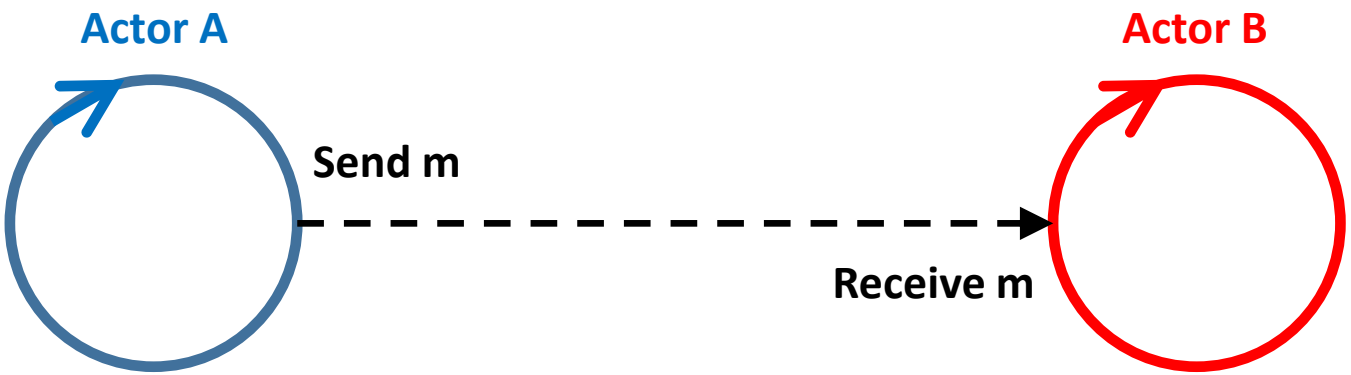
Parallele Programmierung
GPU Parallelisierung 1

Vorlesung 10

Prof. Dr. Luc Bläser

Letzte Vorlesung - Rückblick

Actor Model



Nachrichtenaustausch statt Shared Memory

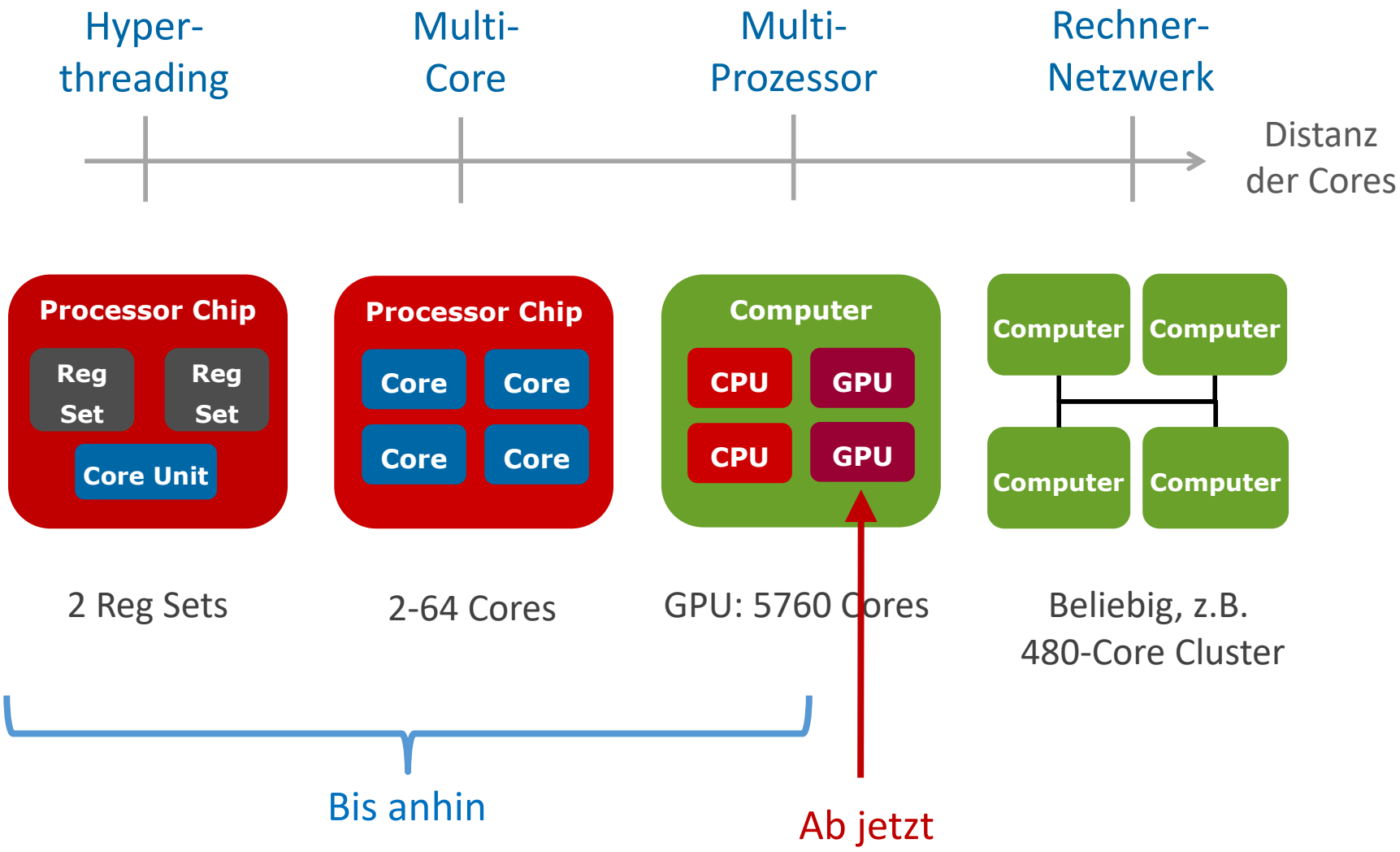
Welche Vorteile bietet das Actor Model für die Parallelisierung?

Möglichst hohe Parallelisierung

- CPUs bieten eher wenig Cores
  - Z.B. 4, 8, 16, 64 Cores
  - Allgemeine schnelle Prozessoren
- GPUs bieten sehr viele Cores
  - 512, 1024, 3584, 5760 Cores
  - Sehr spezifische langsamere Prozessoren

Ziel: GPU Many Cores für Parallelisierung nutzen

Stufen der Parallelisierung



Inhalt Heute

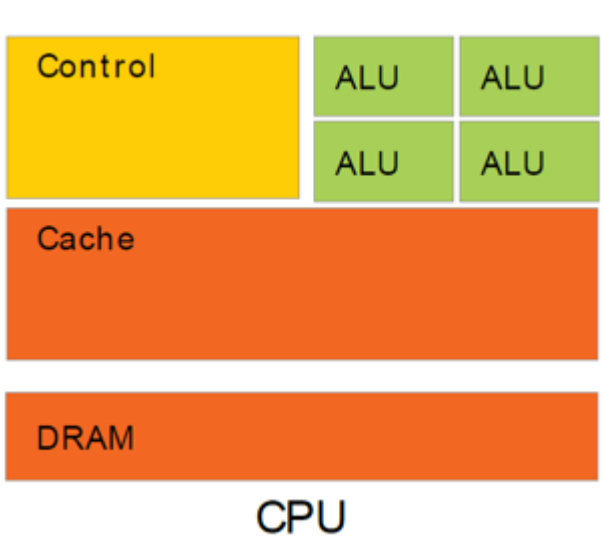
- GPU Co-Prozessor Architektur
- CUDA Programmiermodell
- Kernel Definition und Launch
- Grid, Block, Thread Aufteilung
- Compilation, Execution
- Memory Management
- Anhang: OpenCL Vergleich

Lernziele

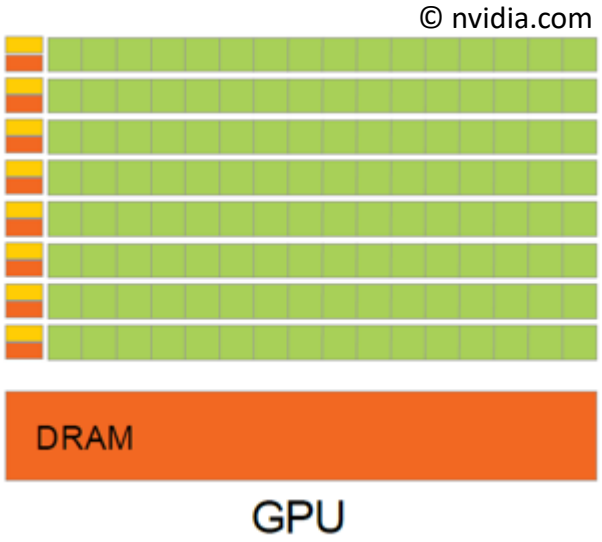
- Die GPU-Prozessorarchitektur verstehen
- CUDA (oder OpenCL) als Programmierframework für GPU-Parallelisierung anwenden können
- Erste Tipps und Tricks für GPU-Parallelisierung kennenlernen

CPU versus GPU

GPUs verwenden mehr Transistoren für Recheneinheiten

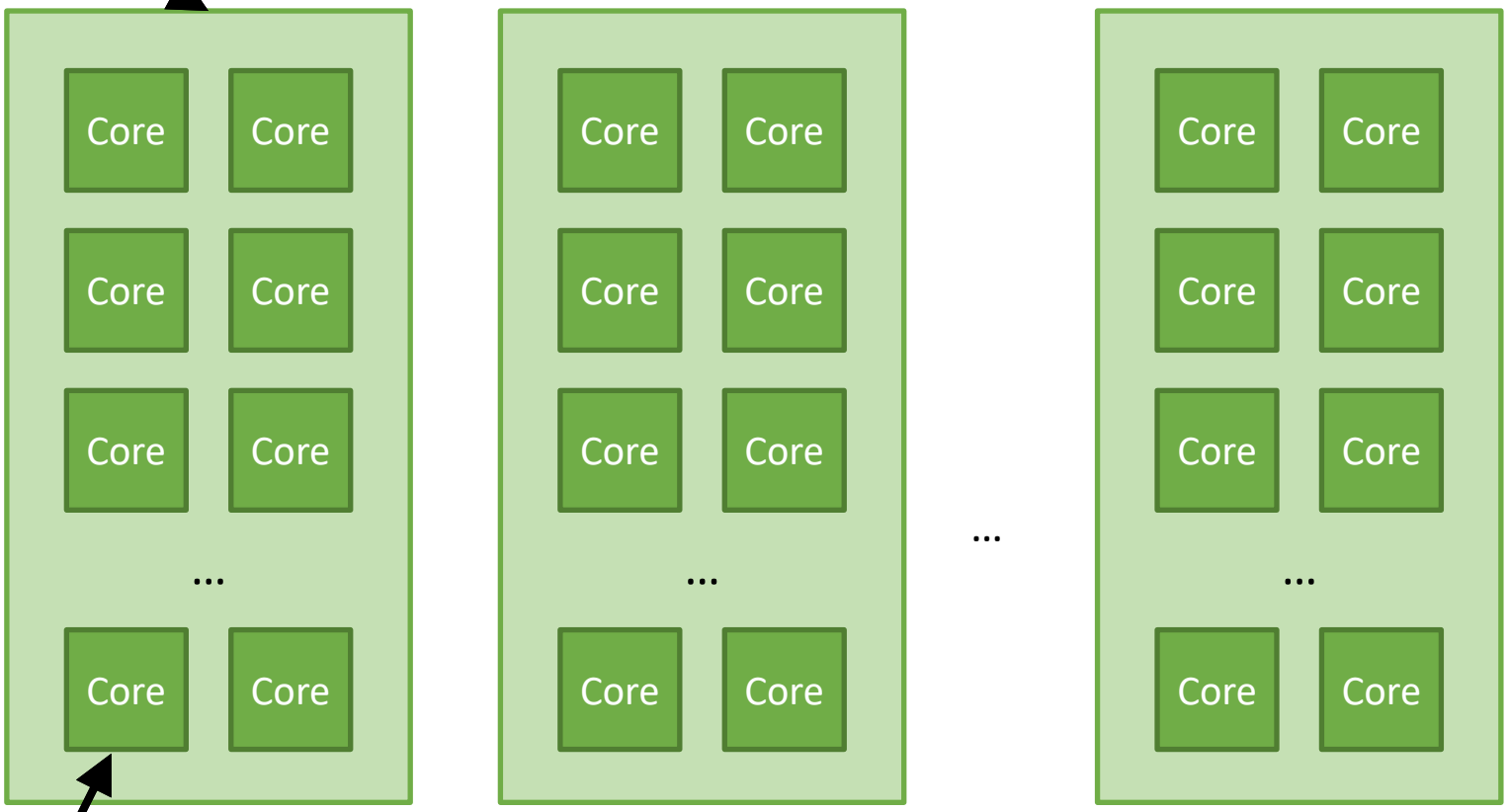


© nvidia.com



GPU Aufbau

SM (Streaming Multiprocessor)

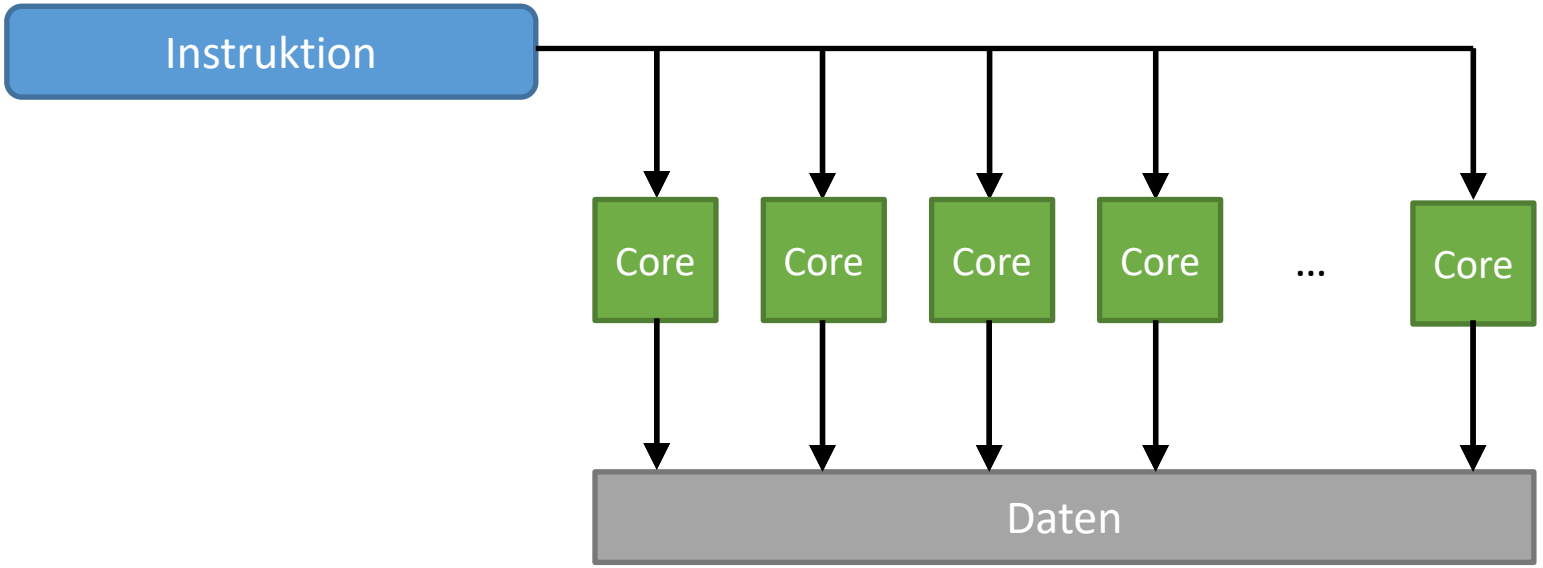


SP (Streaming Processor)

z.B. 1-30 SM
8-192 SPs pro SM

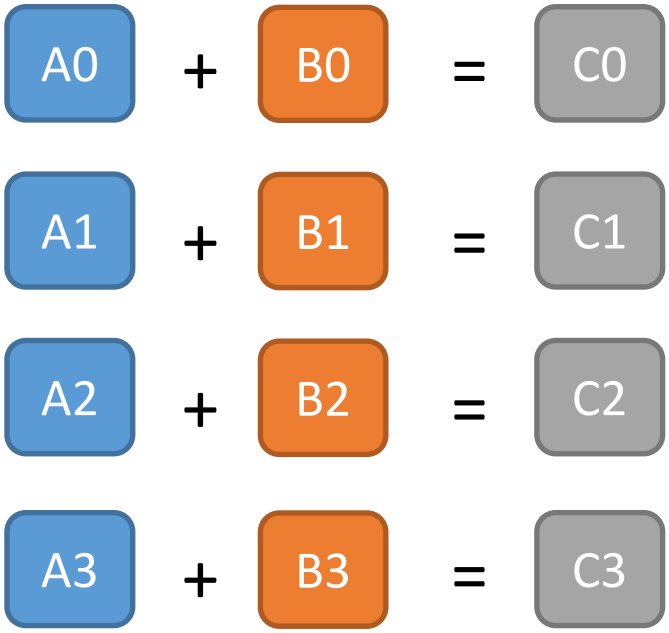
SIMD

- Streaming Multiprocessor ist prinzipiell SIMD (Single Instruction Multiple Data)
  - Cores führen dieselbe Instruktion auf unterschiedlichen Daten/Speicherstellen aus

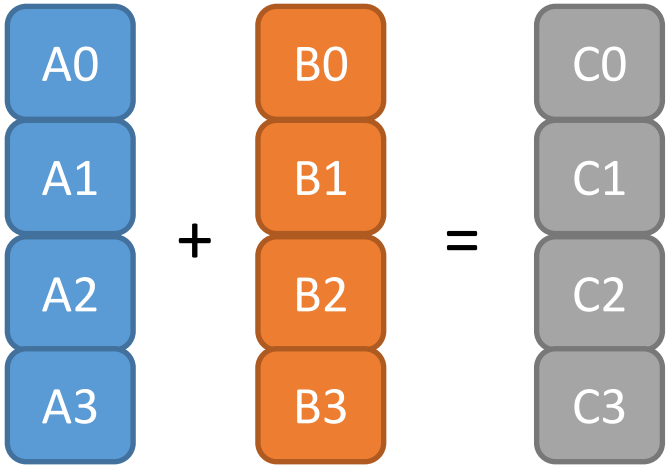


SIMD = Vektorparallelität

Scalar Operation (SISD)



Vektor Operation (SIMD)



Spare Instruction Fetch & Decode pro Datenoperation

GPU Parallelisierungspotential

- Cores innerhalb SM nur mit Vektor-Parallelisierung effizient nutzbar
  - Alle Cores führen gleiche Instruktion aus
  - Einzelne können auch die Instruktion nicht ausführen

Welche Programme können damit effizient parallelisiert werden? Bzw. welche nicht?

GPUs vs. CPUs

- GPU: Video Gaming
  - Extrem hohe Datenparallelität
  - Wenig Verzweigungen
  - Kein beliebiges Warten bei Parallelität
  - Einfache, aber viele Cores
  - Kleine Caches pro Core

Ziel: Hoher Gesamtdurchsatz

- CPU: General Purpose
  - Niedrige Datenparallelität
  - Viel Verzweigungen
  - Beliebige Thread-Synchronisation
  - Wenige, aber mächtige Cores
  - Grössere Caches in Chip

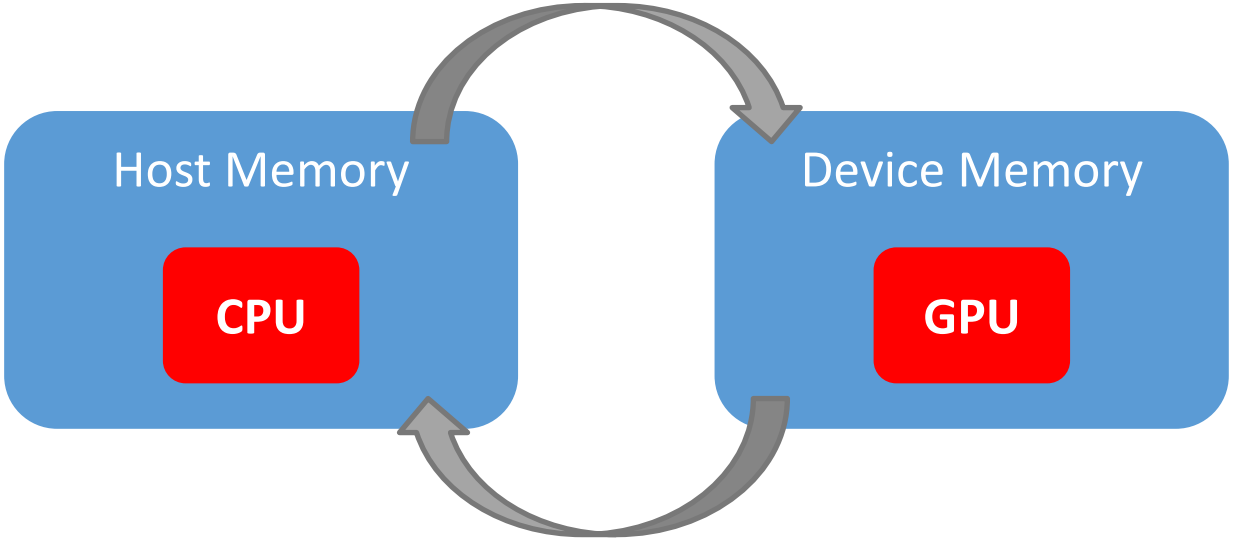
Ziel: Niedrige Latenz pro Thread

Vergleich CPU/GPU

	CPU Intel Core i9 11900K Ice Lake (Q1 2021)	GPU Nvidia Titan RTX 3060 Ampere (Q1 2021)
Cores	8	28 SM mit 64 Cores mit je 2 ALUs = 3584 Cores
Taktrate	5.3 GHz	1.78 GHz
Logische Cores / logische Threads	16	28 SM mit je 1536 Resident Threads = 43’008
Memory Bandbreite	50 GB/s	448 GB/s
L1 Cache Grösse	48 KB/Core	128 KB/SM = 3.5 MB
L2 Cache Grösse	512 KB/Core	4 MB
L3 Cache Grösse	16 MB zusammen	-

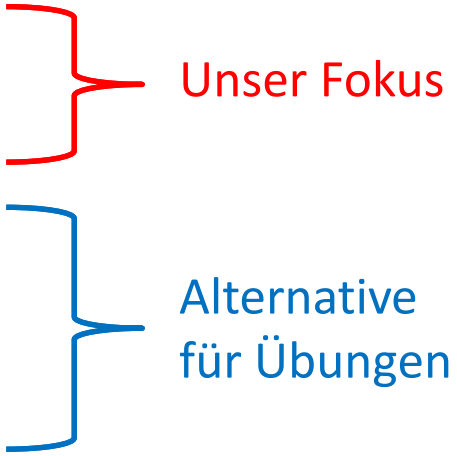
NUMA Modell

- NUMA: Non-Uniform Memory Access
  - Kein gemeinsamer Hauptspeicher zwischen GPU und CPU
    - Explizites Übertragen zwischen CPU und GPU
  - Unterschiedlicher Instruktionssatz/Architektur
    - Code für GPU kompilieren/designen



GPU Parallel-Programmiermodelle

- CUDA: für Nvidia Karten
  - API/Compiler für C/C++
- OpenCL: Unabhängiger Standard
  - HW-unabhängig
  - Ähnlich zu CUDA, weniger Komfort
- C++ AMP: Microsoft Technologie
  - Konzeptionell einfacher (Parallele Loops)
- OpenACC: Offener Standard
  - Analog zu C++ AMP (Pragma Annotationen)



CUDA

- Computer Unified Device Architecture
  - Proprietär für Nvidia Grafikkarten
  - Unterschiedliche Fähigkeiten (Compute Capabilities)
  - Aktuell Version 11.3 (Apr. 2021)
- General Purpose Programming Model
  - Parallelisierung mit sehr vielen (vektorparallelen) Threads
  - Video-Computing und CUDA teilen sich GPUs
  - API und Compiler für Programmiersprache C/C++
- Unterlagen primär für CUDA

OpenCL

- Hardware-unabhängig
  - Z.B. für AMD, ATI, Nvidia, Intel etc.
  - Von Khronos Group
  - Aktuell Version 3.0 (Sep. 2020)
- Ähnliches Model zu CUDA
  - Konzeptionelle Unterschiede
- Informationen siehe Anhang
  - Alternative für Übungen

Beispiel: Vektoraddition

Sequentiell

```
void VectorAdd(float *A, float *B, float *C, int N)  {
  for (int i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
  }
}
```

Parallelisieren: N Threads

Pro Thread i (i = 0 .. N-1):

```
C[i] = A[i] + B[i];
```

CUDA Kernel

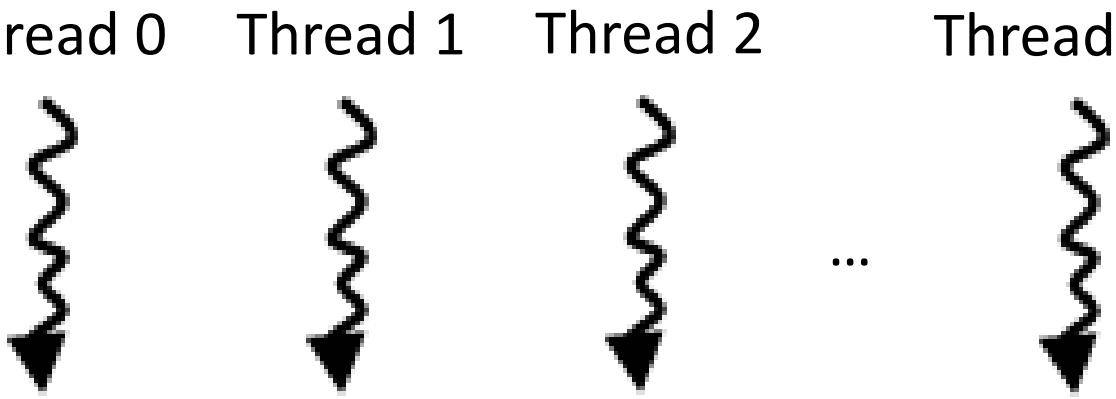
```
// kernel definition
__global__
void VectorAddKernel(float *A, float *B, float *C) {
  int i = threadIdx.x;
  C[i] = A[i] + B[i];
}

int main() {
  ...
  // kernel invocation
  VectorAddKernel<<<1, N>>>(A, B, C);
  ...
}
```

GPU (Device)

CPU (Host)

CUDA Threads

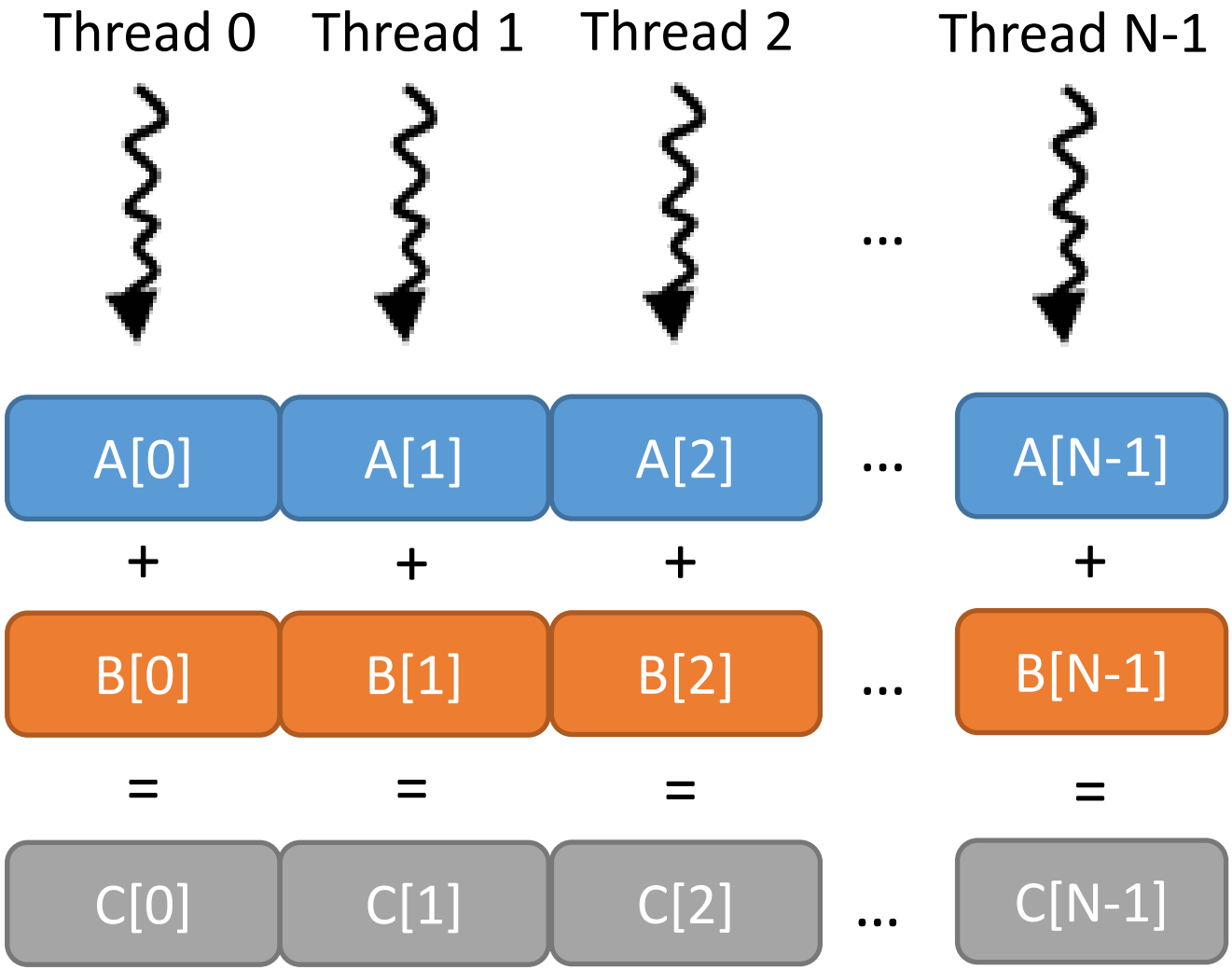
Gleicher Kernel wird von mehreren Threads ausgeführt




```
__global__
void VectorAddKernel(float *A, float *B, float *C)  {
  int i = threadIdx.x;
  C[i] = A[i] + B[i];
}
```

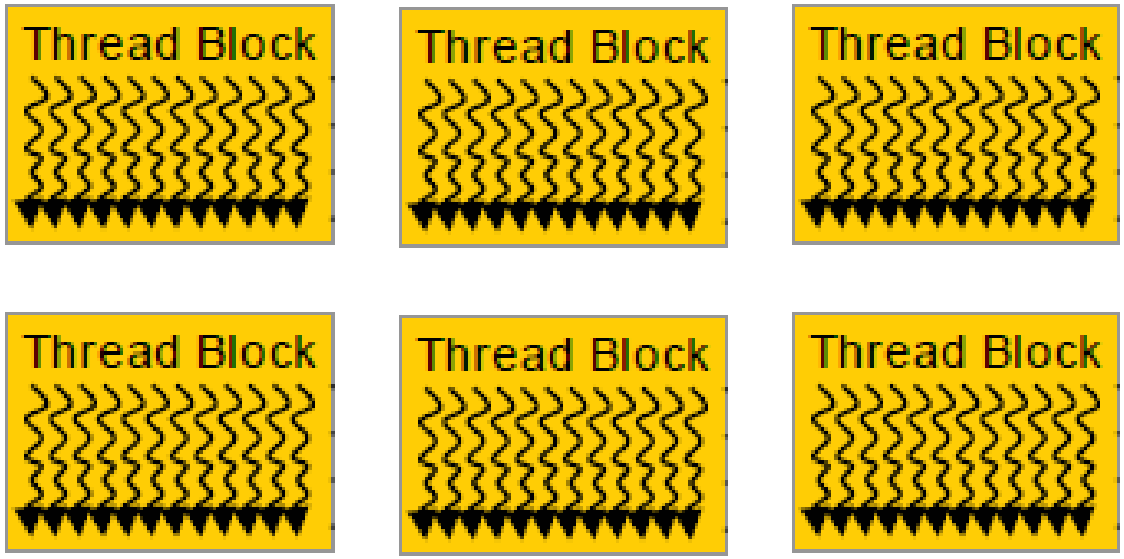
Thread Id (0 .. N-1)

SIMT: Single Instruction Multiple Threads



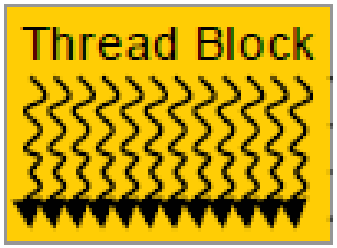
CUDA Blocks

- Threads sind in Blöcke gruppiert
  - Ein Block => gleicher Streaming Multiprozessor
  - Wird im Programmiermodell sichtbar
  - Threads können innerhalb Block interagieren



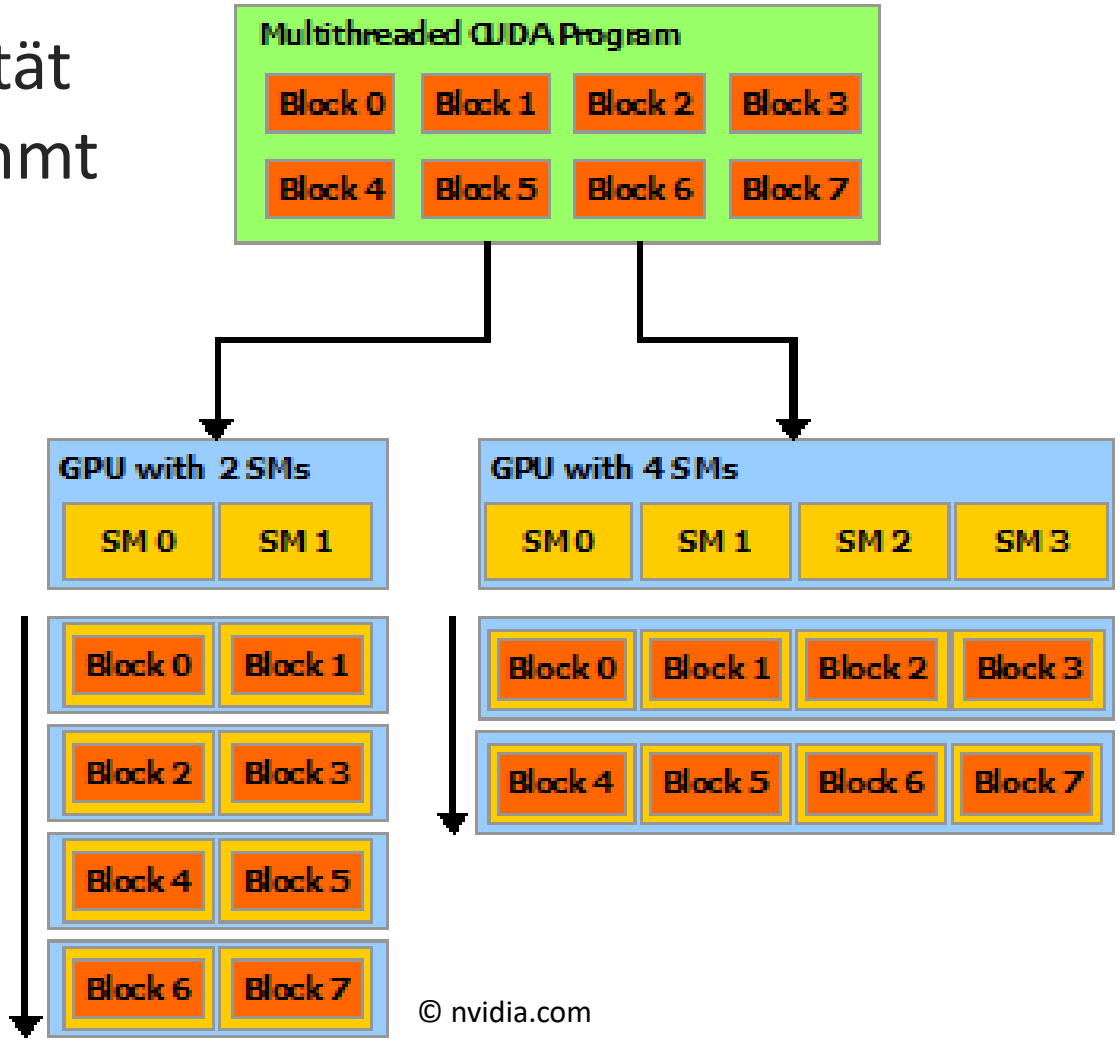
CUDA Ausführungsmodell

- Thread = virtueller Skalarprozessor
- Block = virtueller Multiprozessor
- Blöcke müssen unabhängig sein
  - Run To Completion
  - Beliebige Ausführungsreihenfolge
    - Sequentiell oder parallel
  - Blocks in Thread Pool (Thread Execution Manager)
- Ab Cuda 9: Cooperative Groups
  - Synchronisation über mehrere Blöcke möglich
  - Nur mit neuen Prozessoren möglich, mit Limiten



CUDA Thread Pool Abstraktion

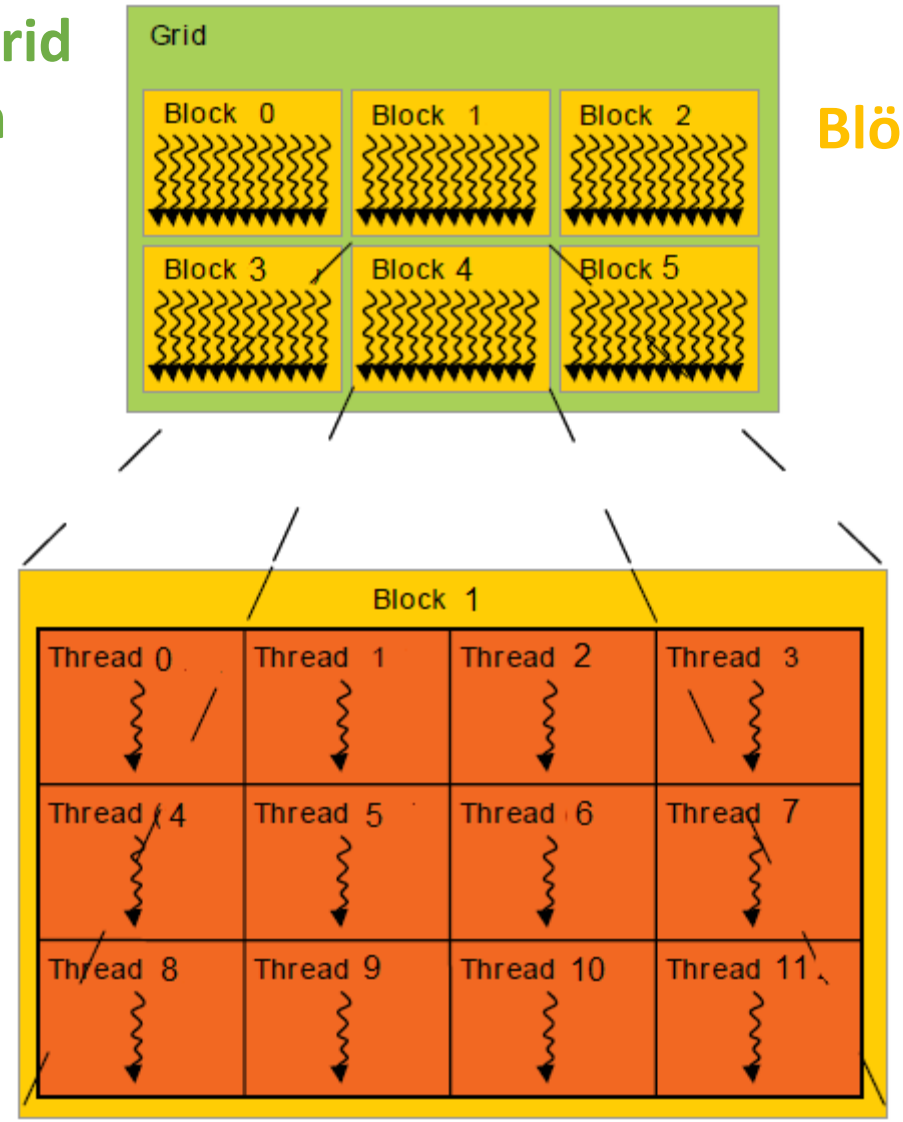
- Grad der Parallelität durch GPU bestimmt
- Automatische Skalierung



© nvidia.com

Thread Hierarchie

Kernel auf Grid ausführen



Blöcke mit Id

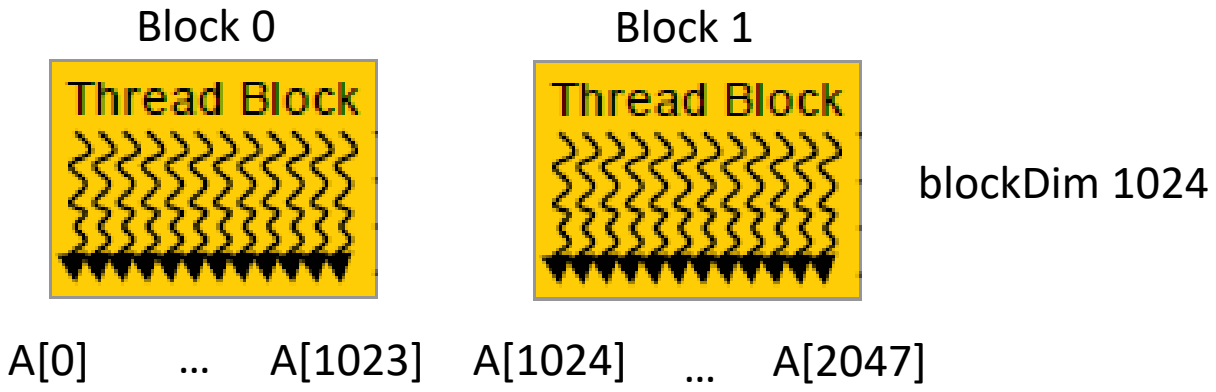
Thread mit Id

© nvidia.com, angepasst

Datenaufteilung

- Jede Kernel-Ausführung bestimmt seinen Datenteil
  - threadIdx.x: Nummer des Threads innerhalb Block
  - blockIdx.x: Nummer des Blocks
  - blockDim.x: Blockgrösse
  - Weitere Dimensionen y, z nutzbar

Programmierende modellieren Datenaufteilung selber



Aufteilung in Blöcke

```
__global__
void VectorAddKernel(float *A, float *B, float *C, int N) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  C[i] = A[i] + B[i];
}
```

Eindeutiger Index basierend auf (Block ID, Thread ID)

```
// kernel invocation
VectorAddKernel<<<4, 1024>>>(A, B, C, 4096);
```

4 Blöcke zu je 1024 Threads

4 * 1024 Threads = Elemente

Allgemeinere Aufteilung

```
__global__
void VectorAddKernel(float *A, float *B, float *C, int N) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < N) {
    C[i] = A[i] + B[i];
  }
}
```

N / blockSize aufgerundet

```
int blockSize = 1024;
int gridSize = (N + blockSize – 1) / blockSize;
VectorAddKernel<<<gridSize, blockSize>>>(A, B, C, N);
```

Wieso braucht es die if-Bedingung im Kernel?

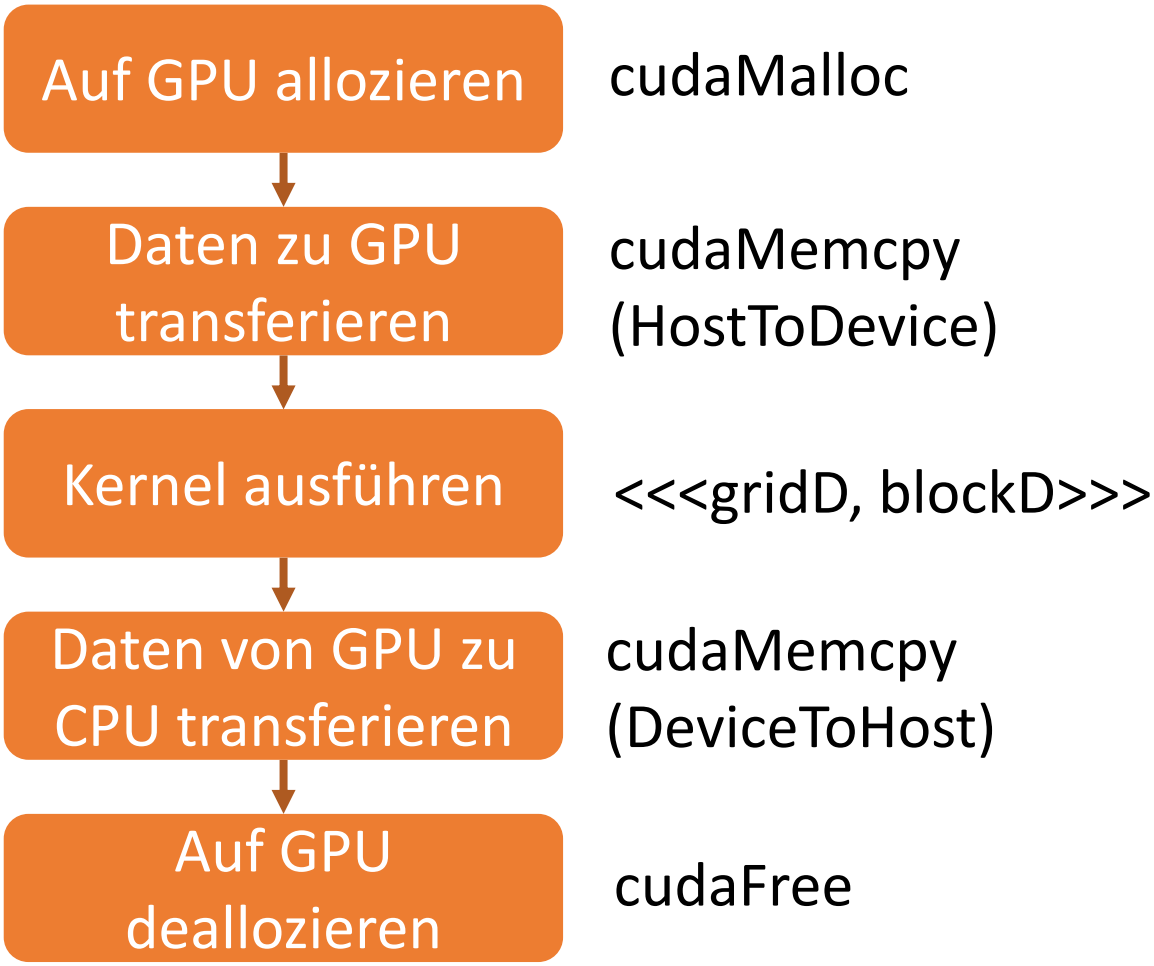
Boundary Check

- Mehr Threads als zu bearbeitende Daten
  - Beispielsweise N = 4000
  - 4 Blöcke mit 1024 Threads => 96 Threads sind unnütz
  - Threads mit i >= N dürfen Array nicht zugreifen

```
__global__
void VectorAddKernel(float *A, float *B, float *C, int N) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < N) {
    C[i] = A[i] + B[i];
  }
}
```

Überflüssige Threads machen nichts

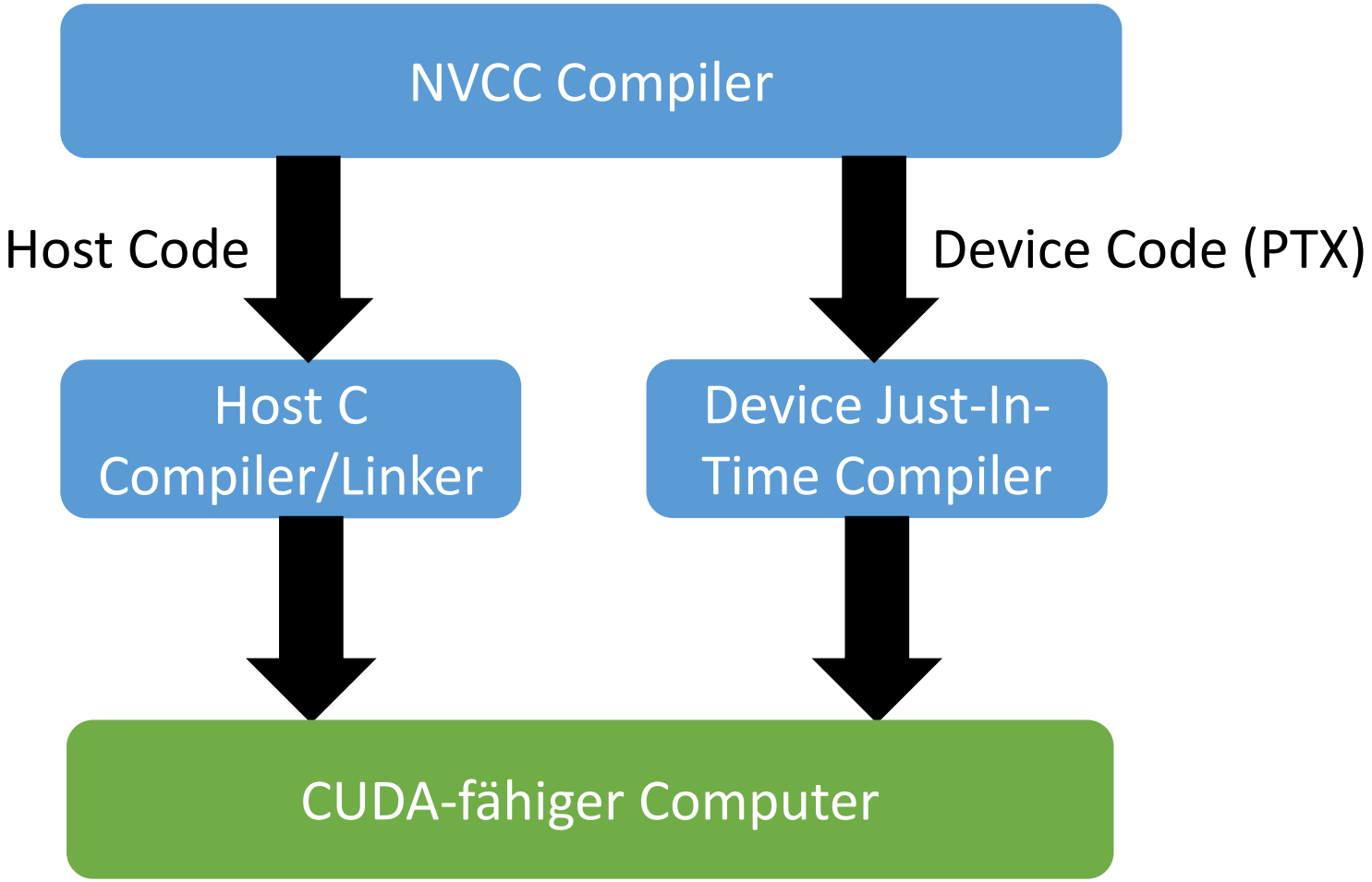
CUDA Ausführung



Programmgerüst

```
void CudaVectorAdd(float* A, float* B, float* C, int N) {
  size_t size = N * sizeof(float);
  float *d_A, *d_B, *d_C;

  cudaMalloc(&d_A, size);
  cudaMalloc(&d_B, size);
  cudaMalloc(&d_C, size);

  cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

  int blockSize = 1024;
  int gridSize = (N + blockSize - 1) / blockSize;
  VectorAddKernel<<<gridSize, blockSize>>>(d_A, d_B, d_C, N);

  cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

  cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

CUDA Compilation



Quelle: B. Kirk, W. Hwu. Programming Massively Parallel Processors, MK 2013.

CUDA Speicher-Funktionen

- `cudaMalloc`
  - Alloziert Objekt in Device Global Memory
  - Parameter: Pointer-Adresse, Size (Bytes)
- `cudaFree`
  - Dealloziert Objekt in Device Global Memory
  - Parameter: Pointer-Adresse
- `cudaMemcpy`
  - Kopiert Speicher zwischen CPU/GPU
  - Target Pointer, Source Pointer, Size, Destination (zu oder von GPU)

Fehlerbehandlung

- Return Codes von CUDA-Funktionen beachten
  - Fehler, falls Code != cudaSuccess

```
cudaError error;

error = cudaMalloc(&d_A, size);
if (error != cudaSuccess) {
  fprintf(stderr, "Failed to allocate device vector A: %s\n",
    cudaGetErrorString(error));
  exit(EXIT_FAILURE);
}
```

Hilfsfunktion für Fehlerbehandlung erstellen

Verbessertes CUDA Gerüst

```
handleCudaError(cudaMalloc(&d_A, size));
handleCudaError(cudaMalloc(&d_B, size));
handleCudaError(cudaMalloc(&d_C, size));

handleCudaError(cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice));
handleCudaError(cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice));

int blockSize = 1024, gridSize = (N + blockSize - 1) / blockSize;
VectorAddKernel<<<gridSize, blockSize>>>(d_A, d_B, d_C, N);
handleCudaError(cudaGetLastError());

handleCudaError(cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost));

handleCudaError(cudaFree(d_A));
handleCudaError(cudaFree(d_B));
handleCudaError(cudaFree(d_C));
```

```
void handleCudaError(cudaError error) {
  if (error != cudaSuccess) {
    fprintf(stderr, "CUDA: %s!\n",
      cudaGetErrorString(error));
    exit(EXIT_FAILURE);
  }
}
```

Unified Memory

- Automatischer Transfer CPU ⇔ GPU
  - Keine expliziten Memory Copies mehr
- Dafür andere Regeln

Unified Memory: Beispiel

CPU Code

```
A = (float*)malloc(size);
B = (float*)malloc(size);
C = (float*)malloc(size);

vectorAdd(A, B, C, N);


free(A);
free(B);
free(C);
```

GPU Code

```
cudaMallocManaged(&A, size);
cudaMallocManaged(&B, size);
cudaMallocManaged(&C, size);

VectorAddKernel<<<..,..>>>(A, B, C, N);
cudaDeviceSynchronize();

cudaFree(A);
cudaFree(B);
cudaFree(C);
```

Warten auf Ende aller GPU Befehle

handleCudaError auch noch benutzen

Device Query

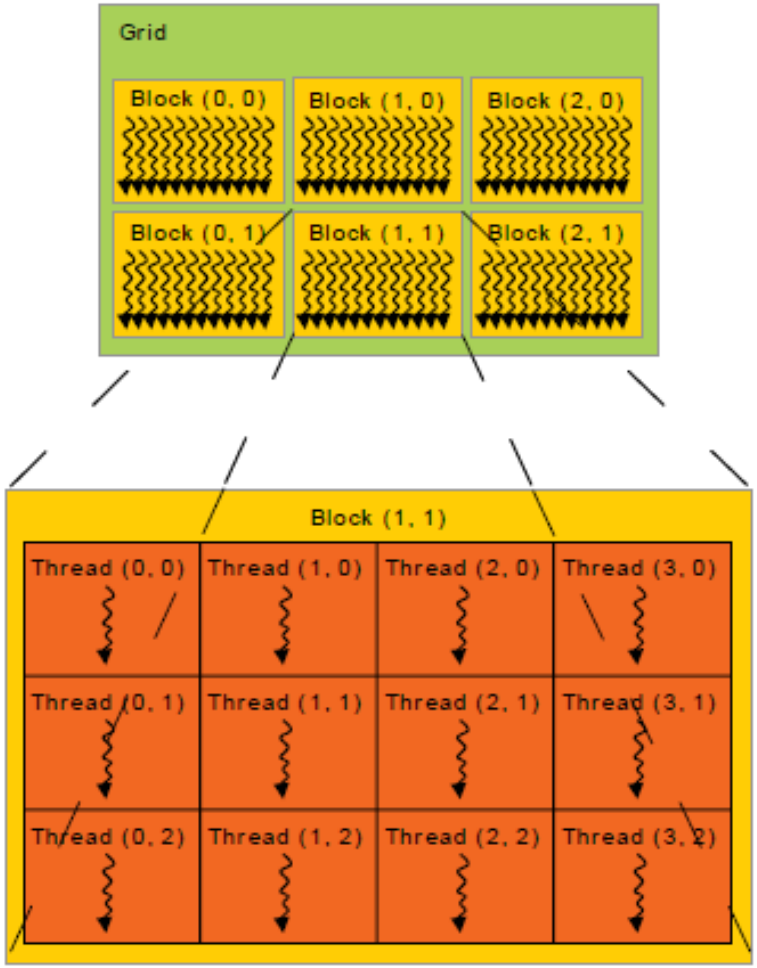
- GPU Fähigkeiten und Limiten abfragen
  - `cudaGetDeviceProperties()`

	Compute Capability												
Technical Specifications	**3.5**	**3.7**	**5.0**	**5.2**	**5.3**	**6.0**	**6.1**	**6.2**	**7.0**	**7.2**	**7.5**	**8.0**	**8.6**
Maximum number of resident grids per device (Concurrent Kernel Execution)	32				16	128	32	16	128	16	128		
Maximum dimensionality of grid of thread blocks	3												
Maximum x-dimension of a grid of thread blocks	$2^{31}-1$												
Maximum y- or z-dimension of a grid of thread blocks	65535												
Maximum dimensionality of a thread block	3												
Maximum x- or y-dimension of a block	1024												
Maximum z-dimension of a block	64												
Maximum number of threads per block	1024												
Warp size	32												
Maximum number of resident blocks per SM	16		32								16	32	16
Maximum number of resident warps per SM	64										32	64	48
Maximum number of resident threads per SM	2048										1024	2048	1536

Quelle: https://docs.nvidia.com/cuda/cuda-c-programming-guide

3D Thread Hierarchie

- (x, y, z) für `threadIdx` und `blockIdx`
  - 1D falls y, z ungenutzt
  - 2D falls z ungenutzt
- Modellierhilfe: 2D Matrizen, 3D Würfel



© nvidia.com

```
dim3 gridSize(3, 2, 1);
dim3 blockSize(4, 3, 1);
VectorAddKernel<<<gridSize, blockSize>>>(...);
```

C Limitation

- Mehrdimensionales Array nicht direkt unterstützt
  - Jagged Array float** keine Hilfe: Array to Pointer of Array

Gewünscht, aber nicht unterstützt

```
float[,] matrix = new float[NofRows, NofCols];
matrix[row, col] = …
```

In C manuell Low-Level linearisieren

```
float *matrix =
   (float *)malloc(NofRows * NofCols * sizeof(float));
matrix[row * NofCols + col] = …
```

Ende der Vorlesung

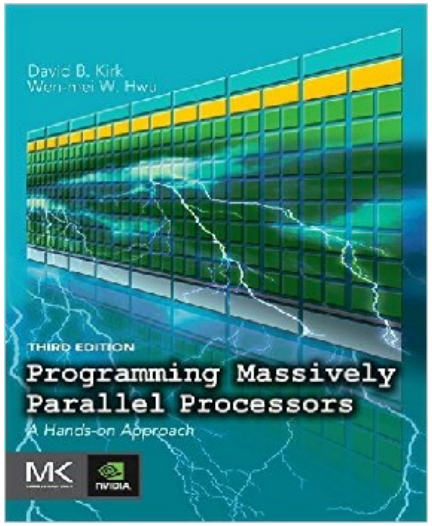
- Nächste Vorlesung: CUDA-Fortsetzung
  - Block-Konfiguration
  - Speichermodell
  - Synchronisation
  - Warps und Divergenz
  - Coalescing
- Anhang: OpenCL Unterschied

Rückblick: Lernziele

- Die GPU-Prozessorarchitektur verstehen
- CUDA (oder OpenCL) als Programmierframework für GPU-Parallelisierung anwenden können
- Erste Tipps und Tricks für GPU-Parallelisierung kennenlernen

Literatur zum Nachschlagen

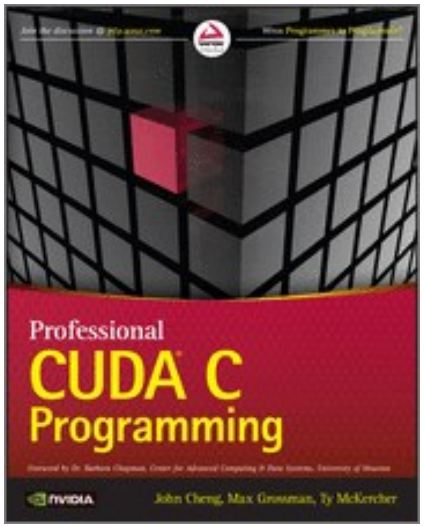
- D. B. Kirk and W.-m. W. Hwu. Programming Massively Parallel Processors, 3rd Edition, Morgan Kaufmann, 2016.
  - CUDA und OpenCL

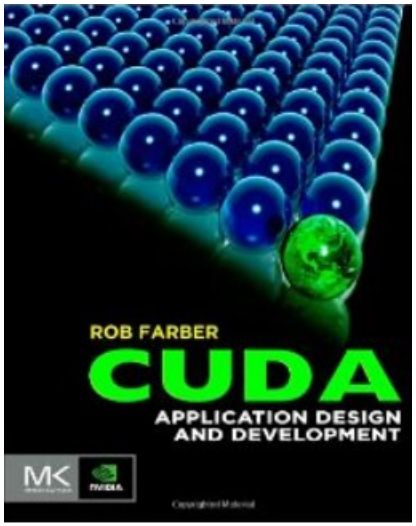


- Nvidia CUDA C Programming Guide
  - https://docs.nvidia.com/cuda/cuda-c-programming-guide

Weitere Literatur (nach Bedarf)

- John Cheng, Max Grossman, Ty McKercher. Professional CUDA C Programming, Wrox, 2014.
- R. Farber. CUDA: Application Design and Development, Morgan Kaufmann, 2011.





Anhang: OpenCL

OpenCL Hauptunterschiede

- Herstellerunabhängig (nicht nur Nvidia)
- Explizite Kompilation (Kernel-Code als Source)
- Aufwändige Initialisierung / Device-Auswahl
- Keine Block/Thread-Unterteilung im Kernel
  - Globale Id
  - Kernel Launch mit Work Size (Blöcke)

OpenCL Kernel

```
__kernel
void VectorAddKernel(__global float *A,
                     __global float *B,
                     __global float *C,
                     int N) {
  int i = get_global_id(0);
  if (i < N) {
    C[i] = A[i] + B[i];
  }
}
```

Transparente Aufteilung in Workgroups (Blöcke)

In separates Kernel File speichern

OpenCL Initialisierung

```
cl_int error;
cl_platform_id platform;
handleClError(
  clGetPlatformIDs(1, &platform, NULL));

cl_device_id device;
handleClError(
  clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL));


cl_context context =
  clCreateContext(0, 1, &device, NULL, NULL, &error);
handleClError(error);

cl_command_queue commandQueue =
  clCreateCommandQueue(context, device, 0, &error);
handleClError(error);
```

OpenCL Driver auswählen

Erste GPU auswählen

OpenCL Umgebung

Für GPU-Befehle

OpenCL Kompilation

```
char *kernelSource = NULL;
size_t kernelLength = 0;
  readSourceFromFile(KERNEL_FILE, &kernelSource, &kernelLength);

cl_program program =
  clCreateProgramWithSource(context, 1,
      (const char **)&kernelSource, &kernelLength, &error);
handleClError(error);
handleClError(
  clBuildProgram(program, 0, NULL, NULL, NULL, NULL));
cl_kernel kernel =
  clCreateKernel(program, KERNEL_FUNCTION, &error);
handleClError(error);
```

Kernel-File laden

Kernel kompilieren

OpenCL Speicher allozieren

```
cl_mem d_A =
  clCreateBuffer(context, CL_MEM_READ_ONLY, size, NULL, &error);
handleClError(error);

cl_mem d_B =
  clCreateBuffer(context, CL_MEM_READ_ONLY, size, NULL, &error);
handleClError(error);

cl_mem d_C =
  clCreateBuffer(context, CL_MEM_WRITE_ONLY, size, NULL, &error);
handleClError(error);
```

A, B und C in Device Memory allozieren

OpenCL Kernel-Argumente setzen

```
handleClError(
  clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_A));

handleClError(
  clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_B));

handleClError(
  clSetKernelArg(kernel, 2, sizeof(cl_mem), &d_C));

handleClError(
  clSetKernelArg(kernel, 3, sizeof(int), &N));
```

Kernel-Parameter

```
(/* 0 */ __global float *A,
 /* 1 */ __global const float *B,
 /* 2 */ __global float *C,
 /* 3 */ int N)
```

Open CL Input kopieren

```
handleClError(
  clEnqueueWriteBuffer(commandQueue, d_A, CL_FALSE, 0, size,
                       A, 0, NULL, NULL));

handleClError(
  clEnqueueWriteBuffer(commandQueue, d_B, CL_FALSE, 0, size,
                       B, 0, NULL, NULL));
```

A und B von Host zu Device kopieren

Open CL Kernel Launch

Entspricht Blockgrösse

```
size_t localWorkSize[] = { 1024 };
size_t globalWorkSize[] = { (N + 1023) / 1024 * 1024 };
```

Totale Anzahl Threads (1024 Vielfaches)

1-dimensional (nur x in CUDA)

```
handleClError(
  clEnqueueNDRangeKernel(commandQueue, kernel, 1, NULL,
     globalWorkSize, localWorkSize, 0, NULL, NULL));
```

CUDA-Analogon: `VectorAddKernel<<<(N + 1023) / 1024, 1024>>>(…)`

Open CL Output kopieren

```
handleClError(
  clEnqueueReadBuffer(commandQueue, d_C, CL_TRUE, 0, size,
                      C, 0, NULL, NULL));
```

C von Device zu Host zurückkopieren

OpenCL freigeben

```
handleClError(clReleaseMemObject(d_A));
handleClError(clReleaseMemObject(d_B));
handleClError(clReleaseMemObject(d_C));

handleClError(clReleaseKernel(kernel));
handleClError(clReleaseProgram(program));

free(kernelSource);
free(kernelPath);

handleClError(clReleaseCommandQueue(commandQueue));
handleClError(clReleaseContext(context));
```

Device Memory deallozieren

Kernel Code deallozieren

OpenCL beenden

Begriffe OpenCL/CUDA

OpenCL	CUDA
Kernel	Kernel
Host Programm	Host Programm
NDRange	Grid
Work Item	Thread
Work Group	Block

Unterschiede OpenCL/CUDA

Aspekt	OpenCL	CUDA
Kernel Function	__kernel	__global__
Kernel Parameter	__global	(keine)
Thread-Identifikation	Global Id	Block und Thread
Initialisierung	Langwierig	Komfortabel
Kernel Kompilation	Explizit als Source	Integrierter Compiler
Launch Config	Blockgrösse und totale Anzahl Threads	Blockgrösse und Anzahl Blöcke
Befehle an GPU	Explizite Queue	Implizite Queue

OpenCL

Generelle Information

- https://www.khronos.org/opencl/

Driver von Hersteller

- Intel: https://software.intel.com/en-us/articles/opencl-drivers
- Nvidia: https://developer.nvidia.com/opencl