technische universität
dortmund

fakultät für informatik
informatik 12

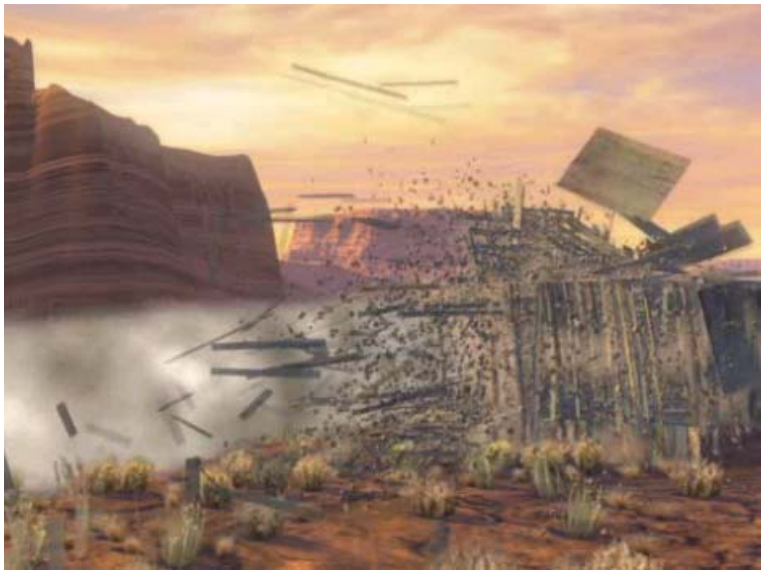
GPGPU-Programming

Constantin Timm
Informatik 12
TU Dortmund

2012/04/09

Motivation (1)

- General Purpose Computing on Graphics Processing Units (GPGPU)
- Einführung um CPU (bei Spielen) zu entlasten

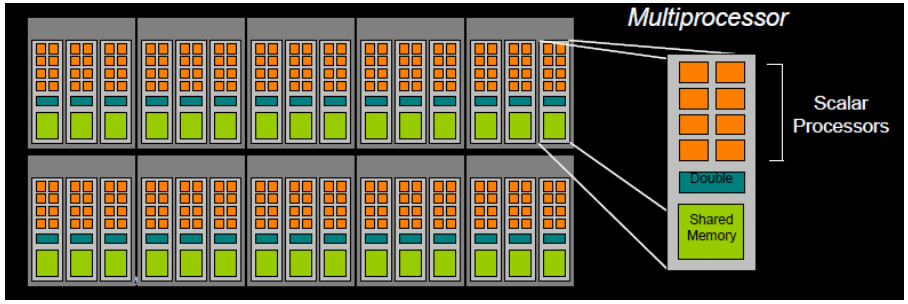


Physikalische Berechnungen

Künstliche Intelligenz

Motivation (2)

- GPUs haben eine große Anzahl von parallelen Rechenkernen
- Wie kann man diese effizient programmieren?

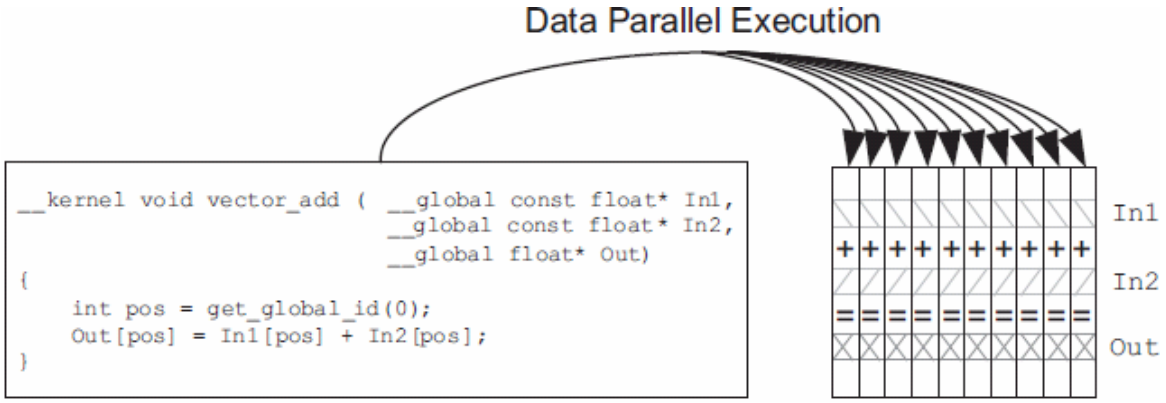


Gut für datenparallele Programme

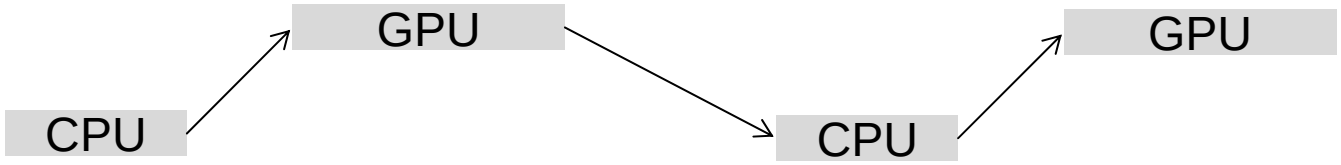
```
void add_vector(int* in1, int *in2, int *out) {
    for ( int i = 0; i < N; ++i ) {
        out[i] = in1[i] + in2[i] ;
    }
}
```

Motivation (3)

- Was sollte man bzgl. GPGPU-Applikationen beachten?
  - Parallele Applikationen mit möglichst unabhängigen Operationen



  - Kopier-Overhead für Daten

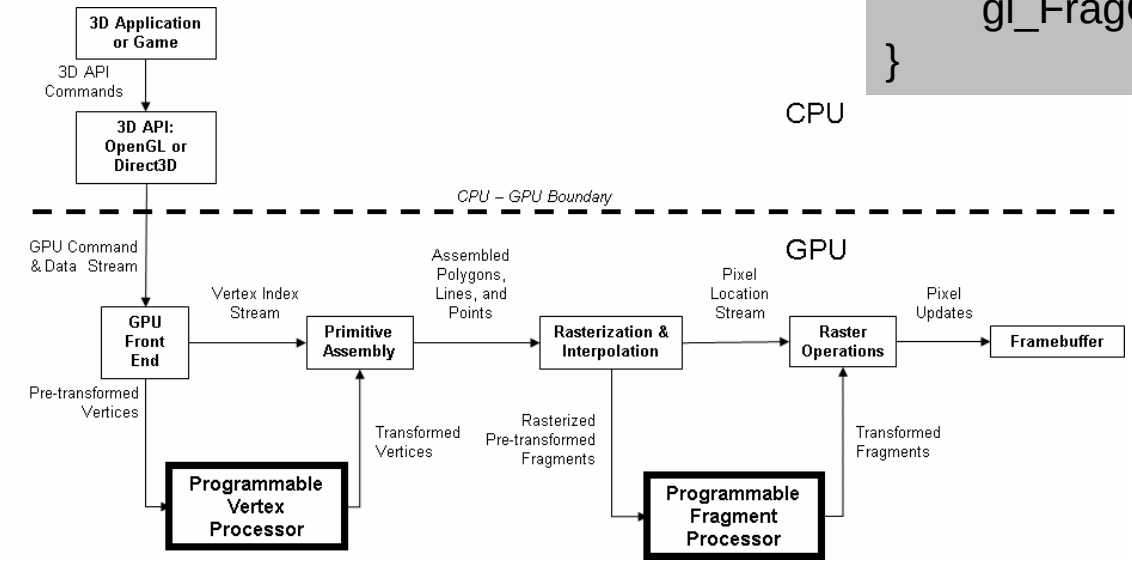


Geschichte der GPGPU-Programmierung (1)

Bis zirka 2003 – 2004

- Benutzung von Shader-Sprachen zur Programmierung
- Vertex- und Fragmentshaderprogramme

```
void main(void) {
        gl_FragColor = vec4(0.0, 1.0, 0.0, 1.0);
}
```



Geschichte der GPGPU-Programmierung (2)

Ab 2004

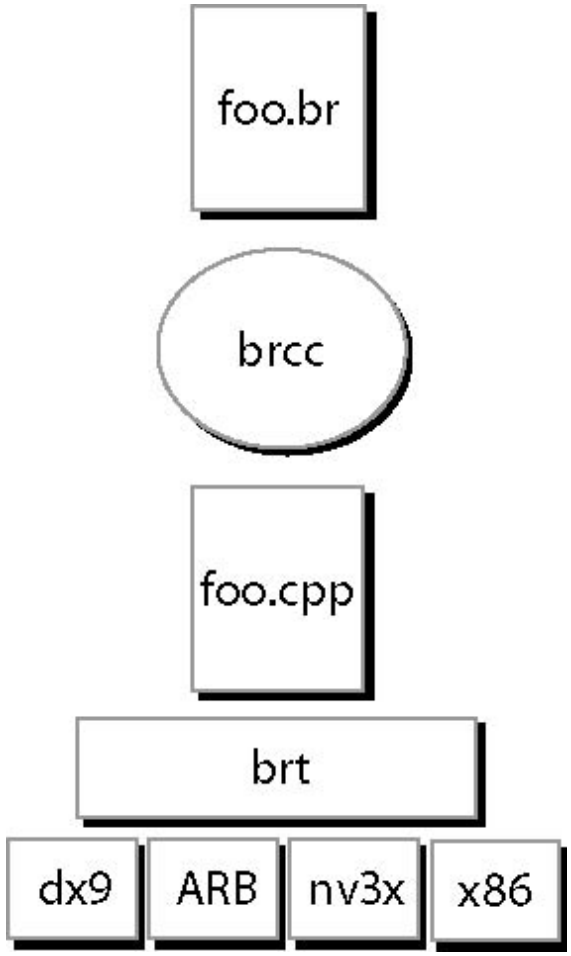
- Einführung von Sprachen zum Streamprocessing, z.B. BrookGPU
  - Von der Stanford University
  - Nutzung von GPUs als Coprocessor / Beschleuniger
  - Versteckt Komplexität

```
kernel void foo (float a<>, float b<>, out float result<>) {
        result = a + b;
}
float a<100>;
float b<100>;
float c<100>;
foo(a,b,c);
```

Geschichte der GPGPU-Programmierung (3)

Ab 2004

- BrookGPU
  - Komplexer Kompilervorgang
  - Plattform-unabhängig
- BrookGPU-Kompilervorgang
  - „brcc“: S2S-Compiler
  - „brt“: Runtime



Geschichte der GPGPU-Programmierung (4)

Ab 2007

- Einführung von CUDA
  - „Compute Unified Device Architecture“
  - Framework für Streamprocessing auf Nvidia Grafikkarten
  - Ursprünglich nur für Datenparallelität konzipiert

Ab 2008

- Einführung von OpenCL
  - Allgemeines Framework für Streamprocessing auf Multi- und Manycore-Architekturen
  - Für Daten- und Taskparallelität konzipiert
  - Spezifikation durch Khronos: AMD, Apple, ARM, Creative, Google, Intel, TI, Samsung, Nvidia

CUDA

Adaptiert das Streamprocessing-Konzept

- Elementare Programmierkomponente => Kernel
  - Keine Rekursion
  - Parameteranzahl ist nicht variabel
- Unterscheidung von Host- und GPU-Code

```
void add_vector(int* in1, int *in2, int *out) {
    for ( int i = 0; i < N; ++i ) {
        out[i] = in1[i] + in2[i] ;
    }
}
```

Vektoraddition in C

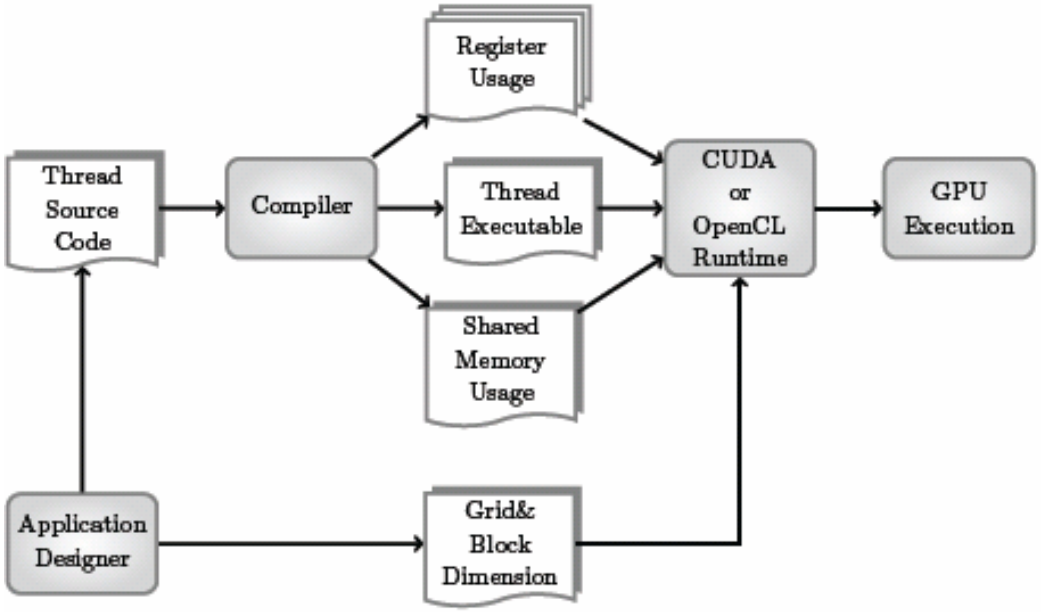
```
__global__
void add_vector (int* in1, int *in2, int *out) {
        int i = (blockIdx.x*blockDim.x)+threadIdx.x;
        out[i] = in1[i] + in2[i];
}

add_vector<<<N,1>>>( in1, in2, out );
```

Vektoraddition in Cuda

CUDA – Entwicklungsprozess

Mehrstufig und kompliziert

- Programmierung von Code für einen Thread
- Spezifikation der Parallelität per Hand
- Viele statisch vorgegebene Größen

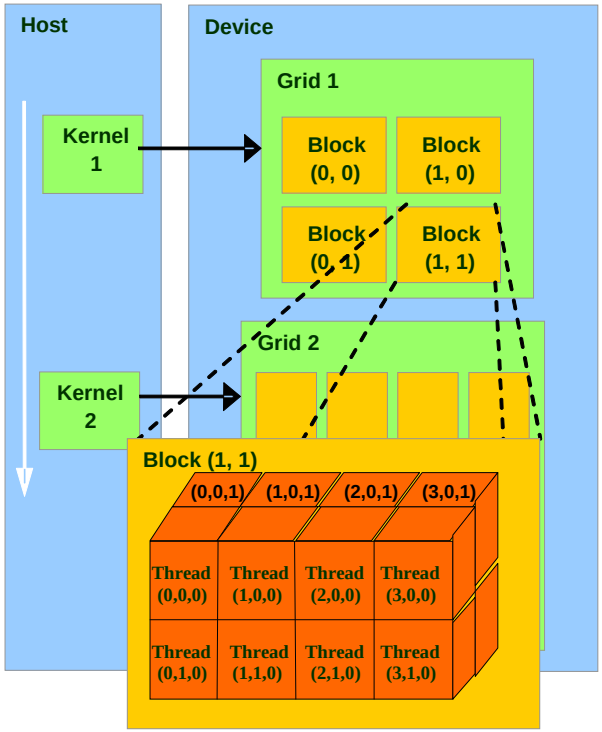


CUDA – Elemente des Frameworks

Thread
- Instanz eines Kernels

Block
- Gruppe von Threads

Grid
- Gesamtheit aller Blocks

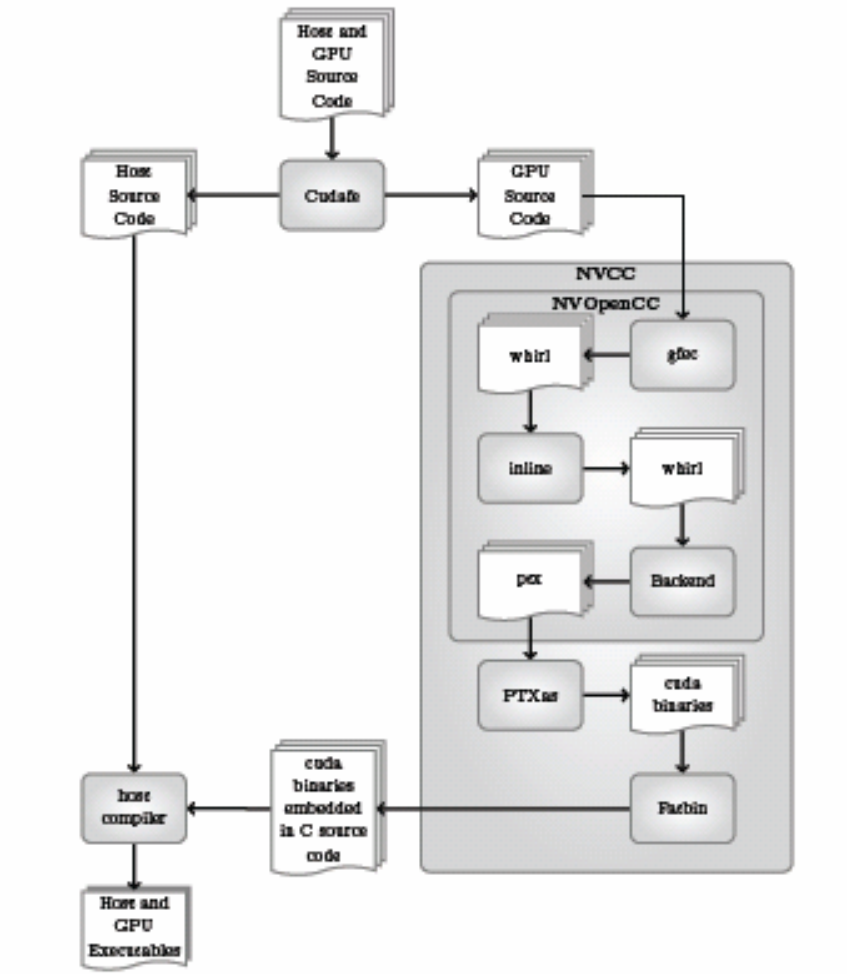



```
__global__
void add_vector (int* in1, int *in2, int *out) {
        int i = (blockIdx.x*blockDim.x)+threadIdx.x;
        out[i] = in1[i] + in2[i];
}
```

CUDA – Kompilierung

CUDA-Programme werden in einem mehrstufigen Verfahren kompiliert

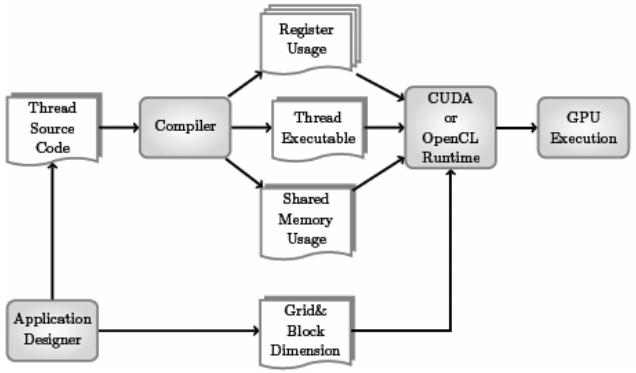
- GPU- und Host-Code getrennt kompiliert
- GPU-Binaries in Host-Code eingebettet
- Neuster Compiler von Nvidia basiert auf LLVM



CUDA – Abbildungsbeispiel

Kernel benötigt folgende Ressourcen

- 5 Register pro Thread
- 1052 Bytes Shared Memory per Block
- Grid size: 64 Blocks
- Block size: 256 Threads



Grafikkarte (1 Streaming-Multiprocessor)

- 8152 Register
- 16384 Bytes Scratchpad-Speicher
- Max 768 Threads, 8 Blocks und 24 Warps

Auslastung der Grafikkarte

- 768 Threads, 3 Blocks, 4608 Bytes Shared Memory und 3840 Register

CUDA - Speicherallokation

cudaMalloc()

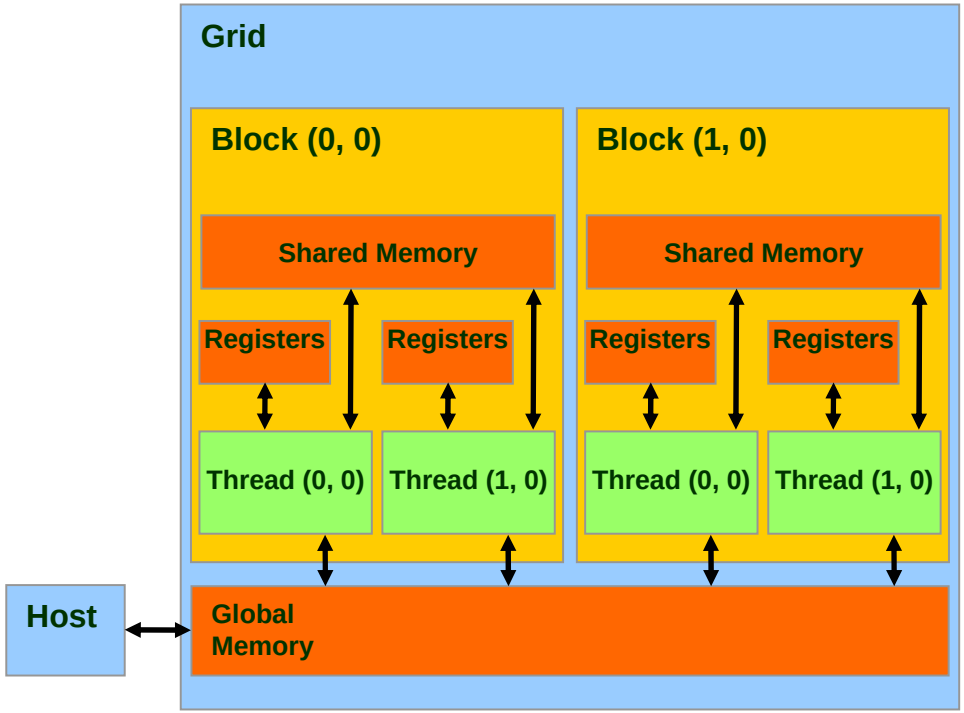
- Allokiert globalen Speicher auf der Grafikkarte

cudaFree()

- Gibt allokierten Speicher auf der Grafikkarte frei

cudaMemcpy()

- Kopiert in/aus/im globalen Speicher auf der Grafikkarte



CUDA - Speichertransfers

Kopieren in/aus/im globalen Speicher

- Vom Host zur Grafikkarte

```
int *d_x;
cudaMalloc((void**) &d_x, sizeof(int));
int x=0;
cudaMemcpy(d_x, &x, sizeof(int), cudaMemcpyHostToDevice);
```

- Von der Grafikkarte zum Host

```
int *d_y;
cudaMalloc((void**) &d_y, sizeof(int));
int y=0;
…
cudaMemcpy(&y, d_y, sizeof(int), cudaMemcpyDeviceToHost);
```

- Auf der Grafikkarte

```
int *d_x,d_y;
cudaMalloc((void**) &d_x, sizeof(int));
cudaMalloc((void**) &d_y, sizeof(int));
cudaMemcpy(d_x, d_y, sizeof(int), cudaMemcpyDeviceToDevice);
```

CUDA – Speicherzugriff

Globaler/Shared Memory-Speicherzugriff

- Zugriff auf globalen/shared Speicher nicht synchronisiert
- Ausgang von Schreibe-/Leseoperationen auf gemeinsamen Speicher?
  - Lösung: Atomare Operationen

```
__global__
void add_vector_gpu (int* out) {
        *out+=5;
}

add_vector_gpu<<<1,5>>>(out);
```

Ergebnis? => out = 5

```
__global__
void add_vector_gpu (int* out) {
        atomicAdd(out,5);
}

add_vector_gpu<<<1,5>>>(out);
```

Ergebnis? => out = 25

CUDA – Thread Divergence (1)

Ablauf der Threads

- Vorhersage der tatsächlichen Reihenfolge schwierig
- Kann von Programmierer erzwungen werden

```
__global__ void update(int* x, int* y) {
        int i = threadIdx.x + blockDim.x * blockIdx.x;
        if (i == 60){
                sharedX = *x;
                sharedX = 1;
        }

        if (i == 100) *y = sharedX;
}

update <<<2,512>>>(in,out);
```

Ergebnis? => *out = 1?;

CUDA – Thread Divergence (2)

Synchronisation über Blockgrenzen

- Problemlos, wenn Blocks auf gleichem SM allokiert
- Sonst Synchronisation problematisch

```
__global__ void update_1(int* x, int* y) {
        int i = threadIdx.x + blockDim.x * blockIdx.x;
        if (i == 600) *x = 1;
}
__global__ void update_2(int* x, int* y) {
        int i = threadIdx.x + blockDim.x * blockIdx.x;
        if (i == 0) *y = *x;
}

update_1 <<<2,512>>>(in,out);
update_2 <<<2,512>>>(in,out);
```

Ergebnis? => *out = ?;

CUDA – Inline Assembly

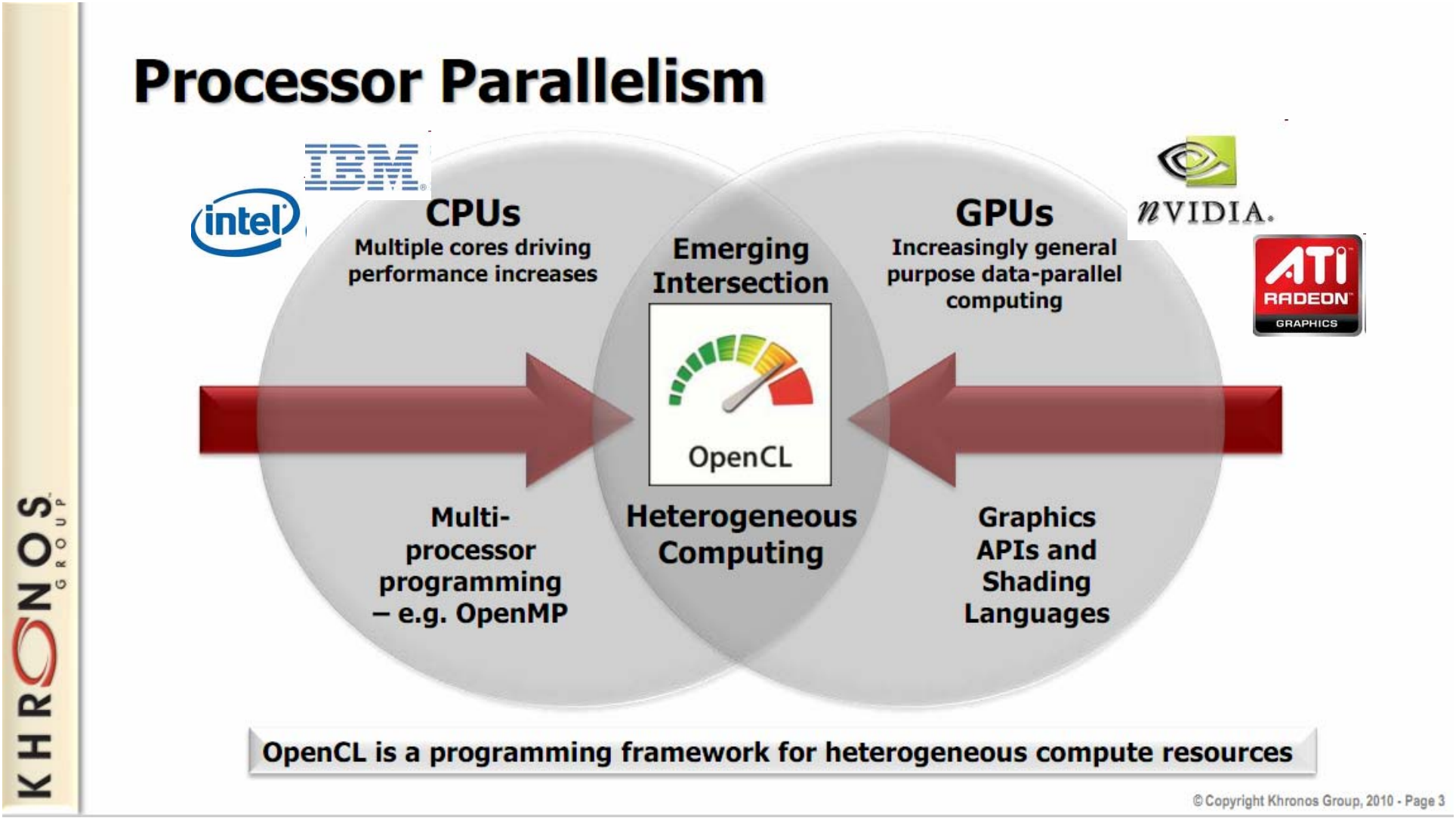
PTX-Code kann direkt im Kernel benutzt werden

- Code meist effizienter
- PTX-Instruktionen keine Hardwarebefehle

```
__global__ void kern(int* x, int* y)
{
        int i = threadIdx.x + …;
        if (i == 0) *x += 1;
        syncthreads();
        if (i == 1) *y = *x;
}
```

```
__global__ void kern(int* x, int* y)
{
        int i = threadIdx.x + blockDim.x * blockIdx.x;
        if (i == 0)
                asm("ld.global.s32   %r9, [%0+0];"
                    "add.s32  %r9, %r9, 1;"
                    "st.global.s32   [%0+0], %r9;"
                    : :"r"(x));
        syncthreads();
        if (i == 1) *y = *x;
}
```

*Open C*ompute *L*anguage - OpenCL



© http://www.khronos.org/developers/library/overview/opencl_overview.pdf

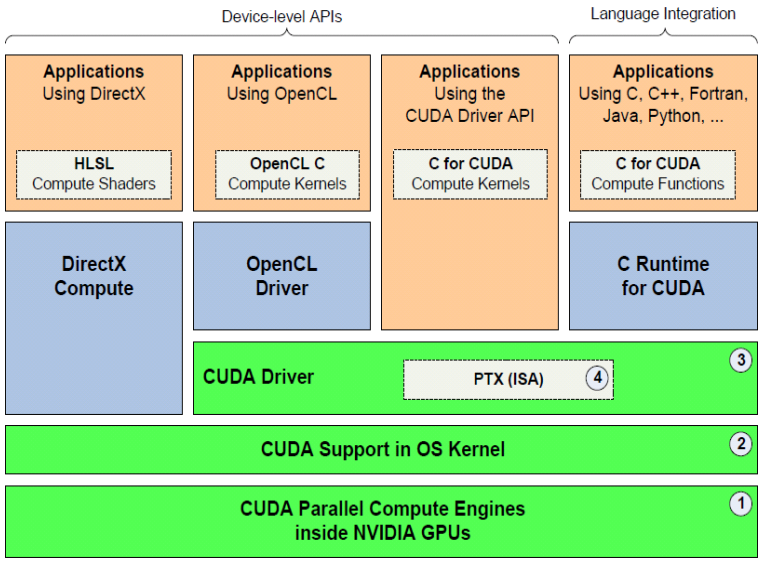
OpenCL: Vergleich zu CUDA

Unterschied zum CUDA-Ansatz

- Taskparallelität kann modelliert werden
- OpenCL-Programme werden online compiliert
- Unterstützung von heterogenen Systemen
  - Befehle, kleinster Nenner

Ausführung auf Nvidia-GPU

- Nur anderes Frontend + API
- Leider schlechterer Compiler



OpenCL für heterogene Systeme

OpenCL auf verschiedensten Plattformen zu finden

ZiiLabs Tablets

Samsung SnuCore

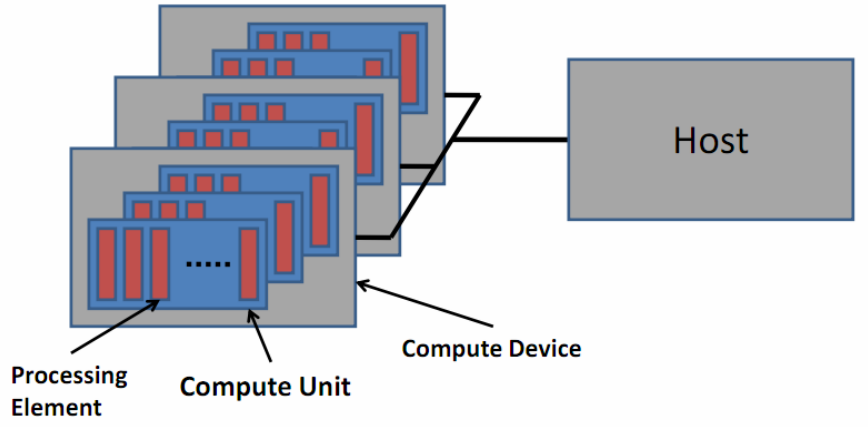
OpenCL vs. CUDA: Platform Model

Compute Device

- CUDA: Nvidia Grafikkarte
- IBM CELL Board

Compute Unit

- CUDA: Streaming Multiprozessor
- IBM CELL PPU

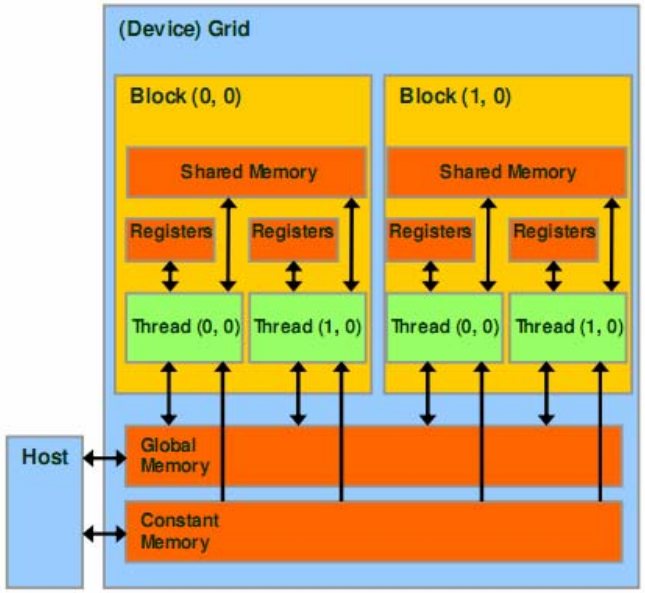


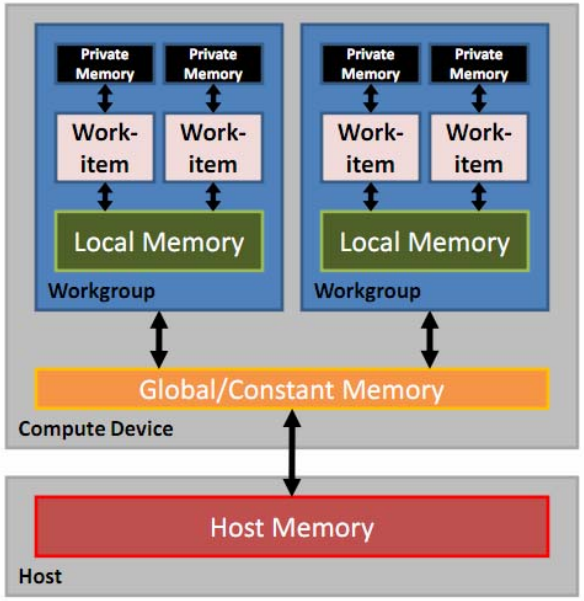
Processing Element

- CUDA: Streaming Prozessor
- IBM CELL SPU

OpenCL vs. CUDA: Memory Model

CUDA	OpenCL
Global memory	Global memory
Constant memory	Constant memory
Shared memory	Local memory
Register	Private memory





OpenCL vs. CUDA: Execution/Program. Model

CUDA	OpenCL
Kernel	Kernel
Host program	Host program
Thread	Work item
Block	Work group
Grid	NDRange (index space)

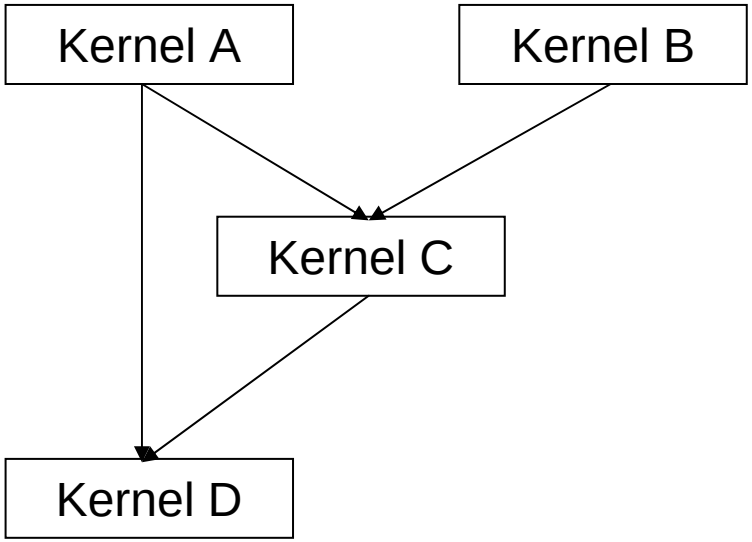
```
__global__ void vecAdd(float *a,
  float *b, float *c)
{
  int i = blockIdx.x* blockDim.x
          + threadIdx.x;
  c[i] = a[i] + b[i];
}
```

```
__kernel void vecAdd(__global
  const float *a, __global const
  float *b, __global float *c)
{
  int i = get_global_id(0);
  c[i] = a[i] + b[i];
}
```

OpenCL vs. CUDA: Task-Parallelität

Einsortierung von OpenCL-Kernel in „Command Queue“

- Synchrone Ausführung
- Asynchrone Ausführung



Zusammenfassung

- Grafikkarten können effizient zur Beschleunigung von parallelen Programmen eingesetzt werden
- NVIDIA setzt auf CUDA und OpenCL
- CUDA Programmierung ist zeitaufwendig
- OpenCL bietet
  - Alternative zu CUDA
  - Portablen Code
  - Unterstützt Taskparallelität direkt