GPU computing mit CUDA

VL ROOT Datenanalyse – Jörg Marks

Eric Volkmann

05.07.2019

Universität Heidelberg

Table of contents

Grundlagen GPU

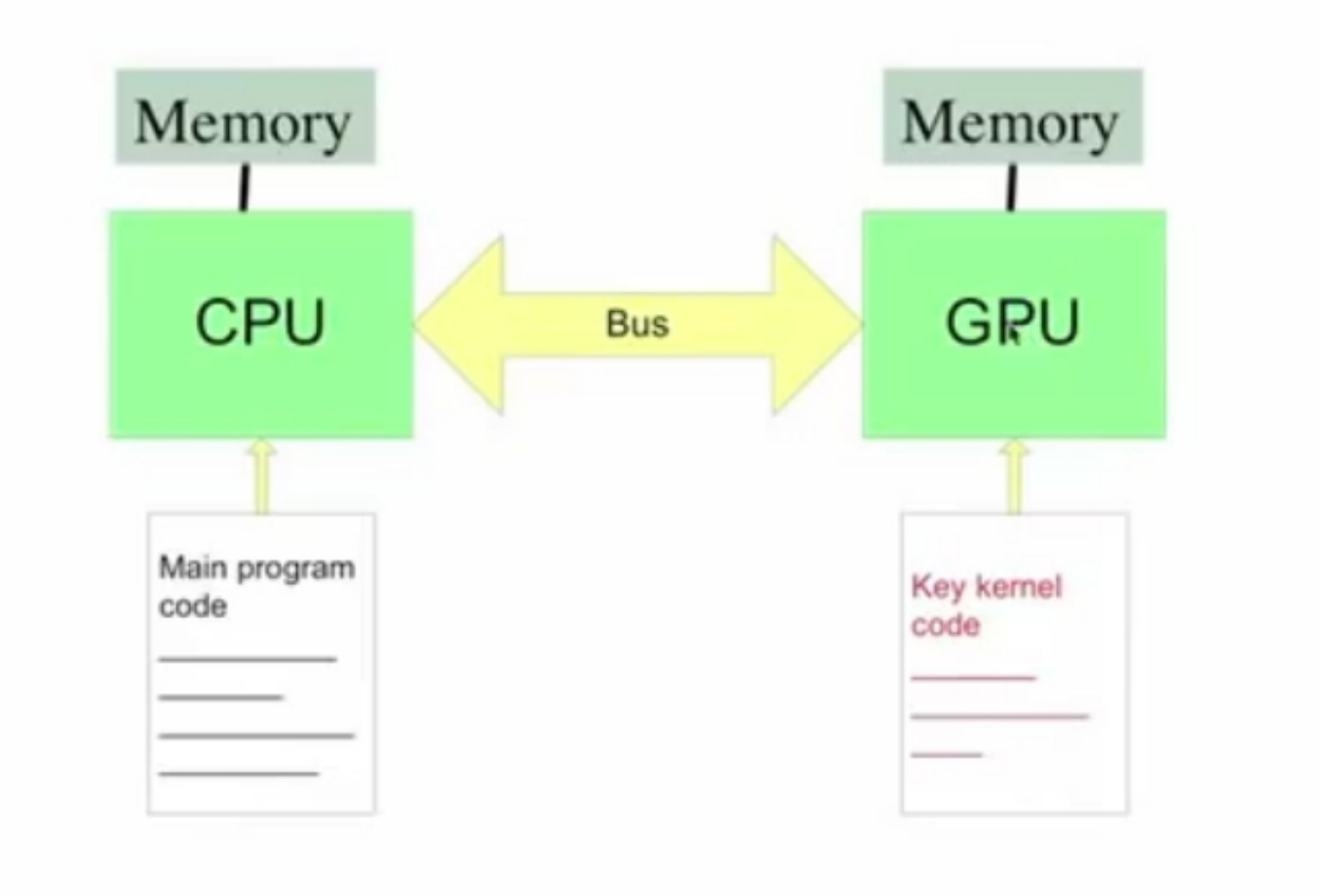
- GPU´s sind auf Grafikberechnungen spezialisierte Prozessoren
- Hohes Maß an Parallelisierung → theo. höhere Rechenleistung als eine CPU
- Moderne GPU´s rechnen mit double precision

Grundlagen GPU

GPU können niemals alleine arbeiten, sondern müssen in Kombination mit einer CPU verwendet werden

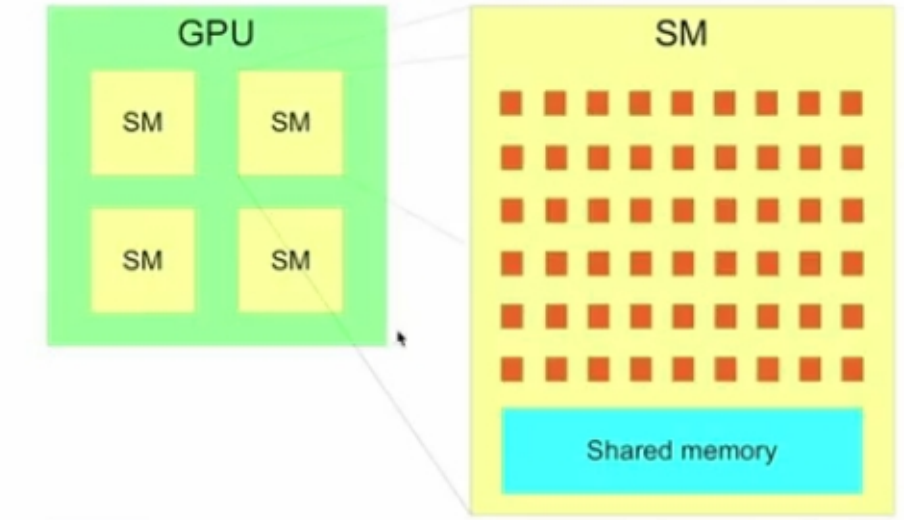
- GPU beschleunigt rechenaufwendige Teile des code, die Kernals
- Kernals werden aufgespalten um parallel auf mehreren Kernen zu laufen

Grundlagen GPU



Grundlagen GPU

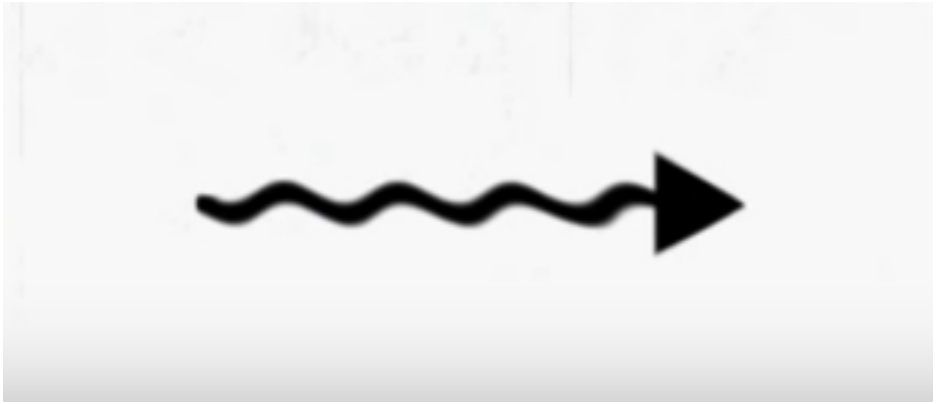
- NVIDIA GPU´s haben eine 2-level Hierarchie
- SM: Streaming Multiprocessor
- Jeder dieser SMs besitzt mehrere Kerne
- Anzahl der SMs und Kerne pro SM variiert über GPU Modelle



Threads

- Einzelne execution unit, die auf den Kernels der GPU laufen
- Analog zu CPU threads
- Viel mehr als auf einer CPU

Werden oft als Pfeile gezeichnet



Warps

- Warp: In einem SM werden 32 threads zu einem Warp zusammengefasst und ausgeführt
- In jedem thread eines warps wird derselbe Code ausgeführt
- Dies ist auf Hardware-Seite festgelegt!

Blocks

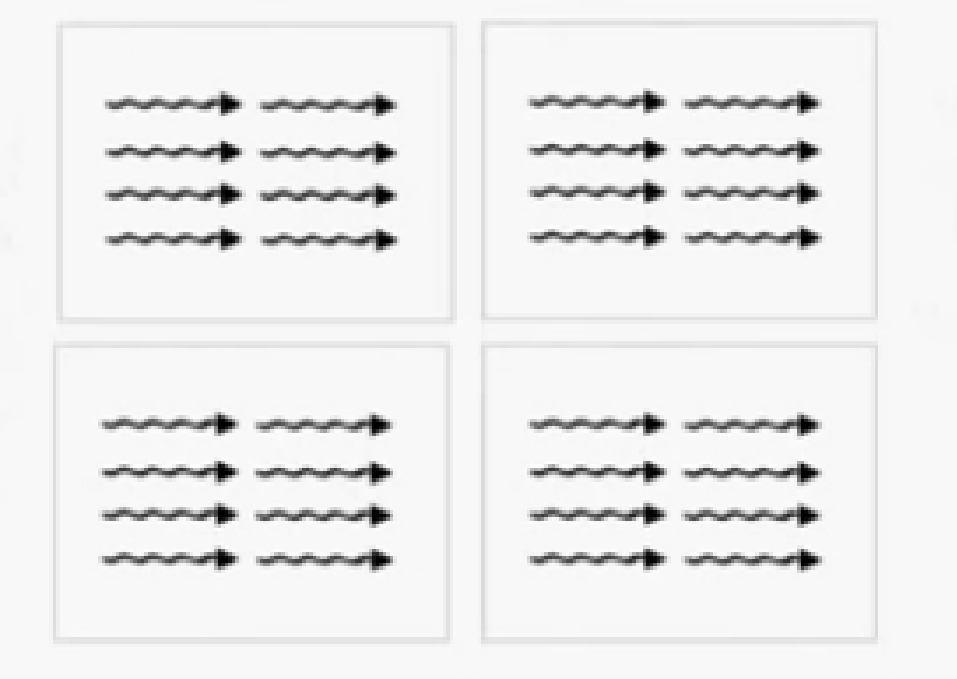
- Gruppierung von Threads
- Alle Threads in einem Block können kommunizieren
- Wollen immer ein Vielfaches von 32 an Threds pro Block

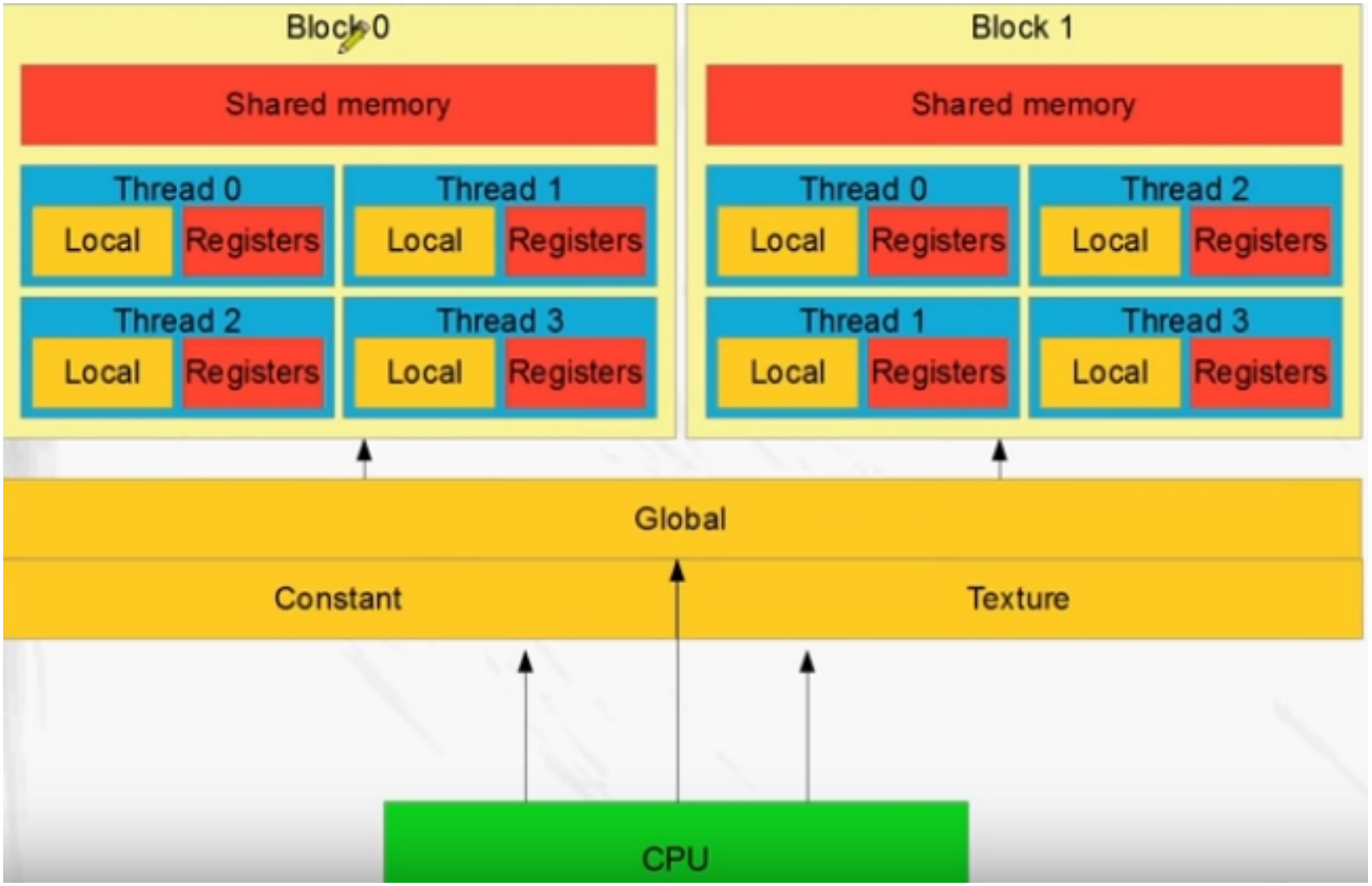
Thread Blocks werden als Boxen mit enthaltenen Threads gezeichnet

Grids

- Ein Kernel wird als Grid gestartet
- Ein Grid ist eine Sammlung von Thread Blocks
- Können mit bis zu 3D initialisiert werden

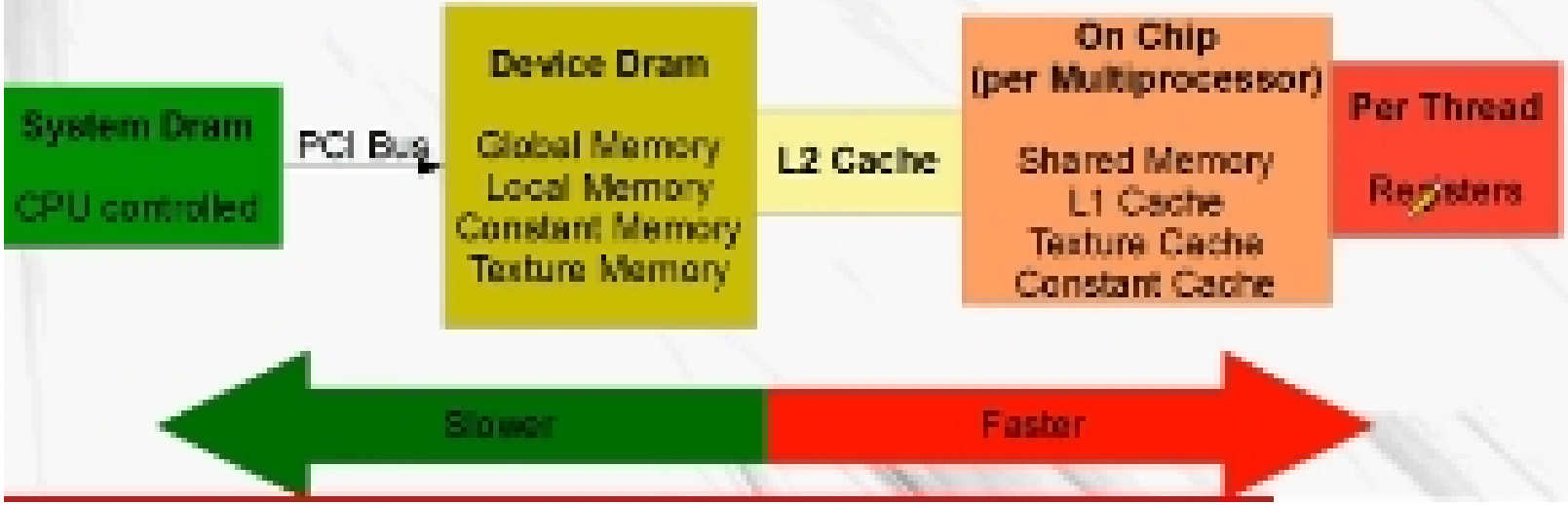
Skizze:




Block 0
Shared memory
Thread 0
Local
Registers
Thread 1
Local
Registers
Thread 2
Local
Registers
Thread 3
Local
Registers
Block 1
Shared memory
Thread 0
Local
Registers
Thread 2
Local
Registers
Thread 1
Local
Registers
Thread 3
Local
Registers
Global
Constant
Texture
CPU

Summary

Memory	Access	Scope	Lifetime	Speed	Note
Global	RW	All threads and CPU	All	Slow, cached	Large
Constant	R	All threads and CPU	All	Slow, cached	Read same address
Texture	R	All threads and CPU	All	Slow, cached	Addressing perks
Local	RW	Per thread	Thread	Slow, cached	Register spilling
Shared	RW	Per block	Block	Fast	Fast com. ↔ threads
Registers	RW	Per thread	Thread	Fast	Don't use too many!



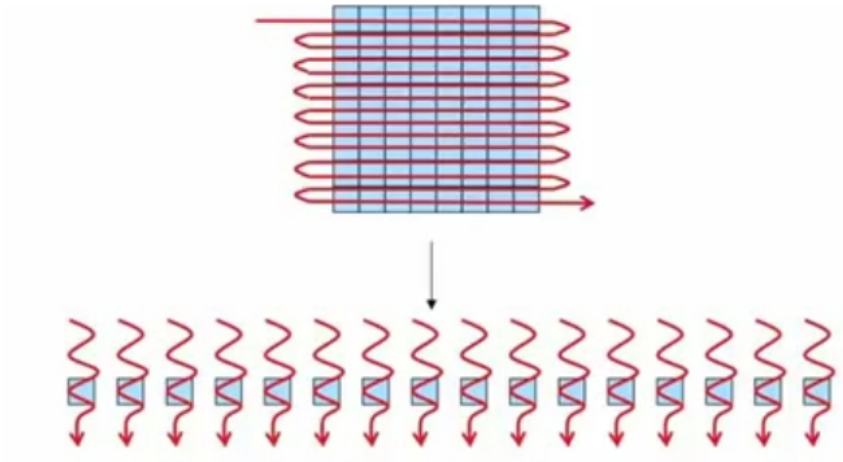
Größenordnungen in einer GPU

- Man kann bis zu 1024 threads pro Block starten

  (512 wenn die compute capability 1.3 oder weniger ist)
- Man kann bis zu 2^(32)-1 Blocks starten

  (oder 2^(16)-1 wenn die compute capability 2.0 oder weniger ist)
- Also kann unsere einfache GPU problemlos 67.108.864 Threads initialisieren

Intuition von Parallelisierung



- Daten werden in einen Strom von Elementen aufgeteilt
- Eine einzelne Funktion/Berechnung operiert auf jedem Element
- Mehrere Kerne können je verschiedene Elemente parallel verarbeiten
- Lösen von “data-parallel” Aufgaben


CUDA.
HOTEL
CRU

CUDA Hotel

- Serielle Lösung:
  - An der Rezeption wird nacheinander jede neue Nummer berechnet
  - Gäste werden einzeln nacheinander ins neue Zimmer bewegt

```
for (i=0;i<N;i++){
  result[i] = 2*i;
}
```

- Parallele Lösung:
  - Jeder überprüft seine Zimmernummer
  - Berechnet neue Nummer
  - Geht ins neue Zimmer

```
__global__ void myKernel(int *result)
{
  int i = threadIdx.x;
  result[i] = 2*i;
}
```

CUDA Grundlagen: Dim3

- Dim3 ist eine 3d Struktur bzw Vektor mit 3 int´s x, y, z
- Initialisierung
  - dim3 grid(512); //512 x 1 x 1
  - dim3 block(1024, 1024) //1024 x 1024 x 1
  - dim3 anotherOne (10, 54, 32); // 10 x 54 x 32

  - Analog existiert float3

Function Qualifiers

Werden genutzt, um die Rolle einer Funktion festzulegen:

__global__ vom Host gerufen, wird auf dem Device ausgeführt, kernals werden so markiert

__device__ vom Device gerufen, wird auf dem Device ausgeführt

__host__ (oder kein Qualifier) normale Host Funktion

Kernel Invocation

- Aufruf eines Kernel´s durch den Host mit <<< >>>
- In die Chevrons kommen die Anzahl der Blöcke und Anzahl der Threads pro Block
  - SomeKernel <<<100,256>>>(…);

- Mit dim3 Variablen können auch 3d grids und blocks initialisiert werden

```
dim3 blocksPerGrid(10,10,1);
dim3 threadsPerBlock(1024,1,1);
myKernel<<<blocksPerGrid, threadsPerBlock>>>(result);
```

Jeder Thread ist individuell

Jeder Thread weiß folgendes über sich:

- ThreadIdx ← Thread Index innerhalb eines Blocks
- BlockIdx ← Block Index innerhab des Grids
- BlockDim ← Block Dimension
- GridDim ← Grid Dimension

Dies sind alles Variablen vom Typ dim3.

Eine Id, die für jeden Thread individuell ist:

```
int id = blocksId.x*blockDim.x + threadIdx.x;
```

GPU Memory

- Die GPU hat KEINEN Zugang zum System Ram, alle Daten müssen zwischen Host und Device explizit ausgetauscht werden
- Daten, die in den Kernels verarbeitet werden, müssen auf dem GPU Memory sein
- Wir müssen den Datenaustausch regeln
- Dies ist ein riesiger Bottleneck

Speicherallozііerung

- CudaMalloc alloziiert GPU Memory
- CudaFree lässt den Memory wieder frei

```
float *a;
cudaMalloc(&a, soizeof(float));
…
cudaFree(a);
```

Datenaustausch

- Sobald wir Memory alloziiert haben, können wir Daten hin und her kopieren

```
cudaMemcpy(array_device, array_host, N*sizeof(float),
      cudaMemcpyHostToDevice);
cudaMemcpy(array_host, array_device, N*sizeof(float),
                  cudaMemcpyDeviceToHost);
```

- Erstes Argument ist das Ziel des Transfers
- CudaMemcpy Aufrufe sind blocking

Blocking ↔ Non-blocking

- Kernel Aufrufe sind non-blocking
- Das bedeutet, dass das Host Programm nach dem Aufruf weiterläuft und NICHT auf ein Ergebnis wartet
- Nutze cudaThreadSynchronize() oder cudaDeviceSynchronize() um auf Beendigung des Kernels zu warten

Beispiel: Matrixaddition

- Matrixaddition parallelisiert:

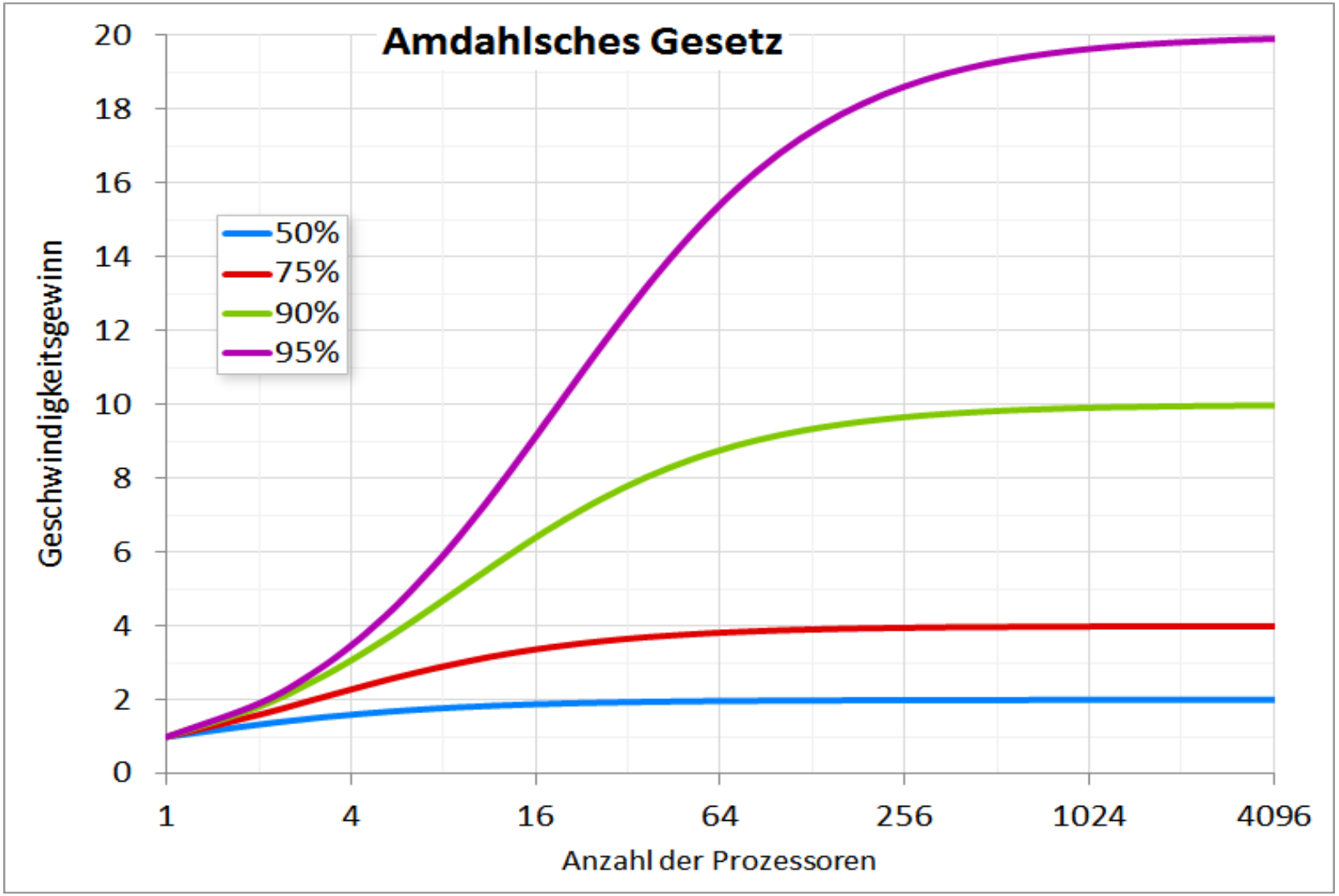
```
__global__ void matrixAdd(float a[N][N], float b[N][N], float c[N][N])
{
  int j = blockIdx.x * blockDim.x + threadIdx.x;
  int i = blockIdx.y * blockDim.y + threadIdx.y;

  c[i][j] = a[i][j] + b[i][j];
}

int main()
{
  dim3 blocksPerGrid(N/16,N/16,1); // (N/16)x(N/16) blocks/grid (2D)
  dim3 threadsPerBlock(16,16,1); // 16x16=256 threads/block (2D)
  matrixAdd<<<blocksPerGrid, threadsPerBlock>>>(a, b, c);
}
```

GPU performance inhibitors

- Datenübertragung von/zur GPU
- Zu geringe Ausnutzung des Device
- GPU memory latency
- GPU memory bandwidth
- Code branching

Ausnutzung der GPU

- GPU performance hängt am Grad der Parallellisierung
- In einem Programm muss so viel parallelism wie möglich ausgenutzt werden
- Teilweise muss der Code komplett überarbeitet werden
- Beschleunigung durch GPU wird durch die seriellen Teile des Code beschränkt (Amdahlsches Gesetz)


Amdahlsches Gesetz
Geschwindigkeitsgewinn
Anzahl der Prozessoren
50%
75%
90%
95%
0
2
4
6
8
10
12
14
16
18
20
1
4
16
64
256
1024
4096

Occupancy und Memory Latency

- Wir verwandeln z.B. die Schleifen im code in threads

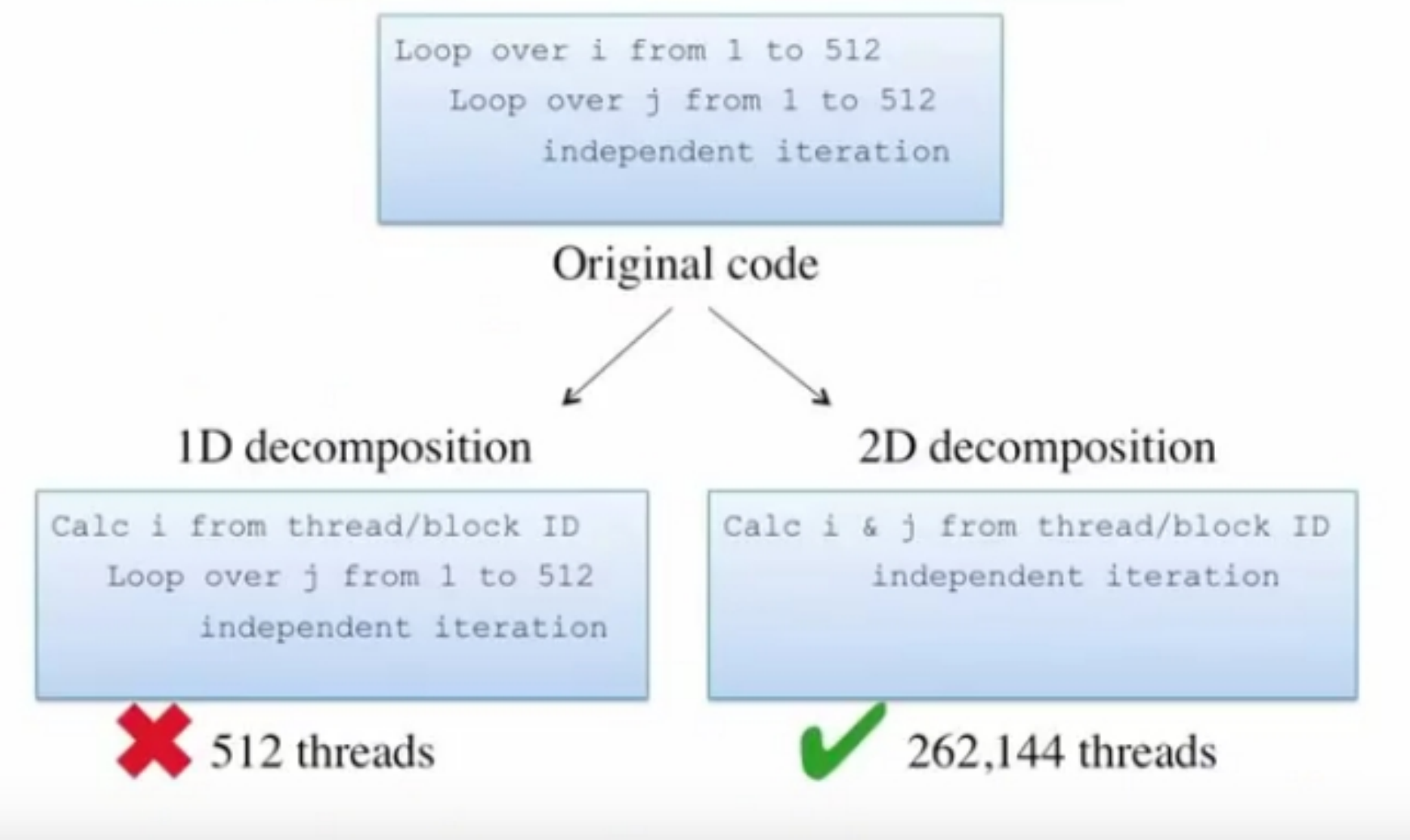
  → natürlich müssen wir min. soviele threads wie Kerne haben, sonst bleibt ein Teil im idle

- Für beste Performance brauchen wir

  #threads >> #cores

- NVIDIA GPUs besitzen sehr schnelles thread switching
- Der Zugang zu GPU Memory hat recht hohe Latenzzeiten
  - Während ein Thread noch auf Daten wartet, kann ein anderer während der Latenzzeit ausgeführt werden

Beispiel: Ausreizen von Parallelism



Memory Coalescing

- GPUs haben eine hohe memory bandwidth
- Diese wird nur ausgenutzt, wenn auf Daten für verschiedene Threads in einer einzigen Übertragung zugegriffen wird

  → Aufeinander folgende Threads greifen auf aufeinander folgende memory locations zu
- Ohne dies wird der Zugriff seriell ausgeführt

  → Performanceverluste

Beispiel: Memory Coalescing

```
i = blockIdx.x*blockDim.x + threadIdx.x;
for (j=0; j<N; j++) {
output[ i ] [ j ] = 2*input[ i ] [ j ]; }
```

Oder

```
j = blockIdx.x*blockDim.x + threadIdx.x;
for (i=0; i<N; i++) {
    output[ i ] [ j ] = 2*input[ i ] [ j ]; }
```

Code Branching

- Auf GPUs gibt es hardwareseitig weniger Einheiten, die Instruktionen zuweisen, als Kerne
- Daher die Warps, die Gruppen von 32 billden
- Innerhalb eines Warps wird von allen Kernen der gleiche Code ausgeführt
- Müssen Verzweigung innerhalb eines Warps vermeiden
- CUDA erlaubt diese Verzweigungen zwar, aber sie vermindern performance

```
i = blockIdx.x*blockDim.x + threadIdx.x;
if (i%2 == 0)
   …
else
   …
```

✖ Threads within warp diverge

```
i = blockIdx.x*blockDim.x + threadIdx.x;
if ((i/32)%2 == 0)
   …
else
   …
```

✔ Threads within warp follow same path

Quellen

- “Learn CUDA in an Afternoon: an Online Hands-on Tutorial” von EdUniPhysicsAstro, Dr. Alan Gray, https://www.youtube.com/watch?v=_41LCMFpsFs
- “CUDA Tutorials” von What´s a Creel?
- https://www.youtube.com/playlist?list=PLKK11Ligqititws0ZOoGk3SW-TZCar4dK
- NVIDIA: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Aufgaben

- Nr. 1: nutze Vorlage Nr1.cu
  - Alloziiere device memory für das array h_a
  - Kopiere das array vom Host zur GPU
  - Kopiere das array von der GPU zum Host
  - Gib den Memory wieder frei
  - Implimentiere den Kernel negate() mit einem 1d Grid und single thread blocks
  - Implimentiere den Kernel negate_multiblock() mit multiple thread blocks

Aufgaben

- Nr. 2: nutze die Vorlage Nr2.cu

  In der Vorlage wird das Array points mit random float3 Werten gefüllt. Es soll für einen beliebigen Eintrag der Vektor mit dem geringsten Abstand gefunden werden. Abstand ist über euklid. Metrik definiert

  Eine Serielle Lösung auf der CPU ist gegeben

  Implimentieren sie eine parallele Lösung auf der GPU

  Prüfen sie die Geschwindigkeit mit nvprof