GPGPU-Programmierung

Pascal Libuschewski
Informatik 12
TU Dortmund

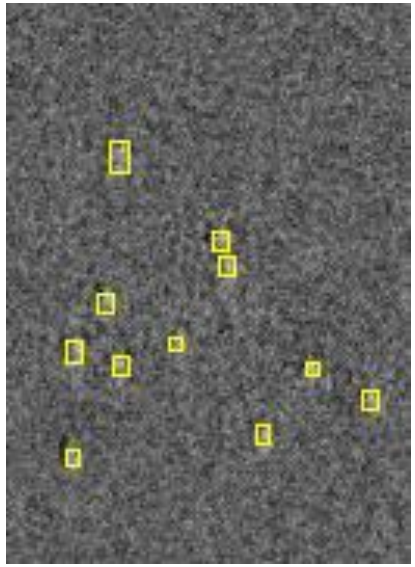
2014/04/29

Motivation (1)

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

Physikalische Berechnungen

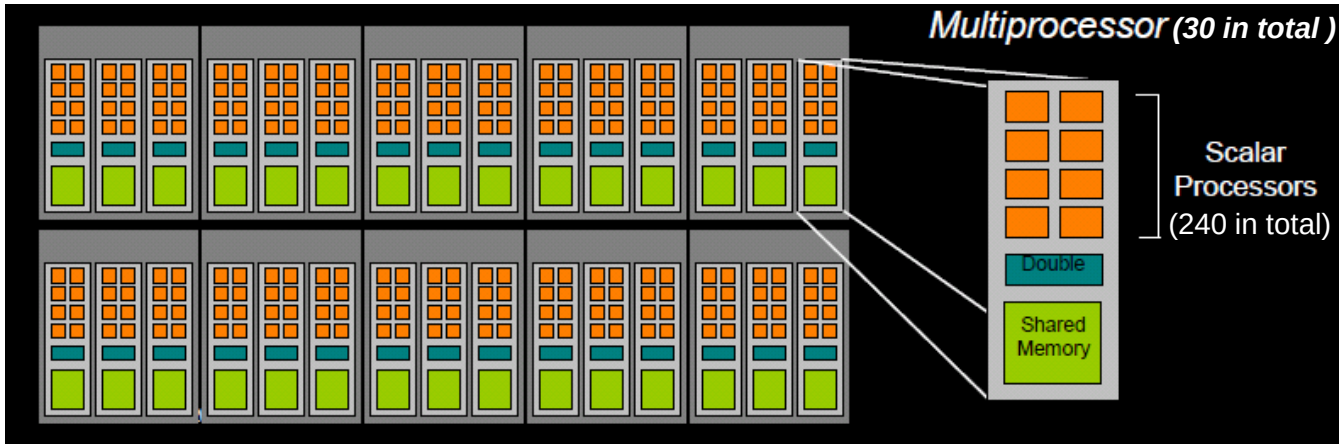
Künstliche Intelligenz



Medizinische Bildverarbeitung

Motivation (2)

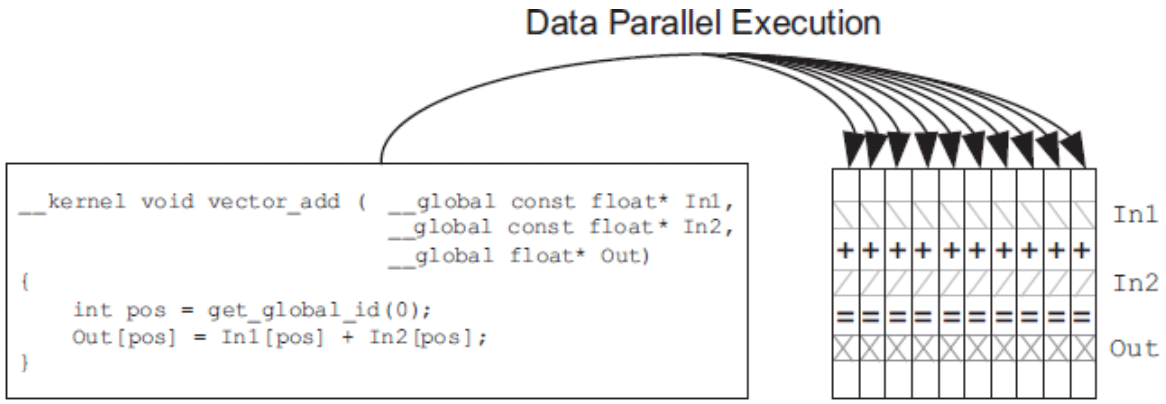
- GPUs haben eine große Anzahl von parallelen Rechenkernen
- Gut für datenparallele Programme



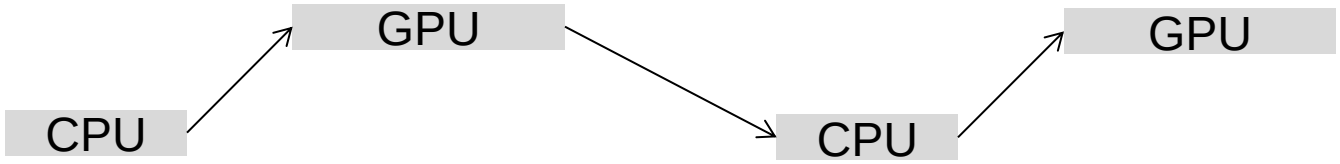
- Die GeForce GTX Titan hat z.B. 2688 Prozessoren
- Wie können diese effizient programmiert werden?

Motivation (3)

- Was sollte bzgl. GPGPU-Applikationen beachtet werden?
  - Threads sollten möglichst unabhängig voneinander sein



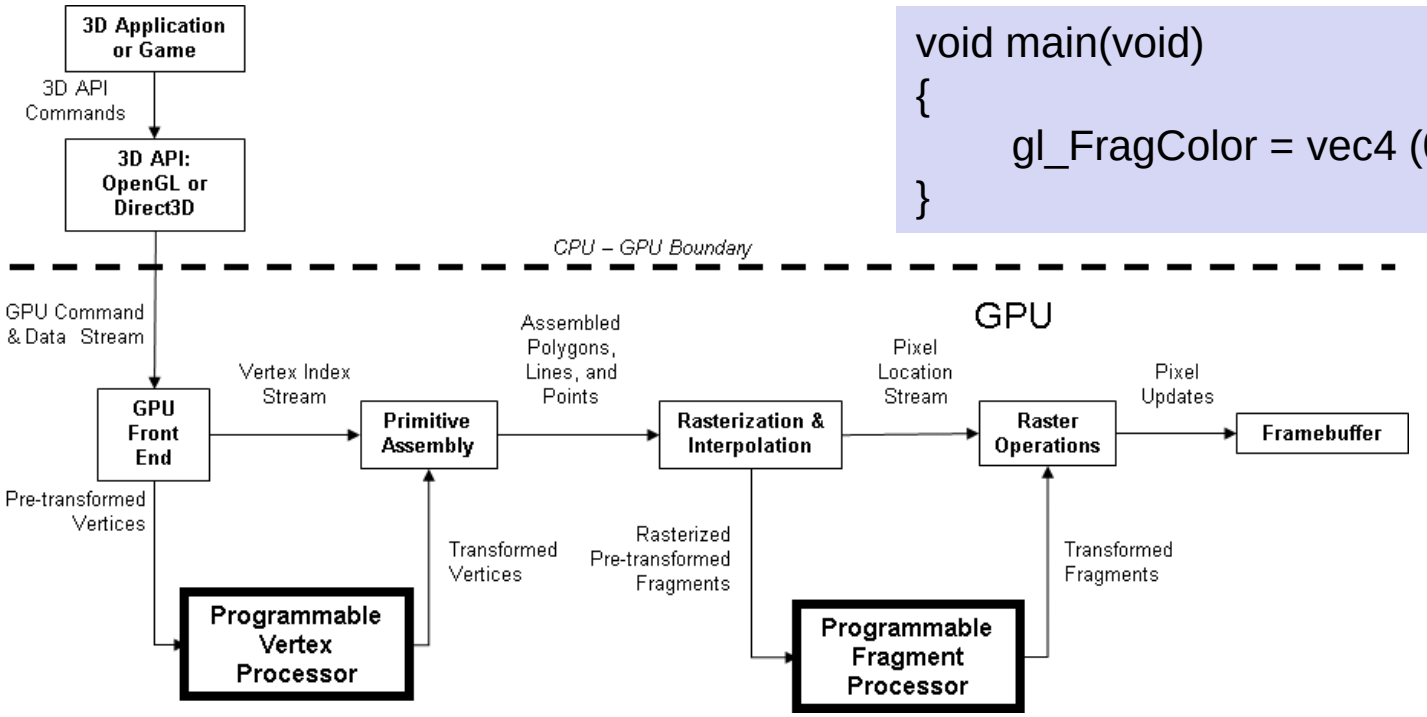
  - Zusätzlicher Kopier-Overhead für Daten



Geschichte der GPGPU-Programmierung (1)

Bis zirka 2003 – 2004

- Shader-Sprachen wurden zur Programmierung benutzt
- Vertex- und Fragment-Shader-Programme




```
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 add_vector(float in1<>, float in2<>, out float out<>)
{
        out = in1 + in2;
}

float in1<100> = {1.0f, 2.0f, ...};
float in2<100> = {2.0f, 3.0f, ...};
float out<100>;
add_vector(in1,in2,out);
```

Vektoraddition in BrookGPU Code

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 die Khronos Group: AMD, Apple, ARM, Creative, Google, Intel, Texas Instruments, 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 id = 0; id < N; id++ )
    {
        out[id] = in1[id] + in2[id] ;
    }
}
```

Vektoraddition in C

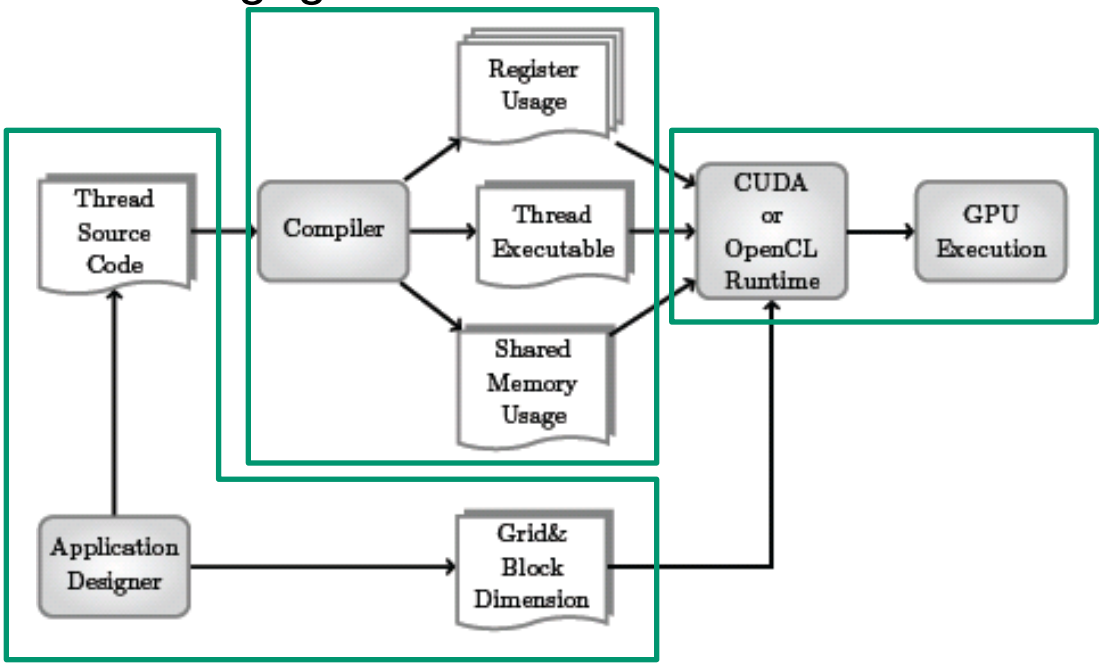
```
__global__ void add_vector(int* in1, int* in2, int* out)
{
        int id = (blockIdx.x*blockDim.x)+threadIdx.x;
        out[id] = in1[id] + in2[id];
}

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

Vektoraddition in Cuda

CUDA – Entwicklungsprozess

Mehrstufig und kompliziert

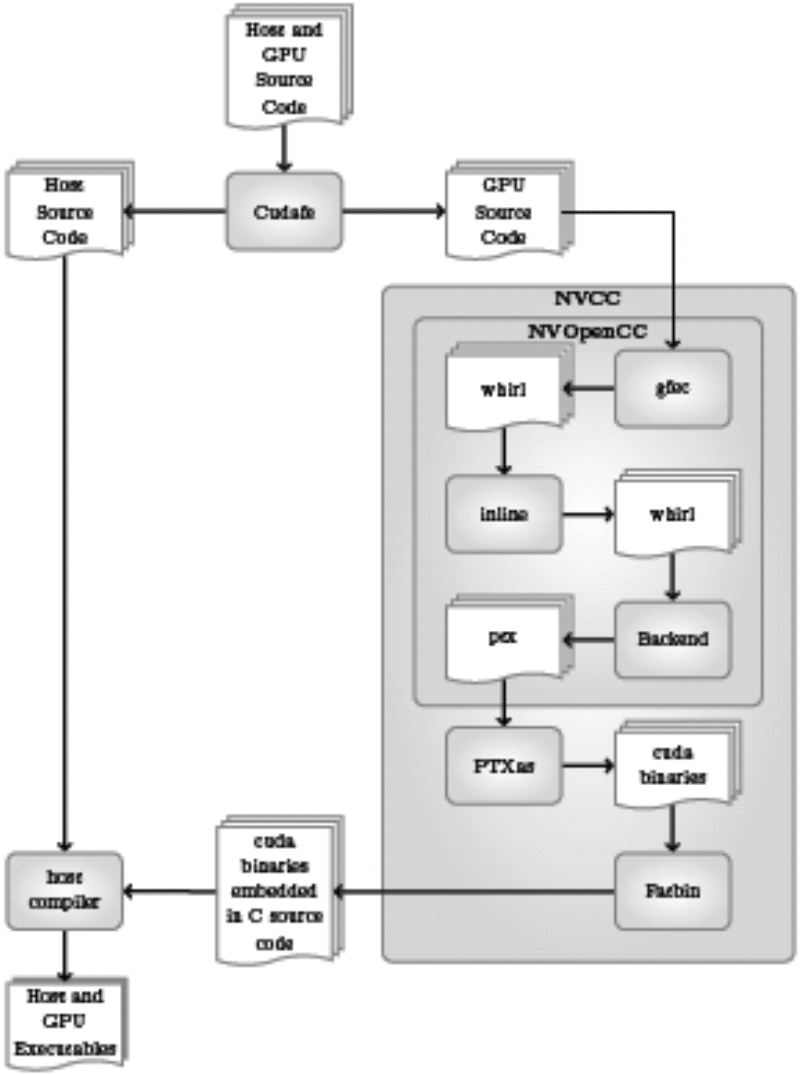
- Programmierung von Code für *einen* Thread
- Spezifikation der Parallelität per Hand
- Einige statisch vorgegebene Größen müssen beachtet werden



CUDA – Kompilierung

Mehrstufigen Verfahren für die Kompilierung von CUDA-Programmen

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



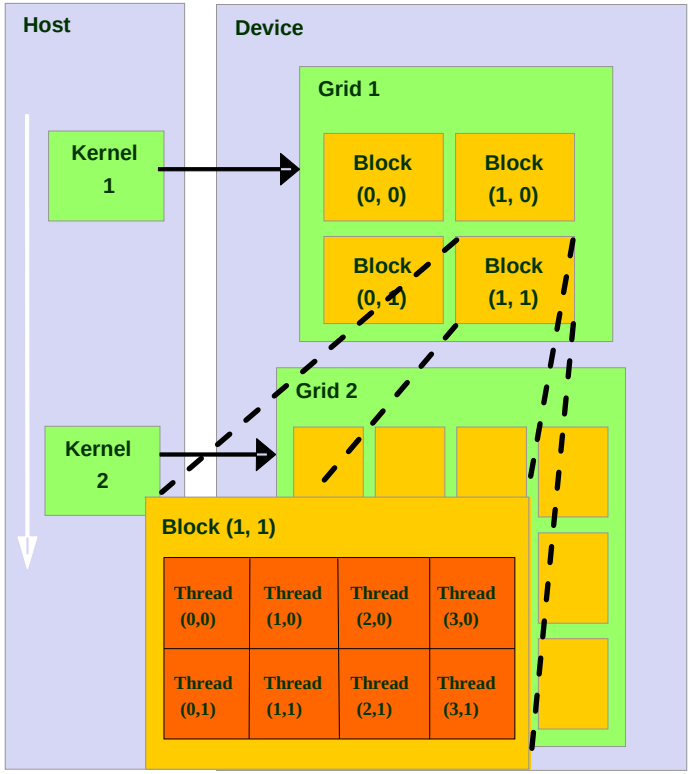
CUDA – Elemente des Frameworks

Thread
- Instanz eines Kernels

Block
- Gruppe von Threads

Grid
- Gesamtheit aller Blocks

Für Thread (2,1) in Block (1,1):
threadIdx.x: 2
threadIdx.y: 1
blockIdx.x: 1
blockIdx.y: 1
blockDim.x: 4
blockDim.y: 2
gridDim.x: 2
gridDim.y: 2



CUDA – Abbildungsbeispiel

Ein Kernel benötigt z.B. folgende Ressourcen

- Grid size: 8x8 Blöcke
- Block size: 16x16 Threads (ergibt insgesamt 128x128 Threads)
- 1024 Bytes Shared Memory per Block (z.B. 1 Float Variable pro Thread)

Beispiel Grafikkarte

- Max. 8 Blocks, 24 Warps (Warpgröße 32), 768 Threads
- Max. 16 Kilobytes Scratchpad-Speicher

Auslastung der Grafikkarte

- 3 Blocks, 24 Warps, 768 Threads, 3072 Bytes Shared Memory
  (768/32) (3*16*16) (3*1024)

CUDA - Speicherallokation

cudaMalloc()

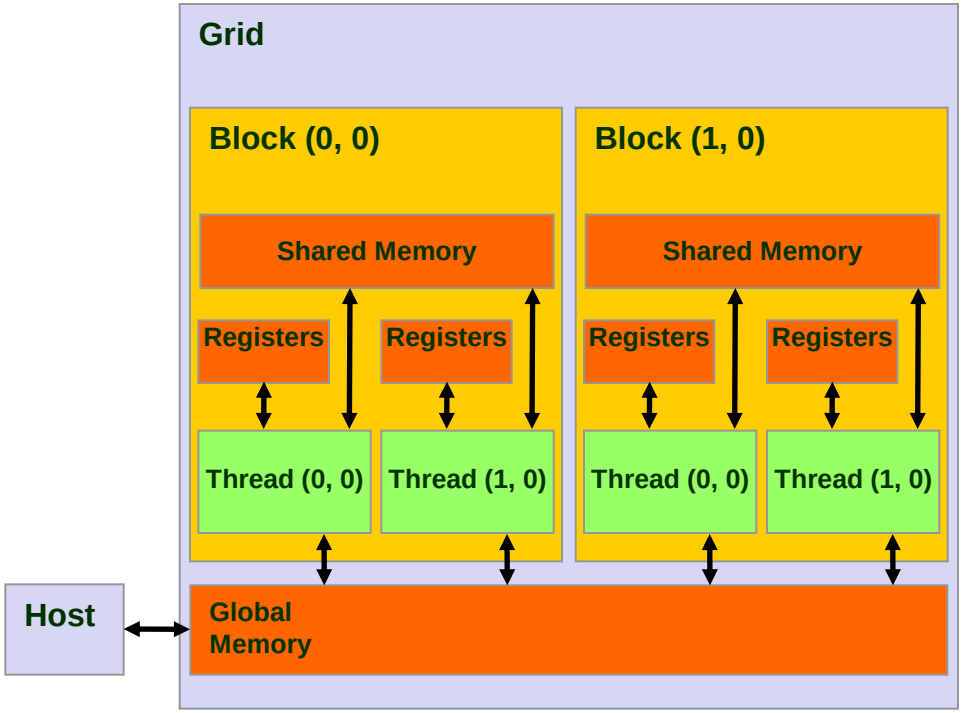
- Allokiert globalen Speicher auf der Grafikkarte

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*devMemX;
int* hostMemX = malloc(1024*sizeof(int));
cudaMalloc((void**) &devMemX, 1024*sizeof(int));
for(int i=0; i<1024; i++)
{
        hostMemX[i] = rand() % 100;
}
cudaMemcpy(devMemX, &hostMemX, 1024sizeof(int), cudaMemcpyHostToDevice);
```

- Von der Grafikkarte zum Host

```
cudaMemcpy(&hostMemX, devMemX, 1024*sizeof(int), cudaMemcpyDeviceToHost);
```

- Auf der Grafikkarte

```
cudaMemcpy(devMemX, devMemY, 1024*sizeof(int), cudaMemcpyDeviceToDevice);
```

CUDA – Speicherzugriff

Globaler/Shared Memory-Speicherzugriff

- Zugriff auf globalen/shared Speicher ist nicht synchronisiert!
- Ergebnis von Schreib-/Leseoperationen auf gemeinsamen Speicher?
  - Lösung: Atomare Operationen
  - Vorsicht: Die Ausführungsreihenfolge ist immer noch undefiniert!

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

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

Ergebnis? => out = {5,10,15,20,25} ?

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

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

Ergebnis? => out = 25

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 id = threadIdx.x + …;
        if (id == 0)
                *x += 5;
        syncthreads();
        if (id == 1)
                *y = *x;
}
```

```
__global__ void kern(int* x, int* y)
{
        int id = threadIdx.x + blockDim.x * blockIdx.x;
        if (id == 0)
                asm("ld.global.s32   %r9, [%0+0];"
                    "add.s32  %r9, %r9, 5;"
                    "st.global.s32   [%0+0], %r9;"
                    : :"r"(x));
        syncthreads();
        if (id == 1)
                *y = *x;
}
```

CUDA – Thread Divergenz (1)

Ablauf der Threads

- Vorhersage der tatsächlichen Reihenfolge ist nicht möglich!
- Programmierer kann aber Synchronisationspunkte setzen

```
__global__ void update(int* x, int* y) {
        int id = threadIdx.x + blockDim.x * blockIdx.x;
        if (id == 5){
                sharedX = *x;
                sharedX = 1;
        }

        if (id == 0)
                *y = sharedX;
}

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

Ergebnis? => *out = 1;

CUDA – Thread Divergenz (2)

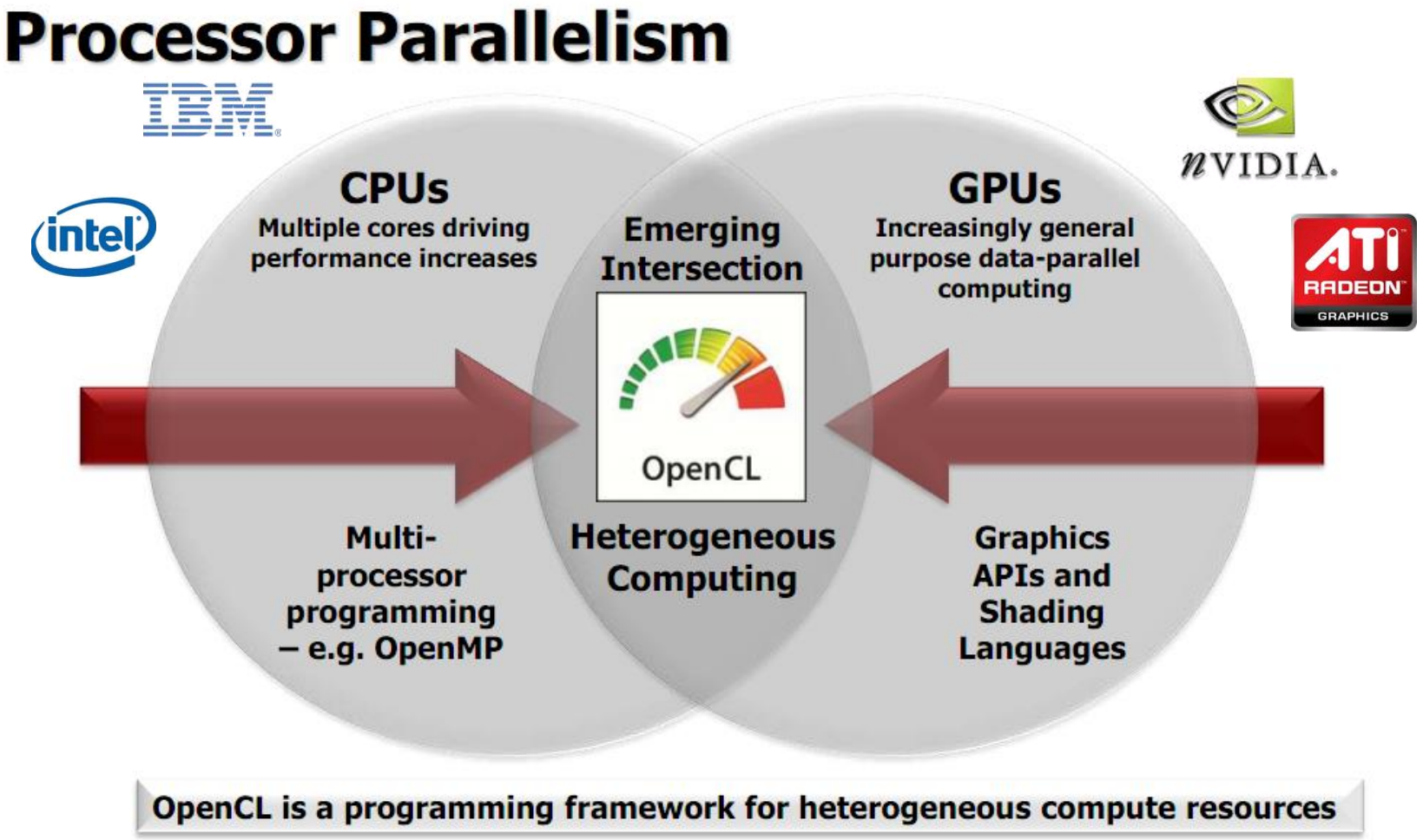
Synchronisation über Blockgrenzen

- Keine globale Synchronisation innerhalb eines Kernels!
- Lösung: Kernel aufsplitten und nacheinander ausführen

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

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

Ergebnis? => *out = 1;

Open *C*ompute *L*anguage - OpenCL



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

OpenCL für heterogene Systeme

OpenCL ist auf verschiedensten Plattformen zu finden

ZiiLabs Tablets

Samsung SnuCore

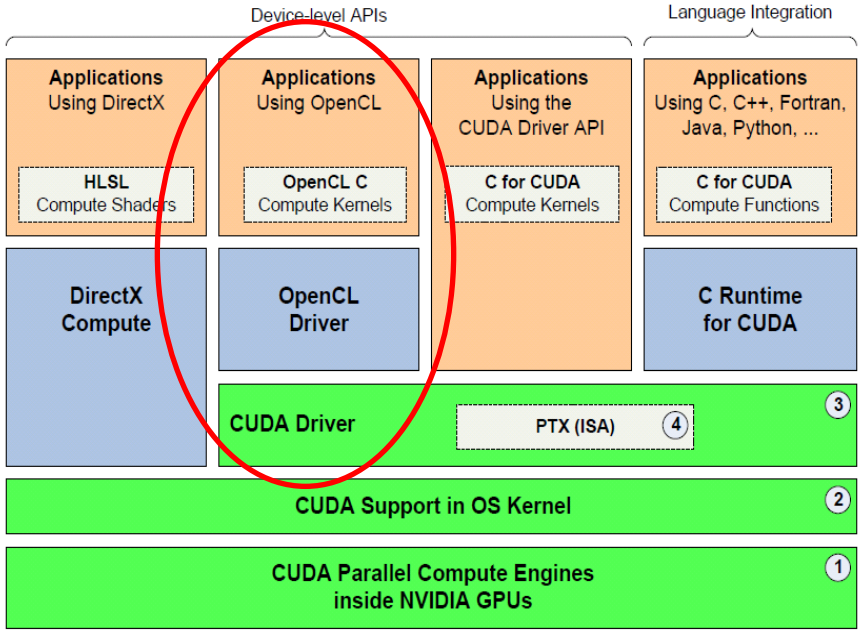
CUDA vs. OpenCL

Unterschied zum CUDA-Ansatz

- Taskparallelität kann modelliert werden
- OpenCL-Programme werden online kompiliert
- Unterstützung von heterogenen Systemen (GPUs, CPUs, Cell, ...)

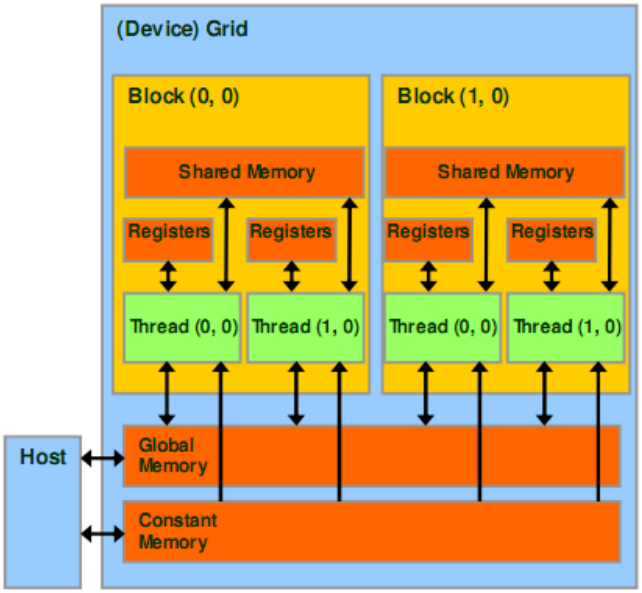
Ausführung auf Nvidia-GPU

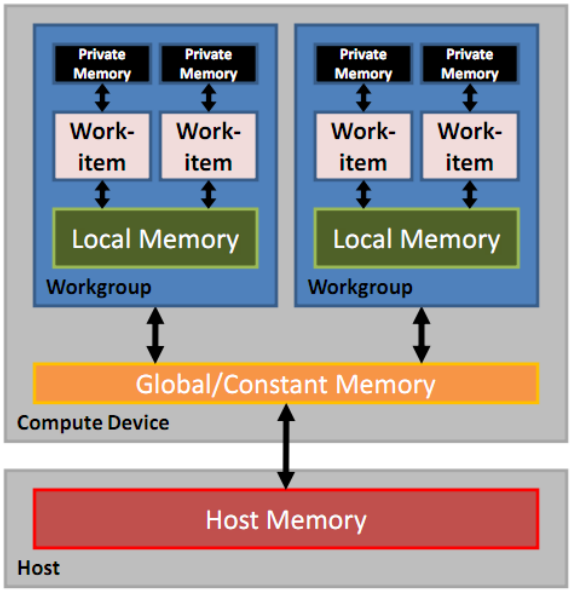
- Nur anderes Frontend + API
- Leider schlechterer Compiler
- OpenCL kann nicht alle CUDA Spezialbefehle nutzen



CUDA vs. OpenCL: Speicher Model

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





CUDA vs. OpenCL : Ausführungs/Progr. Model

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

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

Vektoraddition in Cuda

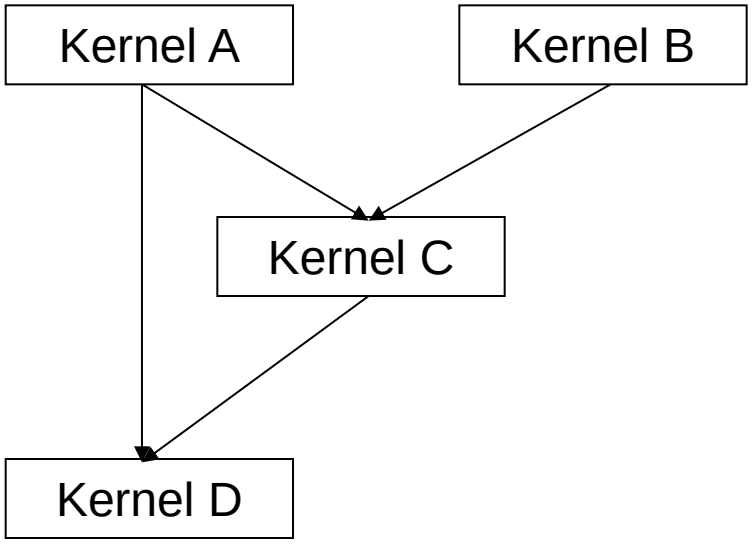
```
__kernel void add_vector (
   __global int in1,
   __global int in2,
   __global int out)
{
        int id = get_global_id(0);
        out[id] = in1[id] + in2[id];
}
```

Vektoraddition in OpenCL

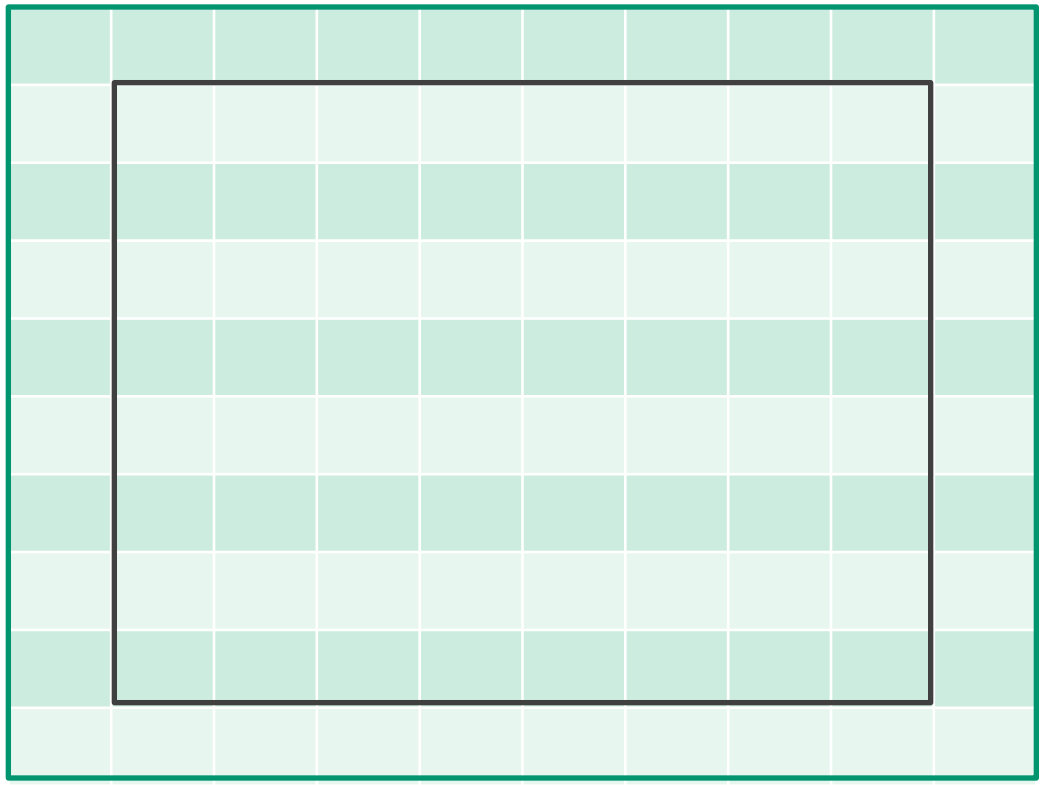
OpenCL vs. CUDA: Task-Parallelität

Einsortierung von OpenCL-Kernel in „Command Queue“

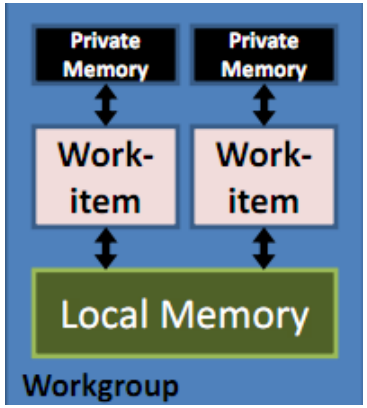
- Synchrone Ausführung
- Asynchrone Ausführung



Lokalen Speicher effizient füllen (1)

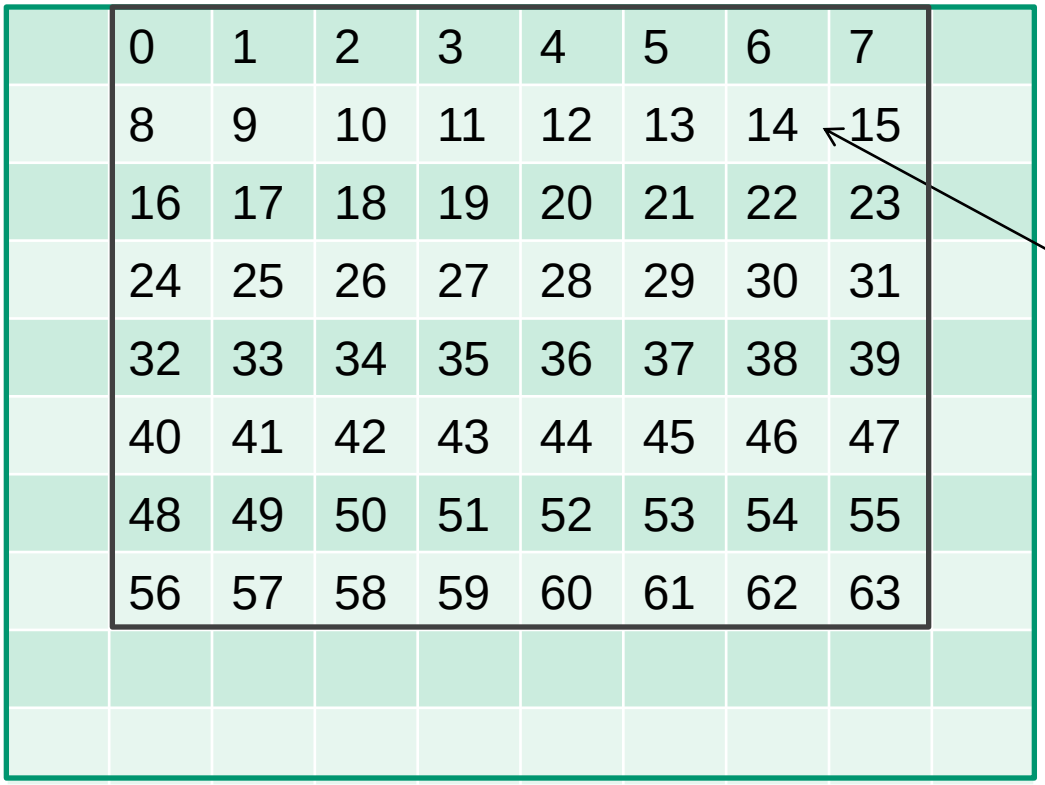


8x8 Threads laden einen
10x10 großen lokalen Speicher



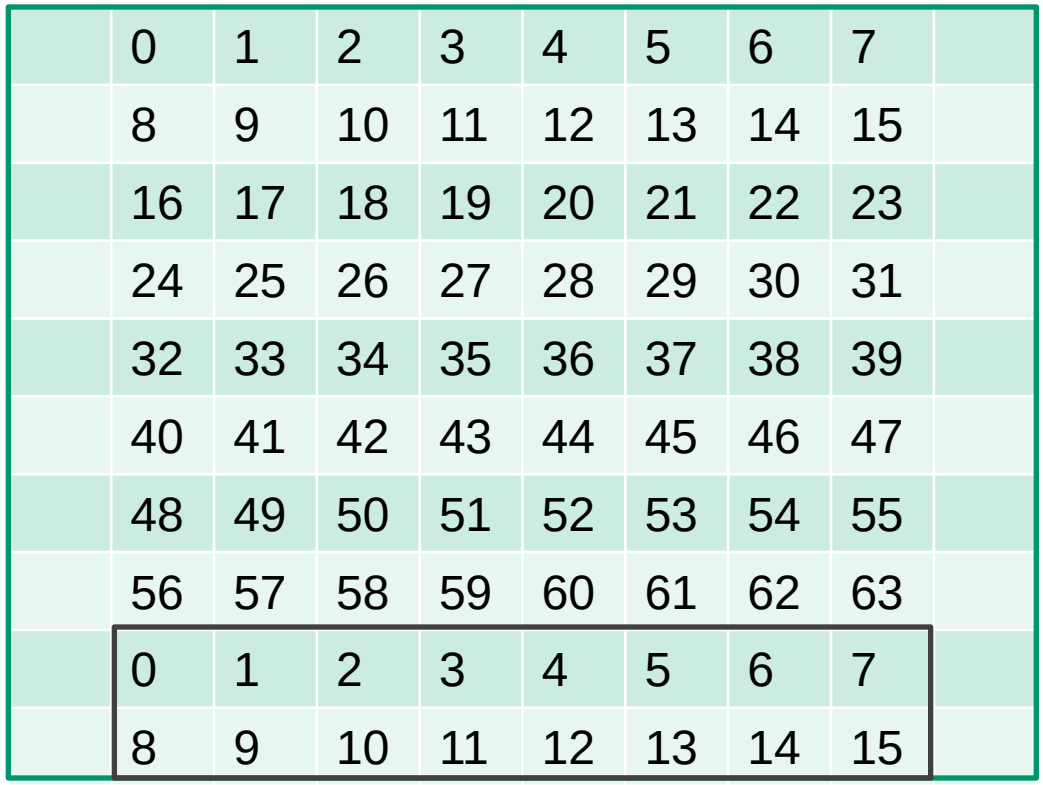
64 work items in einer work group

Lokalen Speicher effizient füllen (2)



8x8 Threads laden einen
10x10 großen lokalen Speicher

Lokalen Speicher effizient füllen (3)



8x8 Threads laden einen
10x10 großen lokalen Speicher

Lokalen Speicher effizient füllen (4)

7	0	1	2	3	4	5	6	7	
15	8	9	10	11	12	13	14	15	
23	16	17	18	19	20	21	22	23	
31	24	25	26	27	28	29	30	31	
39	32	33	34	35	36	37	38	39	
47	40	41	42	43	44	45	46	47	
55	48	49	50	51	52	53	54	55	
63	56	57	58	59	60	61	62	63	
	0	1	2	3	4	5	6	7	
	8	9	10	11	12	13	14	15	

8x8 Threads laden einen
10x10 großen lokalen Speicher

Lokalen Speicher effizient füllen (5)

7	0	1	2	3	4	5	6	7	
15	8	9	10	11	12	13	14	15	
23	16	17	18	19	20	21	22	23	
31	24	25	26	27	28	29	30	31	
39	32	33	34	35	36	37	38	39	
47	40	41	42	43	44	45	46	47	
55	48	49	50	51	52	53	54	55	
63	56	57	58	59	60	61	62	63	
7	0	1	2	3	4	5	6	7	
15	8	9	10	11	12	13	14	15	

8x8 Threads laden einen
10x10 großen lokalen Speicher

Lokalen Speicher effizient füllen (6)

7	0	1	2	3	4	5	6	7	0
15	8	9	10	11	12	13	14	15	8
23	16	17	18	19	20	21	22	23	16
31	24	25	26	27	28	29	30	31	24
39	32	33	34	35	36	37	38	39	32
47	40	41	42	43	44	45	46	47	40
55	48	49	50	51	52	53	54	55	48
63	56	57	58	59	60	61	62	63	56
7	0	1	2	3	4	5	6	7	
15	8	9	10	11	12	13	14	15	

8x8 Threads laden einen
10x10 großen lokalen Speicher

Lokalen Speicher effizient füllen (7)

7	0	1	2	3	4	5	6	7	0
15	8	9	10	11	12	13	14	15	8
23	16	17	18	19	20	21	22	23	16
31	24	25	26	27	28	29	30	31	24
39	32	33	34	35	36	37	38	39	32
47	40	41	42	43	44	45	46	47	40
55	48	49	50	51	52	53	54	55	48
63	56	57	58	59	60	61	62	63	56
7	0	1	2	3	4	5	6	7	0
15	8	9	10	11	12	13	14	15	8

8x8 Threads laden einen
10x10 großen lokalen Speicher

Lokalen Speicher effizient füllen (8)

63	56	57	58	59	60	61	62	63	56
7	0	1	2	3	4	5	6	7	0
15	8	9	10	11	12	13	14	15	8
23	16	17	18	19	20	21	22	23	16
31	24	25	26	27	28	29	30	31	24
39	32	33	34	35	36	37	38	39	32
47	40	41	42	43	44	45	46	47	40
55	48	49	50	51	52	53	54	55	48
63	56	57	58	59	60	61	62	63	56
7	0	1	2	3	4	5	6	7	0

„Natürlichstes“ Zugriffsmuster

0	1	2	3	4	5	6	7	0	1
8	9	10	11	12	13	14	15	8	9
16	17	18	19	20	21	22	23	16	17
24	25	26	27	28	29	30	31	24	25
32	33	34	35	36	37	38	39	32	33
40	41	42	43	44	45	46	47	40	41
48	49	50	51	52	53	54	55	48	49
56	57	58	59	60	61	62	63	56	57
0	1	2	3	4	5	6	7	0	1
8	9	10	11	12	13	14	15	8	9

Zugriffsmuster mit den wenigsten „Branches“

7	0	1	2	3	4	5	6	7	0
15	8	9	10	11	12	13	14	15	8
23	16	17	18	19	20	21	22	23	16
31	24	25	26	27	28	29	30	31	24
39	32	33	34	35	36	37	38	39	32
47	40	41	42	43	44	45	46	47	40
55	48	49	50	51	52	53	54	55	48
63	56	57	58	59	60	61	62	63	56
7	0	1	2	3	4	5	6	7	0
15	8	9	10	11	12	13	14	15	8

Effizientestes Zugriffsmuster (bei Nvidia GPU)

Vorteile/Nachteile

- GPU Programmierung ist aufwändig
- Projekt mit 11.000 Zeilen OpenCL Code
- hat 10.000 Zeilen C Host Code
- Ca. 90% Overhead
- Minimum/Maximum Berechnung eines Arrays
  - In C++: 6 Zeilen
  - In OpenCL/CUDA: 80 Zeilen OpenCL Code + 130 Zeilen C Host Code
- Dafür sehr schnell:
  - 1.2 Millisekunden auf CPU
  - 0.012 Millisekunden auf GPU

Zusammenfassung

- Grafikkarten können effizient zur Beschleunigung von parallelen Programmen eingesetzt werden
- Nvidia setzt auf CUDA (und OpenCL)
- GPGPU Programmierung ist zeitaufwendig

- OpenCL bietet
  - Task- und Datenparallelität
  - Eine offene Alternative zu CUDA
  - Einen portablen Code für eine Vielzahl von Geräten