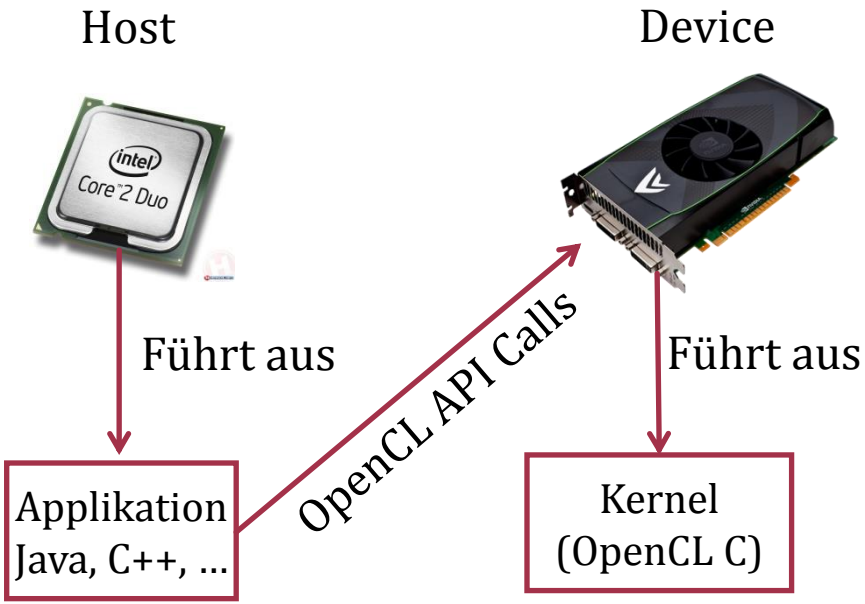
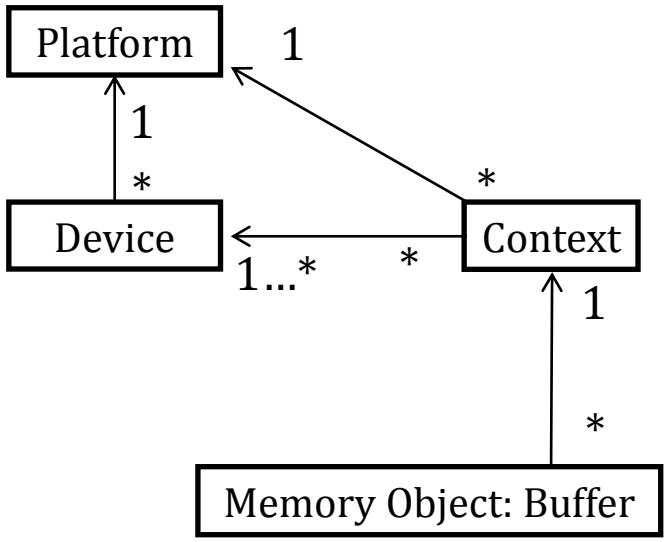
Parallele Algorithmen mit OpenCL

Universität Osnabrück, Henning Wenke, 2013-04-24

Was bisher geschah

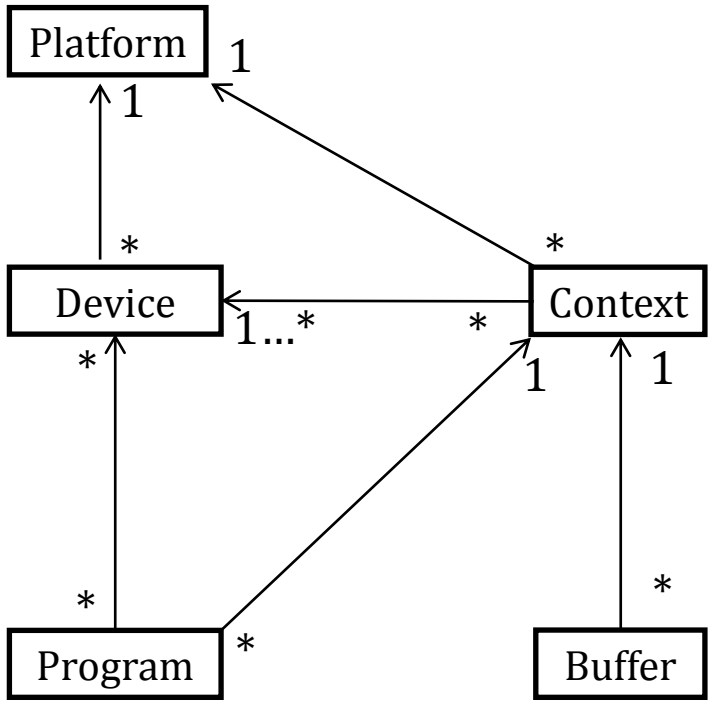





```
kernel void vec_add(
 global int* a,
 global int* b,
 global int* c){
    int i = get_global_id(0);
    c[i] = a[i] + b[i];
}
```

Program

- OpenCL C Code mit mindestens einer Kernel-Funktion
- Genau einem Context…
- … und beim „builden“ mindestens einem Device daraus zugeordnet
- Wird zur Laufzeit durch OpenCL API calls übersetzt
- Enthält optional, z.B.:
  - Funktionen
  - Konstanten
  - Verwendbar in allen Kernels des Program Objects



Erzeugen eines Program Objects

- Durch LWJGL-Klasse CLProgram repräsentiert
- Typischerweise basierend auf Source-Code
- Instanz liefert dann OpenCL Funktion:

```
CLProgram clCreateProgramWithSource(
    CLContext context,     // Context, mit dem Program Object
                           //  assoziiert sein soll
    String string,         // OpenCL C Sourcecode
    IntBuffer errcode_ret // Rückgabe einer Fehlerkonstante
)
```

Beispiel: Program mit Kernel `vec_add`

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


// Erzeuge Program Object 'program' aus OpenCL C Code
CLProgram program = clCreateProgramWithSource(
   context,          // Unser Context 'context'
                     //  (Folie 21, letzte Woche)

   programSource,    // Siehe oben

   null              // Keine Fehlerbehandlung heute
);
```

Kompilieren und Linken eines Programs

- Funktion `clBuildProgram` kompiliert und linkt Program Object für Teilmenge der Devices des zugehörigen Context
- Kann asynchron ausgeführt werden

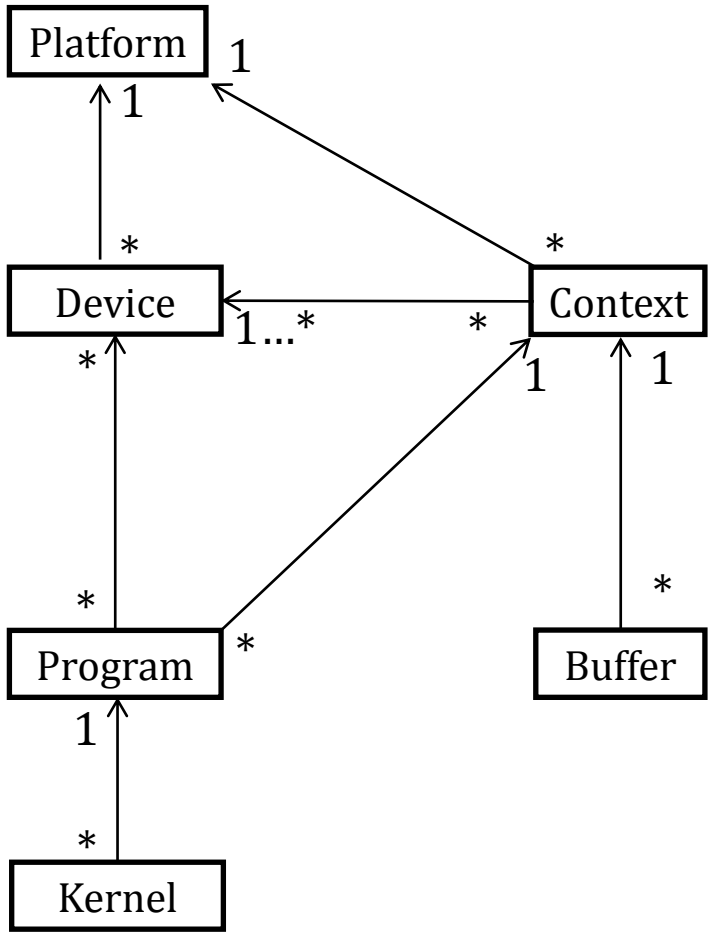
```
int clBuildProgram(                          // Rückgabe: Fehlerkonstante
   CLProgram program,                        // Zu „buildendes" Program Object

       CLDevice device                       // Ein Device,
Oder:  PointerBuffer device_list,            // IDs mehrerer Devices des
                                             // 'program' zugeordneten Context

   String options,                           // Z.B. Optimierungen
   CLBuildProgramCallback pfn_notify         // Zur asynchronen Ausführung
                                             //  Diese Veranstaltung: Synchron
                                             //  ausführen (null übergeben)
)
```

Beispiel

```
// Übersetze und linke Program Object 'program' (Folie 5)
//  für unser Device 'device'
clBuildProgram(   // Keine Fehlerbehandlung heute
   program,

   device,        // (siehe Folie 19, letzte Woche)

   "",            // Keine Optionen

   null           // Befehl synchron ausführen
);
```

Kernel

- Repräsentiert Funktion für parallele Berechnungen
- Funktion ohne Rückgabe mit Qualifier `kernel`
- Genau einem Program zugeordnet



Erzeugen eines Kernels Objects

- Durch LWJGL-Klasse CLKernel repräsentiert
- Instanz davon liefert OpenCL Funktion `clCreateKernel`

```
CLKernel clCreateKernel(
   CLProgram program,      // Ausführbares Program Objekt,
                           //  welches den Kernel enthält
   String kernel_name,     // Name des Kernels im Sourcecode des
                           //  Program Objects 'program'
   IntBuffer errcode_ret   // Rückgabe einer Fehlerkonstante
)
```

```
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];
}
```

```
// Beispiel: Erzeuge Kernel Object 'kernel'
//  für Vektoraddition
CLKernel kernel = clCreateKernel(
   program,                // Siehe Folien 5 und 7
   "vec_add",              // Name der Kernel-Funktion
   null                    // Keine Fehlerbehandlung heute
);
```

Argumentübergabe an Kernel

- OpenCL Funktion `clSetKernelArg` setzt je ein Argument eines Kernels
- Hinweis: Kernelparameter sind persistent
- Hier nur Auszug übergebbarer Informationen

```
int clSetKernelArg (    // Rückgabe: Fehlerkonstante
   CLKernel kernel,     // Kernel, dessen Argument zu setzen ist

   int arg_index,       // 0: Erstes bis n-1: Letztes Argument

       CLObject arg_value  // Z.B.: Memory Objects
Oder:  java.nio.Buffer,    // Für Konstanten
)
```

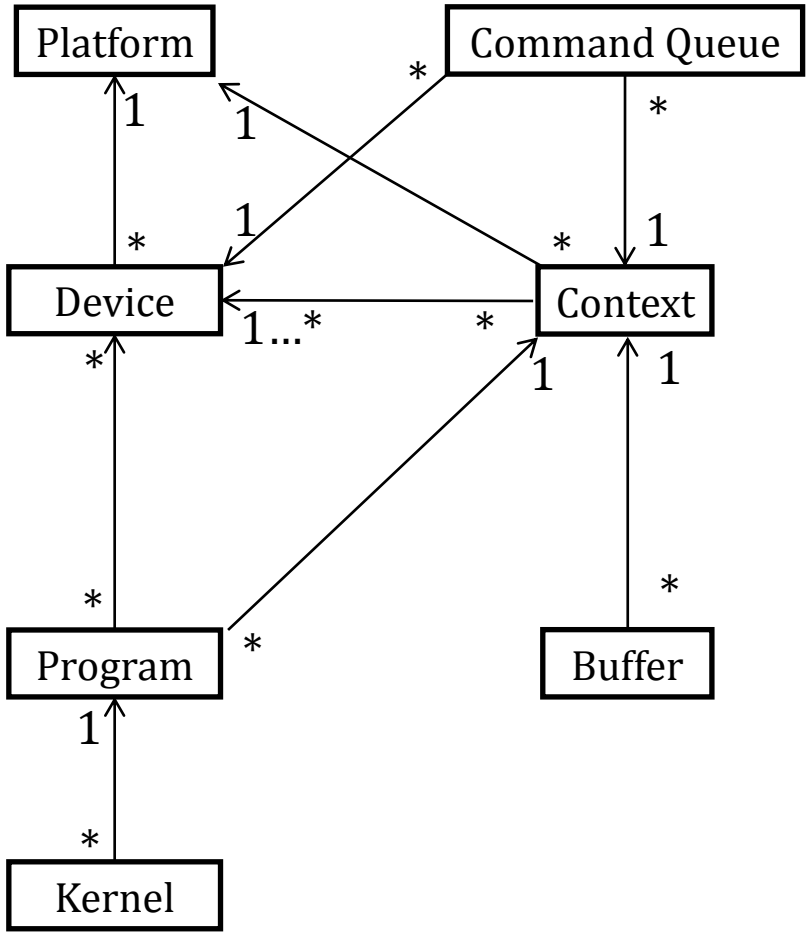
Beispiel

```
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];
}
```

```
// Setze Parameter 'a' des Kernels 'kernel'
clSetKernelArg( // Keine Fehlerbehandlung heute
   kernel,      // Setze Argument des auf Folie 9 erzeugten Kernels
                //  'kernel'

   0,           // Betrifft erstes Argument

   a            // Setze Buffer Object 'a' (Folie 24, letzte Woche)
);

// Beispiel 2: Setze Buffer 'c' (Folie 25, letzte Woche)
clSetKernelArg(kernel, 2, c);
```

Command Queue

- Regelt Host-Device Kommunikation
- Für bestimmten Context & Device erzeugt
- Drei Arten Befehle einhängbar
  - Ausführung der Kernel
  - Datenoperationen
  - Synchronisation
- Ausführung asynchron zu Host / anderen Queues
- Optional: Out-Of-Order Execution



Erzeugen einer Command Queue

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

```
CLCommandQueue clCreateCommandQueue(
   CLContext context,          // Context, mit welchem Command Queue
                               //  assoziiert sein soll
   CLDevice device,            // Ein Device des
                               //  zugeordneten Contexts 'context'
   long properties,            // CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE,
                               //  CL_QUEUE_PROFILING_ENABLE
   IntBuffer errcode_ret       // Rückgabe einer Fehlerkonstante
)
```

```
// Beispiel: Initialisiere Command Queue 'queue'
//  für unsere Konfiguration
CLCommandQueue queue = clCreateCommandQueue(
   context, device,  // Siehe Folien 19 & 21 (letzte Woche)
   0,                // In-order & kein Profiling
   null              // Heute keine Fehlerbehandlung
);
```

Kernel Ausführung

- OpenCL Kernel werden zur Ausführung in Command Queue eingehängt
- Achtung: Wird immer asynchron ausgeführt
- Vereinfacht, bis wir das Parallelitätsmodell besprechen

```
int clEnqueueNDRangeKernel(          // Rückgabe: Fehlerkonstante
  CLCommandQueue command_queue,      // Queue, in die eingeh. w. soll
  CLKernel kernel,                   // Einzuhängender Kernel
  int work_dim,                      // Dimension der Kernel-Indizierung
  PointerBuffer global_work_offset,  // Index-Offset pro Dimension
                                     //  null: Index beginnt mit 0
  PointerBuffer global_work_size,    // Globale Anzahl Work Items pro Dim
                                     //  ~ "parfor Schleifenlänge"
  PointerBuffer local_work_size,     // Kapitel: Parallelitätsmodell
                                     //  null: OpenCL entscheiden lassen
  PointerBuffer event_wait_list,     // Events: später.
  PointerBuffer event                //  null: Keine Events
)
```

Beispiel

```
int indexDim = 1;                  // Verwende eindimensionale Indizierung

// Erzeuge LWJGL-PointerBuffer mit einer Komponente
PointerBuffer compCnt = new PointerBuffer(indexDim);
compCnt.put(0, hostA.capacity());

// Hänge unseren Kernel 'kernel' (Folie 9) zur Ausführung in Command
// Queue 'queue' (Folie 13) ein.
clEnqueueNDRangeKernel(            // Heute keine Fehlerbehandlung
  queue, kernel,
  indexDim,                        // 1-Dimensionale Indizierung
  null,                            // Index beginnt mit 0
  compCnt,                         // Vektorkomponentenanzahl
                                   //  ~ "parfor Schleifenlänge"

  null,                            // OpenCL lokale Parallelität
                                   //  festlegen lassen
  null, null                       // Keine Events
);
```

Auswirkungen auf diesen 1D-Kernel

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

clSetKernelArg

```
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];
}
```

```
clEnqueueNDRangeKernel(...,
 PointerBuffer global_work_size,
 PointerBuffer global_work_offset,
...)
```

- Es werden `global_work_size` Instanzen des Kernels erzeugt
- Verarbeitung potentiell parallel
- Globale Daten in allen Instanzen sichtbar
- Für jede Instanz liefert `get_global_id(0)` einen Index
- Beginnt bei: `global_work_offset`
- Endet bei: `global_work_offset + global_work_size - 1`
- Hinweis: Gilt für weitere Indexdimensionen analog

Daten zurück zum Host kopieren

- Kopiert lineare Daten aus Buffer Object in Host-Datenstruktur
- Wird in Command Queue eingehängt und ggf. asynchron ausgeführt
- Command Queue und Buffer müssen gleichem Context zugeordnet sein
- Hinweis: Schreiben analog mit **`clEnqueueWriteBuffer`**

```
int clEnqueueReadBuffer(          // Rückgabe: Fehlerkonstante
   CLCommandQueue command_queue, // Queue, in die eingehängt w. soll
   CLMem buffer,          // Quelle-Daten: Buffer Object

   int blocking_read,     // CL_TRUE: Führe Befehl synchron aus
                          // CL_FALSE: Führe Befehl asynchron aus

   long offset,           // Erster Wert in 'buffer', angegeben in Bytes

   java.nio.Buffer ptr, // Ziel-Datenstruktur des Host

   PointerBuffer event_wait_list, PointerBuffer event // Später...
)
```

Beispiel

```
// Lies Buffer Object 'c' zurück in Host Datenstruktur
//  java.nio.IntBuffer 'cHost'
clEnqueueReadBuffer(
   queue,                  // Folie 13

   c,                      // 'c' definiert auf Folie 25, letzte Woche

   CL_FALSE,               // Führe Befehl asynchron aus

   0,                      // Von Anfang an kopieren. Außerdem: Bis Ende.
                           //  (diesem Code nicht entnehmbar)

   cHost,                  // Host-Datenstruktur, Folie 25, letzte Woche

   null, null              // Keine Events
);
```

Command Queue Abarbeitung erzwingen

- `clFinish` blockiert, bis alle in `command_queue` eingehängten Befehle abgearbeitet sind
- "Live-Beispiel"
- Warum bei "blocking_read" `clFinish` in gezeigten Beispiel unnötig?

```
int clFinish(       // Rückgabe: Fehlerkonstante
    CLCommandQueue command_queue
)
```