

# Computergrafik

Universität Osnabrück, Henning Wenke, 2012-06-25

# Kapitel XV:



**Parallele Algorithmen mit OpenCL**

# 15.1

---

## Parallele Programmierung

Quellen: V.a. Wikipedia.  
Leistungsdaten unter Vorbehalt. Bitte nicht zitieren.

# Motivation

---

- Gegeben: Array von  $10^7$  floats Zufallszahlen
- Berechne für jedes Element tmp:

```
for (int k = 0; k < 1000; k++)  
    tmp= (tmp* (tmp+k) * (tmp*k*k) * (tmp*k*k*k)) *tmp  
        / (tmp+10) +tmp*k*k*k*k*k /1000.0f;
```

- Miss Zeit mit
  - Java, Intel Core i7 860, seriell: 125 sec
  - OpenCL mit Intel Core i7 860: 21 sec
  - OpenCL mit Nvidia GTX 470: 0.4 sec

# CPUs

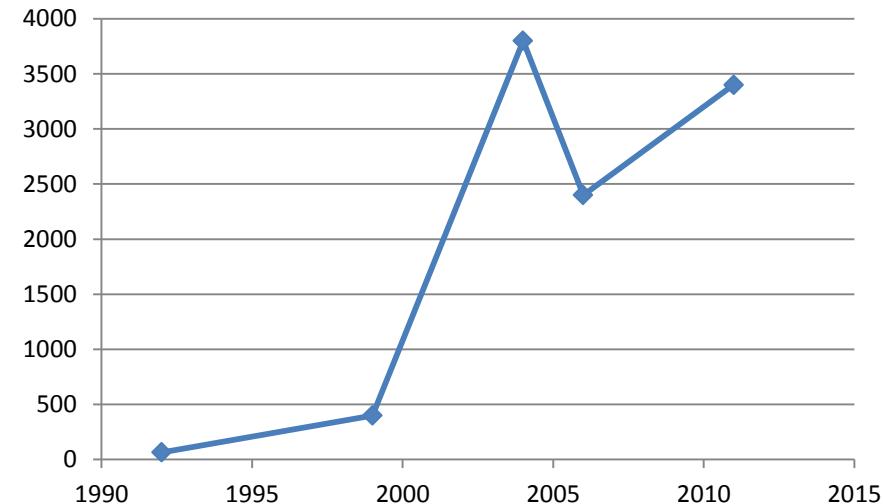
---

	Jahr	Takt [MHz]	TDP [W]	Kerne
Intel 486 DX2 40	1992	66	6	1
Intel P2 400	1999	400	24	1
Intel P4 3800	2004	3800	115	1
Intel Core 2 E6600	2006	2400	65	2
Intel Core i7 2600	2011	3400	958*	

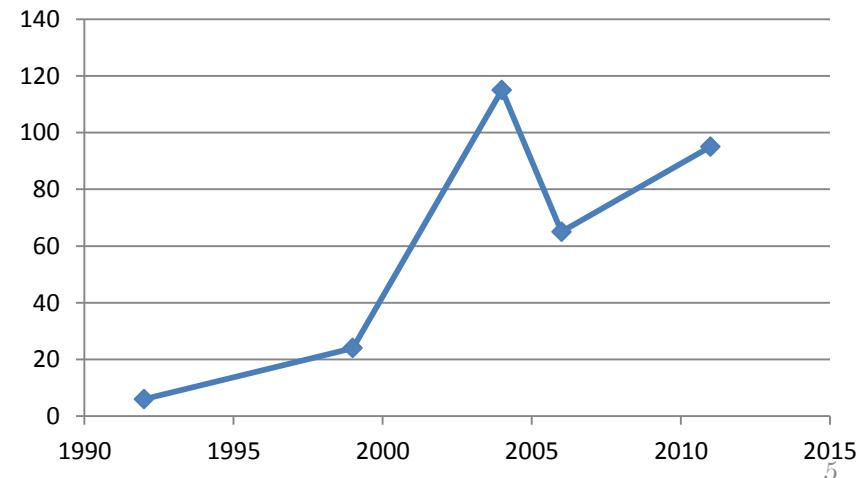
$$P = ACV^2F + VI_{leak}, \text{ mit:}$$

$F$  Frequenz,  $V$  Spannung

**Takt [MHz]**

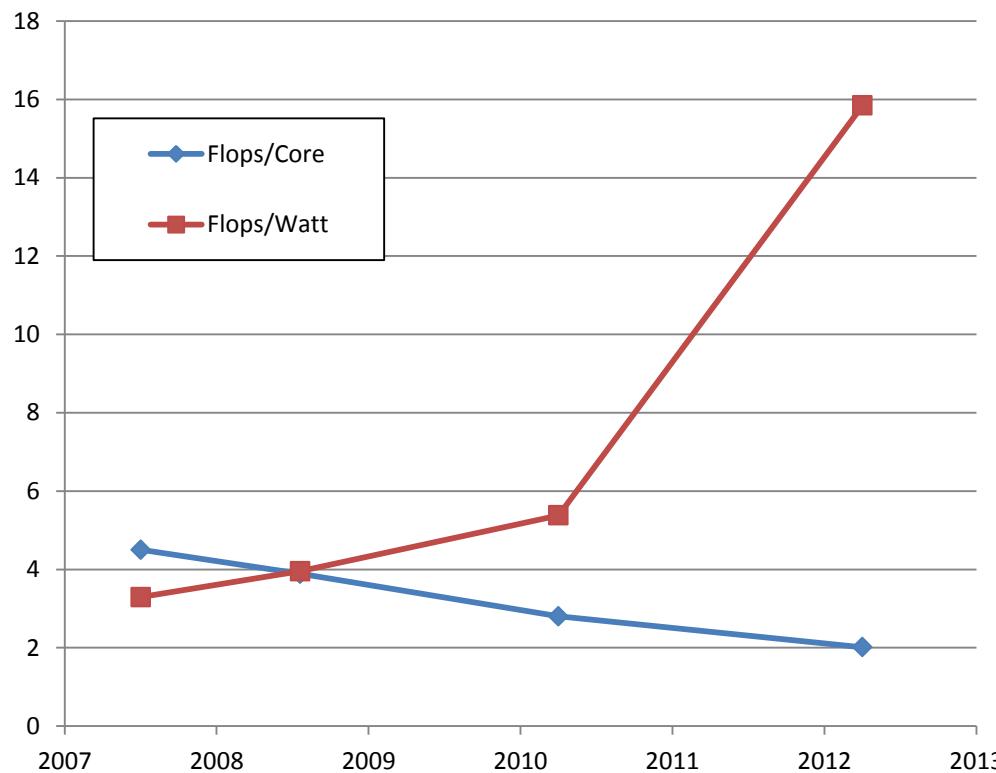


**TDP [W]**



# GPUs

	Jahr	Cores	FLOPS	Band	Tdp	Flops/Core	Flops/Watt
GTX 8800 Ultra	2007-05	128	576	104	175	4,5	3,29142857
GTX 280	2008-06	240	933	141	236	3,8875	3,95338983
GTX 480	2010-03	480	1345	177	250	2,80208333	5,38
GTX 680	2012-03	1536	3090	192	195	2,01171875	15,8461538



# Arten der Parallelität in OpenCL

## ➤ Task Parallelität (wenn Tasks jeweils seriell)

$a = \{1, 2, 3, 4\}$



task1(i) {return i\*i;}

$a = \{1, 4, 9, 16\}$

$b = \{5, 6, 7, 8\}$



task2(i) {return i;}

$b = \{5, 6, 7, 8\}$

## ➤ Datenparallelität (Single Instruction Multiple Data)

$a = \{1, 2, 3, 4, 5, 6, 7, 8\}$



task(i) {return i\*i;}

$a = \{1, 4, 9, 16, 25, 36, 49, 64\}$

## ➤ Datenparallelität (Single Program Multiple Data)

$a = \{1, 2, 3, 4, 5, 6, 7, 8\}$



```
task(i) {
    if(Arrayindex < 4) return i*i;
    else
        return i;
}
```

$a = \{1, 4, 9, 16, 5, 6, 7, 8\}$

## 15.2

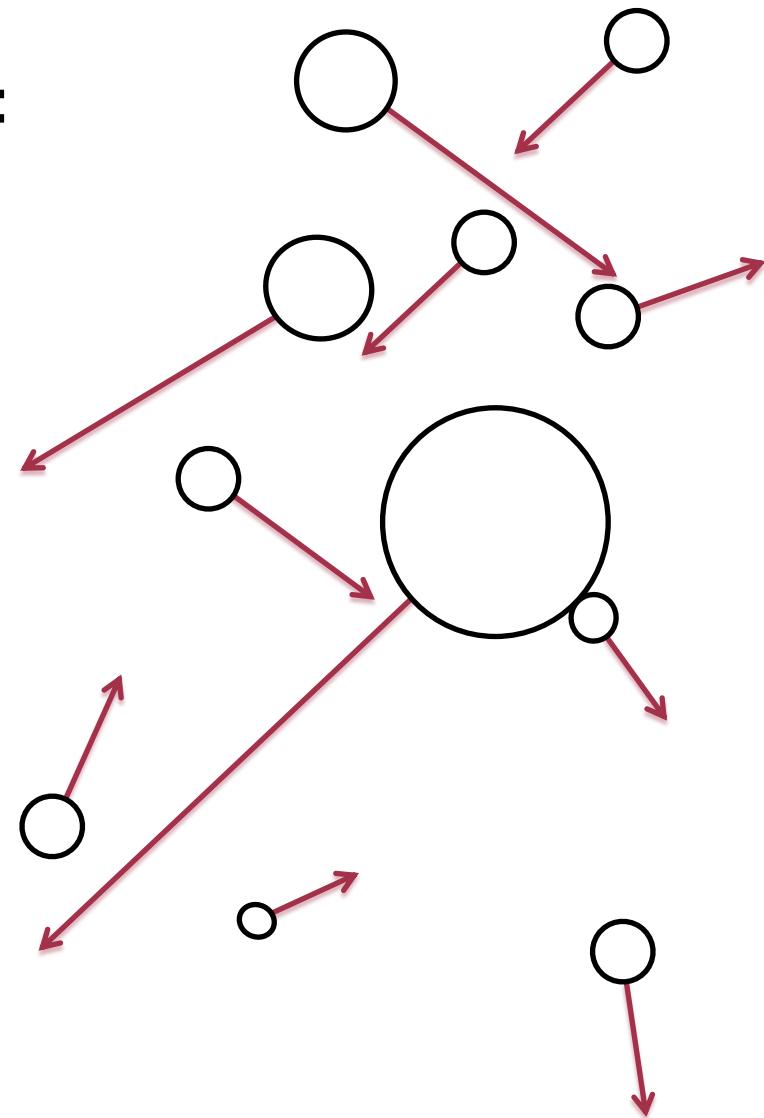
---

Vorschau: Prozedurale Modellierung / Partikelsysteme

# Partikel

---

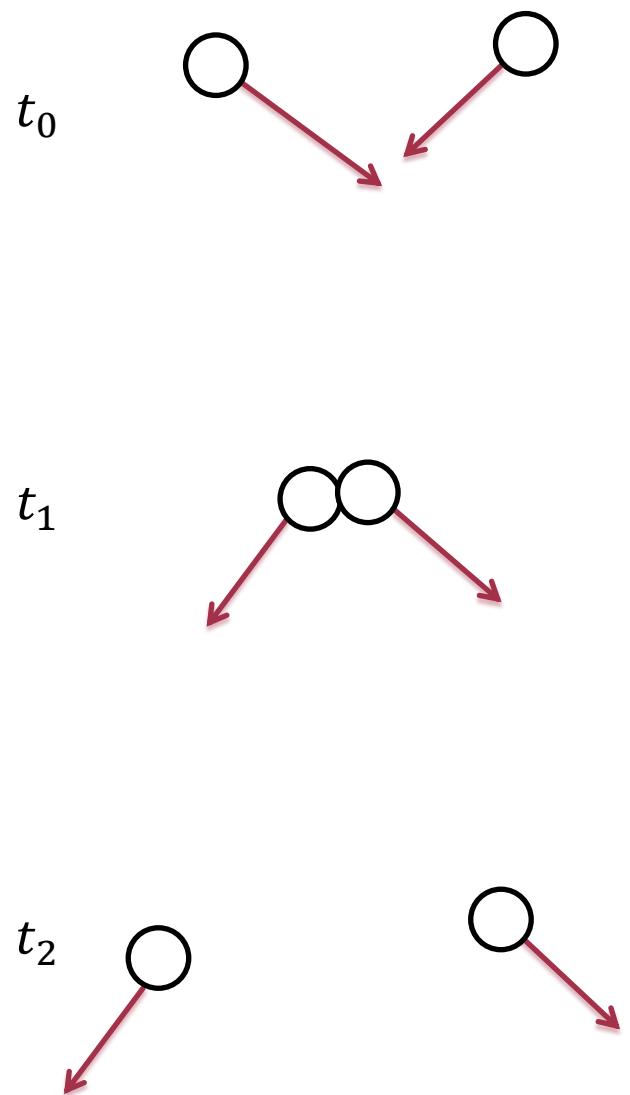
- Beispiel für Partikel: Vertex mit:
  - Position
  - Ausdehnung
  - Aktuelle Geschwindigkeit
- Zentrale Eigenschaft Partikel?
  - Keine Topologie
- Simulation v.s. Rendering
  - Partikel kann z.B. als Kugelmesh gerendert werden
  - Instancing



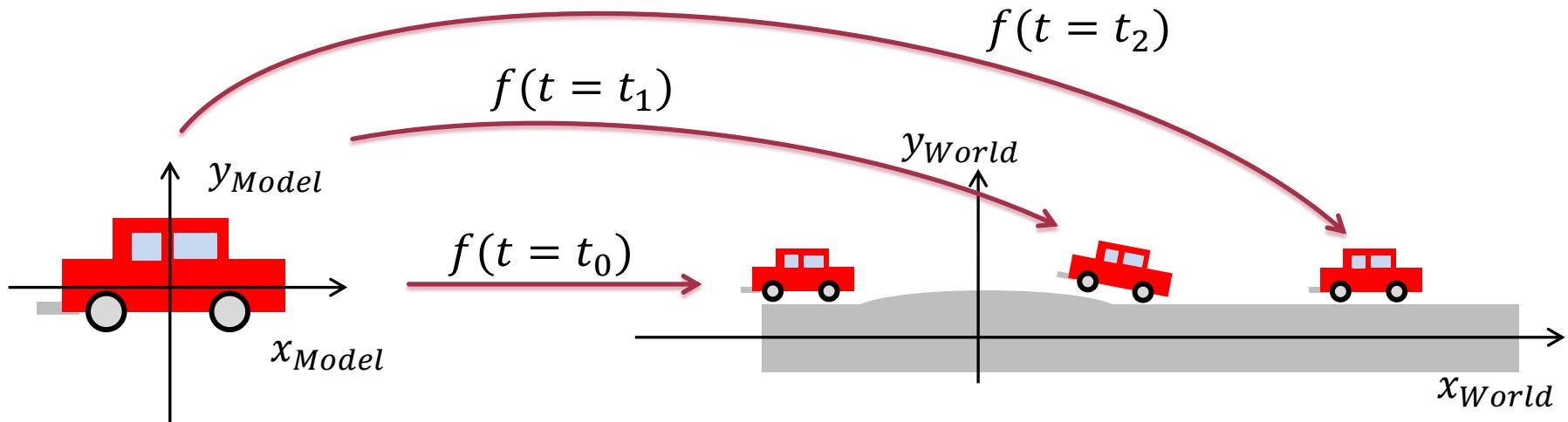
# Kollision

---

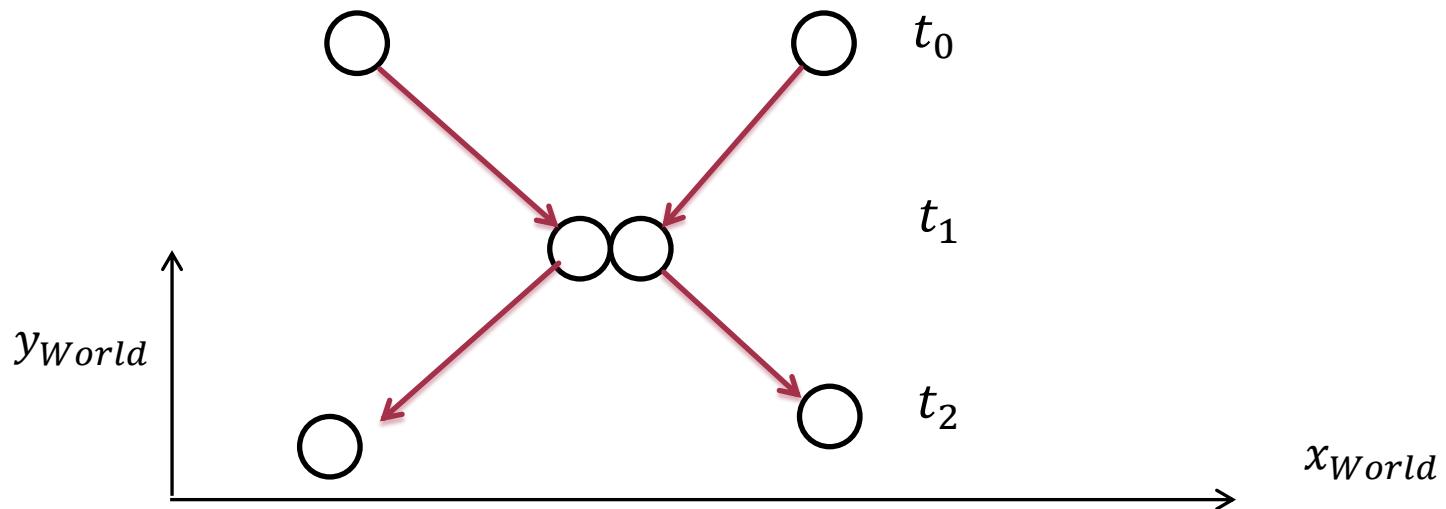
- Inwiefern Berechnungen anders als bisher?
- Berechnungen für Partikel abhängig
- Nicht im Vertex Shader lösbar



# Update



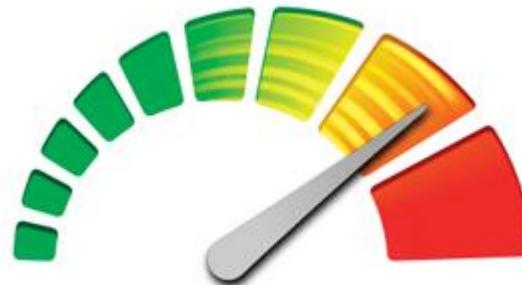
- Bisher: Vorbestimmte, zeitabhängige Funktion
- Jetzt: Berechne basierend auf letztem Zeitschritt neue Position
- Mit OpenGL sehr umständlich (Transform Feedback)



15.3

---

OpenCL



# Allgemeines

---

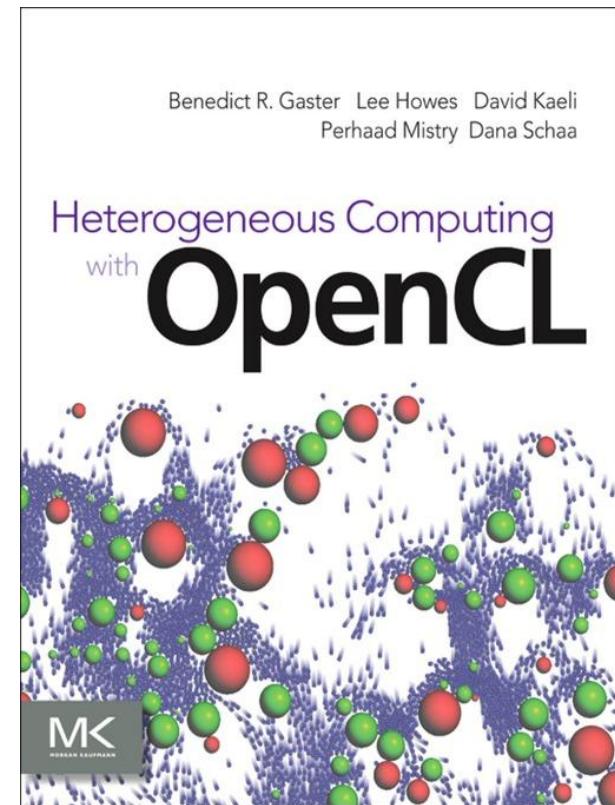
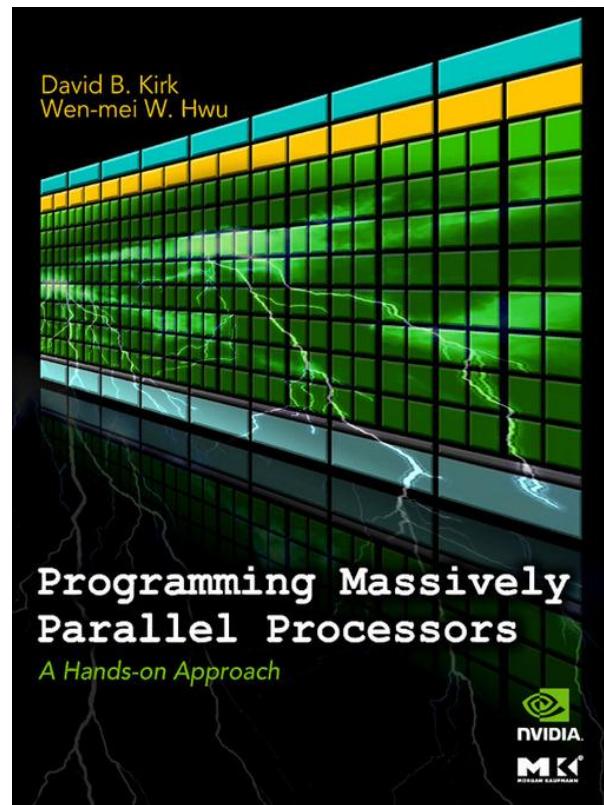
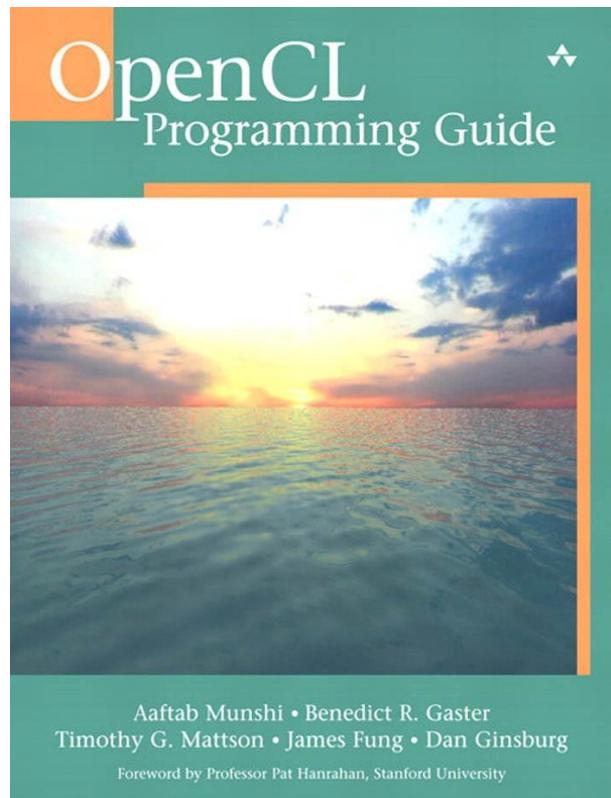
- **Open Compute Language:** API für einheitliche parallele Programmierung heterogener Systeme
  - GPUs, CPUs, APUs, FPGAs...
  - Oder Kombinationen daraus
  - Hinweis: Optimierung typischerweise nicht einheitlich
- Initiiert 2008-12 durch Apple, betreut d. Khronos Group
- Erste Implementation: 2009-08 (Mac OS X 10.6)
- Plattform, Betriebssystem & Sprachunabhängig
- OpenCL C
  - Zugehörige, an C (ISO C99) angelehnte, Sprache mit Erweiterungen für parallele Algorithmen
  - Kein dynamisches Allokieren von Speicher
  - Keine Rekursion
  - Damit geschriebenes Programm heißt Kernel
  - Gibt nur einen Typ
- Arbeitet direkt mit OpenGL zusammen

# OpenGL vs. OpenCL

---

- Ebenfalls Host – Device(s) Architektur
  - Etwa: Host: Applikation, Devices: 2 Grafikkarten + 1 CPU
- OpenCL C  $\approx$  GLSL
- Kernel  $\approx$  Shader
  - Einer für alles
  - Mächtiger als alle Shader zusammen
- Keine feste Pipeline, Kernel werden direkt aufgerufen
- Ebenfalls asynchrone Befehlsausführung
- Keine State Machine
- Keine speziellen oder High Level Funktionen
  - Keine Grafikfunktionen (Rastern, Culling, Blending, Framebuffer)
  - Auch nicht: Invertierung großer Matrizen, o.ä.
  - Ausnahme: Texturen. Sind aber nur Datenstrukturen

# Literatur



OpenCL Reference Pages:

<http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/>

## 15.3

---

### OpenCL C Beispiele

# Pseudo Vertex Shader v.s. Vertex Shader

```
// Unser Pseudo VS zur Translation
// beliebiger Geometrie

for all(int v in(0, ,vertexCnt-1)
      in parallel do{

    vec4 posMC;
    vec4 posWC;
    mat4 trans;

    <Read_Vertex_Data(v)>
        posMC = readPosMC(v);
        trans = readTrans();

    <Do_Calculations(v)>
        posWC = trans * posMC;

    <Write_Data(v)>
        writePosWC(v, posWC);

}
```

```
// Entsprechender echter
// Vertex Shader
#version 330 core

in vec4 posMC;
uniform mat4 trans;
out vec4 posWC;

void main() {
    posWC = trans * posMC;
}
```

# Vertex Shader v.s. Kernel

```
// Entsprechender echter
// Vertex Shader
#version 330 core

in vec4 posMC;
uniform mat4 trans;
out vec4 posWC;

void main() {
    posWC = trans * posMC;
}
```

```
// Methode: Vektor-Matrix-Produkt a * b
float4 mul(float16 a, float4 b) {...};

// Startpunkt eines Kernels "vertexShader"
kernel void vertexShader(
    global float4* posMC, // Direkter Zugriff
    global float4* posWC, // auf globale
    global float16* trans) { // Daten

    int wiId = get_global_id(0);

    // Lies Daten dieses Work Items
    float4 posMC_TT = posMC[wiId];

    // Do Calculations
    float4 posWC_TT = mul(trans[0], posMC_TT);

    // Schreib Ergebnis in Global Memory
    posWC[wiId] = posWC_TT;
}
```

# Update

```
// Methode: Vektor-Matrix-Produkt
float4 mul(float16 a, float4 b) {...};

// Startpunkt eines Kernels "vertexShader"
kernel void vertexShader(
    global float4* pos,
    global float16* trans) {

    int wiId = get_global_id(0);

    // Lies Daten dieses Work Items
    float4 pos_TT = pos[wiId];

    // Do Calculations
    float4 pos_TT = mul(trans, pos_TT);

    // Schreib Ergebnis in global Mem
    pos[wiId] = pos_TT;
}
```

# Gather

```
float4 mul(float16 a, float4 b) {...};

// Startpunkt eines Kernels "vertexShader"
kernel void vertexShader(
    global float4* posMC,
    global float4* posWC,
    const float16* trans) {
    int wiId = get_global_id(0);

    // read Data of this Thread
    float4 posMC_TT = posMC[wiId];
    if(wiId % 2 == 0) // Lies auch Daten eines anderen Vertex
        posMC_TT += posMC[wiId + 1] * 0.25;
    else
        posMC_TT += posMC[wiId - 1] * 0.25;
    // Do Calculations
    float4 posWC_TT = mul(trans, posMC_TT);
    // Schreib Ergebnis in global Mem
    posWC[wiId] = posWC_TT;
}
```

# Gather / Update

```
float4 mul(float16 a, float4 b) {...};

kernel void vertexShader(
    global float4* pos,
    const float16* trans) {
    int wiId = get_global_id(0);
    // read Data of this Thread
    float4 pos_TT = pos[wiId];

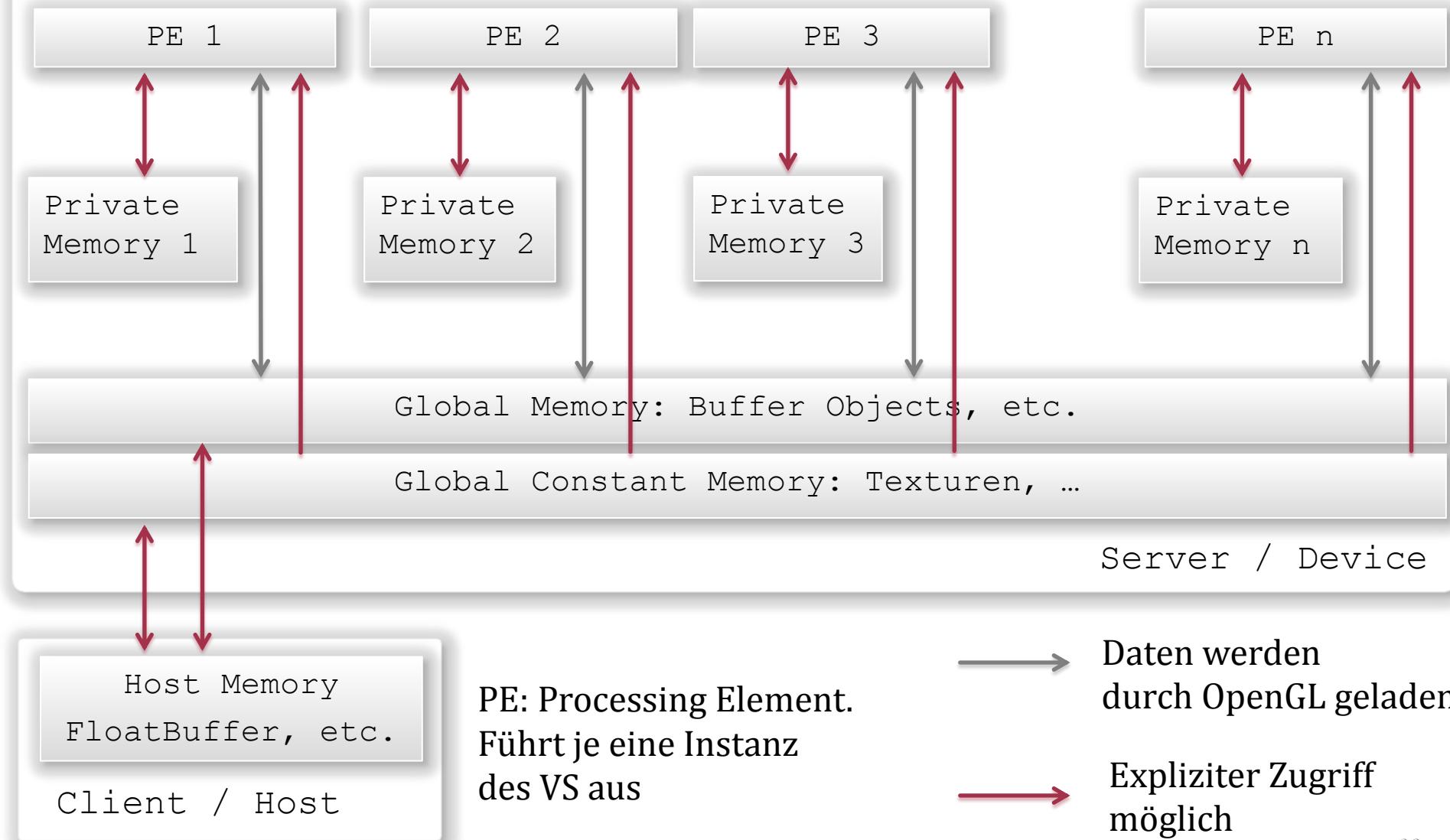
    // Muss syncronisiert werden da pos[wiId - 1] und
    // pos[wiId + 1] bereits verarbeitet sein können
    if(wiId % 2 == 0) // Lies auch Daten eines anderen Vertex
        pos_TT += pos[wiId + 1] * 0.25;
    else
        pos_TT += pos[wiId - 1] * 0.25;
    // Do Calculations
    float4 pos_TT = mul(trans, pos_TT);
    // Schreib Ergebnis in global Mem
    pos[wiId] = pos_TT;
}
```

## 15.4

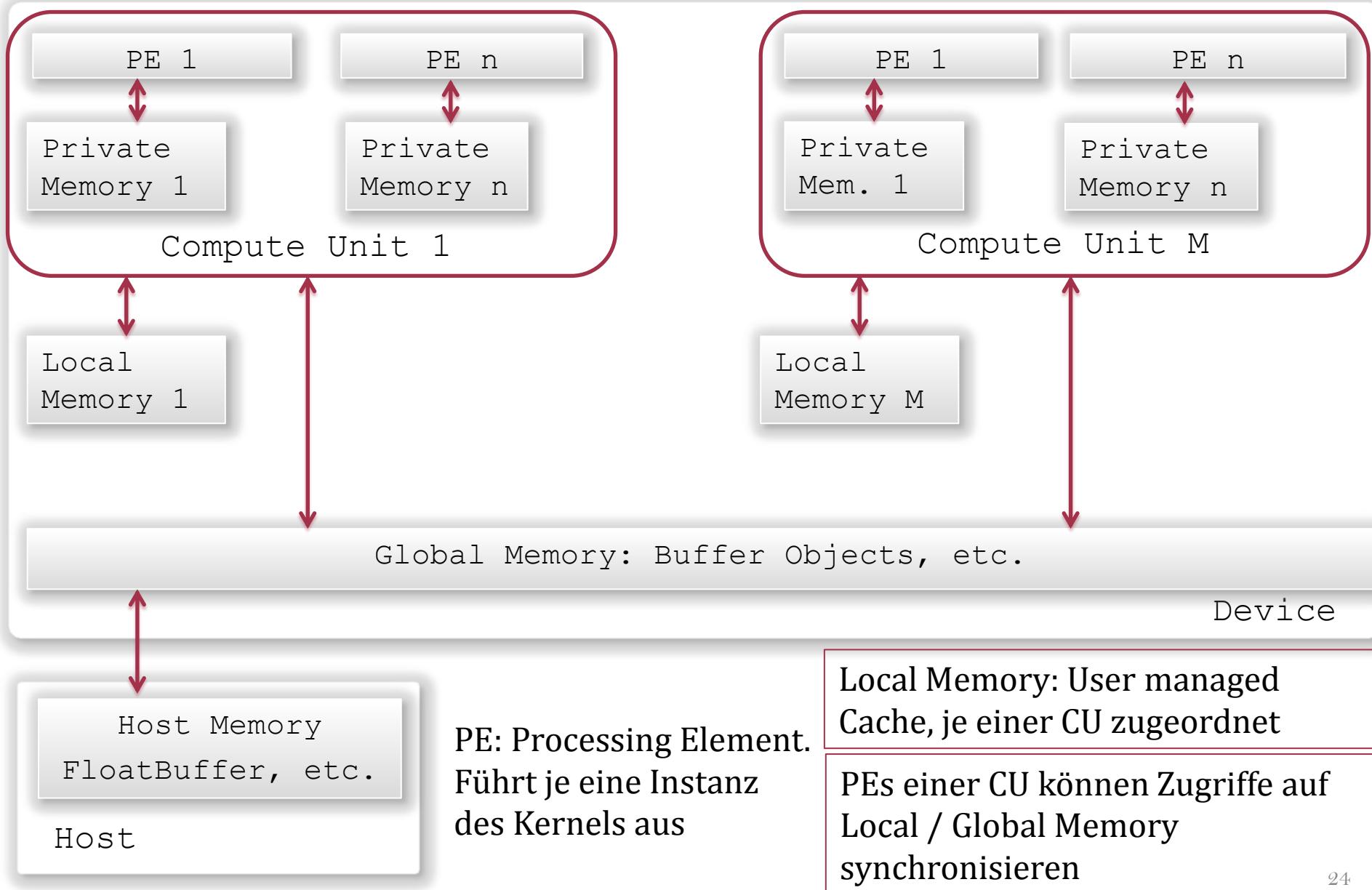
---

### Speicher & Indizierung

# OpenGL VS Speichermodell



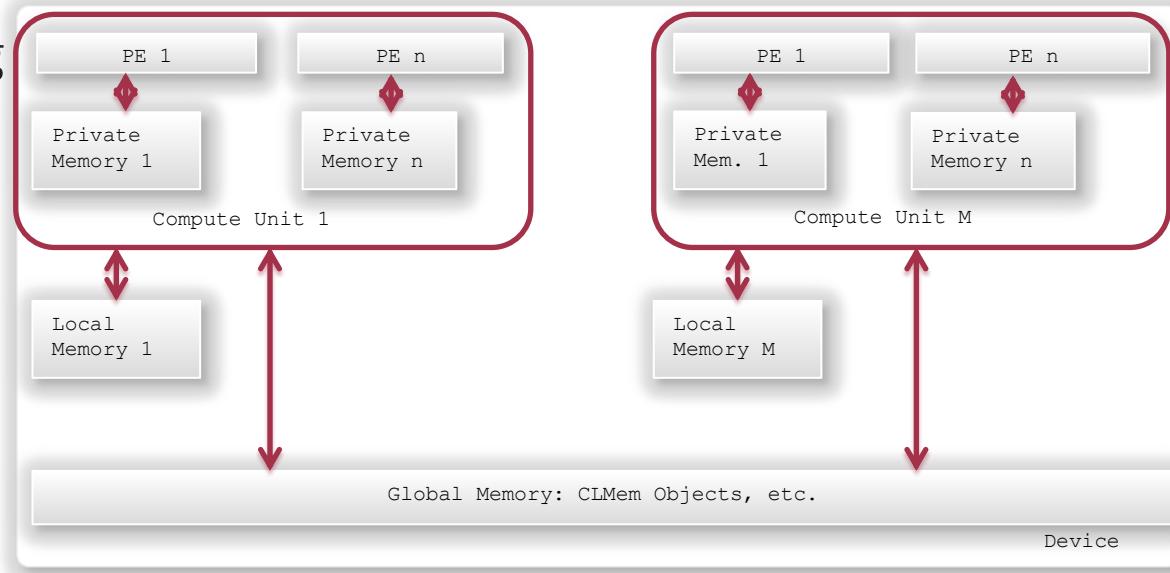
# OpenCL Speichermodell



# Indizierung: Wünsche

- Zugriff auf Global Memory:
  - Global eindeutiger Index nötig
- Zugriff auf Local Memory:
  - Lokal, also in einer CU, eindeutiger Index nötig
- Weiterer Index für CU und Indexzahlen hilfreich

```
kernel void aKernel(  
    global float4* a,  
    local  float4* b) {  
    ...  
}
```



# 1D Speichermodell

gId = 0  
lId = 0

gId = 1  
lId = 1

gId = 2  
lId = 2

groupId = 0

gId = 3  
lId = 0

gId = 4  
lId = 1

gId = 5  
lId = 2

groupId = 1

gId = 6  
lId = 0

gId = 7  
lId = 1

gId = 8  
lId = 2

groupId = 2

gId = 9  
lId = 0

gId = 10  
lId = 1

gId = 11  
lId = 2

groupId = 3

```
get_num_groups(0); // Liefert Anzahl der Work Groups. Hier: 4  
get_local_size(0); // Liefert Anzahl Work Items einer Work Group. Hier: 3  
get_global_size(0); // Liefert Gesamtzahl der Work Items. Hier: 12
```

Work Group  
/ Compute Unit

Work Item /  
Processing Element

- ‘gId’: Globale Id eines Work Items: `get_global_id(0)`;
- ‘lId’: Lokale Id eines Work Items: `get_local_id(0)`;
- ‘groupId’ Id einer Work Group: `get_group_id(0)`;

## 15.5

---

# Programmierung mit OpenCL C (Kurzfassung)

# OpenCL C Basics I

```
// Startpunkt eines Kernels des Namens "aKernel"
kernel void aKernel(...) { ...}

// Datentypen z.B.
float, float2, float16, int8, uint2, ...

// Initialisierung & Zugriff
float8 val = (float8)(1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0);
val.x // Liefert 1.0
val.s0 // Liefert 1.0
val.s7310 // Liefert (8.0, 4.0, 2.0, 1.0)
// Ab Komponente 10: .sa, .sb, etc.

// Alle Operationen Komponentenweise
float4 a = (float4)(8.0, 4.0, 2.0, 1.0);
float4 b = (float4)(0.0, 0.0, 2.0, 1.0);
float4 ab = a * b; // Liefert: (0.0, 0.0, 4.0, 1.0)
```

# OpenCL C Basics II

```
// Daten werden als Parameter des Kernels deklariert
kernel void aKernel(
    const int a,           // Konstante
    global float4* b,   // Pointer
                           // Global les- & schreibbare Daten
    local  float4* c   // Pointer
                           // Lokal in der Workgroup les- & schreibbar
                           // Müssen erst durch Kernel geschrieben werden
) {
    // Typischer Zugriff auf globale Daten
    float4 myB = b[get_global_id(0)];

    float4 myC;
    // Vermutlich keine gute Idee:
    // 1. c noch gar nicht geschrieben.
    // 2. Index nur sinnvoll, wenn es nur eine Work Group gibt.
    myC = c[get_global_id(0)];
}
```

# OpenCL C Basics III

```
kernel void aKernel(
    const int a,
    global float4* b,
    local  float4* c) {

    float4 myB = b[get_global_id(0)];

    // Schreibe einen Wert in den lokalen Speicher an eine Position des lokalen
    // Speichers dieser Workgroup. Passiert in allen WGs, immer wieder die
    // gleichen Positionen aber anderer Speicher.
    c[get_local_id(0)] = myB;

    // Blockiert, bis alle WEs dieser WG ihre Zugriffe auf den lokalen Speicher
    // beendet haben
    barrier(CLK_LOCAL_MEM_FENCE);

    // Jetzt können die in c geschriebenen Werte in allen Work Elements der WG,
    // etwa das Erste und das Letzte, gelesen werden:
    float4 firstC = c[0];
    float4 lastC   = c[get_local_size(0)-1];
}
```

# OpenCL C Basics IV

---

```
// Blockiert, bis alle WEs dieser WG ihre Zugriffe auf  
// den globalen Speicher beendet haben  
barrier(CLK_GLOBAL_MEM_FENCE);  
  
// Achtung: Keine globale Synchronisation.  
// Globaler Synchronisationspunkt erst nach vollständig  
// ausgeführtem Kernel möglich  
  
// Mathematische Build-In Funktionen auch vorhanden, z.B.:  
cos(), dot(), cross(), ...
```