Parallele Algorithmen mit OpenCL

Universität Osnabrück, Henning Wenke, 2013-04-17

Kapitel I

OpenCL Einführung

Allgemeines

- **Open C**ompute **L**anguage: API für einheitliche parallele Programmierung heterogener Systeme
  - GPU (Grafikprozessor, hohe Parallelität, API zwingend)
  - CPU (Hauptprozessor, geringe Parallelität, API optional)
  - APU
  - FPGA
  - …
  - Oder Kombinationen daraus
- Prozedural, low-level
- Initiiert durch Apple
- Danach betreut durch KHRONOS GROUP
- Plattform, Betriebssystem & Sprachunabhängig

Vorläufige Vereinfachungen

- Parallelitätsmodell
- Speicherhierarchie der Devices
- Events, etc.
- „Heterogene Systeme“
- Fehlerbehandlung

Host – Device(s) Architektur

- Genau ein Host
  - Für Koordination (Berechnungen, Datentransfers, Synchronisation, ...)
  - Geschieht mit OpenCL API calls
  - Immer CPU
- Ein oder mehrere Device(s)
  - Zum Ausführen der parallelen Berechnungen
  - Auf CPU(s), GPU(s), APU(s), …
- Programme für Devices heißen Kernel
  - Sind in OpenCL C formuliert
  - Syntaktisch: Funktion ohne Rückgabe mit Qualifier `kernel`
  - Durch OpenCL API calls übersetzt, mit Daten versorgt & ausgeführt
- OpenCL C
  - Zugehörige, an C (ISO C99) angelehnte, Sprache
  - Zusätzlich Erweiterungen für parallele Algorithmen
  - Kein dynamisches Allokieren von Speicher
  - Keine Arrays variabler Länge
  - Keine Rekursion

Beispielumgebung: Rechner in 31/145

Host:
CPU: Intel Core 2 Duo
Hinweis: Kann auch Device sein

Device:
GPU: Nvidia GTS 450

Führt aus

Applikation
Java, C++, ...

OpenCL API Calls

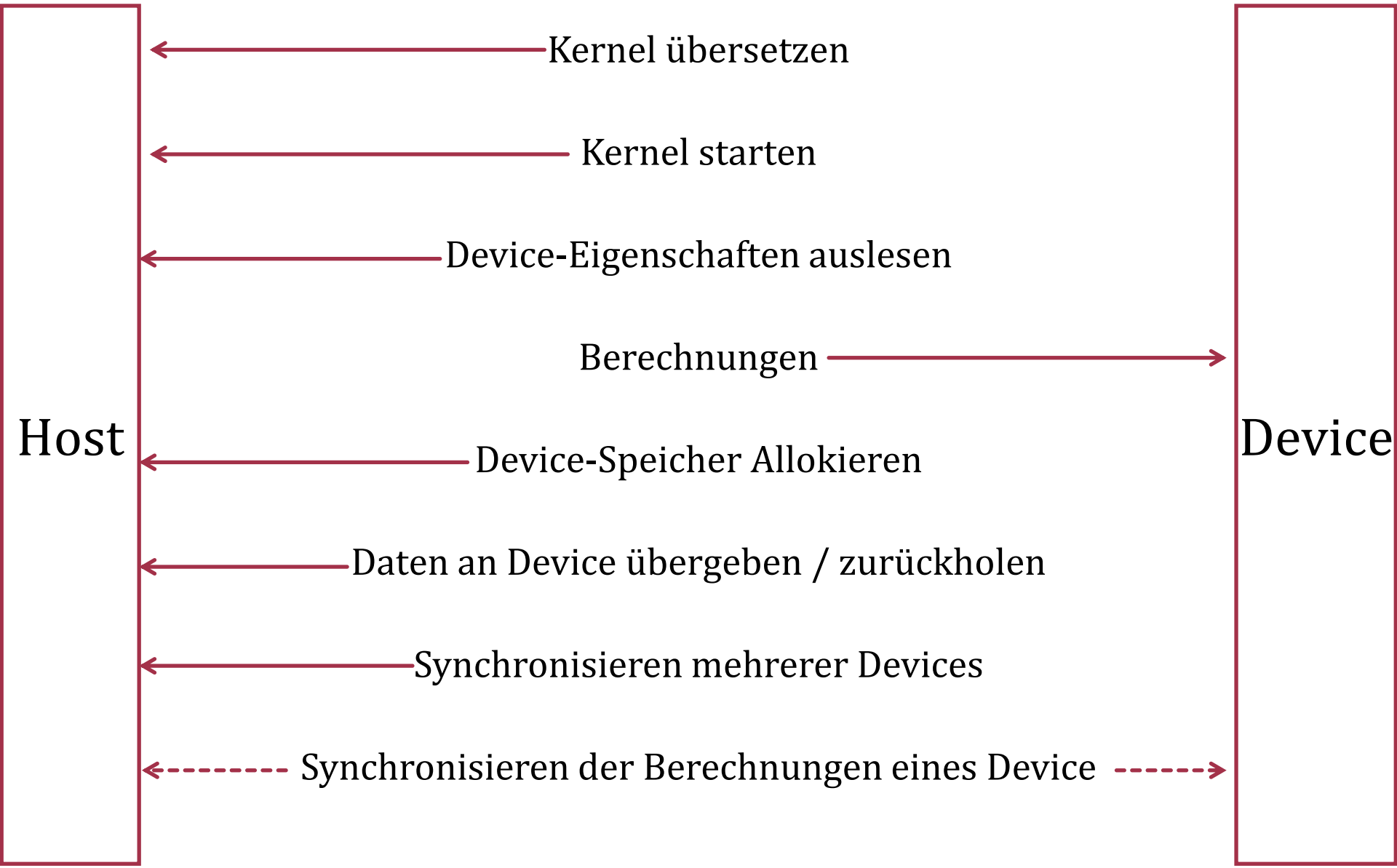
Führt aus

Kernel
(OpenCL C)

```
... // LWJGL Imports
public class HelloOpenCL {
  public static void main(String[] args) {
      ... // Vorbereitungen
      ... // Daten übergeben
      // Berechnung ausführen
      clEnqueueNDRangeKernel(...);
      ... // Ergebnis holen
      ... // Aufräumen
  }
}
```

```
kernel void vec_add(
 global int* a,
 global int* b,
 global int* c){
    int i = get_global_id(0);
    c[i] = a[i] + b[i];
}
```

Wer ist zuständig?



1.1

OpenCL Syntax & LWJGL

OpenCL API Calls

- Ziel: Aus vielen Sprachen ansprechbar
- Befehlsausführung teilweise asynchron
- Eigene Datentypen
- Interne Objekte
  - Zugriff nur über OpenCL API Calls
  - Host erhält nur ID, welche ein Objekt repräsentiert
  - Muss als Argument übergeben werden
  - Oft gekapselt (Struct, etwa C), oder Java Objekte

OpenCL API Syntax

- Form der Befehle folgt folgendem Schema:
  - `<Rückgabe> cl<Befehlsname>(parameter)`
  - Beginnen immer mit **`cl`**
  - Befehlsname, ggf. Parameter
  - Oft auch Rückgabewert
- Rückgabewert zusätzlich auch über Pointer
- Beispiel:

```
// OpenCL API-Call clCreateKernel...
//   ... erzeugt (internes) Kernel Objekt.
// Liefert Datenstruktur cl_kernel, welche dieses Objekt
//   im Host Code repräsentiert und Zugriff darauf ermöglicht.
cl_kernel clCreateKernel(
   cl_program program,       // Übersetztes Program Object
   const char *kernel_name, // Name des Kernels im OpenCL C Code
   cl_int *errcode_ret       // „Rückgabe“ für Fehler
)
```

Frage an CG2012-Experten

Unterschiede zur OpenGL Syntax?

- Keine State-Machiene
- Rückgaben möglich
- Überladen möglich

LWJGL

- Bietet Java über Klassen aus `org.lwjgl.opencl` Zugriff auf native OpenCL Funktionen
- Klasse `CL` initialisiert OpenCL mit `CL.create()`
  - Keine Entsprechung in der C API
- Klasse `CL10` kapselt OpenCL 1.0 Funktionalität
  - Statische Felder repräsentieren OpenCL 1.0 Konstanten
  - Statische Methoden repräsentieren OpenCL 1.0 Funktionen…
  - … und rufen diese via JNI calls auf
- Für OpenCL 1.1 zusätzlich Klasse CL11, usw.
- Eigene Klassen zur Repräsentation OpenCL spezifischer Datenstrukturen und Pointer
- Außerdem abstraktere Hilfsklassen

Vergleich mit „reiner“ OpenCL API

OpenCL API

```
cl_kernel clCreateKernel(
    cl_program    program,
    const char   *kernel_name,
    cl_int       *errcode_ret
)
```

LWJGL

```
CLKernel clCreateKernel(
    CLProgram    program,
    String       kernel_name,
    java.nio.IntBuffer errcode_ret
)
```

(Pointer auf) Structs → Java Objekte

Pointer auf Char → String

Pointer → Von java.nio.Buffer abgeleitete Klassen:
IntBuffer, ByteBuffer, DoubleBuffer, …

Pointer → org.lwjgl.PointerBuffer

Hinweis: Argumentanzahl oft
nicht gesondert zu übergeben

1.2

HelloOpenCL - Vektoraddition

OpenCL mit LWJGL initialisieren

```
import org.lwjgl.opencl.CL;
import static org.lwjgl.opencl.CL10.*;
import java.nio.IntBuffer;
import java.util.List;
import org.lwjgl.BufferUtils;
import org.lwjgl.PointerBuffer;
import org.lwjgl.opencl.CLCommandQueue;
import org.lwjgl.opencl.CLContext;
import org.lwjgl.opencl.CLDevice;
import org.lwjgl.opencl.CLKernel;
import org.lwjgl.opencl.CLMem;
import org.lwjgl.opencl.CLPlatform;
import org.lwjgl.opencl.CLProgram;

public class HelloOpenCL {
   private static final int SIZE_OF_INT = 4; // Größe des Datentyps int in Bytes

   public static void main(String[] args) throws Exception {

      CL.create();    // Initialisiere OpenCL

      /*  Hier kann der Code der folgenden Folien ausgeführt werden  */

      CL.destroy();
   }
}
```

Platform

- Repräsentiert OpenCL-Implementation eines Anbieters
- Interface zum restlichen Programm
- Genau einem Host zugeordnet
- Host kann mehrere Platforms verwalten
- Enthält mindestens ein Device
- Platforms in unserem Beispiel:
  1. Nvidia, im Grafikkartentreiber enthalten
  2. AMD APP Sdk, x86 CPUs
  - In beiden Fällen ist der Intel Core 2 Host
  - Beide Platforms haben keinen Zugriff auf Device der anderen Platform
- Verschiedene Platforms kommunizieren über Host

Initialisieren einer Platform

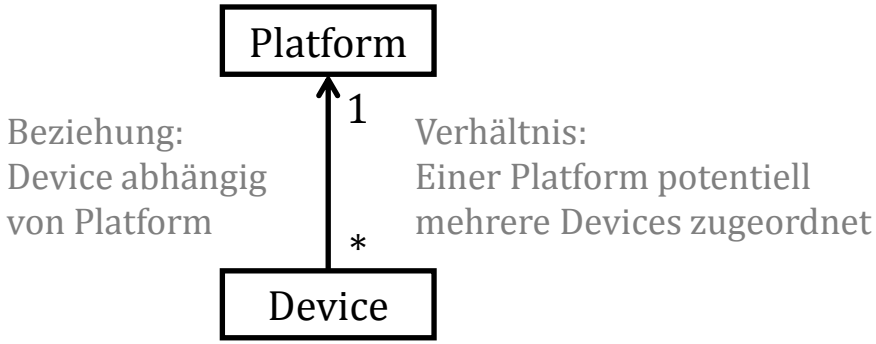
- Durch LWJGL-Klasse CLPlatform repräsentiert
- Erinnerung: Code eingebettet in Java Klasse HelloOpenCL (Folie 15)

```
// Verfügbare Platformen liefert in LWJGL:
List<CLPlatform> platforms = CLPlatform.getPlatforms();

// Wir nehmen – hier – die Erste
CLPlatform platform = platforms.get(0);
```

Hinweis: Funktioniert ohne LWJGL etwas anders:
`clGetPlatformID` liefert Pointer auf Structs des Typs `cl_platform_id`

Device

- Repräsentiert Gerät zur Ausführung paralleler Berechnungen
- Gehört zu einer Platform
- Platform kann Devices verschiedenen Typs enthalten
- Beispiel AMD:
  - GPU
  - CPU



Initialisieren der Devices

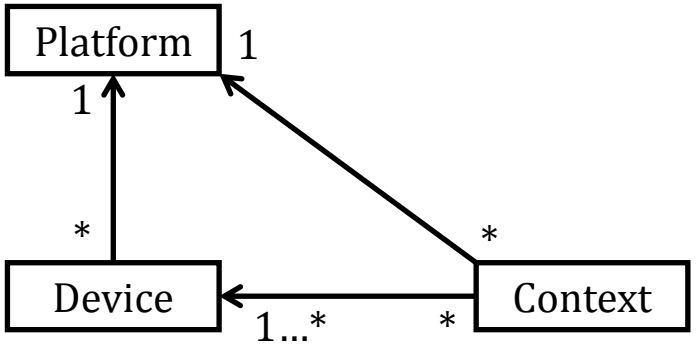
- Device durch LWJGL-Klasse CLDevice repräsentiert
- Methode `getDevices(int deviceType)` einer Instanz der Klasse `CLPlatform` liefert `CLDevice` Objekte der Platform
- Werte für `deviceType`:
  - `CL_DEVICE_TYPE_CPU` `// CPU. (Host, laut Spec (?))`
  - `CL_DEVICE_TYPE_ACCELERATOR` `// Spezialisierte, v. Host getrennte, // Prozessoren`
  - `CL_DEVICE_TYPE_GPU` `// Unterstützt zusätzlich OpenGL / DX`
  - `CL_DEVICE_TYPE_ALL`
  - `...`

```
// Gegeben: LWJGL-Objekt platform (Folie 17)

// Beispiel: Liefere Devices des Typs CPU der Platform platform
List<CLDevice> devices = platform.getDevices(CL_DEVICE_TYPE_CPU);

// Wir nehmen – hier - das Erste
CLDevice device = devices.get(0);
```

Ohne LWJGL: **`clGetDeviceIDs`**

Context

- Abstrakter Container
- Definiert gemeinsame Umgebung…
- … innerhalb einer Plattform mit einigen ihrer Devices…
- … für:
  - Memory Objects
  - Program Objects
  - Kernels
  - Queues



Initialisieren eines Context

- Durch LWJGL-Klasse CLContext repräsentiert
- Instanz liefert Methode…

```
CLContext create(
    CLPlatform platform,      // Eine Platform
    List<CLDevice> devices,   // Mindestens ein Device
    IntBuffer errcode_ret     // Rückgabe v. Fehlerkonst.
)
```

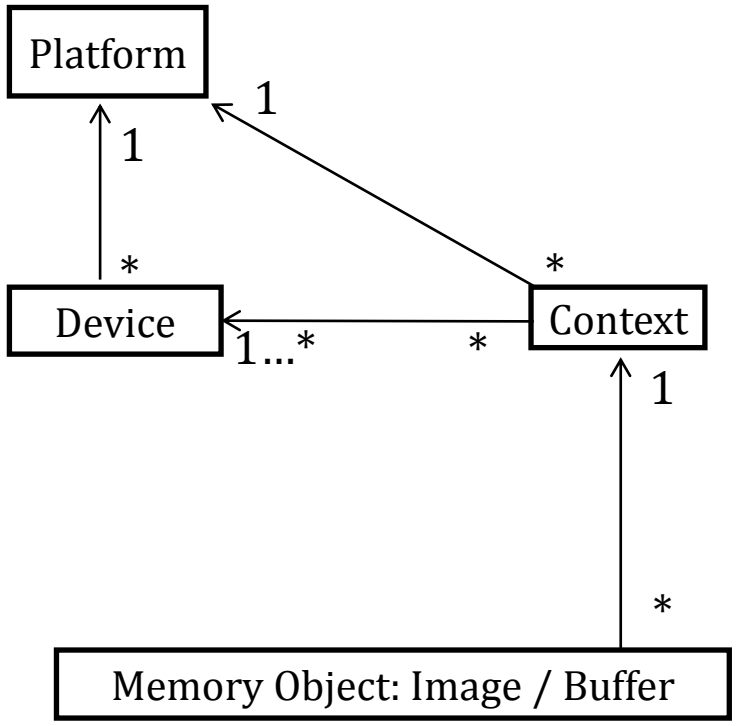
…. der LWJGL-Klasse `CLContext`

```
// Gegeben: LWJGL-Objekte platform (Folie 17) und devices (F. 19)
// Hinweise: - Fehlerkontrolle behandeln wir später
//           - Es existieren weitere Varianten des Befehls
CLContext context = CLContext.create(platform, devices, null);
```

Ohne LWJGL: **`clCreateContext`**

Memory Objects

- Viele parallele Algorithmen in der Praxis speicherbandbreiten-limitiert
- Ziel: Daten möglichst physisch nah an Devices speichern
- Nicht immer möglich / sinnvoll
- Dafür Memory Objects durch OpenCL angeboten
  - Buffer: Linearer Speicher
  - Image (optional): Optimiert für 1, 2 & 3-D Lesezugriff
  - Sind einem Context zugeordnet



Initialisieren eines OpenCL-Buffers

- Durch LWJGL-Klasse CLMem repräsentiert
- Instanz davon liefert OpenCL Funktion:

```
CLMem clCreateBuffer(
      CLContext context,      // Context, zu dem Daten gehören sollen
      long flags,             // Optionen: Speicherort, R/W-Zugriffe

      IntBuffer host_ptr,     // Daten im Host-Speicher; Verwende deren Größe
Oder: long host_ptr_size      // Größe anzulegender (leerer) Datenstruktur
                              //   in Byte

      IntBuffer errcode_ret   // Rückgabe einer Fehlerkonstante
);
```

- Einige Konstanten für `flags`:
  - `CL_MEM_COPY_HOST_PTR` // Kopiere Daten auf Device(s)
  - `CL_MEM_USE_HOST_PTR` // Verwende stattdessen Host-Speicher
  - `CL_MEM_READ_ONLY`
  - `CL_MEM_WRITE_ONLY`
  - `CL_MEM_READ_WRITE` // Default (Übergabe von 0, also kein Bit gesetzt)
  - Können bitweise kombiniert werden
  - Nicht immer alle (Kombinationen) erlaubt

Beispiel: Buffer mit Daten des Host

```
// Gegeben: LWJGL-Objekt context (Folie 21)

// Daten im Host Speicher:
IntBuffer hostA <- {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

// Erzeuge CLMem Objekt „a“, reserviere Devicespeicher & kopiere Daten
CLMem a = clCreateBuffer(
   context,        // Context, für dessen assoziierte Devices der
                   //   Buffer erzeugt wird

   CL_MEM_COPY_HOST_PTR    // Daten kopieren...
   | CL_MEM_READ_ONLY,     // ... und kopierte Daten nur lesbar

   hostA,          // Reserviert in Devices Platz für
                   //   lwjgl_Buffer.capacity() ints
                   // Kopiert dann die Daten aus hostA

   null            // Keine Fehlerbehandlung heute
);
```

Beispiel 2: Initial leerer Buffer

```
// Gegeben: LWJGL-Objekt context (Folie 21)

// Daten im Host Speicher:
IntBuffer cHost <- {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

// Erzeuge CLMem Objekt „c“ & reserviere Device-Speicher
CLMem c = clCreateBuffer(
   context,        // Context, für dessen assoziierte Devices der
                   //   Buffer erzeugt wird

   0,              // Kein Bit gesetzt, also:
                   //   Allokiere Device Memory   (~~CL_MEM_USE_HOST_PTR~~ )
                   //   Kopiere nichts            (~~CL_MEM_COPY_HOST_PTR~~)
                   //   Daten les- und schreibbar (default)

   cHost.capacity() * SIZE_OF_INT,  // Allokiert Platz für
                                    //   cHost.capacity() Ints
   null            // Keine Fehlerbehandlung heute
);
```

Anmerkungen

- Sprachen
  - Java / LWJGL: In Vorlesungsteil zwingend & ausreichend
  - Keine speziellen Fragen dazu in Testaten
  - C++, Fortran, etc. im Projektteil denkbar
  - Hinweise darauf in Vorlesung können ignoriert werden
- APIs
  - OpenCL: Im Vorlesungsteil zwingend & ausreichend
  - CUDA, MPI, etc. im Projektteil denkbar