Bauhaus-Universität Weimar

Parallele und Verteilte Systeme

Grundlagen zur Programmierung von Mehrkern-Systemen, verteilter Berechnung und massiver Parallelität

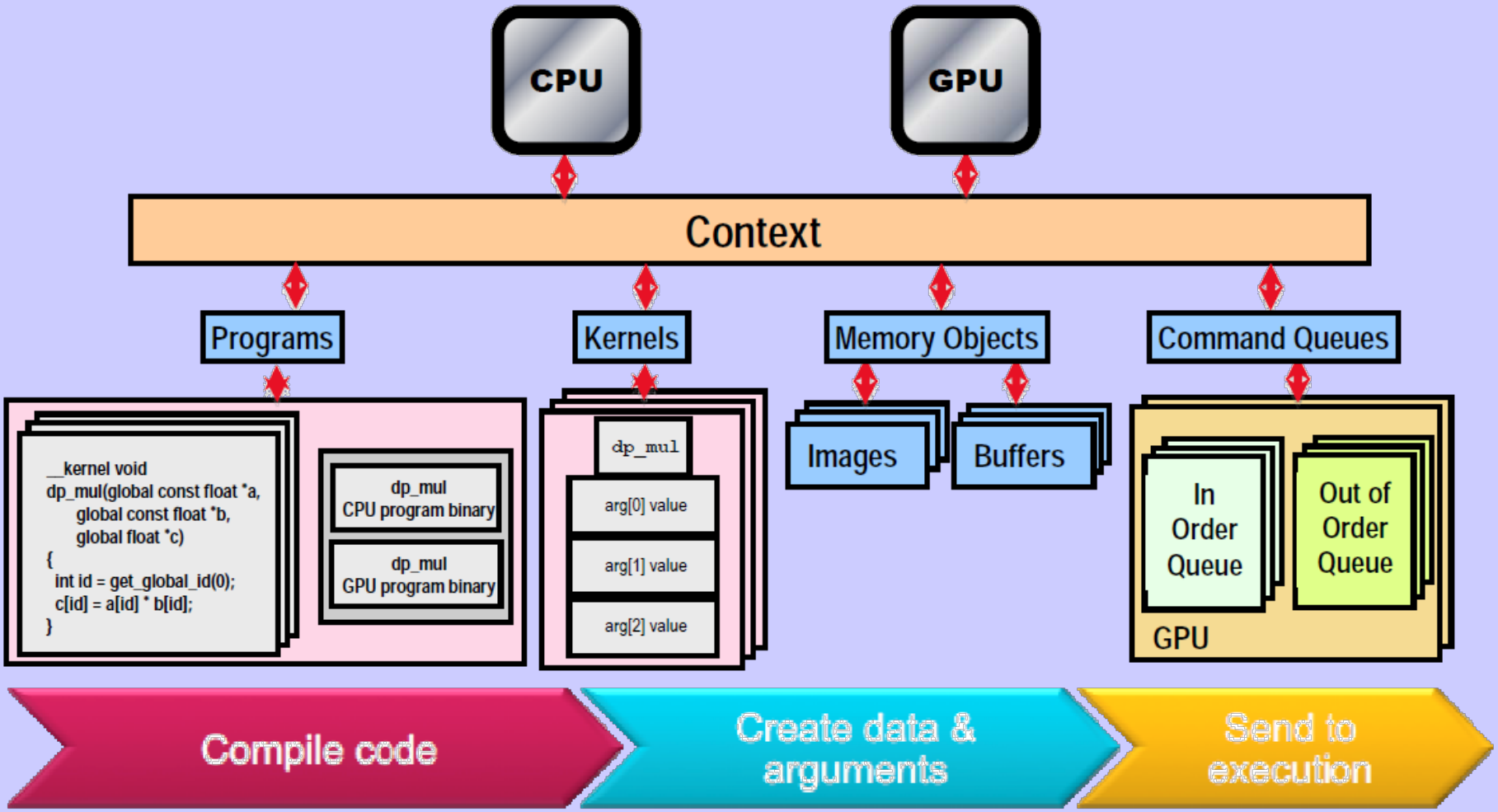
Vorlesung WS 2017/18

(Unterlagen nur für den internen Gebrauch!)

Übersicht

- **Massiv parallele Programmierung**
  - Einführung in OpenCL
    - Plattform-Modell
    - Ausführungs-Modell
    - Speicher-Modell
    - Programmierungs-Modell
  - Praktische Hinweise
    - Geräteinformationen besser abfragen
    - Parameterübergabe
    - Zeitmessung
    - OpenCL Compilermeldungen
    - Watchdog Timer unter Windows

Programmierung mit OpenCL

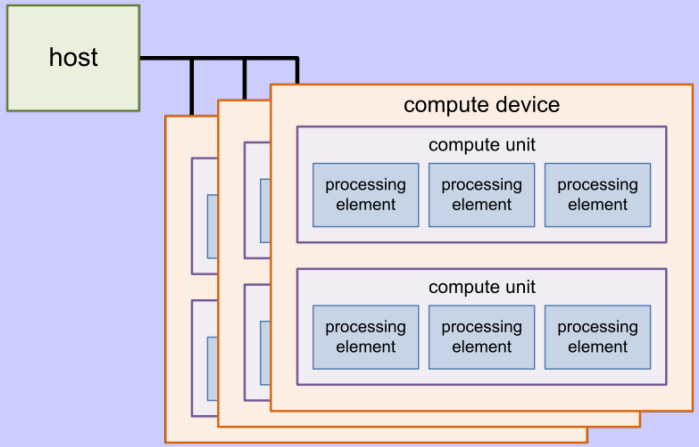


Grundlagen von OpenCL

- **Plattformmodell:**
  Eine abstrakte Beschreibung von heterogenen Systemen mit unterschiedlichen physikalischen Eigenschaften
- **Ausführungsmodell:**
  Eine abstrakte Beschreibung, wie Befehle auf solch heterogenen Plattformen ausgeführt werden
- **Speichermodell:**
  Die unterschiedlichen Speichertypen und wie sie während der Berechnung verwendet werden können
- **Programmiermodell:**
  Die Abstraktion, die ein Programmierer verwenden kann, wenn er Algorithmen entwirft

1. Plattform-Modell

- **Host**
  - **Rechner**, der die "Compute Devices" verwaltet
  - Verteilt die Arbeit an die Devices
- **Compute Device**
  - Genutzte Rechenressource
    (z.B. **Grafikkarte**, Prozessor, Cell-Blade)
- **Compute Unit**
  - Zusammenschluss von „Ausführenden Elementen"
    (z.B. **Shader**, Rechenkern, Coprozessoren)
- **Processing Element**
  - Eigentliches **Rechenelement**

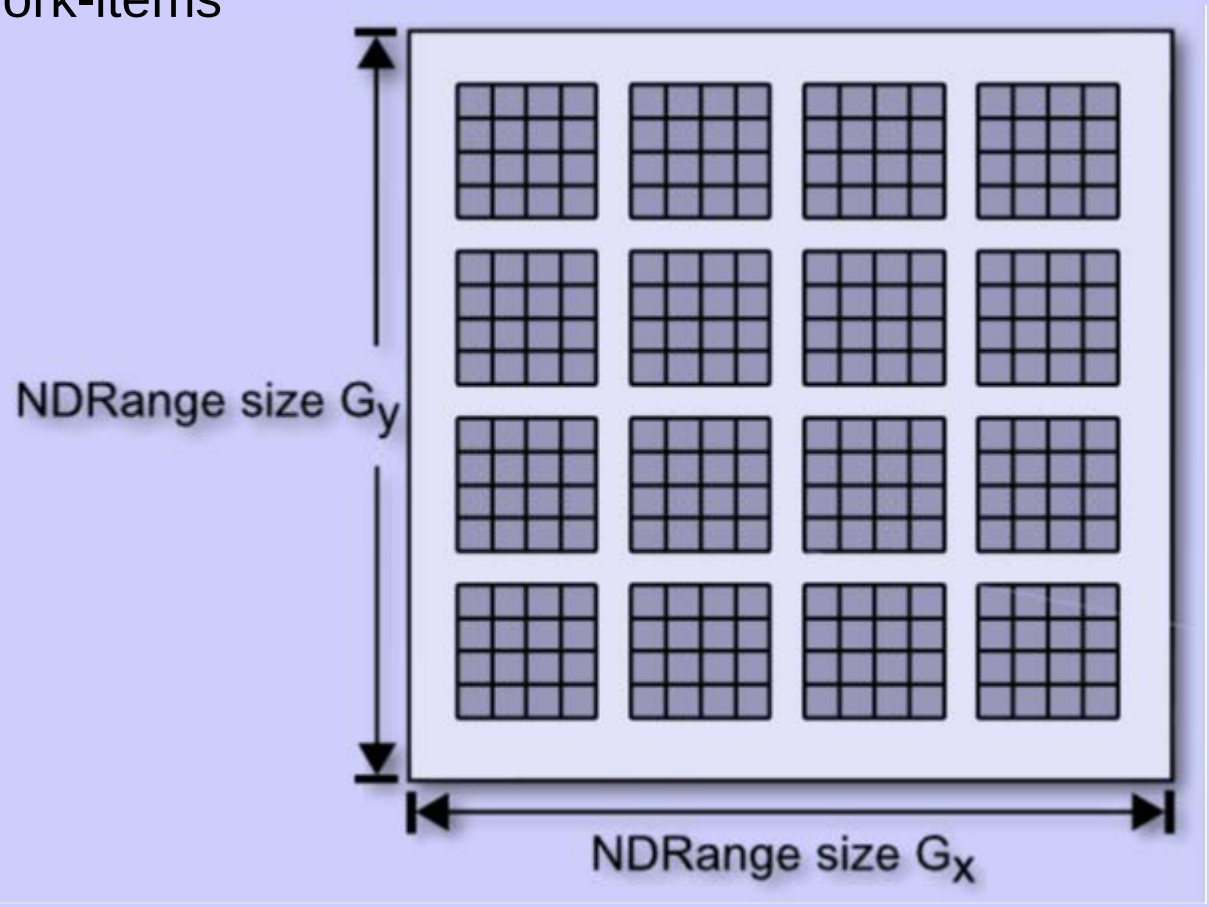


2. Ausführungs-Modell

- **Bestandteile eines OpenCL Programms**
  - Serieller **Host**-Code in C/C++
    (Initialisierung, Speicherverwaltung, etc.)
  - Paralleler **Kernel**-Code in OpenCL-C
    (wird auf dem Device ausgeführt)
- **Kernel Ausführung**
  - Hostprogramm ruft Kernel auf einem **Indexraum** mit 1-3 Dimensionen auf
  - Eine Instanz eines Kernels ist ein (logisches) **Work-item**
  - Einzelne Work-items sind in **Work-groups** zusammengefasst
- **Befehlswarteschlange**
  - Kernelausführung
  - Speichertransfer
  - Synchronisation

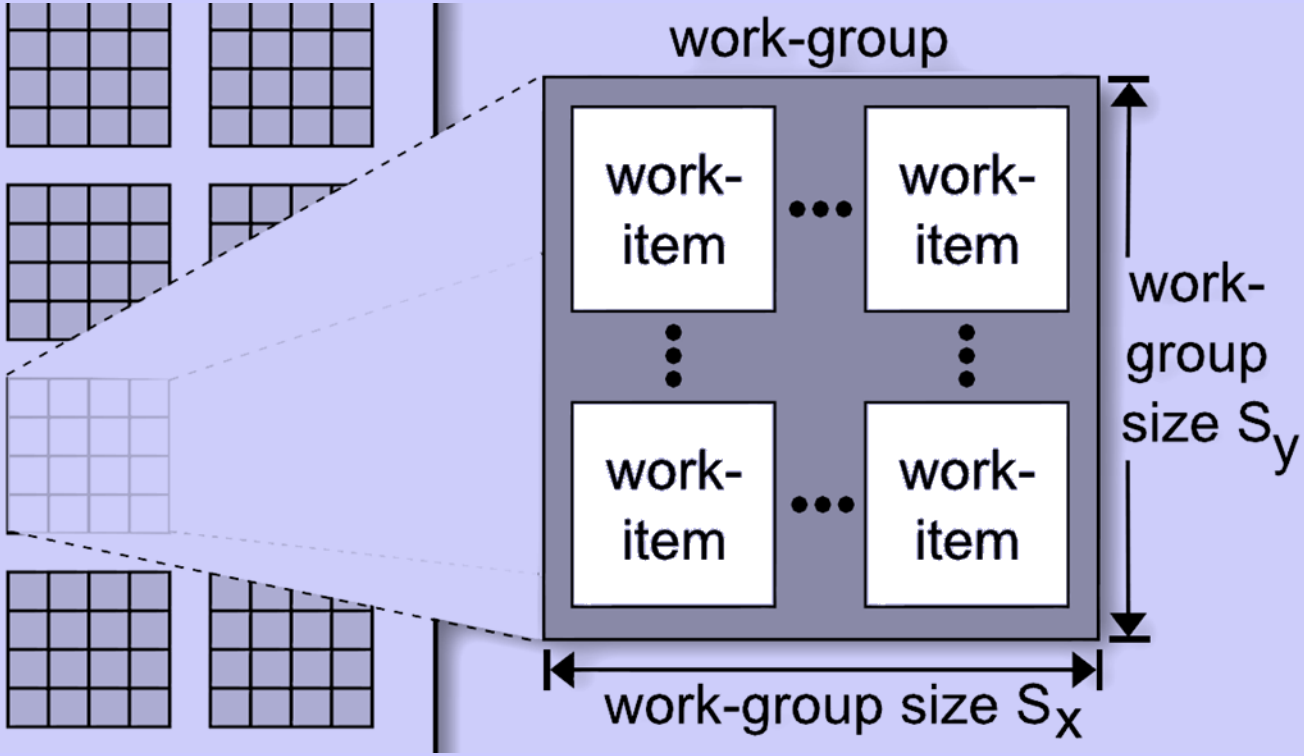
N-dimensionaler Indexraum

- **NDRange**: Aufteilen der gesamten Aufgabe in einen n-dimensionalen Bereich (n = 1,…,3)
- Globale ID eines Work-items

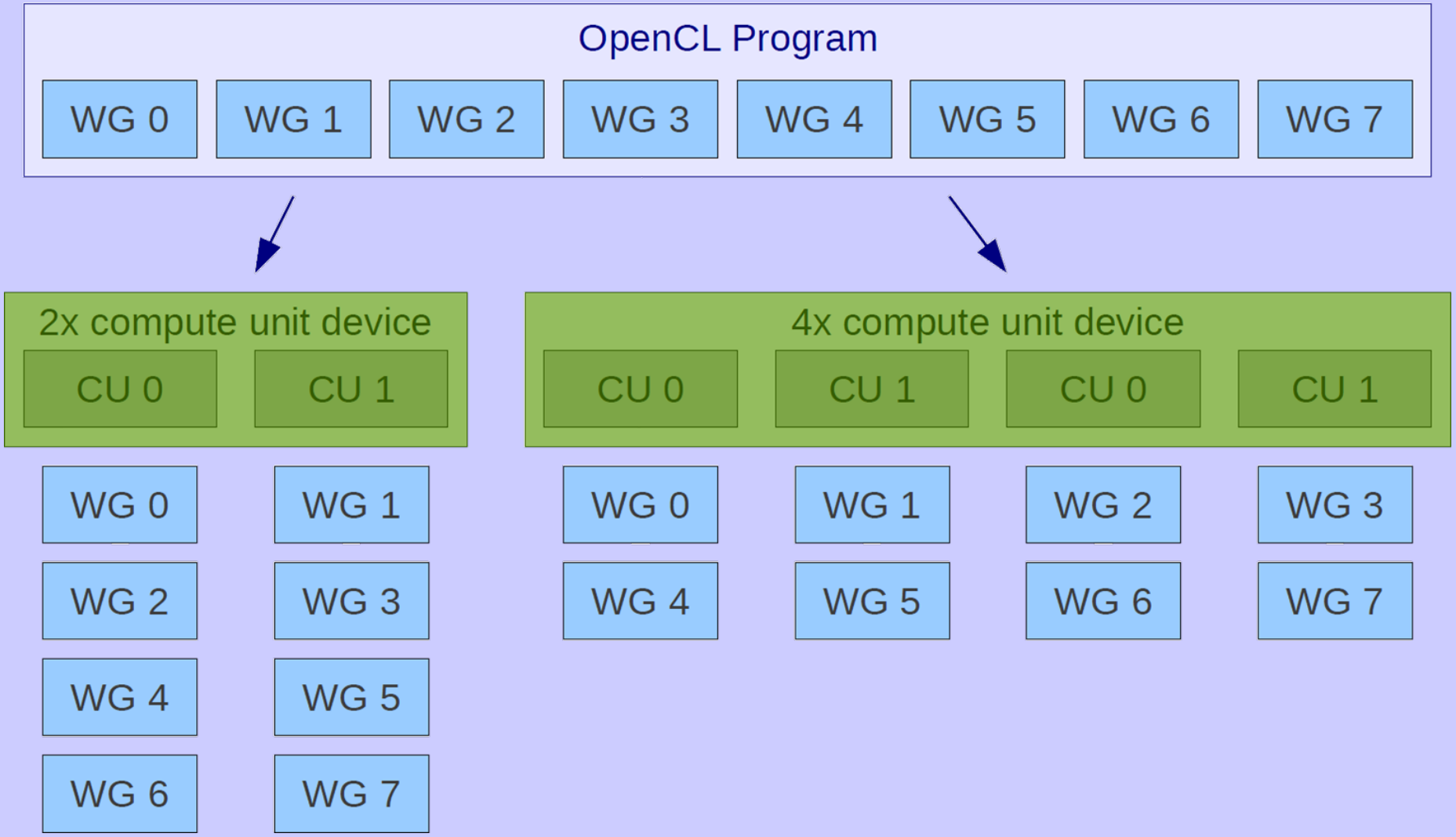


Work-items und Work-groups

- **Work-item**: entspricht einer Kernel-Instanz
- **Work-group**: Zusammenfassung von Work-items mit gemeinsamem Speicher die gleichzeitig Rechnen

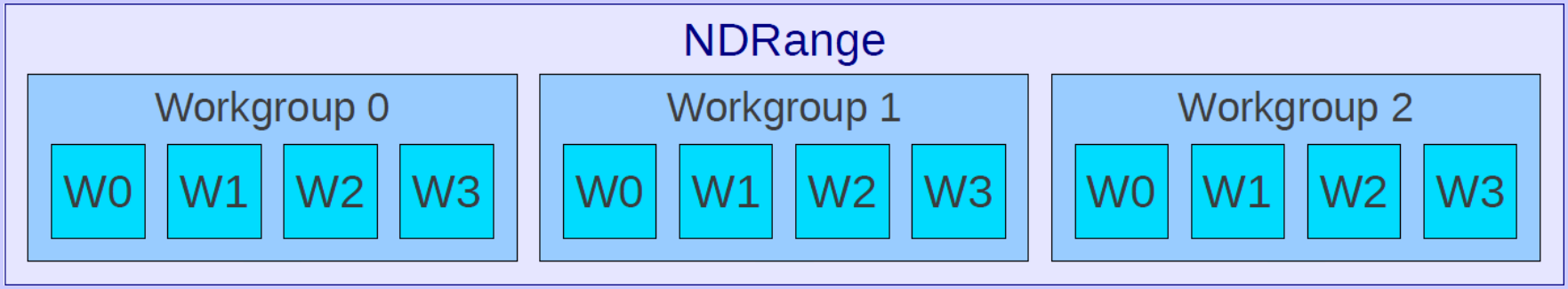


Thread Topologie



Thread Topologie

- OpenCL verwendet ein skalierbares Programmiermodell

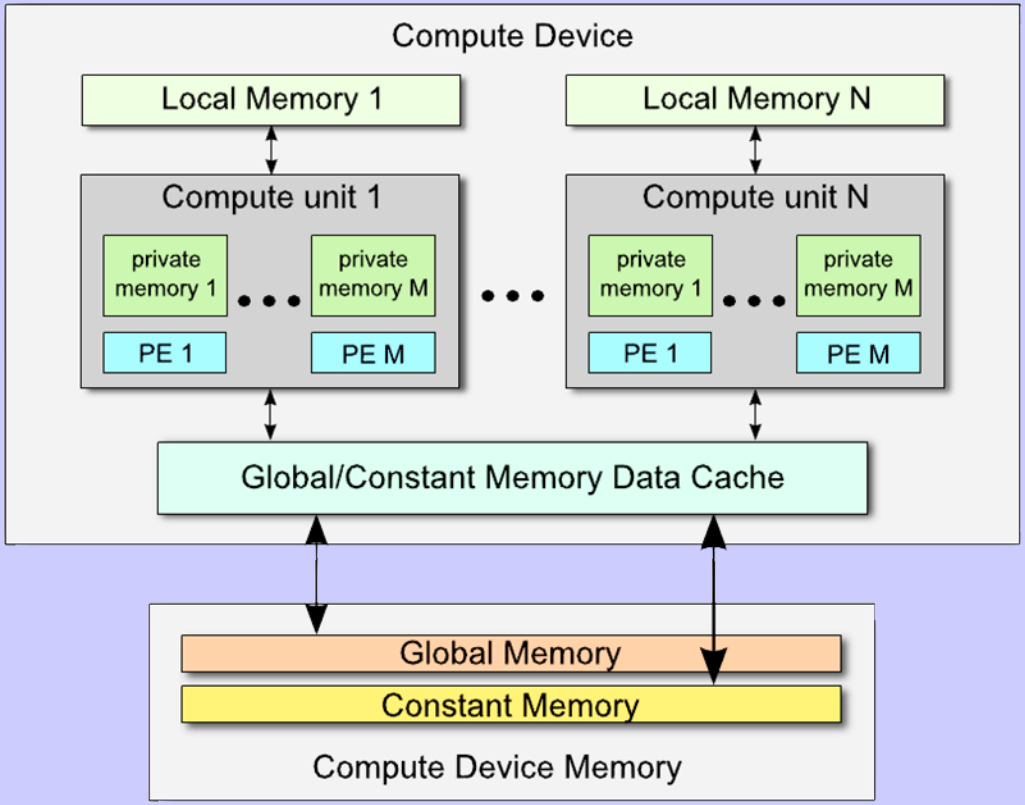


- Jedes Work-item kann die **Dimension** des NDRange, der Work-Group und seinen eigenen **Index** abfragen:

```
uint get_work_dim()
size_t get_global_size(uint d)
size_t get_global_id(uint d)
size_t get_local_size(uint d)
size_t get_local_id(uint d)
```

3. Speichermodell

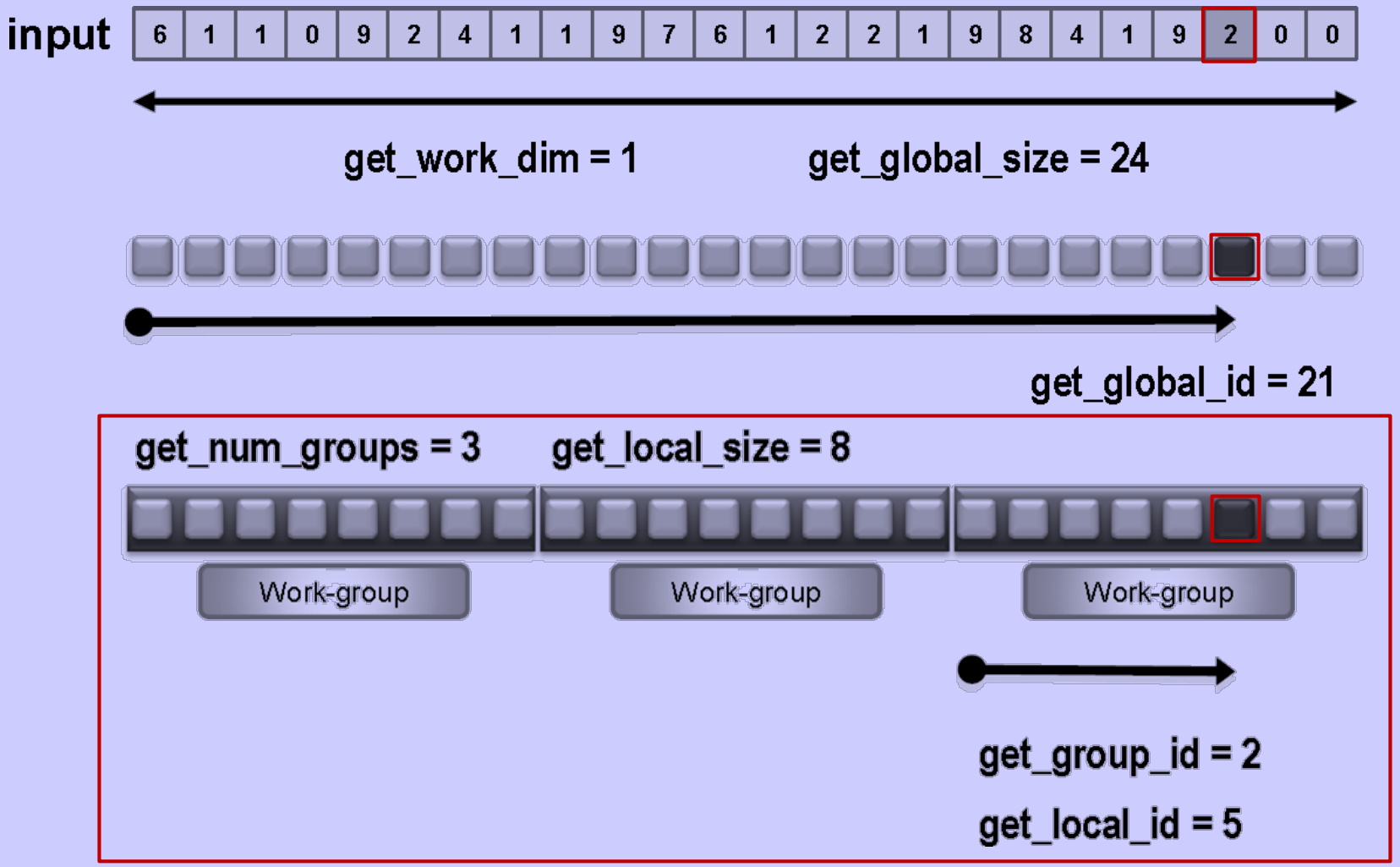
- **Adressräume**
  - **Global:** Arbeitsspeicher des Compute Devices. Von allen Work-items aus allen Work-groups nutzbar
  - **Konstant:** Teil des Global Memory und konstant während der Laufzeit
  - **Daten-Cache:** optional
  - **Lokal:** zu einer Work-group gehörend
  - **Privat:** zu einem Work-item gehörend



Zugriffsrechte für Gerätespeicher

	Global	Constant	Local	Private
Host	Dynamic allocation Read / Write access	Dynamic allocation Read / Write access	Dynamic allocation No access	No allocation No access
Kernel	No allocation Read / Write access	Static allocation Read-only access	Static allocation Read / Write access	Static allocation Read / Write access

Lokaler Speicher und Arbeitsgruppen



4. Programmierungs-Modell

- **Daten-paralleles Modell**
  - Single Instruction Multiple Data (**SIMD**)
  - Single Program Multiple Data (**SPMD**)
  - **Implizites Modell**:
    Der NDRange wird automatisch verteilt
  - **Explizites Modell**:
    Der Programmierer definiert die Größe der Work-groups
- **Task-paralleles Modell**

Praktische Hinweise

Aufruf-Beispiel: Hello.cpp

```
#include "CL/cl.h"
int main(void)
{
  // ...
  clGetPlatformIDs(1, &platform_id, &num_of_platforms);
  clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_GPU, 1, &device_id, &num_of_devices);
  clGetDeviceInfo(device_id, CL_DEVICE_NAME, sizeof(name), name, NULL);
  context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
  command_queue = clCreateCommandQueue(context, device_id, 0, &err);
  program = clCreateProgramWithSource(context, 1, (const char **)&KernelSource, NULL, &err);
  clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
  kernel = clCreateKernel(program, "hello", &err);
  input  = clCreateBuffer(context, CL_MEM_READ_ONLY,  MEM_SIZE, NULL, &err);
  output = clCreateBuffer(context, CL_MEM_WRITE_ONLY, MEM_SIZE, NULL, &err);
  clEnqueueWriteBuffer(command_queue, input, CL_TRUE, 0, MEM_SIZE, data, 0, NULL, NULL);
  clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);
  clSetKernelArg(kernel, 1, sizeof(cl_mem), &output);
  clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, global, NULL, 0, NULL, NULL);
  clFinish(command_queue);
  clEnqueueReadBuffer(command_queue, output, CL_TRUE, 0, MEM_SIZE, results, 0, NULL, NULL);
  // ...
}
```

Problem: Initialisierung der Geräte

```
int main(void)
{
    cl_platform_id platform_id;
    cl_device_id device_id;
    cl_uint pnum = 0, dnum = 0;

    // Plattform ?
    if (clGetPlatformIDs(1, &platform_id, &pnum) != CL_SUCCESS) {
        printf("Unable to get platform id\n"); return 1;
    }
    // GPU Device ?
    if (clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_GPU, 1, &device_id, &dnum) != CL_SUCCESS){
        printf("Unable to get device id\n"); return 1;
    }
    // ...
```

Initialisierung mehrerer Geräte

```
#include <string.h>                                                    // für Zeichenkettenvergleich

int main(void)
{
    cl_uint         pnum = 0, pid = 0, dnum = 0, i;
    cl_platform_id *platforms = NULL;                                  // Leeres Feld für Plattformen
    char            pname[1024];                                                  // Name der Plattform

    if (clGetPlatformIDs(0, NULL, &pnum) != CL_SUCCESS) {              // Anzahl verfügbarer Plattformen
        printf("No platform found.\n"); return 0;
    }
    platforms = (cl_platform_id *)malloc(pnum * sizeof(cl_platform_id)); // Speicher für das Feld. Testen!
    if (clGetPlatformIDs(pnum, platforms, NULL) != CL_SUCCESS) {
        printf("No platform found.\n"); return 0;
    }
    for (i=0; i<pnum; i++) {
        if (clGetPlatformInfo(platforms[i], CL_PLATFORM_NAME, sizeof(pname), pname, NULL) != CL_SUCCESS) {
            printf("Could not get platform information.\n"); return 0;
        }
        if (strstr(pname, "NVIDIA") != NULL) {                     // Vergleich ob Zeichenkette enthalten
            pid = i; break;                                                           // ID merken
        }
    }
    if (clGetDeviceIDs(platforms[pid], CL_DEVICE_TYPE_GPU, 0, NULL, &dnum) != CL_SUCCESS) {// Deviceanzahl
        printf("Could not get device.\n"); return 0;
    }
    // ...
```

Geräteinformation abfragen

```
cl_uint i;
cl_ulong l, a[3] = {0, 0, 0};
char name[48];

clGetDeviceInfo(device_id, CL_DEVICE_NAME, sizeof(name), name, NULL);
printf("Device         : %s\n", name);
clGetDeviceInfo(device_id, CL_DEVICE_MAX_COMPUTE_UNITS, sizeof(i), &i, NULL);
printf("Compute units  : %d\n", i);
clGetDeviceInfo(device_id, CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS, sizeof(i), &i, NULL);
printf("Work item dim  : %d\n", i);
clGetDeviceInfo(device_id, CL_DEVICE_MAX_WORK_ITEM_SIZES, sizeof(a), &a, NULL);
printf("Work item sizes: %d / %d / %d\n", a[0], a[1], a[2]);
clGetDeviceInfo(device_id, CL_DEVICE_MAX_WORK_GROUP_SIZE, sizeof(l), &l, NULL);
printf("Work group size: %d\n", l);
clGetDeviceInfo(device_id, CL_DEVICE_GLOBAL_MEM_SIZE, sizeof(l), &l, NULL);
printf("Global mem size: %d MByte\n", l/1024/1024);
clGetDeviceInfo(device_id, CL_DEVICE_MAX_MEM_ALLOC_SIZE, sizeof(l), &l, NULL);
printf("Max alloc size : %d MByte\n", l/1024/1024);
clGetDeviceInfo(device_id, CL_DEVICE_LOCAL_MEM_SIZE, sizeof(l), &l, NULL);
printf("Local mem size : %d KByte\n", l/1024);
```

Beispiel: Geräteinformation

- **Kapazitäten**

```
Device          : GeForce GTX 260
Compute units   : 24
Work item dim   : 3
Work item sizes: 512 / 512 / 64
Work group size: 512
Global mem size: 1792 MByte
Max alloc size : 448 MByte
Local mem size : 16 KByte
```

- **Abschätzung**

Matrixgröße mit Fließkommazahlen einfacher Genauigkeit, `sizeof(float) = 4` Bytes:

`10.000 x 10.000 x 4 = 400.000.000 =` **`382 MByte`**

Einlesen des Kernel-Quellcodes

```
#define MAX_SOURCE_SIZE (0x100000)  // 1 MB

int main(void)
{
    FILE *fp;
    const char *FileName = "kernel.cl";
    char *KernelSource;

    fp = fopen(FileName, "r");
    if (!fp) {
        printf("Can't open kernel source: %s", FileName); exit(1);
    }
    KernelSource = (char *)malloc(MAX_SOURCE_SIZE);
    if (!KernelSource) {
        printf("Can't allocate kernel"); fclose (fp); exit(1);
    }
    fread(KernelSource, 1, MAX_SOURCE_SIZE, fp);
    fclose(fp);
    // ...
}
```

Dimensionen als Parameter?

```
const char *KernelSource =
"#define DIM 1000                                          // Groesse der Matrix \n"
"__kernel void vecmult(__global float *A, __global float *B, __global float *C) { \n"
"    int i;                                                                       \n"
"    float Al[DIM];                                                               \n"
"    for (i = 0; i < DIM; i++)                                                    \n"
"        //...                                                                    \n"
```

```
const char *KernelSource =
"__kernel void vecmult(__global float *A, __global float *B, __global float *C,   \n"
"                      const int dim) {                      // fuer dim <= 1000! \n"
"    int i;                                                                       \n"
"    float Al[1000];                                                              \n"
"    for (i = 0; i < dim; i++)                                                    \n"
"        //...                                                                    \n"
```

```
int main(void)
{
    cl_int dim = DIM;
    // ...
    clSetKernelArg(kernel, 3, sizeof(cl_int), &dim);
```

Zeitmessung für den Kern

```
#include <omp.h>

double start, end;
start = omp_get_wtime();
//...
end = omp_get_wtime();
printf("time = %f sec\n", end - start);
```

```
cl_event event;
cl_ulong start, end;

command_queue = clCreateCommandQueue(context, device_id, CL_QUEUE_PROFILING_ENABLE, &err);
// ...
clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, &global, &local, 0, NULL, &event);
clFinish(command_queue);
// ...
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START, sizeof(start), &start, NULL);
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END, sizeof(end), &end, NULL);
printf("time = %.1f ms\n", ((end - start) / 1000000.0) );
```

OpenCL Compilermeldungen anzeigen

```
if (clBuildProgram(program, 0, NULL, NULL, NULL, NULL) != CL_SUCCESS) {
    char *log;
    size_t size;

    clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG,
                          0, NULL, &size);            // 1. Länge des Logbuches?
    log = (char *)malloc(size+1);
    if (log) {
        clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG,
                              size, log, NULL);        // 2. Hole das Logbuch ab
        log[size] = '\0';
        printf("%s", log);
        free(log);
    }
    return 1;
}
```

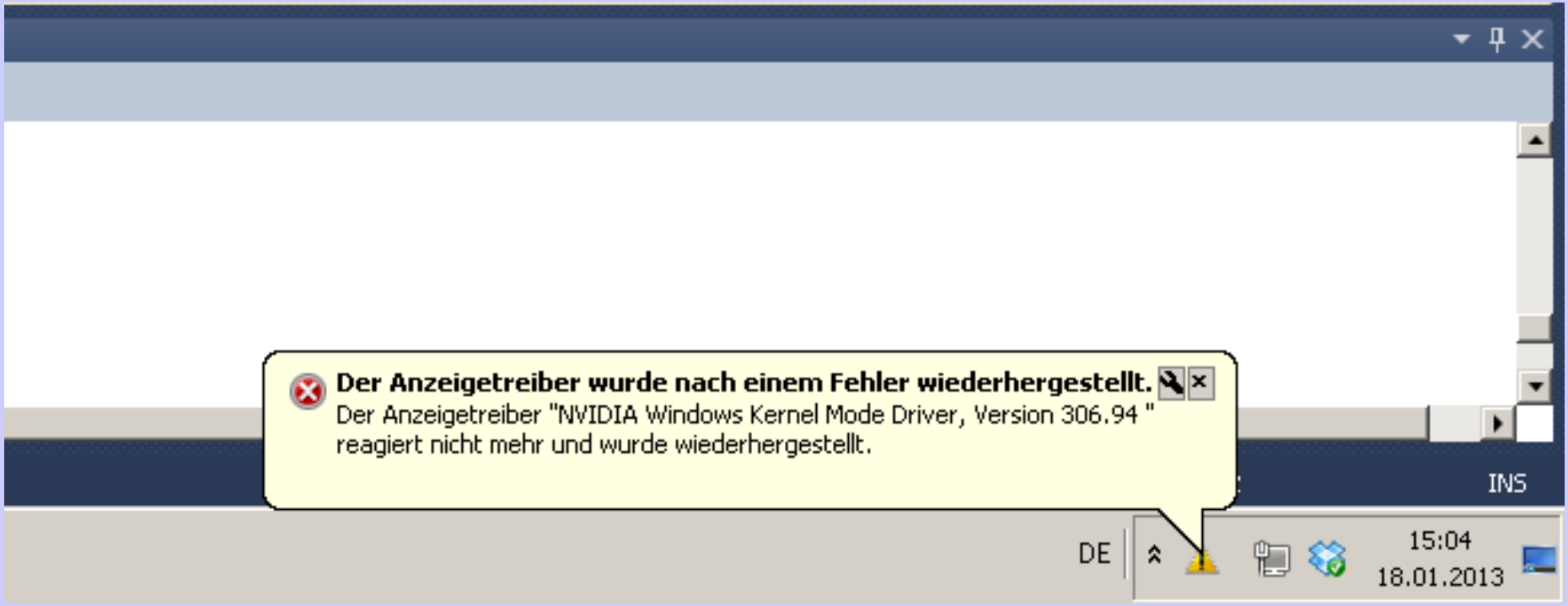
```
:9:39: error: expected ';' after expression
            sum += A[i*n+k] * B[k*n+j]:
                                      ^
                                      ;
:9:39: error: expected expression
```

Speicherverwaltung vereinfachen

```
#define DATA_SIZE 10                                    // Anzahl der Daten
#define MEM_SIZE  DATA_SIZE*sizeof(float)  // Groesse der Daten im Speicher

int main(void)
{
  cl_mem input;
  float data[DATA_SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

  // ... Erzeuge Puffer für Ein- und Ausgabe
  input  = clCreateBuffer(context, CL_MEM_READ_ONLY,  MEM_SIZE, NULL, &err);

  // Kopiere zusammenhängende Daten aus 'data' in den Eingabe-Puffer 'input'
  clEnqueueWriteBuffer(command_queue, input, CL_TRUE, 0, MEM_SIZE, data, 0, NULL, NULL);
```

```
  // Reserviere globalen Speicher und kopiere Daten vom Host in einem Schritt
  input = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
                         MEM_SIZE, data, &err);
```

Absturz des NVIDIA Treibers



NVIDIA Watchdog unter Windows

- **Problem**: Timeout Detection and Recovery (TDR)
  im Windows Display Driver Model (WDDM)
  (`http://msdn.microsoft.com/en-us/windows/hardware/gg487368.aspx`)
- **Idee:** Wenn der Grafiktreiber nicht mehr reagiert, wird er neu gestartet
- **Betroffen**: Windows + NVIDIA + angeschlossener Monitor
- **Lösung 1 (Nicht zu empfehlen, nur für Testzwecke!)**:

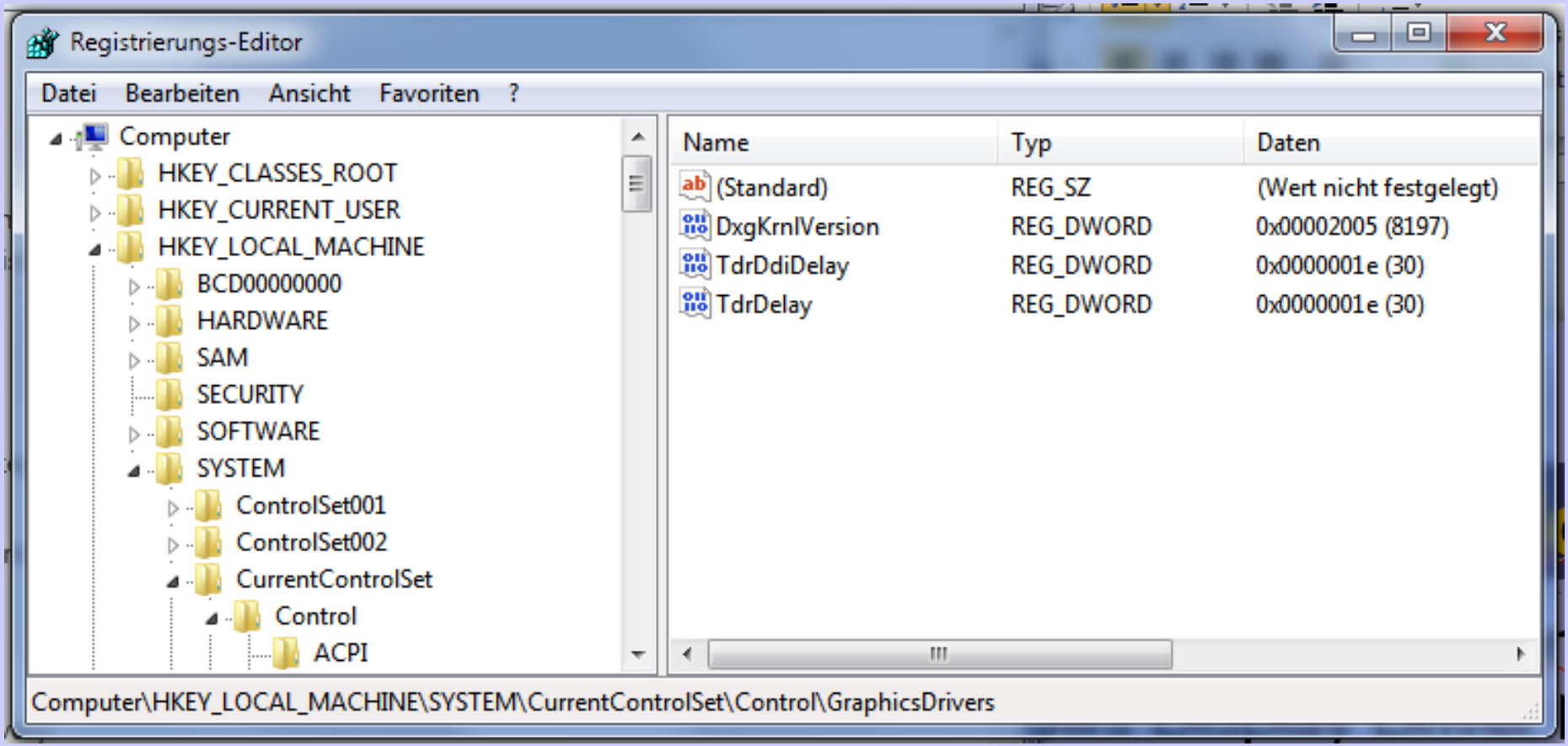
  in der Windows-Registrierungsdatenbank unter
  `HKLM\System\CurrentControlSet\Control\GraphicsDrivers`
  neue Einträge erstellen:

  – **TdrDelay:** REG_DWORD

  – **TdrDdiDelay:** REG_DWORD

  Enthalten die Anzahl an Sekunden, die das Betriebssystem dem Grafikkartentreiber für eine Rückmeldung erlaubt.
  Nach dieser Zeit wird der Treiber zurückgesetzt.
  Die Standardwerte sind **2** bzw. 5!

Beispiel: Erhöhung auf 30 Sekunden



Achtung: In dieser Zeit hat man keine aktualisierte Bildschirmausgabe!

Lösungsvorschlag

- **Lösung 2** (**Empfehlung!**):
  - Die langwierige Aufgabe für den Kernel in mehrere schnellere **Teilaufgaben** splitten (< 2 Sekunden)
  - Alle Teilaufgaben in die Befehlswarteschlange einreihen
  - Das Allokieren, Füllen und Leeren der globalen Puffer sowie das Warten auf Beendigung (`clFinish`) braucht nur **einmal** erfolgen!