

  Gibt eine Koordinate des ersten Gitterplatzes zurück.

Die Verwendung dieser Funktionen wird in den Code-Beispielen in den folgenden Abschnitten dieser Anleitung erläutert.

2.2.4.2 Mathematische Funktionen

Wie auch in der C-Standard-Mathematik-Bibliothek sind in OpenCL zahlreiche mathematische Funktionen vorhanden, die für Berechnungen mit Fließkommazahlen benutzt werden können. Im Gegensatz zu C muss jedoch kein Header dafür eingebunden werden, wie es beim Header `math.h` in C der Fall ist. Auch das Linken mit der Mathematik-Bibliothek entfällt. Stattdessen sind die Funktionen fest in die OpenCL-Sprache integriert. Die folgende Liste gibt einen Einblick in die vorhandenen Funktionen:

- Exponentialfunktion und Logarithmus: `exp`, `exp2`, `exp10`, `log`, `log2`, `log10`.
- Trigonometrische Funktionen: `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`.
- Hyperbolische Funktionen: `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`.
- Wurzeln: `sqrt`, `cbrt`.
- Potenzen: `pow`.
- Rundungsfunktionen: `round`, `floor`, `ceil`.

Eine vollständige Liste inklusive der kompletten Aufruf-Syntax befindet sich in der OpenCL-Spezifikation [3]. Die meisten Funktionen können als Argument entweder einen skalaren Wert vom Typ `float` verarbeiten oder komponentenweise auf einem Vektor-Datentyp vom Typ `float2`, `float3`, `float4`, `float8`, `float16` operieren. Der Rückgabetyp der Funktion entspricht dabei dem Typ des Arguments.

2.3 Laufzeitbibliothek

Um die in OpenCL geschriebenen Kernels auf unterschiedlicher Hardware, insbesondere auf GPUs, ausführen zu können, ist die Unterstützung einer Laufzeitbibliothek notwendig. Aufgabe der Laufzeitbibliothek ist es, die Rechenhardware zu verwalten, Daten zu übertragen und OpenCL-Kernels zur Ausführung zu bringen. In diesem Abschnitt werden ihre wichtigsten Funktionen eingeführt, so weit sie im Praktikum benötigt werden. Es handelt sich nicht um eine vollständige Liste oder eine vollständige Beschreibung der aufgelisteten Funktionen. Eine ausführliche Beschreibung der Laufzeitbibliothek inklusive aller Funktionen befindet sich in der OpenCL-Spezifikation [3].

2.3.1 Einbinden der Laufzeitbibliothek in C

Um die Datentypen, Funktionen und vordefinierten Konstanten der OpenCL-Laufzeitbibliothek in C verwenden zu können, muss eine Header-Datei eingebunden werden, die typischerweise unter dem Dateinamen `cl.h` im Verzeichnis `CL` zu finden ist:

```
1 #include <CL/cl.h>
```

Der Speicherort dieses Ordners und der darin enthaltenen Header-Dateien ist abhängig von der verwendeten Laufzeitumgebung und muss dem C-Compiler mitgeteilt werden, wie in Abschnitt 3.1 beschrieben.

2.3.2 Fehlerbehandlung

Die meisten OpenCL-Funktionen geben einen Fehlercode vom Typ `cl_int` zurück. Dieser gibt an, ob die Funktion erfolgreich ausgeführt wurde oder es einen Fehler bei der Ausführung gab. Im Fall der erfolgreichen Ausführung entspricht der Rückgabewert der vordefinierten Konstanten `CL_SUCCESS`. Anderenfalls wird ein Fehlerwert zurückgegeben. Die Bedeutung der vordefinierten Fehlerwerte ist in der OpenCL-Spezifikation [3] aufgelistet.
In den folgenden Abschnitten wird bei jedem Code-Beispiel ein Fehlercode abgerufen. Dieser wird jedoch nicht weiter auf eine korrekte Ausführung des Programms überprüft. In einem

Programm, das eine Fehlerbehandlung durchführt, müsste der Fehlercode mit dem Wert `CL_SUCCESS` verglichen werden. Im Falle eines Fehlers kann dann z.B. eine Meldung an den Benutzer ausgegeben und das Programm vorzeitig beendet werden.

2.3.3 Plattformen

Die Gesamtheit einer OpenCL-Laufzeitumgebung, bestehend aus Treibern, Rechenhardware und Laufzeitroutinen zum Zugriff auf die Hardware wird als Plattform bezeichnet. Auf einem Host-Rechner können mehrere Plattformen gleichzeitig installiert sein. Jede Plattform wird durch eine ID-Nummer vom Typ `cl_platform_id` repräsentiert. Um mit einer Plattform arbeiten zu können, muss zunächst bestimmt werden, welche Plattformen vorhanden sind. Der folgende Code benutzt die Funktion `clGetPlatformIDs` aus der Laufzeitbibliothek, um eine Liste aller verfügbaren Plattform-IDs abzufragen:

```
1  cl_uint n_plat;            // Anzahl der Plattformen
2  cl_platform_id* platforms; // Speicherplatz für Plattform-IDs
3  cl_int error;              // Fehlercode
4
5  // Bestimme Anzahl der verfügbaren Plattformen.
6  error = clGetPlatformIDs(0, 0, &n_plat);
7
8  // Reserviere Speicherplatz für Plattform-IDs.
9  platforms = (cl_platform_id*)malloc(n_plat * sizeof(cl_platform_id));
10
11 // Frage Liste der Plattform-IDs ab.
12 error = clGetPlatformIDs(n_plat, platforms, 0);
```

Um zu entscheiden, mit welcher Plattform gearbeitet werden soll, ist es sinnvoll, zunächst einige Informationen über die einzelnen Plattformen abzufragen. Dies ist mit der Funktion `clGetPlatformInfo` möglich, wie im folgenden Beispiel erläutert, das den Plattform-Namen für eine vorher festgelegte Plattform mit der ID `platform` abfragt:

```
1  cl_platform_id platform; // ID der abzufragenden Plattform
2  char* info;              // Speicherplatz für die Ausgabe
3  size_t length;           // Anzahl der ausgegebenen Zeichen
4  cl_int error;            // Fehlercode
5
6  // TODO: Wähle eine Plattform aus und speichere ihre ID in platform.
7
8  // Bestimme die Anzahl der Zeichen für die Ausgabe.
9  error = clGetPlatformInfo(platform, CL_PLATFORM_NAME, 0, 0, &length);
10
11 // Reserviere Speicher.
12 info = (char*)malloc(length);
13
14 // Lese den Plattformnamen ein.
15 error = clGetPlatformInfo(platform, CL_PLATFORM_NAME, length, info, 0);
```

Der zweite Parameter gibt an, welche Informationen ausgelesen werden sollen. Dabei sind u.a. folgende Konstanten vordefiniert:

- `CL_PLATFORM_NAME` gibt den Namen der Plattform zurück.
- `CL_PLATFORM_VENDOR` gibt den Hersteller der Plattform zurück.
- `CL_PLATFORM_VERSION` gibt zurück, welche OpenCL-Version von dieser Plattform unterstützt wird.

Die so erhaltenen Informationen können z.B. mittels `printf` ausgegeben werden.

2.3.4 Geräte

Jede Plattform kann eines oder mehrere Geräte umfassen, die die eigentliche Rechenhardware darstellen. Jedem Gerät ist eine ID vom Typ `cl_device_id` zugeordnet. Die zu einer Plattform zugehörigen Geräte lassen sich in gleicher Weise auflisten, wie es im vorhergehenden Abschnitt mit der Auflistung der Plattformen beschrieben wurde. Dafür dient die Funktion `clGetDeviceIDs`, wie im folgenden Code-Beispiel dargestellt:

```
1  cl_platform_id platform; // ID der abzufragenden Plattform
2  cl_uint n_dev;           // Anzahl der vorhandenen Geräte
3  cl_device_id* devices;   // Speicherplatz für Geräte-IDs
4  cl_int error;            // Fehlercode
5
6  // TODO: Wähle eine Plattform aus und speichere ihre ID in platform.
7
8  // Bestimme Anzahl der verfügbaren Geräte.
9  error = clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 0, 0, &n_dev);
10
11 // Reserviere Speicherplatz für Geräte-IDs.
12 devices = (cl_device_id*)malloc(n_dev * sizeof(cl_device_id));
13
14 // Frage Liste der Geräte-IDs ab.
15 error = clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, n_dev, devices, 0);
```

Das zweite Argument bestimmt, welche Geräte aufgelistet werden sollen. Folgende Konstanten sind vordefiniert:

- `CL_DEVICE_TYPE_ALL`: alle Geräte.
- `CL_DEVICE_TYPE_DEFAULT`: das Standard-Gerät.
- `CL_DEVICE_TYPE_CPU`: nur Host-CPUs.
- `CL_DEVICE_TYPE_GPU`: nur GPUs / Grafikgeräte.
- `CL_DEVICE_TYPE_ACCELERATOR`: nur eigens für OpenCL vorgesehene Recheneinheiten.

Mittels der Funktion `clGetDeviceInfo` können verschiedene Informationen über die einzelnen Geräte abgerufen werden. Der folgende Code ruft den Namen, die Verfügbarkeit und die globale Speichergröße eines Gerätes ab:

```
1  cl_device_id device; // ID des abzufragenden Gerätes
2  char* name;         // Speicherplatz für den Gerätenamen
3  size_t length;      // Länge des Gerätenamens
4  cl_bool available;  // Verfügbarkeit des Gerätes
5  cl_ulong memsize;   // Größe des globalen Speichers
6  cl_int error;       // Fehlercode
7
8  // TODO: Wähle ein Gerät aus und speichere die ID in device.
9
10 // Bestimme die Länge des Gerätenamens.
11 error = clGetDeviceInfo(device, CL_DEVICE_NAME, 0, 0, &length);
12
13 // Reserviere Speicher.
14 name = (char*)malloc(length);
15
16 // Lese den Gerätenamen ein.
17 error = clGetDeviceInfo(device, CL_DEVICE_NAME, length, name, 0);
18
19 // Lese Verfügbarkeit und Größe des globalen Speichers ein.
```

```
20 error = clGetDeviceInfo(device, CL_DEVICE_AVAILABLE,
21     sizeof(cl_bool), &available, 0);
22 error = clGetDeviceInfo(device, CL_DEVICE_GLOBAL_MEM_SIZE,
23     sizeof(cl_ulong), &memsize, 0);
```

Wie auch bei der Funktion `clGetPlatformInfo` gibt der zweite Parameter von `clGetDeviceInfo` an, welche Information abgerufen werden soll. Davon hängt zugleich ab, welchen Zeigertyp die Funktion als vierten Parameter erwartet. Die folgende Tabelle listet einige der vordefinierten Konstanten auf, die an `clGetDeviceInfo` übergeben werden können, sowie den Datentyp, der zurückgegeben wird:

- `CL_DEVICE_NAME` (`char[]`)
  Name des Gerätes.
- `CL_DEVICE_VENDOR` (`char[]`)
  Name des Herstellers.
- `CL_DEVICE_VERSION` (`char[]`)
  Unterstützte OpenCL-Version.
- `CL_DEVICE_EXTENSIONS` (`char[]`)
  Unterstützte Erweiterungen zu OpenCL.
- `CL_DEVICE_AVAILABLE` (`cl_bool`)
  Verfügbarkeit des Gerätes. Wenn der Wert `CL_TRUE` zurückgegeben wird, ist das Gerät verfügbar und kann für OpenCL-Berechnungen benutzt werden. Anderenfalls wird `CL_FALSE` zurückgegeben.
- `CL_DEVICE_TYPE` (`cl_device_type`)
  Der Gerätetyp wird zurückgegeben. Es handelt sich dabei um eine Konstante vom Typ `cl_device_type`, deren Werte bei der Funktion `clGetDeviceIDs` aufgelistet wurden.
- `CL_DEVICE_MAX_COMPUTE_UNITS` (`cl_uint`)
  Anzahl der auf dem Gerät verfügbaren Recheneinheiten. Da jede Recheneinheit zu jeder genau eine Kernel-Instanz ausführt, entspricht dies zugleich der maximale Anzahl an parallel aktiven Kernel-Instanzen.
- `CL_DEVICE_MAX_CLOCK_FREQUENCY` (`cl_uint`)
  Maximale Taktrate des Gerätes in MHz.
- `CL_DEVICE_ADDRESS_BITS` (`cl_uint`)
  Maximale Anzahl an Bits für eine Speicheradresse. Je nach Gerätetyp können dies 32 oder 64 Bits sein.
- `CL_DEVICE_GLOBAL_MEM_SIZE` (`cl_ulong`)
  Größe des globalen Device-RAMs in Bytes.
- `CL_DEVICE_LOCAL_MEM_SIZE` (`cl_ulong`)
  Größe des lokalen Device-RAMs pro Arbeitsgruppe in Bytes.
- `CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS` (`cl_uint`)
  Maximale Anzahl an Arbeitsgruppen-Dimensionen. Dies wird im Abschnitt 2.3.9 beim Aufruf der Funktion `clEnqueueNDRangeKernel` näher erläutert.
- `CL_DEVICE_MAX_WORK_ITEM_SIZES` (`size_t[]`)
  Maximale lokale Arbeitsgruppengröße in jeder Dimension.
- `CL_DEVICE_MAX_WORK_GROUP_SIZE` (`size_t`)
  Maximale lokale Arbeitsgruppengröße insgesamt.

Aus dieser Tabelle ist ersichtlich, dass die Funktion `clGetDeviceInfo`, je nach abgefragter Information, einen skalaren Datentyp oder ein Array zurückgibt. Es ist Aufgabe des Host-Programms, dafür zu sorgen, dass der als viertes Argument übergebene Zeiger diesen Datentyp aufnehmen kann und genügend Speicher für die zurückgegebene Information reserviert ist. Die Größe des reservierten Speichers ist als drittes Argument zu übergeben. Für skalare Datentypen geschieht dies am besten mit dem `sizeof` Operator, wie im Beispiel-Code gezeigt. Für Arrays ist die so bestimmte Größe des Datentyps noch mit der Anzahl der Elemente zu multiplizieren.

2.3.5 Kontexte

Eine Berechnung in OpenCL kann eines oder mehrere Geräte der selben Plattform benutzen. Die Geräte, die für eine gemeinsame Berechnung ausgewählt werden, bilden einen Kontext. Die Funktion `clCreateContext` dient dazu, einen neuen Kontext aus einem oder mehreren Geräten zu erstellen, wie im folgenden Beispiel dargestellt:

```
1  cl_context context;       // Speicherplatz für den Kontext
2  cl_platform_id platform; // Plattform für die Berechnung
3  cl_device_id* devices;   // Liste der zu verwendenden Geräte
4  cl_uint n_dev;           // Anzahl der zu verwendenden Geräte
5  cl_int error;            // Fehlercode
6
7  // TODO: Wähle eine Plattform aus und speichere die ID in platform.
8  // TODO: Bestimme die Anzahl n_dev der zu verwendenden Geräte.
9  // TODO: Erstelle ein Array devices von n_dev Geräte IDs.
10
11 // Speichere die Plattform-ID in ein Array von Typ cl_context_properties.
12 cl_context_properties cprops[3] = { CL_CONTEXT_PLATFORM,
13     (cl_context_properties)platform, 0 };
14
15 // Erstelle den Kontext.
16 context = clCreateContext(cprops, n_dev, devices, 0, 0, &error);
```

Der so erstellte Kontext und die in ihm zusammengefassten Geräte stehen nun für Berechnungen zur Verfügung. Speicher kann belegt und freigegeben werden, Kernels können kompiliert und ausgeführt werden. Dies wird in den folgenden Abschnitten erläutert. Wenn die Berechnung abgeschlossen ist und der Kontext nicht mehr benötigt wird, müssen die belegten Ressourcen mittels `clReleaseContext` wieder freigegeben werden, um für neue Berechnungen zur Verfügung zu stehen:

```
1  cl_context context; // Kontext zur Freigabe
2  cl_int error;       // Fehlercode
3
4  // TODO: Erstelle und verwende den Kontext context.
5
6  // Gebe Ressourcen frei und lösche den Kontext.
7  error = clReleaseContext(context);
```

Die Geräte können nun wieder für andere Aufgaben genutzt werden.

2.3.6 Speicherverwaltung

In Abschnitt 2.2.3 wurde die grundlegende Verwendung von Zeigervariablen in OpenCL erläutert, die auf Speicherabschnitte in unterschiedlichen Bereichen im Device-RAM zeigen. Diese Speicherabschnitte müssen vor ihrer Verwendung durch den Host-Code reserviert und anschließend wieder freigegeben werden. Zum Reservieren von Speicher dient die Funktion `clCreateBuffer`, die den neu reservierten Speicher zugleich mit Werten füllen kann. Ihre grundlegende Verwendung wird im folgenden Code-Beispiel gezeigt:

```
cl_context context;  // Kontext zum Reservieren des Speichers
cl_mem a_dev, b_dev; // Variablen für die OpenCL-Speicherbereiche
int* b_host;         // Zeiger auf ein Array vom Typ int
int b_count;         // Anzahl der Elemente im Array
cl_int error;        // Fehlercode

// TODO: Erstelle den Kontext context.

// Reserviere 0x1000 Bytes globalen Speicher zum Lesen und Schreiben.
a_mem = clCreateBuffer(context, CL_MEM_READ_WRITE, 0x1000, 0, &error);

// TODO: Erstelle ein Array b_host mit b_count Werten vom Typ int.

// Reserviere globalen Speicher und kopiere Daten von b_host.
b_mem = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
    sizeof(int) * b_count, b_host, &error);
```

Das zweite Argument ist vom Typ `cl_mem_flags` und bestimmt die Zugriffsrechte des OpenCL-Codes auf den neu reservierten Speicher sowie den Zusammenhang zum Host-Speicher. Mögliche Werte sind die folgenden vordefinierten Konstanten:

- `CL_MEM_READ_WRITE`: Der Speicher kann vom OpenCL-Code gelesen und geschrieben werden.
- `CL_MEM_READ_ONLY`: Der Speicher kann vom OpenCL-Code nur gelesen werden.
- `CL_MEM_WRITE_ONLY`: Der Speicher kann vom OpenCL-Code nur geschrieben werden.
- `CL_MEM_USE_HOST_PTR`: Statt neuen Speicher im Device-RAM zu reservieren, erwartet die Funktion `clCreateBuffer` als viertes Argument einen Zeiger auf einen Bereich im Host-RAM und erstellt eine Verknüpfung auf diesen Zeiger. OpenCL-Kernels, die diese Verknüpfung benutzen, operieren also auf dem Host-RAM.
- `CL_MEM_ALLOC_HOST_PTR`: Der Speicher wird in einem Bereich reserviert, auf den vom Host-Code aus zugegriffen werden kann (z.B. PCIe-Speicher).
- `CL_MEM_COPY_HOST_PTR`: Die Funktion `clCreateBuffer` reserviert einen Speicherbereich im Device-RAM und füllt diesen mit Werten aus dem Host-RAM. Dafür muss als viertes Argument ein Zeiger auf den Speicherbereich übergeben werden, von dem die Daten kopiert werden sollen.

Mehrere dieser Flags können durch logisches Oder verknüpft und gemeinsam verwendet werden. Dabei kann jedoch jeweils nur ein Wert für die Zugriffsrechte verwendet werden. Außerdem schließen sich die Werte `CL_MEM_USE_HOST_PTR` und `CL_MEM_COPY_HOST_PTR` gegenseitig aus. Der mit `clCreateBuffer` reservierte Speicher kann mit `clReleaseMemObject` wieder freigegeben werden:

```
cl_mem mem;     // Speicherbereich
cl_int error;   // Fehlercode

// TODO: Reserviere Speicherbereich mem.

// Gebe Speicherbereich wieder frei.
error = clReleaseMemObject(mem);
```

Zeiger auf diesen Speicherbereich verlieren damit ihre Gültigkeit. Der physikalische Speicher kann anschließend wieder neu reserviert werden.

2.3.7 Programme

Um auf unterschiedlichen Gerätearchitekturen ausführbar zu sein, kann der OpenCL-Code erst zur Laufzeit kompiliert werden. Daher ist ein OpenCL-Compiler Bestandteil der Laufzeitbibliothek. Um damit einen OpenCL-Quelltext zu kompilieren, muss dieser als Array vom Typ `char` im Arbeitsspeicher vorhanden sein. Dies lässt sich erreichen, indem der Quelltext entweder direkt in das fertige Binärprogramm eingebettet wird, oder durch Laden des Quelltextes aus einer ASCII-Datei. Das so erstellte Array kann dann mit der Funktion `clCreateProgramWithSource` an die OpenCL-Laufzeitbibliothek übergeben und mit der Funktion `clBuildProgram` kompiliert werden, wie im folgenden Beispiel gezeigt:

```
1  cl_program program;    // Speicherplatz für die Programm-ID
2  cl_context context;    // Kontext zum Kompilieren des Programms
3  cl_device_id* devices; // Geräte zur Ausführung des kompilierten Programms
4  cl_uint n_dev;         // Anzahl der zu verwendenden Geräte
5  char* sourcecode;      // Quelltext des zu kompilierenden Programms
6  size_t length;         // Länge des Quelltextes
7  cl_int error;          // Fehlercode
8
9  // TODO: Erstelle einen Kontext context zur Ausführung des Programms.
10 // TODO: Lade das Programm als char-Array sourcecode in den Arbeitsspeicher.
11 // TODO: Setze length auf die Länge des Quelltextes.
12
13 // Erstelle ein neues Programm aus dem Quelltext.
14 program = clCreateProgramWithSource(context, 1, &sourcecode, &length, &error);
15
16 // TODO: Wähle n_dev Geräte devices zur Ausführung des Programms.
17
18 // Kompiliere das Programm für dieses Gerät.
19 error = clBuildProgram(program, n_dev, devices, 0, 0, 0);
```

Der Rückgabecode von `clBuildProgram` gibt an, ob der Quelltext erfolgreich kompiliert wurde. Weitere Informationen in Form von Fehlermeldungen oder Compiler-Warnungen lassen sich in Textform über die Funktion `clGetProgramBuildInfo` abrufen. Das folgende Programm zeigt, wie ein solches Compiler-Log abgerufen wird:

```
1  cl_program program;  // Programm zur Abfrage des Compiler-Logs
2  cl_device_id device; // Gerät, auf dem kompiliert wurde
3  char* log;           // Speicherplatz für das Compiler-Log
4  size_t length;       // Länge des Compiler-Logs
5  cl_int error;        // Fehlercode
6
7  // TODO: Erstelle ein neues Programm program und kompiliere es.
8  // TODO: Wähle eines der Geräte aus, für die kompiliert wurde.
9
10 // Lese die Länge des Compiler-Logs ein.
11 error = clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG,
12     0, 0, &length);
13
14 // Reserviere Speicher.
15 log = (char*)malloc(length);
16
17 // Lese das Compiler-Log ein.
18 error = clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG,
19     length, log, 0);
```

Wenn das Programm erfolgreich kompiliert wurde, liegt der Programmcode in binärer Form im Speicher vor und kann weiter verarbeitet werden. Wenn das Programm ausgeführt wurde

und nicht mehr im Speicher benötigt wird, können die belegten Ressourcen mit `clReleaseProgram` wieder freigegeben werden:

```
1 cl_program program; // Programm zur Freigabe
2 cl_int error;       // Fehlercode
3
4 // TODO: Erstelle ein neues Programm program und kompiliere es.
5
6 // Gebe Ressourcen wieder frei.
7 error = clReleaseProgram(program);
```

Die belegten Ressourcen stehen nun wieder für andere Aufgaben zur Verfügung.

2.3.8 Kernels

Wie in Abschnitt 2.2.1 beschrieben dienen die Kernel-Funktionen als Funktionen innerhalb des OpenCL-Codes, die von außerhalb aufgerufen werden können. Wie dies im Detail geschieht wird im folgenden Abschnitt erläutert. Als Beispiel dient der folgende OpenCL-Code, der zwei Kernels sowie ihre Argumente definiert:

```
1 __kernel void Kernel1(float x, __global float4* y)
2 {
3     // TODO: Führe Berechnung durch.
4 }
5
6 __kernel void Kernel2(int a, int4 b, __global int8* c)
7 {
8     // TODO: Führe Berechnung durch.
9 }
```

Damit diese beiden Kernels ausgeführt werden können, muss der Quelltext zunächst zu einem Programm kompiliert werden, wie im vorherigen Abschnitt beschrieben. Mit der Funktion `clCreateKernel` können anschließend die beiden Kernels extrahiert werden, wie in folgendem Beispiel gezeigt:

```
1 cl_program program;         // Programm (aus OpenCL-Quelltext kompiliert)
2 cl_kernel kernel1, kernel2; // Speicherplatz für die beiden Kernel
3 cl_int error;               // Fehlercode
4
5 // TODO: Lade und kompiliere ein Programm program.
6
7 // Suche die beiden Kernels aus dem Programm heraus.
8 kernel1 = clCreateKernel(program, "Kernel1", &error);
9 kernel2 = clCreateKernel(program, "Kernel2", &error);
```

Damit sind der Laufzeitumgebung die Einsprungpunkte für die beiden Kernels bekannt. Als nächstes müssen nun noch die Argumente der beiden Kernels mit Werten gefüllt werden. Dies geschieht mit der Funktion `clSetKernelArg`. Dabei muss der Datentyp jedes Arguments in der Deklaration des Kernels mit dem übergebenen Datentyp im Host-Code übereinstimmen. OpenCL-Vektordatentypen wie `int4` werden dabei durch die entsprechenden Datentypen wie `cl_int4` im Host-Code dargestellt. Zeigervariablen in OpenCL werden durch Speicherreferenzen vom Typ `cl_mem` im Host-Code dargestellt. Das folgende Beispiel verdeutlicht dies für die beiden obigen Kernels:

```
1 cl_kernel kernel1, kernel2; // Speicherplatz für die beiden Kernel
2 cl_int error;               // Fehlercode
3 int a;                      // Argument vom Typ int
4 cl_int4 b;                  // Argument vom Typ (cl_)int4
5 float x;                    // Argument vom Typ float
```

```
6  cl_mem c_dev, y_dev;        // Zeigerargumente
7
8  // TODO: Extrahiere zwei Kernel kernel1, kernel2.
9
10 // Setze die Argumente für Kernel1.
11 error = clSetKernelArg(kernel1, 0, sizeof(float), &x);
12 error = clSetKernelArg(kernel1, 1, sizeof(cl_mem), &y_dev);
13
14 // Setze die Argumente für Kernel2.
15 error = clSetKernelArg(kernel2, 0, sizeof(int), &a);
16 error = clSetKernelArg(kernel2, 1, sizeof(cl_int4), &b);
17 error = clSetKernelArg(kernel2, 2, sizeof(cl_mem), &c_dev);
```

Die beiden Kernels sind nun bereit zur Ausführung. Wie die Ausführung gestartet wird, wird im nächsten Abschnitt beschrieben. Wenn keine weiteren Berechnungen ausgeführt werden sollen, muss jeder Kernel durch einen Aufruf der Funktion `clReleaseKernel` wieder freigegeben werden, wie in folgendem Beispiel:

```
1  cl_kernel kernel1, kernel2; // Speicherplatz für die beiden Kernel
2  cl_int error;               // Fehlercode
3
4  // TODO: Führe Berechnung mit Kernel1 und / oder Kernel2 durch.
5
6  // Gebe Ressourcen für die beiden Kernel frei.
7  error = clReleaseKernel(kernel1);
8  error = clReleaseKernel(kernel2);
```

Damit stehen die durch die Kernels belegten Ressourcen wieder zur Verfügung.

2.3.9 Warteschlangen

Warteschlangen dienen dazu, Aufgaben an OpenCL-Geräte zu verteilen. Jede Aufgabe, wie das Ausführen eines Kernels oder ein Datentransfer zwischen Host-RAM und Device-RAM, muss in eine Warteschlange eingefügt werden. Dafür muss zunächst eine Warteschlange erstellt werden, die einem Gerät zugewiesen ist. Dies wird mit der Funktion `clCreateCommandQueue` erreicht, die im folgenden Code-Beispiel gezeigt wird:

```
1  cl_context context;     // Kontext für die Warteschlange
2  cl_device_id device;    // ID des zu verwendenden Geräts
3  cl_command_queue queue; // Speicherplatz für die Warteschlange
4  cl_int error;           // Fehlercode
5
6  // TODO: Wähle ein Gerät device aus und erzeuge einen Kontext context.
7
8  // Erzeuge eine neue Warteschlange.
9  queue = clCreateCommandQueue(context, device, 0, &error);
```

Die so erzeugte Warteschlange ist einem bestimmten Gerät zugeordnet und ermöglicht es, Berechnungen auf diesem Gerät auszuführen. Dazu dient die Funktion `clEnqueueNDRangeKernel`, die einen Kernel zur Warteschlange hinzufügt und ihm zugleich einen N-dimensionalen Datenbereich übergibt. Die im OpenCL-Code programmierte Kernel-Funktion wird dann für jedes Element dieses Datenbereichs einmal aufgerufen. Die Dimension und Größe des Datenbereichs muss dafür ebenfalls an die Funktion `clEnqueueNDRangeKernel` übergeben werden, wie im folgenden Beispiel eines zweidimensionalen Datenbereichs gezeigt:

```
1  cl_command_queue queue;     // Warteschlange
2  cl_kernel kernel;           // Kernel für die Berechnung
3  cl_int error;               // Fehlercode
4
```

```
5  // TODO: Kompiliere ein Programm und erstelle einen Kernel kernel.
6  // TODO: Setze die Argumente für den Aufruf des Kernels.
7  // TODO: Erzeuge eine Warteschlange queue zur Ausführung des Kernels.
8
9  // Lege die Dimension des Datenbereichs fest.
10 cl_uint dim = 2;
11
12 // Lokale Arbeitsgruppengröße = Anzahl der Datenelemente in einem Block.
13 size_t local[] = {16, 16};
14
15 // Globale Arbeitsgruppengröße = Anzahl der Datenelemente insgesamt.
16 size_t global[] = {1024, 1024};
17
18 // Führe den Kernel auf 1024 * 1024 Datenelementen aus.
19 error = clEnqueueNDRangeKernel(queue, kernel, dim, 0, global, local, 0, 0, 0);
```

Der Kernel wird nun ausgeführt, sobald die für die Berechnung notwendigen Hardware-Ressourcen verfügbar sind. Je nach Hardware-Kapazität können dabei mehrere Instanzen des gleichen Kernels oder verschiedener Kernels gleichzeitig ausgeführt werden. In ähnlicher Weise lassen sich auch Datentransfers in eine Warteschlange einführen. Das folgende Beispiel benutzt die Funktionen `clEnqueueReadBuffer` und `clEnqueueWriteBuffer`, um in einen Bereich des Device-RAMs zu schreiben und aus einem anderen zu lesen:

```
1  int input_count;         // Anzahl der Eingabe-Daten
2  int output_count;        // Anzahl der Ausgabe-Daten
3  float* input_host;       // Eingabe-Daten im Host-RAM
4  float* output_host;      // Ausgabe-Daten im Host-RAM
5  cl_mem input_dev;        // Eingabe-Daten im Device-RAM
6  cl_mem output_dev;       // Ausgabe-Daten im Device-RAM
7  cl_command_queue queue; // Warteschlange
8  cl_int error;            // Fehlercode
9
10 // TODO: Bestimme Anzahl der Eingabe- und Ausgabe-Daten
11 // TODO: Reserviere Speicher im Host-RAM für Eingabe und Ausgabe.
12 // TODO: Reserviere Speicher im Device-RAM für Eingabe und Ausgabe.
13 // TODO: Fülle Eingabe-Speicher im Host-RAM mit Eingabe-Daten.
14 // TODO: Erzeuge eine Warteschlange queue.
15
16 // Kopiere Daten vom Host-RAM in den Device-RAM.
17 error = clEnqueueWriteBuffer(queue, input_dev, CL_TRUE, 0,
18     sizeof(float) * input_count, input_host, 0, 0, 0);
19
20 // TODO: Führe eine Berechnung durch.
21
22 // Kopiere Daten vom Device-RAM in den Host-RAM.
23 error = clEnqueueReadBuffer(queue, output_dev, CL_TRUE, 0,
24     sizeof(float) * output_count, output_host, 0, 0, 0);
```

Die Argumente zum Aufruf meiner Funktionen sind identisch. Sie unterscheiden sich nur durch die Richtung des Datentransfers. Das dritte Argument gibt ab, ob der Aufruf der Funktion blockierend oder nicht-blockierend sein soll. Dabei haben die übergebenen Werte die folgende Bedeutung:

- `CL_TRUE`: Der Aufruf ist blockierend. Die Funktion kehrt erst zurück, wenn der Datentransfer abgeschlossen ist.
- `CL_FALSE`: Der Aufruf ist nicht-blockierend. Die Funktion kehrt sofort zurück und der Datentransfer wird im Hintergrund ausgeführt.

Wenn keine weiteren Aufgaben ausgeführt werden sollen, muss die Warteschlange mit `clReleaseCommandQueue` wieder freigegeben werden, wie im folgenden Beispiel gezeigt:

```
1  cl_command_queue queue; // Warteschlange
2  cl_int error;           // Fehlercode
3
4  // TODO: Erzeuge Warteschlange queue und führe Aufgaben damit aus.
5
6  // Gebe Warteschlange wieder frei.
7  error = clReleaseCommandQueue(queue);
```

Die durch die Warteschlange belegten Ressourcen werden freigegeben, sobald sämtliche Aufgaben abgearbeitet wurden.

2.3.10 Ereignisse

Die im vorhergehenden Abschnitt beschriebenen Warteschlangen ermöglichen es, Aufgaben auszuführen, sobald die dafür notwendigen Hardware-Ressourcen verfügbar sind. Je nach Hardware-Kapazität können dabei auch mehrere Aufgaben gleichzeitig ausgeführt werden. Häufig ist es jedoch notwendig, dass eine Aufgabe komplett bearbeitet wurde, bevor mit der nächsten Aufgabe begonnen wird. Um die korrekte Hintereinanderausführung von Aufgaben zu gewährleisten, müssen Synchronisationspunkte eingeführt werden. Diese Synchronisationspunkte werden in OpenCL als Ereignisse bezeichnet. Die im vorherigen Abschnitt genannten Funktionen `clEnqueueNDRangeKernel`, `clEnqueueReadBuffer` und `clEnqueueWriteBuffer` erlauben es, vor dem Beginn ihrer Ausführung auf ein Ereignis zu warten und nach dem Ende ihrer Ausführung ein Ereignis auszulösen. Dies wird im folgenden Beispiel erläutert. Darin werden zwei Datensätze nicht-blockierend in den Device-RAM kopiert und eine Berechnung gestartet, sobald beide Kopiervorgänge abgeschlossen sind. Nach Fertigstellung der Berechnung werden die Ergebnisse blockierend in den Host-RAM kopiert, damit das Programm erst nach Abschluss der Kopiervorgangs fortgesetzt wird.

```
1  int in1_count;          // Anzahl der Eingabe-Daten 1
2  int in2_count;          // Anzahl der Eingabe-Daten 2
3  int out_count;          // Anzahl der Ausgabe-Daten
4  float* in1_host;        // Eingabe-Daten 1 im Host-RAM
5  float* in2_host;        // Eingabe-Daten 2 im Host-RAM
6  float* out_host;        // Ausgabe-Daten im Host-RAM
7  cl_mem in1_dev;         // Eingabe-Daten 1 im Device-RAM
8  cl_mem in2_dev;         // Eingabe-Daten 2 im Device-RAM
9  cl_mem out_dev;         // Ausgabe-Daten im Device-RAM
10 int dim;                // Dimension des Rechenbereichs
11 size_t local[];         // Lokale Arbeitsgruppengröße
12 size_t global[];        // Globale Arbeitsgruppengröße
13 cl_command_queue queue; // Warteschlange
14 cl_kernel kernel;       // Kernel für die Berechnung
15 cl_event load[2];       // Ereignisse beim Laden der Eingabe-Daten
16 cl_event compute;       // Ereignis bei Abschluss der Berechnung
17 cl_int error;           // Fehlercode
18
19 // TODO: Bestimme Anzahl der Eingabe- und Ausgabe-Daten
20 // TODO: Reserviere Speicher im Host-RAM für Eingabe und Ausgabe.
21 // TODO: Reserviere Speicher im Device-RAM für Eingabe und Ausgabe.
22 // TODO: Fülle Eingabe-Speicher im Host-RAM mit Eingabe-Daten.
23 // TODO: Lade und kompiliere einen Kernel kernel.
24 // TODO: Setze die Argumente für den Aufruf des Kernels.
25 // TODO: Lege die Dimension und Größe des Rechenbereichs fest.
26 // TODO: Erzeuge eine Warteschlange queue.
```

```
27
28 // Lese Daten ein und erzeuge zwei Ereignisse.
29 error = clEnqueueWriteBuffer(queue, in1_dev, CL_FALSE, 0,
30     sizeof(float) * in1_count, in1_host, 0, 0, &load[0]);
31 error = clEnqueueWriteBuffer(queue, in2_dev, CL_FALSE, 0,
32     sizeof(float) * in2_count, in2_host, 0, 0, &load[1]);
33
34 // Warte auf die Daten, rechne und erzeuge ein weiteres Ereignis.
35 error = clEnqueueNDRangeKernel(queue, kernel, dim, 0,
36     global, local, 2, load, &compute);
37
38 // Warte auf die Berechnung und lese Daten aus.
39 error = clEnqueueReadBuffer(queue, out_dev, CL_TRUE, 0,
40     sizeof(float) * out_count, out_host, 1, &compute, 0);
```

Gelegentlich ist es erforderlich, dass nicht eine Aufgabe in einer Warteschlange, sondern die Ausführung des Host-Programms auf das Eintreten eines Ereignisses (z.B. das Kopieren von Daten) wartet. Dies kann mit der Funktion `clWaitForEvents` erreicht werden, die die Ausführung so lange anhält, bis eines oder mehrere Ereignisse eingetreten sind. Das folgende Beispiel verdeutlicht ihre Verwendung:

```
1 cl_event event[2]; // Ereignisse, auf die gewartet werden soll
2 cl_int error;      // Fehlercode
3
4 // TODO: Starte Aufgaben, die zwei Ereignisse auslösen.
5
6 // Warte, bis beide Ereignisse eingetreten sind.
7 error = clWaitForEvents(2, event);
```

Statt auf das Eintreten eines Ereignisses zu warten und so lange untätig zu sein, kann es auch sinnvoll sein, den Status eines Ereignisses abzufragen und andere Dinge zu erledigen, bis das Ereignis eingetreten ist. Die Funktion `clGetEventInfo` ruft Informationen über ein Ereignis ab, darunter auch seinen Ausführungsstatus:

```
1 cl_event event; // Ereignis, auf das gewartet wird
2 cl_int status;  // Ausführungsstatus des Ereignisses
3 cl_int error;   // Fehlercode
4
5 // Frage Information über das Ereignis ab.
6 error = clGetEventInfo(event, CL_EVENT_COMMAND_EXECUTION_STATUS,
7     sizeof(cl_int), &status, 0);
```

Der Rückgabewert kann dabei eine der folgenden Konstanten annehmen:

- `CL_QUEUED`: Auftrag ist in der Warteschlange.
- `CL_SUBMITTED`: Auftrag wurde ans Gerät übermittelt:
- `CL_RUNNING`: Auftrag wird auf dem Gerät ausgeführt.
- `CL_COMPLETE`: Auftrag wurde bearbeitet.

Wenn bei der Ausführung des Auftrages, der das Ereignis auslöst, ein Fehler aufgetreten ist, wird ein negativer Wert zurückgegeben.

2.3.11 Marker und Barrieren

Eine etwas einfachere Form der Synchronisation lässt sich mit Markern und Barrieren erreichen. Beide stellen einen Synchronisationspunkt innerhalb einer Befehlswarteschlange dar. Bei einem Marker handelt es sich um eine Markierung innerhalb einer Warteschlange, die ein

Ereignis auslöst, sobald sämtliche Aufträge abgeschlossen sind, die vor dem Marker in die Warteschlange eingefügt wurden. Ein Marker lässt sich mit der Funktion `clEnqueueMarker` in eine Warteschlange einfügen, wie im folgenden Beispiel gezeigt:

```
1  cl_command_queue queue; // Warteschlange
2  cl_event marker;        // Marker-Ereignis
3  cl_int error;           // Fehlercode
4
5  // TODO: Erzeuge Warteschlange queue.
6  // TODO: Füge Aufgaben in die Warteschlange ein, auf die gewartet wird.
7
8  // Füge Marker in die Warteschlange ein.
9  error = clEnqueueMarker(queue, &marker);
```

Als logisches Gegenstück zum Marker, der das Warten auf alle vorher in die Warteschlange eingefügten Aufgaben ermöglicht, gibt es auch die Möglichkeit, auf eines oder mehrere Ereignisse zu warten, bevor mit der Ausführung der später eingefügten Aufgaben begonnen wird. Dies geschieht mit der Funktion `clEnqueueWaitForEvents`, wie im folgenden Beispiel:

```
1  cl_command_queue queue; // Warteschlange
2  cl_event* events;       // Ereignisse, aus die gewartet wird
3  int n_events;           // Anzahl der Ereignisse
4  cl_int error;           // Fehlercode
5
6  // TODO: Erzeuge Warteschlange queue.
7  // TODO: Führe Aufgaben aus, um die Ereignisse events auszulösen.
8
9  // Füge Wartebedingung in die Warteschlange ein.
10 error = clEnqueueWaitForEvents(queue, n_events, events);
11
12 // TODO: Füge weitere Aufgaben in die Warteschlange ein.
```

Beide Funktionen lassen sich kombinieren. Es kann also zunächst mittels `clEnqueueMarker` ein Marker eingefügt werden, auf den anschließend mit `clEnqueueWaitForEvents` gewartet wird. Als resultierender Effekt werden sämtliche Aufgaben, die vor dem Marker in die Befehlswarteschlange eingefügt wurden, vollständig abgeschlossen, bevor mit der Ausführung der Aufgaben begonnen wird, die nach dem Wartepunkt in die Warteschlange eingefügt werden. Da es sich dabei um eine häufig auftretende Synchronisationsaufgabe handelt, gibt es eine Funktion, die diese beiden Funktionen vereint und die ohne die explizite Deklaration einer Ereignis-Variablen auskommt. Es handelt sich dabei um die Funktion `clEnqueueBarrier`:

```
1  cl_command_queue queue; // Warteschlange
2  cl_int error;           // Fehlercode
3
4  // TODO: Erzeuge Warteschlange queue.
5  // TODO: Füge Aufgaben in die Warteschlange ein, auf die gewartet wird.
6
7  // Füge Barriere in die Warteschlange ein.
8  error = clEnqueueBarrier(queue);
9
10 // TODO: Füge weitere Aufgaben in die Warteschlange ein.
```

Diese Form der Synchronisation ist immer dann sinnvoll, wenn zwei aufeinander folgende Aufgaben die gleichen Daten benutzen und daher nicht gleichzeitig darauf zugreifen können. Auch der Host-Code kann auf die Ausführung der Aufgaben in einer Warteschlange warten. Für diesen Zweck gibt es die Funktionen `clFlush` und `clFinish`, die wie folgt benutzt werden können:

```
1  cl_command_queue queue; // Warteschlange
```

```
2  cl_int error;           // Fehlercode
3
4  // TODO: Erzeuge Warteschlange queue.
5  // TODO: Füge Aufgaben in die Warteschlange ein.
6
7  // Warte, bis alle Aufgaben an das Gerät übermittelt wurden.
8  error = clFlush(queue);
9
10 // Alternative: Warte, bis alle Aufgaben abgearbeitet wurden.
11 error = clFinish(queue);
```

Die Funktion `clFlush` kehrt zurück, sobald alle Aufgaben, die vor ihrem Aufruf in die Warteschlange eingefügt wurden, an das zugehörige Gerät übermittelt, aber noch nicht notwendigerweise ausgeführt wurden. Die Funktion `clFinish` dagegen wartet, bis die Aufgaben abgeschlossen sind.

Kapitel 3

Die OpenCL-Umgebung

Um mit OpenCL arbeiten zu können, ist eine OpenCL-Umgebung erforderlich. Darunter versteht man die Gesamtheit von ausführbaren Programmen, wie Compiler und Linker, eine Laufzeitbibliothek, Header-Dateien und Treiber für die GPU, auf der gerechnet werden soll. Die Benutzung der OpenCL-Umgebung wird in diesem Kapitel beschrieben.

3.1 Kompilieren und Ausführen eines Programms

In diesem Abschnitt wird beschrieben, wie ein OpenCL-Programm kompiliert und ausgeführt wird. Im Folgenden sei dafür ein einfaches Beispielprogramm betrachtet, das aus zwei Quelltext-Dateien besteht - dem Host-Code in C in der Datei `main.c` und dem Device-Code in OpenCL in der Datei `ocl.cl`.

3.1.1 Kompilieren des Programms

Um aus dem Quelltext ein ausführbares Programm zu erhalten, muss der Host-Code kompiliert werden. Wie bei jedem C-Quelltext wird dafür ein C-Compiler benutzt. Im Falle eines OpenCL-Programms muss zusätzlich die OpenCL-Laufzeitbibliothek eingebunden werden. Diese besteht aus einer Header-Datei, wie in Abschnitt 2.3.1 beschrieben, und dem Bibliothekscode. Damit der C-Compiler die Laufzeitbibliothek findet, müssen ihm die Pfade zum Einbinden dieser Dateien übergeben werden. Dafür bietet es sich zunächst an, diese in zwei Shell-Variablen zu speichern:

```
export OCLLIBPATH=/path/to/opencl/lib
export OCLINCPATH=/path/to/opencl/include
```

Anschließend können diese Variablen beim Aufruf des Compilers benutzt werden. Bei Benutzung des GNU C-Compilers `gcc` lautet der Aufruf:

```
gcc -I$OCLINCPATH -L$OCLLIBPATH -lOpenCL -o program main.c
```

Dieser Aufruf kompiliert die Datei main.c und linkt sie anschließend mit der OpenCL-Laufzeitbibliothek zu einem ausführbaren Programm, das in der Binärdatei `program` abgespeichert wird. Zur Ausführung des Programms ist zusätzlich die OpenCL-Datei `ocl.cl` notwendig, in der sich der Device-Code befindet. Diese Datei muss zur Laufzeit in den Arbeitsspeicher geladen und kompiliert werden. Alternativ gibt es aber auch die Möglichkeit, den OpenCL-Quelltext direkt in die kompilierte Binärdatei einzubinden. Dies wird im nächsten Abschnitt beschrieben.

3.1.2 Einbetten des OpenCL-Codes

Im vorherigen Abschnitt wurde der OpenCL-Code in einer separaten Datei gehalten, die zur Laufzeit in den Arbeitsspeicher eingelesen wird. Um die Ausführung zu vereinfachen, kann man den OpenCL-Code direkt in die kompilierte Binärdatei einbinden. Dafür muss aus der

OpenCL-Datei `ocl.cl` zunächst eine Objekt-Datei erzeugt werden. Das folgende Beispiel benutzt den Linker `ld` der GNU Binutils für diese Aufgabe:

```
ld -r -b binary -o ocl.o ocl.cl
```

Das Ergebnis ist eine Objekt-Datei, die vom Linker in das kompilierte Programm eingebunden werden kann. Um vom Host-Code aus darauf zuzugreifen, definiert `ld` mehrere Symbole in der Objekt-Datei. Diese können im Host-Code genutzt werden, wie in folgendem Beispiel gezeigt:

```
1  extern char _binary_ocl_cl_start[]; // Beginn des OpenCL-Quelltextes
2  extern char _binary_ocl_cl_end[];   // Ende des OpenCL-Quelltextes
3
4  void Compile(void)
5  {
6      cl_program program;    // Speicherplatz für die Programm-ID
7      cl_context context;    // Kontext zum Kompilieren des Programms
8      cl_device_id* devices; // Geräte zur Ausführung des kompilierten Programms
9      cl_uint n_dev;         // Anzahl der zu verwendenden Geräte
10     char* source;          // Anfangsadresse des Quelltextes
11     size_t length;         // Länge des Quelltextes
12     cl_int error;          // Fehlercode
13
14     // TODO: Erstelle einen Kontext context zur Ausführung des Programms.
15
16     // Bestimme Anfang und Länge des OpenCL-Quelltextes.
17     source = _binary_ocl_cl_start;
18     length = (size_t)(_binary_ocl_cl_end - _binary_ocl_cl_start);
19
20     // Erstelle ein neues Programm aus dem Quelltext.
21     program = clCreateProgramWithSource(context, 1, &source, &length, &error);
22
23     // TODO: Wähle n_dev Geräte devices zur Ausführung des Programms.
24
25     // Kompiliere das Programm für dieses Gerät.
26     error = clBuildProgram(program, n_dev, devices, 0, 0, 0);
27 }
```

Beim Linken des Host-Codes muss nun die Objekt-Datei `ocl.o` eingebunden werden. Beim GNU C-Compiler kann dies mit folgendem Aufruf erreicht werden:

```
gcc -I$OCLINCPATH -L$OCLLIBPATH -lOpenCL -o program main.c ocl.o
```

Die so erstellte Binärdatei `program` enthält nun den OpenCL-Quelltext und kann daher auch ohne die Datei `ocl.cl` benutzt werden.

3.1.3 Ausführen des Programms

Um die Größe des ausführbaren Programms gering zu halten, wird die OpenCL-Laufzeitbibliothek nicht statisch, sondern dynamisch eingebunden, d.h. in einer separaten Datei gehalten. Diese Datei wird erst bei der Programmausführung vom dynamischen Linker geladen. Damit die Laufzeitbibliothek vom dynamischen Linker gefunden werden kann, muss das Verzeichnis, in dem sich die Datei befindet, zur Variable `LD_LIBRARY_PATH` hinzugefügt werden. Der Inhalt dieser Variablen lässt sich mittels

```
echo $LD_LIBRARY_PATH
```

anzeigen. Wenn die Variable leer ist, kann der korrekte Pfad mit

```
export LD_LIBRARY_PATH=$OCLLIBPATH
```

gesetzt werden. Ansonsten muss er durch einen Doppelpunkt getrennt hinzugefügt werden:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$OCLLIBPATH
```

Nun kann das eigentliche Programm durch einen Aufruf von

```
./program
```

ausgeführt werden.

3.2 OpenCL-Distributionen

Die im vorherigen Abschnitt genannten Verzeichnisse für die OpenCL-Laufzeitbibliothek und die zugehörigen Header-Dateien hängen von der installierten OpenCL-Umgebung und den Installationspfaden ab. Auf den Praktikumsrechnern sind zwei verschiedene OpenCL-Distributionen installiert. Welche Pfade vor dem Kompilieren und Ausführen eines Programms eingestellt werden müssen, hängt davon ab, welche der Distributionen genutzt werden soll.

3.2.1 NVidia

Um NVidia-Grafikkarten und insbesondere die Tesla-Serie zu verwenden, muss die von NVidia bereitgestellte OpenCL-Distribution benutzt werden. Dafür müssen die im vorhergehenden Abschnitt genannten Shell-Variablen `OCLINCPATH` und `OCLLIBPATH` auf die Werte

```
export OCLLIBPATH=/usr/lib/nvidia-current
export OCLINCPATH=/usr/local/cuda/include
```

gesetzt werden.

3.2.2 AMD

Mit der OpenCL-Distribution von AMD lassen sich nicht nur GPUs der Radeon-Serie benutzen, sondern auch CPUs mit der x86 / x86_64-Architektur, die mindestens über den SSE3-Befehlssatz verfügen. Auf 32-Bit-Systemen sind dafür die Shell-Variablen auf die Werte

```
export OCLLIBPATH=/opt/AMD-APP-SDK-v2.4-lnx64/lib/x86
export OCLINCPATH=/opt/AMD-APP-SDK-v2.4-lnx64/include
```

zu setzen, während auf 64-Bit-Systemen die Pfade

```
export OCLLIBPATH=/opt/AMD-APP-SDK-v2.4-lnx64/lib/x86_64
export OCLINCPATH=/opt/AMD-APP-SDK-v2.4-lnx64/include
```

gesetzt werden müssen.

Kapitel 4

Aufgaben

Im Laufe der Praktikumswoche sollen einige Programmieraufgaben gelöst werden, die in diesem Kapitel erläutert werden. Ziel ist es jeweils, ein vollständiges, lauffähiges Programm zu schreiben und einige Male mit unterschiedlichen Eingabe-Daten auszuführen. Dabei kann das Programm schrittweise optimiert werden. Dokumentieren Sie, welche Änderungen Sie am Code vornehmen und welche Auswirkungen diese Änderungen haben.

4.1 Übungsaufgaben

Um mit der OpenCL-Umgebung vertraut zu werden, sollen zunächst einige einfache Übungsaufgaben gelöst werden. Die in diesem Abschnitt vorgestellten Aufgaben müssen nicht exakt wie vorgegeben bearbeitet werden. Stattdessen sollen die Code-Beispiele als Anregung und Hilfe für eigene Programme dienen.

4.1.1 Anzeigen der Plattformen und Geräte

Bevor eine Berechnung durchgeführt wird, ist es sinnvoll, die zur Verfügung stehende Hardware und ihre Rechenkapazitäten zu erkunden. Ziel dieser Aufgabe ist es, ein Programm zu schreiben, das die vorhandenen OpenCL-Plattformen und Geräte auflistet und ihre Eigenschaften anzeigt. Dafür ist kein OpenCL-Kernel erforderlich, sondern ausschließlich ein Host-Programm, das auf die Funktionen der Laufzeitbibliothek zugreift. Die wichtigsten Funktionen, die dafür nötig sind, wurden bereits in den Abschnitten 2.3.3 und 2.3.4 erläutert und sollen nun in der Praxis angewandt werden. Als Grundlage soll der folgende Beispiel-Code dienen:

```
1  #include <CL/cl.h>
2  #include <stdio.h>
3
4  int main(int argc, char** argv)
5  {
6      cl_uint n_plat;            // Anzahl der Plattformen
7      cl_uint n_dev;             // Anzahl der Geräte
8      cl_platform_id* platforms; // Speicherplatz für Plattform-IDs
9      cl_device_id* devices;     // Speicherplatz für Geräte-IDs
10     cl_int error;              // Fehlercode
11     int i, j;                  // Schleifenzähler
12     char* info;                // Speicherplatz für die Ausgabe
13     size_t length;             // Länge der Ausgabe
14
15     // Frage die vorhandenen Plattformen ab.
16     error = clGetPlatformIDs(0, 0, &n_plat);
17     printf("Anzahl der Plattformen: %d\n", n_plat);
```

```
18      platforms = (cl_platform_id*)malloc(n_plat * sizeof(cl_platform_id));
19      error = clGetPlatformIDs(n_plat, platforms, 0);
20
21      // Iteriere über alle Plattformen.
22      for(i = 0; i < n_plat; i++)
23      {
24          // Frage den Namen der Plattform ab.
25          error = clGetPlatformInfo(platforms[i], CL_PLATFORM_NAME,
26              0, 0, &length);
27          info = (char*)malloc(length);
28          error = clGetPlatformInfo(platforms[i], CL_PLATFORM_NAME,
29              length, info, 0);
30          printf("CL_PLATFORM_NAME: %s\n", info);
31          free(info);
32
33          // Frage die vorhandenen Geräte ab.
34          error = clGetDeviceIDs(platforms[i], CL_DEVICE_TYPE_ALL,
35              0, 0, &n_dev);
36          printf("Anzahl der Geräte: %d\n", n_dev);
37          devices = (cl_device_id*)malloc(n_dev * sizeof(cl_device_id));
38          error = clGetDeviceIDs(platforms[i], CL_DEVICE_TYPE_ALL,
39              n_dev, devices, 0);
40
41          // Iteriere über alle Geräte.
42          for(j = 0; j < n_dev; j++)
43          {
44              // Frage den Namen des Gerätes ab.
45              error = clGetDeviceInfo(devices[i], CL_DEVICE_NAME,
46                  0, 0, &length);
47              info = (char*)malloc(length);
48              error = clGetDeviceInfo(devices[i], CL_DEVICE_NAME,
49                  length, info, 0);
50              printf("CL_DEVICE_NAME: %s\n", info);
51              free(info);
52          }
53
54          // Gebe Speicher wieder frei.
55          free(devices);
56      }
57
58      // Gebe Speicher wieder frei.
59      free(platforms);
60
61      return 0;
62  }
```

Lösen Sie basierend auf diesem Code die folgenden Aufgaben:

1. Kompilieren Sie den obigen Beispiel-Code und führen Sie das Programm aus, um die vorhandenen Plattformen und Geräte aufzulisten. Gehen Sie dabei vor, wie in Abschnitt 3.1 beschrieben.

2. Fügen Sie weitere Aufrufe der Funktion `clGetPlatformInfo` hinzu, um die folgenden Informationen abzufragen und anzuzeigen:

   - Name der Plattform.
   - Name des Herstellers.

- Unterstützte OpenCL-Version.
- Unterstützte Erweiterungen zu OpenCL.

Denken Sie daran, den Speicher für die auszugebende Zeichenkette korrekt zu belegen. Alternativ können Sie auch einen hinreichend großen statischen Puffer verwenden.

3. Fügen Sie weitere Aufrufe der Funktion `clGetDeviceInfo` hinzu, um die folgenden Informationen abzufragen und anzuzeigen:

   - Name des Gerätes.
   - Name des Herstellers.
   - Unterstützte OpenCL-Version.
   - Unterstützte Erweiterungen zu OpenCL.
   - Gerätetyp. Statt den numerischen Wert der zurückgegebenen Konstanten anzuzeigen, kann es hier sinnvoll sein, eine Fallunterscheidung durchzuführen und einen Text auszugeben.
   - Maximale Taktrate.
   - Größe des lokalen und globalen Device-RAMs.
   - Anzahl der Recheneinheiten.
   - Maximalwerte für die lokalen und globalen Arbeitsgruppengrößen. Diese Werte können Ihnen später nützlich sein, wenn Sie einen Kernel auf dem Gerät ausführen.

   Beachten Sie, dass die Funktion `clGetDeviceInfo` unterschiedliche Datentypen zurückgibt, wenn unterschiedliche Informationen abgefragt werden.

4. Fügen Sie dem Programm eine Funktion zur Fehlerbehandlung hinzu und überprüfen Sie nach jedem Funktionsaufruf den zurückgegebenen Fehlercode. Lassen Sie im Fall eines Fehlers das Programm beenden und geben Sie eine Fehlermeldung aus, die den Grund des Fehlers anzeigt.

Protokollieren Sie, welche Plattformen und Geräte auf Ihrem Rechner vorhanden sind und welche Eigenschaften diese Geräte haben. Überlegen Sie, welche Sie davon im Laufe des weiteren Versuchs benutzen werden. Die in diesem Versuchsteil ermittelten IDs für Plattformen und Geräte werden Sie in den nachfolgenden Versuchen benötigen.

4.1.2 Kompilieren von OpenCL-Code

In dieser Aufgabe geht es darum, den OpenCL-Compiler kennenzulernen, der in der Laufzeitbibliothek enthalten ist. Ziel der Aufgabe ist es, ein paar einfache Funktionen in OpenCL zu schreiben und zu kompilieren. Als Ausgangspunkt kann Ihnen der folgende Code dienen, der zwei komplexe Zahlen multipliziert:

```
1  // Multipliziere zwei komplexe Zahlen.
2  float2 Multiply(float2 a, float2 b)
3  {
4      return (float2)(a.x * b.x - a.y * b.y, a.x * b.y + a.y * b.x);
5  }
6
7  // Multipliziere ein Array aus komplexen Zahlen mit einer Konstanten.
8  __kernel void MultiplyArray(float2 a, __global float2* b, __global float2* c)
9  {
10     int i = get_global_id(0);
11     c[i] = Multiply(a, b[i]);
12 }
```

Um diesen Code zu kompilieren, soll er wie in Abschnitt 3.1.2 beschrieben in das Host-Programm eingebunden werden. Damit der OpenCL-Code vom Host-Programm gefunden wird, muss seine Speicheradresse im Host-Code eingebunden werden, wie in folgendem Beispiel-Code gezeigt:

```
1  #include <CL/cl.h>
2
3  extern char _binary_ocl_cl_start[]; // Beginn des OpenCL-Quelltextes
4  extern char _binary_ocl_cl_end[];   // Ende des OpenCL-Quelltextes
5
6  int main(int argc, char** argv)
7  {
8      cl_platform_id platform; // Plattform
9      cl_device_id device;     // Gerät
10     cl_context context;      // Kontext für die Kompilierung
11     cl_program program;      // Programm (aus OpenCL-Quelltext)
12     cl_int error;            // Fehlercode
13     size_t length;           // Länge des Quelltextes / Compiler-Logs
14     char* source;            // Anfangsadresse des Quelltextes
15     char* log;               // Speicherplatz für das Compiler-Log
16
17     // TODO: Wähle eine Plattform platform und ein Gerät device aus.
18
19     // Speichere die Plattform-ID in ein Array von Typ cl_context_properties.
20     cl_context_properties cprops[3] = { CL_CONTEXT_PLATFORM,
21         (cl_context_properties)platform, 0 };
22
23     // Erstelle einen Kontext.
24     context = clCreateContext(cprops, 1, &device, 0, 0, &error);
25
26     // Bestimme Anfang und Länge des OpenCL-Quelltextes.
27     source = _binary_ocl_cl_start;
28     length = (size_t)(_binary_ocl_cl_end - _binary_ocl_cl_start);
29
30     // Erstelle ein neues Programm aus dem Quelltext.
31     program = clCreateProgramWithSource(context, 1, _binary_ocl_cl_start,
32         &length, &error);
33
34     // Kompiliere das Programm für das gewählte Gerät.
35     error = clBuildProgram(program, 1, &device, 0, 0, 0);
36
37     // Lese die Länge des Compiler-Logs ein.
38     error = clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG,
39         0, 0, &length);
40
41     // Reserviere Speicher.
42     log = (char*)malloc(length);
43
44     // Lese das Compiler-Log ein.
45     error = clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG,
46         length, log, 0);
47
48     // TODO: Zeige das Compiler-Log an.
49
50     // Gebe Ressourcen wieder frei.
51     error = clReleaseProgram(program);
```

```
52      error = clReleaseContext(context);
53
54      return 0;
55  }
```

Zu Beginn des Codes werden zwei Symbole als `extern char[]` deklariert, die vom Linker erzeugt werden, wenn die OpenCL-Quelltextdatei `ocl.cl` wie in Abschnitt 3.1.2 beschrieben in eine Objektdatei umgewandelt wird. Diese markieren Anfang und Ende des OpenCL-Quelltextes im Speicher. Vervollständigen Sie das Programm, damit es die folgenden Aufgaben erfüllt:

1. Auswahl einer Plattform und eines Gerätes. In der vorherigen Übungsaufgabe haben Sie gelernt, wie man die vorhandenen Plattformen und Geräte auflistet und ihre Plattform-ID und Geräte-ID bestimmt. Wählen Sie nun eine Plattform und ein Gerät aus und bestimmen Sie die zugehörigen IDs.
2. Erstellen eines Kontextes.
3. Erstellen und Kompilieren des OpenCL-Programms.
4. Abrufen des Compiler-Logs.
5. Ausgabe des Compiler-Logs mittels `printf`.
6. Freigeben der belegten Ressourcen.

Kompilieren Sie das Programm und führen Sie es aus. Wird der OpenCL-Code korrekt kompiliert? Verändern Sie den OpenCL-Code und experimentieren Sie damit, indem Sie neue Funktionen hinzufügen und die bestehenden Funktionen verändern. Bauen Sie bewusst Fehler ein und beobachten Sie die Auswirkungen, wenn Sie den Code kompilieren. Versuchen Sie, aus dem Compiler-Log die eingebauten Fehler zu bestimmen.

4.1.3 Ausführen eines Kernels

Der Kernel, der in der vorherigen Übungsaufgabe kompiliert wurde, soll nun zur Ausführung gebracht werden. Dafür sind einige weitere Aufrufe der Laufzeitbibliothek erforderlich, wie im folgenden Beispielcode gezeigt:

```
1  int main(int argc, char** argv)
2  {
3      cl_platform_id platform; // Plattform für die Berechnung
4      cl_device_id device;     // Gerät für die Berechnung
5      cl_command_queue queue;  // Befehlswarteschlange
6      cl_kernel kernel;        // Kernel für die komplexe Multiplikation
7      cl_int error;            // Fehlercode
8
9      cl_float2 a;             // Konstante für die Multiplikation
10     cl_float2* b_host;       // Eingabedaten im Host-RAM
11     cl_float2* c_host;       // Ausgabedaten im Host-RAM
12     cl_mem b_dev;            // Eingabedaten im Device-RAM
13     cl_mem c_dev;            // Ausgabedaten im Device-RAM
14     int count;               // Anzahl der komplexen Zahlen
15
16     // TODO: Wähle eine Plattform und ein Gerät aus.
17     // TODO: Erzeuge einen neuen Kontext context.
18     // TODO: Erstelle ein Programm aus dem Quelltext und kompiliere es.
19
20     // TODO: Lege Anzahl count der komplexen Zahlen fest.
21     // TODO: Reserviere Speicher für Ein- und Ausgabedaten im Host-RAM.
```

```
22      // TODO: Lege komplexe Konstante a fest.
23      // TODO: Fülle Speicher b_host für Eingabedaten mit komplexen Zahlen.
24
25      // Reserviere Speicher und kopiere die Eingabedaten in den Device-RAM.
26      b_dev = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
27          sizeof(cl_float2) * count, b_host, 0);
28
29      // Reserviere Speicher für die Ausgabedaten im Device-RAM.
30      c_dev = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
31          sizeof(cl_float2) * count, 0, 0);
32
33      // Erzeuge eine neue Warteschlange.
34      queue = clCreateCommandQueue(context, device, 0, &error);
35
36      // Extrahiere den Kernel.
37      kernel = clCreateKernel(program, "MultiplyArray", &error);
38
39      // Setze die Argumente des Kernels.
40      error = clSetKernelArg(kernel, 0, sizeof(cl_float2), &a);
41      error = clSetKernelArg(kernel, 1, sizeof(cl_mem), &b_dev);
42      error = clSetKernelArg(kernel, 2, sizeof(cl_mem), &c_dev);
43
44      // Lege lokale und globale Arbeitsgruppengröße fest.
45      size_t lws = 256;
46      size_t gws = count;
47
48      // Führe den Kernel aus.
49      error = clEnqueueNDRangeKernel(queue, matmul, 1, 0, &gws, &lws, 0, 0, 0);
50
51      // Lese Ergebnis aus dem Device-RAM.
52      error = clEnqueueReadBuffer(queue, c_dev, CL_TRUE, 0,
53          sizeof(cl_float2) * count, c_host, 0, 0, 0);
54
55      // TODO: Gebe Ergebnisdaten c_host aus.
56
57      // Gebe Kernel-Ressourcen wieder frei.
58      error = clReleaseKernel(kernel);
59
60      // Gebe Warteschlange wieder frei.
61      error = clReleaseCommandQueue(queue);
62
63      // TODO: Gebe übrige Ressourcen wieder frei.
64
65      return 0;
66  }
```

Benutzen Sie Ihren Code aus der vorherigen Übungsaufgabe und ergänzen Sie ihn basierend auf dem oben beschriebenen Beispiel-Code, um ein Programm zu schreiben, das die folgenden Aufgaben erfüllt:

1. Auswählen einer Plattform und eines Gerätes.
2. Erzeugen eines Kontextes und einer Warteschlange.
3. Kompilieren eines Programms und Extrahieren des Kernels.
4. Festlegen der Eingabedaten. Die Konstante `a` können Sie entweder statisch im Code festlegen oder zur Laufzeit vom Benutzer abfragen. Für das Array `b_host` bietet es sich

an, Zufallszahlen zu verwenden oder das Array in einer Schleife mit Werten zu füllen, die Sie aus einer Formel erzeugen.

5. Reservieren von Speicher im Device-RAM und Kopieren der Eingabedaten.

6. Festlegen der Argumente für den Kernel-Aufruf.

7. Festlegen der lokalen und globalen Arbeitsgruppengröße. Den maximalen Wert für die lokale Arbeitsgruppengröße können Sie der ersten Übungsaufgabe entnehmen. Die globale Arbeitsgruppengröße muss der Anzahl der Elemente im Ein- und Ausgabearray `b_host` bzw. `c_host` entsprechen. Dabei sollte die globale Arbeitsgruppengröße ein ganzzahliges Vielfaches der lokalen Arbeitsgruppengröße sein.

8. Einfügen des Kernels in die Warteschlange.

9. Auslesen der Ausgabedaten.

10. Ausgabe der Ausgabedaten, z.B. in eine Datei oder einen Plot.

11. Freigabe aller Ressourcen.

Kompilieren Sie das Programm und führen Sie es für unterschiedliche Eingabedaten aus. Überprüfen Sie für einige Elemente des Arrays die Richtigkeit des Ergebnisses. Erhöhen Sie anschließend die Anzahl der Elemente des Arrays und untersuchen Sie, wie sich die Ausführungszeit des Programms dabei verändert. Führen Sie die Berechnung auf unterschiedlichen OpenCL-Geräten durch und vergleichen Sie die Ausführungszeiten.

4.2 Matrixmultiplikation

Zum Einstieg in die GPU-Programmierung soll zunächst eine mathematisch recht einfache Aufgabe gelöst werden, die sich kanonisch parallelisieren lässt. Als Beispiel soll hier die Multiplikation von zwei Matrizen behandelt werden.

4.2.1 Grundlagen

Betrachten Sie zwei Matrizen $\underline{\underline{A}}$ und $\underline{\underline{B}}$ der Dimension $m \times n$ und $n \times p$, deren Komponenten gegeben sind durch

$$\underline{\underline{A}} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \tag{4.2.1}$$

und analog für $\underline{\underline{B}}$. Ihr Produkt $\underline{\underline{C}} = \underline{\underline{A}} \cdot \underline{\underline{B}}$ ist eine Matrix der Dimension $m \times p$, deren Komponenten gegeben sind durch

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \tag{4.2.2}$$

für $1 \leq i \leq m$ und $1 \leq j \leq p$. Jede dieser Komponenten $m \times p$ kann unabhängig von den anderen berechnet werden. Jede dieser Berechnungen erfordert n Multiplikationen und die Summation dieser n Produkte. Es bietet sich daher an, die Berechnung parallel durchzuführen.

4.2.2 OpenCL-Kernel

Für die Berechnung von (4.2.2) bietet es sich an, die $m \times p$ Komponenten c_{ij} als zweidimensionales, endliches Gitter zu betrachten. Jeder Gitterpunkt entspricht einer Komponente der Matrix. Da sich jede Komponente c_{ij} unabhängig von den anderen Komponenten berechnen lässt, ist es sinnvoll, einen Kernel zu schreiben, der für jeden Gitterpunkt einmal aufgerufen wird und eine Komponente berechnet. Der folgende Beispiel-Code zeigt, wie ein solcher Kernel aufgebaut sein könnte:

```
1  __kernel void MatMul(int m, int n, int p,
2      __global float* a, __global float* b, __global float* c)
3  {
4      float value = 0;          // Zwischenspeicher zur Berechnung der Summe
5      int i = get_global_id(0); // Zeilenindex von C
6      int j = get_global_id(1); // Spaltenindex von C
7      int k;                    // Laufvariable zur Summation
8
9      for(k = 0; k < n; k++)
10     {
11         // Addiere a_ik b_kj zur Summe.
12         value += a[i * n + k] * b[k * p + j];
13     }
14
15     c[i * p + j] = value; // Schreibe Ergebnis in c_ij.
16 }
```

Als Parameter werden an den Kernel die Dimensionen der einzelnen Matrizen sowie Zeiger auf ihre Komponenten übergeben. Der Kernel benutzt anschließend die Funktion `get_global_id`, um herauszufinden, für welchen Gitterpunkt er gerade ausgeführt wird. Schließlich berechnet er die entsprechende Komponente c_{ij} und trägt sie in die Matrix ein.

4.2.3 Host-Programm

Um den im vorherigen Abschnitt vorgestellten Kernel auszuführen, ist ein Host-Programm notwendig, das die Eingabedaten bereitstellt, den Kernel aufruft und schließlich die Ausgabedaten verwertet. Der folgende Beispiel-Code zeigt den grundlegenden Ablauf eines solchen Host-Programms und die wesentlichen Aufgaben:

```
1  int main(int argc, char** argv)
2  {
3      int m, n, p;                      // Dimensionen der Matrizen
4      float *a_host, *b_host, *c_host; // A, B, C im Host-RAM
5
6      cl_context context;               // Gerätekontext
7      cl_command_queue queue;           // Befehlswarteschlange
8      cl_kernel matmul;                 // Kernel zur Matrixmultiplikation
9
10     cl_mem a_dev, b_dev, c_dev;       // A, B, C im Device-RAM
11
12     int blocksize = 16;               // GPU-Blockgröße
13
14     // TODO: Lese die Matrix-Dimensionen m, n, p ein.
15     // TODO: Reserviere Speicherplatz für a_host, b_host, c_host im Host-RAM.
16     // TODO: Fülle a_host und b_host mit Werten.
17     // TODO: Erzeuge Gerätekontext context.
18     // TODO: Erzeuge Befehlswarteschlange queue.
19     // TODO: Lade und kompiliere Kernel matmul.
```

```
20
21      // Reserviere Device-RAM für A und B und kopiere Daten.
22      a_dev = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
23          sizeof(float) * m * n, a_host, 0);
24      b_dev = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
25          sizeof(float) * n * p, b_host, 0);
26
27      // Reserviere Device-RAM für C.
28      c_dev = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
29          sizeof(float) * m * p, 0, 0);
30
31      // Setze Argumente für den Kernelaufruf.
32      error = clSetKernelArg(matmul, 0, sizeof(int), &m);
33      error = clSetKernelArg(matmul, 1, sizeof(int), &n);
34      error = clSetKernelArg(matmul, 2, sizeof(int), &p);
35      error = clSetKernelArg(matmul, 3, sizeof(cl_mem), &a_dev);
36      error = clSetKernelArg(matmul, 4, sizeof(cl_mem), &b_dev);
37      error = clSetKernelArg(matmul, 5, sizeof(cl_mem), &c_dev);
38
39      // Bestimme globale und lokale Arbeitsgruppengröße.
40      size_t gws[] = {m, p};
41      size_t lws[] = {blocksize, blocksize};
42
43      // Führe den Kernel aus.
44      error = clEnqueueNDRangeKernel(queue, matmul, 2, 0, gws, lws, 0, 0, 0);
45
46      // Lese Matrix C aus dem Device-RAM.
47      error = clEnqueueReadBuffer(queue, c_dev, CL_TRUE, 0,
48          sizeof(float) * m * p, c_host, 0, 0, 0);
49
50      // TODO: Ausgabe von C.
51      // TODO: Freigabe der belegten Ressourcen.
52
53      return 0;
54  }
```

Bevor die eigentliche Berechnung beginnen kann, müssen die Eingabedaten festgelegt werden. Insbesondere muss die Dimension m, n, p der Matrizen bekannt sein. Diese können Sie als Konstanten definieren oder vom Benutzer zur Laufzeit abfragen. Wenn die Dimensionen feststehen, muss der Speicherplatz für die Matrizen $\underline{\underline{A}}, \underline{\underline{B}}, \underline{\underline{C}}$ im Host-RAM reserviert werden. Anschließend können die Matrizen $\underline{\underline{A}}$ und $\underline{\underline{B}}$ mit Eingabewerten gefüllt werden. Am einfachsten ist die Verwendung von (Pseudo-)Zufallszahlen. Alternativ können Sie aber auch eine Funktion schreiben, die Eingabedaten aus einer Formel generiert.
Als nächstes muss die Hardware für die Berechnung vorbereitet werden. In den Übungsaufgaben im Abschnitt 4.1 haben Sie in Erfahrung gebracht, welche Plattformen und Geräte für die Berechnung zur Verfügung stehen. Wählen Sie nun eine Plattform und ein Gerät aus und ermitteln Sie die zugehörigen Werte vom Typ `cl_platform_id` und `cl_device_id`. Erstellen Sie anschließend einen Kontext, der dieses Gerät umfasst.

4.2.4 Aufgaben

Vervollständigen Sie das in diesem Abschnitt beschriebene OpenCL-Programm, damit es die folgenden Aufgaben erfüllt:

1. Einlesen der Matrix-Dimensionen m, n, p.
2. Reservieren von Speicherplatz für $\underline{\underline{A}}, \underline{\underline{B}}, \underline{\underline{C}}$ im Host-RAM.

3. Füllen von $\underline{\underline{A}}$ und $\underline{\underline{B}}$ mit Zahlenwerten. Hier können Sie Zufallszahlen, Werte aus einer Datei oder formelgenerierte Werte verwenden.

4. Reservieren von Speicherplatz für $\underline{\underline{A}}, \underline{\underline{B}}, \underline{\underline{C}}$ im Device-RAM und Kopieren der Matrizen $\underline{\underline{A}}$ und $\underline{\underline{B}}$.

5. Kompilieren und Starten des Kernels.

6. Kopieren der Matrix $\underline{\underline{C}}$ in den Host-RAM.

7. Ausgabe von $\underline{\underline{C}}$ (z.B. in eine Datei oder grafisch mittels Gnuplot).

Multiplizieren Sie einige größere Matrizen miteinander und versuchen Sie, die benötigte Rechenzeit zu messen. Dafür bietet es sich an, die Größe der Matrizen so zu wählen, dass die Rechenzeit im Bereich von einigen Sekunden bis Minuten liegt. Wie hängt die Rechenzeit von m, n, p ab? Für die Zeitmessung können Sie eine externe Uhr oder die C-Funktion `clock` benutzen.

4.3 Mandelbrot-Menge

Bei der Mandelbrot-Menge handelt es sich um eine Teilmenge der komplexen Zahlen $\mathbb{C}$. Ziel dieser Aufgabe ist es, die Mandelbrot-Menge numerisch zu approximieren.

4.3.1 Grundlagen

Betrachten Sie die Folge $(z_n, n \in \mathbb{N})$, die gegeben ist durch die logistische Abbildung

$$z_{n+1} = z_n^2 + c \tag{4.3.1}$$

mit Anfangswert $z_0 = 0$ und einem konstanten Parameter $c \in \mathbb{C}$. Das Verhalten dieser Folge hängt stark von der Wahl des Parameters c ab. Für große Werte von $|c|$ ist die Folge divergent. Für kleine Werte von $|c|$ dagegen konvergiert sie gegen einen Fixpunkt. Im Bereich dazwischen zeigt sich teils periodisches, teils chaotisches Verhalten. Bei genauerer Betrachtung zeigt sich, dass die Menge der Parameterwerte c, für die die Folge beschränkt ist, zusammenhängend ist. Diese zusammenhängende Menge wird als Mandelbrot-Menge bezeichnet. Ihr Rand bildet ein Fraktal.

4.3.2 OpenCL-Kernel

Zur Darstellung von komplexen Zahlen bietet es sich an, den OpenCL-Vektordatentyp `float2` zu verwenden, der aus zwei Gleitkommazahlen besteht. Die Verwendung von Vektordatentypen wird im Abschnitt 2.2.2.2 erläutert. Das folgende Beispiel veranschaulicht die Darstellung von komplexen Zahlen durch den Datentyp `float2`:

```
1  float2 Square(float2 z)
2  {
3      // Das Quadrat einer komplexen Zahl, in Real- und Imaginärteil.
4      return (float2)(z.x * z.x - z.y * z.y, 2 * z.x * z.y);
5  }
6
7  int Mandel(float2 c, int n)
8  {
9      // Initialisiere komplexe Variable auf den Wert 0.
10     float2 z = (float2)(0.0);
11     int i;
12
13     for(i = 0; i < n; i++)
14     {
```

```
15          // Berechne das Quadrat von z und addiere eine Konstante.
16          z = Square(z) + c;
17
18          // TODO: Wenn Abbruchkriterium erfüllt, Schleife verlassen.
19      }
20
21      return i;
22  }
```

Die Funktion `Square` berechnet das Quadrat einer komplexen Zahl. Der Zugriff auf die einzelnen Komponenten der vektorwertigen Variable `z` erfolgt durch `z.x` und `z.y`. Im hier gezeigten Beispiel stellt `z.x` den Realteil und `z.y` den Imaginärteil von `z` dar. Diese Funktion wird innerhalb der Funktion `Mandel` benutzt, die wiederholt die logistische Abbildung berechnet und zurückgibt, nach wie vielen Iterationen ein Abbruchkriterium erreicht ist.

4.3.3 Aufgaben

Schreiben Sie zunächst einen OpenCL-Kernel, der die folgenden Aufgaben erfüllt:

1. Ermitteln der zweidimensionalen Gitterkoordinaten.
2. Berechnen der zugehörigen komplexen Zahl c.
3. Setzen des Startwertes $z = 0$.
4. Iteration der logistischen Abbildung (4.3.1).
5. Zählen der Iterationsschritte bis zum Erreichen der Abbruchbedingung.
6. Eintragen der Schrittzahl in ein Ausgabearray.

Schreiben Sie anschließend ein Host-Programm, das die folgenden Aufgaben erfüllt:

1. Einlesen der Grenzen eines rechteckigen Bereichs der komplexen Zahlenebene.
2. Aufteilen des Bereichs in Gitterpunkte.
3. Einlesen der maximalen Anzahl an Iterationen.
4. Iteration der logistischen Abbildung für jeden Gitterpunkt, bis das Abbruchkriterium erreicht ist.
5. Ausgabe der Anzahl der ausgeführten Iterationen für jeden Gitterpunkt.

Für die Datenausgabe bietet es sich an, ein Dateiformat zu wählen, das z.B. in GnuPlot eingelesen werden kann. Finden Sie eine geeignete grafische Darstellung für Ihre Ergebnisse (z.B. durch eine Farbcodierung für die maximale Iterationszahl) und plotten Sie das Ergebnis für einige Bereiche der Mandelbrot-Menge, die Ihnen interessant erscheinen.

4.4 Vielteilchen-System

In dieser Aufgabe soll die Dynamik eines wechselwirkenden Vielteilchen-Systems simuliert werden.

4.4.1 Definition des Systems

Betrachten Sie die Dynamik eines Vielteilchen-Systems aus N Teilchen, dessen Dynamik gegeben ist durch die Lagrange-Funktion

$$L = \sum_{i=1}^{N} \frac{m}{2} \dot{\vec{x}}_i^2 - \sum_{i=1}^{N} V(\vec{x}_i) - \sum_{i<j} U(|\vec{x}_i - \vec{x}_j|)\,. \tag{4.4.1}$$

Dabei sind $\vec{x}_i$ die Koordinaten der N Teilchen und m ihre Masse. Die Dynamik wird bestimmt durch ein äußeres Potential $V(\vec{x}_i)$, in dem sich die Teilchen bewegen, sowie durch ein Wechselwirkungspotential $U(|\vec{x}_i - \vec{x}_j|)$, das nur vom Abstand der Teilchen abhängen soll. Die Bewegungsgleichungen sind gegeben durch die Euler-Lagrange-Gleichungen

$$0 = \frac{d}{dt}\frac{dL}{d\dot{\vec{x}}_i} - \frac{dL}{d\vec{x}_i} = m\ddot{\vec{x}}_i + \vec{\nabla} V(\vec{x}_i) + \sum_{j\neq i} \frac{\vec{x}_i - \vec{x}_j}{|\vec{x}_i - \vec{x}_j|} \left.\frac{d}{dr} U(r)\right|_{r=|\vec{x}_i - \vec{x}_j|}\,. \tag{4.4.2}$$

Ein einfaches Beispiel ist ein System von Teilchen, die sich in einem quadratischen Potential $V(\vec{x}) = a\vec{x}^2$ befinden und der Coulomb-Wechselwirkung $U(r) = b/r$ unterliegen, wobei a und b Konstanten sind. In diesem Fall lauten die Bewegungsgleichungen

$$0 = m\ddot{\vec{x}}_i + a\vec{x}_i - b\sum_{j\neq i} \frac{\vec{x}_i - \vec{x}_j}{|\vec{x}_i - \vec{x}_j|^3}\,. \tag{4.4.3}$$

Je nach Vorzeichen von b kann die Wechselwirkung anziehend sein, wie im Fall der Gravitation, oder abstoßend, wie im Fall der elektrostatischen Wechselwirkung.

4.4.2 Lösungsverfahren

Um die Bewegungsgleichung (4.4.2) des Vielteilchen-Systems zu lösen, bietet es sich an, das Euler-Verfahren zu verwenden. Dafür führt man zunächst die Geschwindigkeit $\vec{v}_i = \dot{\vec{x}}_i$ als neue Variable ein und erhält so ein Differentialgleichungssystem 1. Ordnung:

$$\dot{\vec{x}}_i = \vec{v}_i\,, \tag{4.4.4a}$$

$$\dot{\vec{v}}_i = -\frac{1}{m}\left(\vec{\nabla} V(\vec{x}_i) + \sum_{j\neq i} \frac{\vec{x}_i - \vec{x}_j}{|\vec{x}_i - \vec{x}_j|} \left.\frac{d}{dr} U(r)\right|_{r=|\vec{x}_i - \vec{x}_j|}\right)\,. \tag{4.4.4b}$$

Beim Euler-Verfahren werden nun die Zeitableitungen durch Differenzenquotienten ersetzt:

$$\dot{\vec{x}}_i(t) = \frac{d}{dt}\vec{x}_i(t) \approx \frac{\vec{x}_i(t+\Delta t) - \vec{x}_i(t)}{\Delta t} \tag{4.4.5}$$

und analog für $\dot{\vec{v}}$. Daraus lässt sich die Lösung schrittweise (iterativ) bestimmen:

$$\vec{x}_i(t+\Delta t) = \vec{x}_i(t) + \Delta t \cdot \vec{v}_i\,, \tag{4.4.6a}$$

$$\vec{v}_i(t+\Delta t) = \vec{v}_i(t) - \frac{\Delta t}{m}\left(\vec{\nabla} V(\vec{x}_i) + \sum_{j\neq i} \frac{\vec{x}_i - \vec{x}_j}{|\vec{x}_i - \vec{x}_j|} \left.\frac{d}{dr} U(r)\right|_{r=|\vec{x}_i - \vec{x}_j|}\right)\,. \tag{4.4.6b}$$

Dieses Verfahren soll nun zur numerischen Lösung des Vielteilchen-Problems angewandt werden.

4.4.3 OpenCL-Kernel

Die Struktur der Bewegungsgleichungen 4.4.6 legt es nahe, das Problem für jeden Zeitschritt Δt in zwei Teilschritten zu lösen, die sich abwechseln:

- Die neue Geschwindigkeit $\vec{v}_i(t + \Delta t)$ eines Teilchens hängt nur von seiner alten Geschwindigkeit $\vec{v}_i(t)$ und der Kraft auf dieses Teilchen ab, die durch die Positionen aller Teilchen gegeben ist. Es bietet sich daher an, in einem Schritt die Geschwindigkeit aller Teilchen neu zu berechnen und den alten durch den neuen Wert zu ersetzen. Dieser Rechenschritt kann für alle Teilchen parallel durchgeführt werden, da dabei nur die Geschwindigkeit $\vec{v}_i$ jedes Teilchens verändert wird, die in die Berechnung der Geschwindigkeiten der anderen Teilchen nicht eingeht.

- Die neue Position $\vec{x}_i(t+\Delta t)$ eines Teilchens hängt nur von seiner alten Position $\vec{x}_i(t)$ und seiner Geschwindigkeit $\vec{v}_i(t)$ ab. Im zweiten Schritt bietet es sich daher an, die Position aller Teilchen neu zu berechnen und den alten durch den neuen Wert zu ersetzen. Auch dieser Rechenschritt kann für alle Teilchen parallel durchgeführt werden, da hier nur die Position $\vec{x}_i$ jedes Teilchens verändert wird, was aber die Positionsberechnung der anderen Teilchen nicht beeinflusst.

Diese Zerlegung in zwei Teilschritte, die abwechselnd ausgeführt werden, lässt sich am besten durch zwei Kernels realisieren, die nacheinander aufgerufen werden. Im Folgenden sei der erste Kernel, der die Kraft auf jedes Teilchen berechnet, mit `Force` bezeichnet, während der zweite Kernel, der das Voranschreiten der Teilchen berechnet, mit `Advance` bezeichnet sei. Die beiden Kernels können durch folgende Struktur realisiert werden:

```
1  __kernel void Force(__global float3* x, __global float3* v)
2  {
3      // Geschwindigkeitsänderung durch die Kraftwirkung
4      float3 deltav;
5
6      // Index i des zu betrachtenden Teilchens
7      int i = get_global_id(0);
8
9      // TODO: Berechne Geschwindigkeitsänderung durch äußeres Potential.
10     // TODO: Berechne Geschwindigkeitsänderung durch Wechselwirkung.
11
12     // Nutze Addition von float3 Vektoren.
13     v[i] += deltav;
14 }
15
16 __kernel void Advance(__global float3* x, __global float3* v)
17 {
18     // Zeitschritt (an anderer Stelle im Programm festgelegt)
19     float deltat;
20
21     // Index i des zu betrachtenden Teilchens
22     int i = get_global_id(0);
23
24     // Nutze (Vektor + (Skalar * Vektor)).
25     x[i] += deltat * v[i];
26 }
```

Die Positionen und Geschwindigkeiten der Teilchen wurden hier durch zwei Arrays vom Typ `float3` realisiert, um Vektoroperationen benutzen zu können.

4.4.4 Host-Programm mit Ereignissen

Für die korrekte Ausführung des Programms ist es notwendig, dass die beiden Kernel abwechselnd ausgeführt werden und dass die Ausführung eines Kernels vollständig abgeschlossen ist, bevor mit der Ausführung des anderen Kernels begonnen wird. Um dies zu gewährleisten, werden beide Kernels abwechselnd in die gleiche Befehlswarteschlange eingefügt und dabei

jeweils auf ein Ereignis gewartet, dass das Ende einer Berechnung signalisiert. Der folgende Beispiel-Code verdeutlicht diese Vorgehensweise:

```
1  int main(int argc, char** argv)
2  {
3      cl_context context;          // Gerätekontext
4      cl_command_queue queue;      // Befehlswarteschlange
5      cl_kernel force, advance;    // Kernel für die Berechnung
6      cl_event event1, event2;     // Ereignisse für die Synchronisation
7      cl_int error;                // Fehlercode
8
9      int bodies;                  // Anzahl der Teilchen
10     cl_float3 *x_host, *v_host; // Orte und Geschwindigkeiten im Host-RAM
11     cl_mem x_dev, v_dev;         // Orte und Geschwindigkeiten im Device-RAM
12
13     int n;                       // Anzahl der Schleifendurchläufe
14     bool finish;                 // Test zum Beenden der Simulation
15     int outstep;                 // Anzahl der Durchläufe pro Ausgabe
16     int blocksize = 512;         // GPU-Blockgröße
17
18     // TODO: Wähle eine Plattform und ein Gerät aus.
19     // TODO: Erzeuge einen Kontext und eine Warteschlange.
20     // TODO: Lade und kompiliere den OpenCL-Code.
21     // TODO: Erzeuge zwei Kernel für die Berechnung
22     // TODO: Reserviere Speicher im Host-RAM für Orte und Geschwindigkeiten.
23     // TODO: Fülle x_host und v_host mit Anfangswerten.
24
25     // Reserviere Speicher im Device-RAM und kopiere die Anfangswerte.
26     x_dev = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
27         sizeof(cl_float3) * bodies, x_host, &error);
28     v_dev = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
29         sizeof(cl_float3) * bodies, v_host, &error);
30
31     // Setze Argumente für die beiden Kernel.
32     error = clSetKernelArg(force, 0, sizeof(cl_mem), &x_dev);
33     error = clSetKernelArg(force, 1, sizeof(cl_mem), &v_dev);
34     error = clSetKernelArg(advance, 0, sizeof(cl_mem), &x_dev);
35     error = clSetKernelArg(advance, 1, sizeof(cl_mem), &v_dev);
36
37     // Setze globale und lokale Arbeitsgruppengröße.
38     size_t lws = blocksize;
39     size_t gws = bodies;
40
41     // Rufe Kernel einmal auf, um Ereignis zu initialisieren.
42     error = clEnqueueNDRangeKernel(queue, advance, 1, 0,
43         &gws, &lws, 0, 0, &event1);
44
45     // Initialisiere Abbruchvariable.
46     finish = false;
47
48     // Führe Iterationsschleife aus.
49     for(n = 0; !finish; n++)
50     {
51         // Gebe alle outstep Schritte die Daten aus.
52         if(n % outstep == 0)
53         {
```

```
54              // Warte auf das Ende der Berechnung und kopiere Daten.
55              error = clEnqueueReadBuffer(queue, x_dev, CL_FALSE, 0,
56                  sizeof(cl_float3) * bodies, x_host, 1, &event1, &event2);
57              error = clEnqueueReadBuffer(queue, v_dev, CL_TRUE, 0,
58                  sizeof(cl_float3) * bodies, v_host, 1, &event2, &event1);
59
60              // TODO: Speichere Daten in eine Datei oder zeige sie an.
61          }
62
63          // Führe nacheinander beide Kernel aus.
64          error = clEnqueueNDRangeKernel(queue, advance, 1, 0,
65              &gws, &lws, 1, &event2, &event1);
66          error = clEnqueueNDRangeKernel(queue, force, 1, 0,
67              &gws, &lws, 1, &event1, &event2);
68      }
69
70      // TODO: Gebe belegte Ressourcen wieder frei;
71
72      return 0;
73  }
```

Durch das Warten auf die beiden Ereignisse `event1` und `event2` ist sichergestellt, dass ein Kernel beendet wurde, bevor der andere gestartet wird. Es wird außerdem sichergestellt, dass keine Berechnung stattfindet, so lange noch Daten vom Device-RAM in den Host-RAM kopiert werden.

4.4.5 Host-Programm mit Barrieren

Eine alternative Möglichkeit, die Synchronisation zwischen mehreren Kernels zu gewährleisten, ist die Verwendung von Barrieren, wie in Abschnitt 2.3.11 erläutert. Der folgende Beispiel-Code verdeutlicht die Verwendung von Barrieren:

```
1  int main(int argc, char** argv)
2  {
3      cl_context context;          // Gerätekontext
4      cl_command_queue queue;      // Befehlswarteschlange
5      cl_kernel force, advance;    // Kernel für die Berechnung
6      cl_int error;                // Fehlercode
7
8      int bodies;                  // Anzahl der Teilchen
9      cl_float3 *x_host, *v_host; // Orte und Geschwindigkeiten im Host-RAM
10     cl_mem x_dev, v_dev;         // Orte und Geschwindigkeiten im Device-RAM
11
12     int n;                       // Anzahl der Schleifendurchläufe
13     bool finish;                 // Test zum Beenden der Simulation
14     int outstep;                 // Anzahl der Durchläufe pro Ausgabe
15     int blocksize = 512;         // GPU-Blockgröße
16
17     // TODO: Wähle eine Plattform und ein Gerät aus.
18     // TODO: Erzeuge einen Kontext und eine Warteschlange.
19     // TODO: Lade und kompiliere den OpenCL-Code.
20     // TODO: Erzeuge zwei Kernel für die Berechnung
21     // TODO: Reserviere Speicher im Host-RAM für Orte und Geschwindigkeiten.
22     // TODO: Fülle x_host und v_host mit Anfangswerten.
23
24     // Reserviere Speicher im Device-RAM und kopiere die Anfangswerte.
25     x_dev = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
```

```
26          sizeof(cl_float3) * bodies, x_host, &error);
27      v_dev = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
28          sizeof(cl_float3) * bodies, v_host, &error);
29
30      // Setze Argumente für die beiden Kernel.
31      error = clSetKernelArg(force, 0, sizeof(cl_mem), &x_dev);
32      error = clSetKernelArg(force, 1, sizeof(cl_mem), &v_dev);
33      error = clSetKernelArg(advance, 0, sizeof(cl_mem), &x_dev);
34      error = clSetKernelArg(advance, 1, sizeof(cl_mem), &v_dev);
35
36      // Setze globale und lokale Arbeitsgruppengröße.
37      size_t lws = blocksize;
38      size_t gws = bodies;
39
40      // Initialisiere Abbruchvariable.
41      finish = false;
42
43      // Führe Iterationsschleife aus.
44      for(n = 0; !finish; n++)
45      {
46          // Gebe alle outstep Schritte die Daten aus.
47          if(n % outstep == 0)
48          {
49              // Kopiere Daten.
50              error = clEnqueueReadBuffer(queue, x_dev, CL_TRUE, 0,
51                  sizeof(cl_float3) * bodies, x_host, 0, 0, 0);
52              error = clEnqueueReadBuffer(queue, v_dev, CL_TRUE, 0,
53                  sizeof(cl_float3) * bodies, v_host, 0, 0, 0);
54
55              // TODO: Speichere Daten in eine Datei oder zeige sie an.
56          }
57
58          // Führe nacheinander beide Kernel aus.
59          error = clEnqueueNDRangeKernel(queue, advance, 1, 0,
60              &gws, &lws, 0, 0, 0);
61          clEnqueueBarrier(queue);
62          error = clEnqueueNDRangeKernel(queue, force, 1, 0,
63              &gws, &lws, 0, 0, 0);
64          clEnqueueBarrier(queue);
65      }
66
67      // TODO: Gebe belegte Ressourcen wieder frei;
68
69      return 0;
70  }
```

Dieser Code kommt ohne die explizite Verwendung von Ereignissen aus und ist daher einfacher strukturiert.

4.4.6 Aufgaben

Vervollständigen Sie zunächst die beiden OpenCL-Kernel. Der Kernel `Force` berechnet die Kraft auf die einzelnen Teilchen und die daraus resultierende Geschwindigkeitsänderung. Der Kernel `Advance` berechnet die Bewegung der Teilchen. Verwenden Sie dabei das quadratische Potential $V(\vec{x}) = a\vec{x}^2$ und die Coulomb-Wechselwirkung $U(r) = b/r$. Vervollständigen Sie außerdem die beiden Varianten des Host-Programms, um die folgenden Aufgaben zu lösen:

1. Zufällige Verteilung der Anfangspositionen der Teilchen.
2. Setzen der Anfangsgeschwindigkeiten aller Teilchen auf 0.
3. Laden und Kompilieren der beiden Kernel.
4. Iterative Anwendung des Euler-Verfahrens.
5. Regelmäßige Ausgabe der Daten in eine Datei oder einen Plot.

Testen Sie anschließend beide Varianten und vergleichen Sie ihre Ausführungszeiten. Führen Sie die Simulation für anziehende (gravitative) und abstoßende (elektrostatische) Coulomb-Wechselwirkung zwischen den Teilchen durch. Danach können Sie weitere Modifikationen an Ihrer Simulation vornehmen. Einige Beispiele für mögliche Modifikationen sind:

- Ersetzen des äußeren Potentials, z.B. durch ein quartisches Potential $V(\vec{x}) = a\vec{x}^4$ oder ein periodisches Potential.
- Ersetzen des Wechselwirkungspotentials, z.B. durch ein Yukawa-Potential $U(r) = b\exp(-r/r_0)/r$.
- Beschränken des Bewegungsbereiches der Teilchen nicht durch ein äußeres Potential, sondern durch Randbedingungen, z.B. durch einen starren Kasten (Teilchen werden an den Wänden reflektiert) oder periodische Randbedingungen (Teilchen, die den "Kasten" auf einer Seite verlassen, tauchen auf der gegenüberliegenden Seite wieder auf, entsprechend der Topologie eines Torus T^3).
- Hinzufügen einer geschwindigkeitsproportionalen Reibung.

Dokumentieren Sie jeweils Ihre Modifikationen und die Ergebnisse.

Literaturverzeichnis

[1] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors - A Hands-on Approach*, Morgan Kaufman, 2010.

[2] A. Munshi, B. R. Gaster, T. G. Mattson, J. Fung and D. Ginsburg, *OpenCL Programming Guide*, Addison-Wesley, 2012.

[3] The OpenCL Specification, Khronos OpenCL Working Group, http://www.khronos.org

- http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf
- http://www.khronos.org/registry/cl/specs/opencl-1.2-extensions.pdf
- http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/
- http://www.khronos.org/registry/cl/sdk/1.2/docs/OpenCL-1.2-refcard.pdf

[4] NVidia OpenCL Programming Guide:
http://developer.nvidia.com/opencl

[5] AMD OpenCL Programming Guide:
http://developer.amd.com/sdks/AMDAPPSDK/documentation/Pages/default.aspx

[6] Intel OpenCL Programming Guide:
http://software.intel.com/en-us/articles/vcsource-tools-opencl-sdk