CUDA
René Fischer, Konrad Gräfe (IF09w1-M)
Hochschule Mittweida (FH)
19. Januar 2011

FAKULTÄT MATHEMATIK/NATURWISSENSCHAFTEN/INFORMATIK

HOCHSCHULE MITTWEIDA
UNIVERSITY OF APPLIED SCIENCES

CUDA

René Fischer, Konrad Gräfe (IF09w1-M)
Hochschule Mittweida (FH)

19. Januar 2011

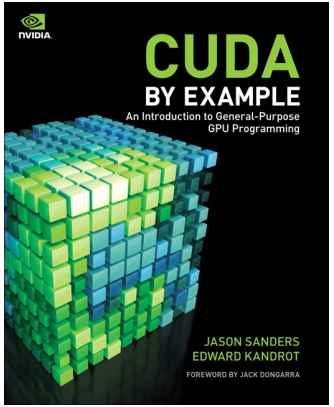
Gliederung
CUDA-Hardware
Einführung CUDA-Entwicklung
Programmier-Modell
Quellen

Gliederung

HOCHSCHULE MITTWEIDA
UNIVERSITY OF APPLIED SCIENCES

Literaturhinweis

Literaturhinweis



CUDA BY EXAMPLE
An Introduction to General-Purpose GPU Programming

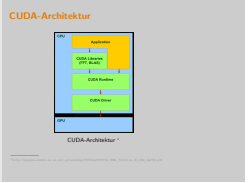
Jason Sanders
Edward Kandrot

ISBN: 0131387685

CUDA
- CUDA: Computer Unified Device Architectur
- Architektur aus Soft- und Hardware
- massiv paralleles Rechnen
- Anwendung im wissenschaftlichen und technischen Bereich (z.B. seti@home)
- Programmiersprachen: C, C++, Java, Python, Matlab

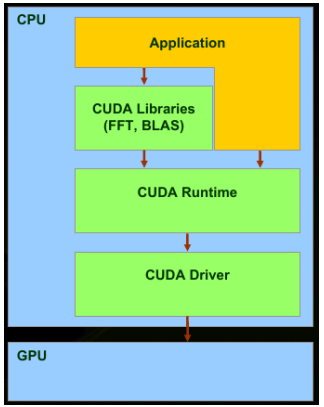
CUDA

- CUDA: Computer Unified Device Architectur
- Architektur aus Soft- und Hardware
- massiv paralleles Rechnen
- Anwendung im wissenschaftlichen und technischen Bereich (z.B. seti@home)
- Programmiersprachen: C, C++, Java, Python, Matlab



1. CUDA: Computer Unified Device Architectur
2. Architektur aus Soft- und Hardware

CUDA-Architektur



CUDA-Architektur *

*http://people.maths.ox.ac.uk/~gilesm/hpc/NVIDIA/NVIDIA_CUDA_Tutorial_No_NDA_Apr08.pdf

CUDA-Hardware
unterstützte Hardware
Fermi-Architektur

CUDA-Hardware

unterstützte Hardware
Fermi-Architektur

unterstützte Hardware

unterstützte GPUs:
- G 8X
- GT 2XX
- GF 1XX

unterstütze Kartenfamilien:
- GeForce / GeForce Mobile
- Quadro / Quadro Mobile

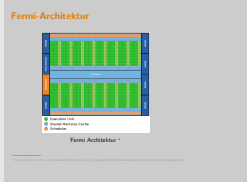
unterstütze Architekturen:
- Tesla
- Fermi

unterstützte Hardware

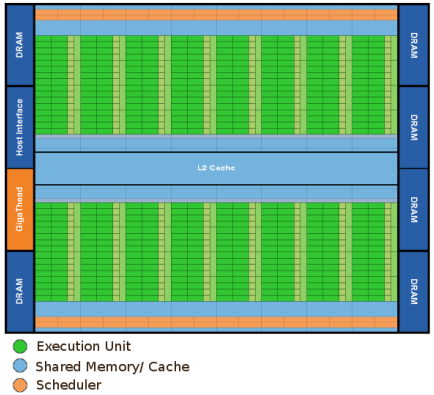
unterstützte GPUs:
- G 8X
- GT 2XX
- GF 1XX

unterstütze Kartenfamilien:
- GeForce / GeForce Mobile
- Quadro / Quadro Mobile

unterstütze Architekturen:
- Tesla
- Fermi

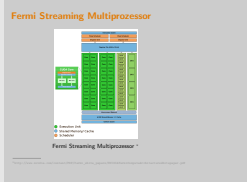


Fermi-Architektur

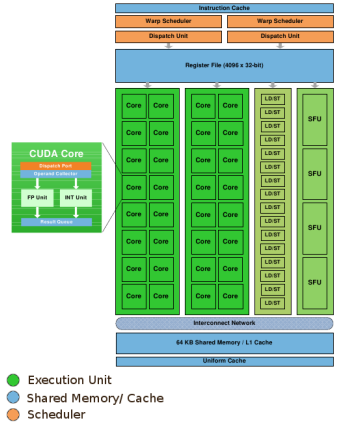


Fermi Architektur *

*http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf



Fermi Streaming Multiprozessor



Fermi Streaming Multiprozessor *

*http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf

Einführung CUDA-Entwicklung
Vorraussetzungen
CUDA C
Entwicklungswerkzeuge

Einführung CUDA-Entwicklung

- Vorraussetzungen
- CUDA C
- Entwicklungswerkzeuge

Vorraussetzungen

- CUDA-fähiger Grafikprozessor (www.nvidia.com/cuda)
- NVIDIA Gerätetreiber
- CUDA Entwicklungsumgebung (nvcc, CUDA-Bibliotheken)
- Standard C Compiler
  - Windows: Visual Studio 2005/2008
  - Linux: gcc
  - Mac OS X (ab Version 10.5.7): gcc

Vorraussetzungen

- CUDA-fähiger Grafikprozessor (www.nvidia.com/cuda)
- NVIDIA Gerätetreiber
- CUDA Entwicklungsumgebung (nvcc, CUDA-Bibliotheken)
- Standard C Compiler
  - Windows: Visual Studio 2005/2008
  - Linux: gcc
  - Mac OS X (ab Version 10.5.7): gcc

Begriffe

Host: Rechner der die main-Methode ausführt. (i.d.R. PC)
Device: Grafikprozessor (GPU) und Grafikspeicher (DRAM)
Kernel: Funktion die auf dem Device ausgeführt und vom Host aufgerufen wird. Parallele Ausführung dieser Funktion auf mehreren Prozessoren.
Devicefunktion: Funktion auf dem Device die nicht vom Host aus aufrufbar ist.

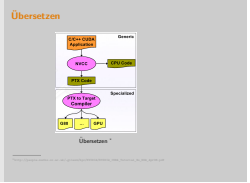
Begriffe

Host: Rechner der die main-Methode ausführt. (i.d.R. PC)

Device: Grafikprozessor (GPU) und Grafikspeicher (DRAM)

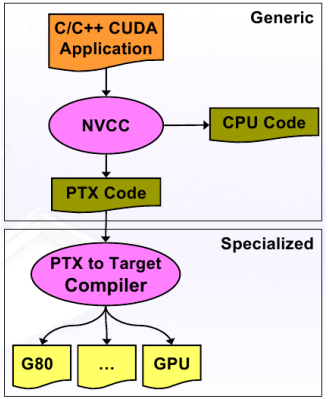
Kernel: Funktion die auf dem Device ausgeführt und vom Host aufgerufen wird. Parallele Ausführung dieser Funktion auf mehreren Prozessoren.

Devicefunktion: Funktion auf dem Device die nicht vom Host aus aufrufbar ist.



ptx: Parallel Thread Execution

Übersetzen



Übersetzen *

*http://people.maths.ox.ac.uk/~gilesm/hpc/NVIDIA/NVIDIA_CUDA_Tutorial_No_NDA_Apr08.pdf

C-Erweiterungen

- Deklarationen für Funktionen
  __global__ void KernelFunc(...); // Kernelfunktion auf Device
  __device__ void Device(...); // Devicefunktion
- Deklarationen für Variablen
  __device__ int GlobalVar; // Variable in Device-Memory
  __shared__ int SharedVar; // Variable in Shared Memory
- Erweiterte Aufrufsyntax für Kernelfunktionen
  KernelFunc<<<500,128>>>(...); // 500 Blöcke a 128 Threads

C-Erweiterungen

- Deklarationen für Funktionen

```
__global__ void KernelFunc(...);    // Kernelfunktion auf Device
__device__ void Device(...);        // Devicefunktion
```

- Deklarationen für Variablen

```
__device__ int GlobalVar;    // Variable in Device-Memory
__shared__ int SharedVar;    // Variable in Shared Memory
```

- Erweiterte Aufrufsyntax für Kernelfunktionen

```
KernelFunc<<<500,128>>>(...); // 500 Blöcke a 128 Threads
```

Minimalbeispiel

```
__global__ void kernel( void ) {
}

int main( void ) {
    kernel<<<1,1>>>();
    printf( "Hello, World!\n" );
    return 0;
}
```

Minimalbeispiel

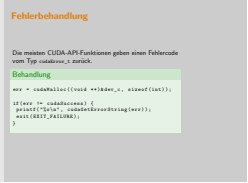
```
__global__ void kernel( void ) {
}

int main( void ) {
    kernel<<<1,1>>>();
    printf( "Hello, World!\n" );
    return 0;
}
```

Einschränkungen für Kernel- und Devicefunktionen
- keine Rekursionen
- keine statischen Variablendeklarationen
- keine variable Argumentenanzahl
- automatische Variablen befinden sich in Registern
- Kernelfunktionen haben keinen Rückgabewert

Einschränkungen für Kernel- und Devicefunktionen

- keine Rekursionen
- keine statischen Variablendeklarationen
- keine variable Argumentenanzahl
- automatische Variablen befinden sich in Registern
- Kernelfunktionen haben keinen Rückgabewert



Fehlerbehandlung

Die meisten CUDA-API-Funktionen geben einen Fehlercode vom Typ `cudaError_t` zurück.

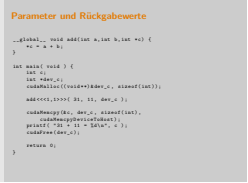
Behandlung

```
err = cudaMalloc((void **)&dev_c, sizeof(int));

if(err != cudaSuccess) {
 printf("%s\n", cudaGetErrorString(err));
 exit(EXIT_FAILURE);
}
```

Grafikspeichernutzung

- Speicher reservieren
  - `cudaError_t cudaMalloc(void **devPtr,size_t size)`
- Speicher kopieren
  - `cudaError_t cudaMemcpy(void *dst,const void *src, size_t count,enum cudaMemcpyKind kind)`
  - enum cudaMemcpyKind
    - cudaMemcpyHostToHost *Host* → *Host*
    - cudaMemcpyHostToDevice *Host* → *Device*
    - cudaMemcpyDeviceToHost *Device* → *Host*
    - cudaMemcpyDeviceToDevice *Device* → *Device*
- Speicher freigeben
  - `cudaError_t cudaFree ( void *devPtr )`



Parameter und Rückgabewerte

```
__global__ void add(int a,int b,int *c) {
    *c = a + b;
}

int main( void ) {
    int c;
    int *dev_c;
    cudaMalloc((void**)&dev_c, sizeof(int));

    add<<<1,1>>>( 31, 11, dev_c );

    cudaMemcpy(&c, dev_c, sizeof(int),
        cudaMemcpyDeviceToHost);
    printf( "31 + 11 = %d\n", c );
    cudaFree(dev_c);

    return 0;
}
```

Debuggen
- CUDA-gdb
- Device Emulate Mode (nvcc -deviceemu)
  - keine Hardware oder Treiber notwendig
  - Device Threads werden auf dem Host emuliert
  - Verwendung von Hostfunktionen (printf) in Kernel-funktionen möglich
  - Zugriff auf Devicespezifische Daten vom Host möglich und vice versa

Debuggen

- CUDA-gdb
- Device Emulate Mode (nvcc -deviceemu)
  - keine Hardware oder Treiber notwendig
  - Device Threads werden auf dem Host emuliert
  - Verwendung von Hostfunktionen (printf) in Kernel-funktionen möglich
  - Zugriff auf Devicespezifische Daten vom Host möglich und vice versa

Debuggen

- CUDA-gdb
- Device Emulate Mode (nvcc -deviceemu)
  - keine Hardware oder Treiber notwendig
  - Device Threads werden auf dem Host emuliert
  - Verwendung von Hostfunktionen (printf) in Kernelfunktionen möglich
  - Zugriff auf Devicespezifische Daten vom Host möglich und vice versa

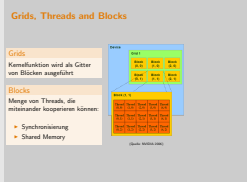
Fallstricke des Device Emulate Mode

- Devicethreads werden sequentiell ausgeführt
- Ergebnisse von Gleitpunktoperationen können verschieden sein
  - verschiedene Compileroutputs
  - verschiedene Befehlssätze
  - andere Präzisionen für Zwischenergebnisse

Programmier-Modell
Grids, Threads and Blocks
Speicherarten
Wichtige Funktionen und Variablen

Programmier-Modell

- Grids, Threads and Blocks
- Speicherarten
- Wichtige Funktionen und Variablen



Grids, Threads and Blocks

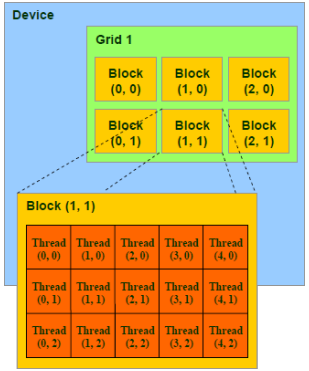
Grids

Kernelfunktion wird als Gitter von Blöcken ausgeführt

Blocks

Menge von Threads, die miteinander kooperieren können:

- Synchronisierung
- Shared Memory

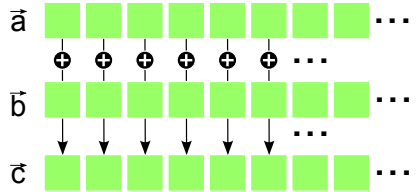


(Quelle: NVIDIA 2006)

Beispiel: Vektoraddition

- $\vec{c} = \vec{a} + \vec{b}$
- Größe jedes Vektors: *WIDTH*
- jeder Block bearbeitet ein Element von $\vec{c}$

Beispiel: Vektoraddition



- $\vec{c} = \vec{a} + \vec{b}$
- Größe jedes Vektors: *WIDTH*
- jeder Block bearbeitet ein Element von $\vec{c}$

Konfiguration der Grids/Blocks

- wird im Kernelaufruf festgelegt:

```
__global__ void vecadd(int* a,int* b,int* c)
    { ... }

int main() {
    ...
    // Kernelaufruf <<<dimGrid, dimBlock>>>
    vecadd<<<WIDTH, 1>>>(a, b, c);
    ...
}
```

- max. 512 Threads pro Block

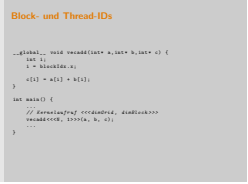
Konfiguration der Grids/Blocks

- wird im Kernelaufruf festgelegt:

```
__global__ void vecadd(int* a,int* b,int* c)
    { ... }

int main() {
    ...
    // Kernelaufruf <<<dimGrid, dimBlock>>>
    vecadd<<<WIDTH, 1>>>(a, b, c);
    ...
}
```

- max. 512 Threads pro Block



Block- und Thread-IDs

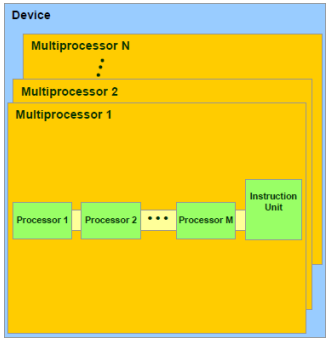
```
__global__ void vecadd(int* a,int* b,int* c) {
    int i;
    i = blockIdx.x;

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

int main() {
    ...
    // Kernelaufruf <<<dimGrid, dimBlock>>>
    vecadd<<<N, 1>>>(a, b, c);
    ...
}
```

Multiprozessoren: SPMD & SIMD
SPMD: Single Program Multiple Data
SIMD: Single Instruction Multiple Data

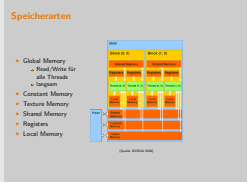
Multiprozessoren: SPMD & SIMD



(Quelle: NVIDIA 2006)

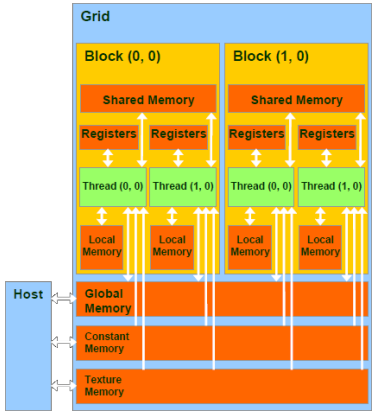
SPMD: Single Program Multiple Data

SIMD: Single Instruction Multiple Data

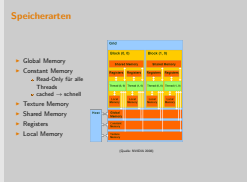


Speicherarten

- Global Memory
  - Read/Write für alle Threads
  - langsam
- Constant Memory
- Texture Memory
- Shared Memory
- Registers
- Local Memory

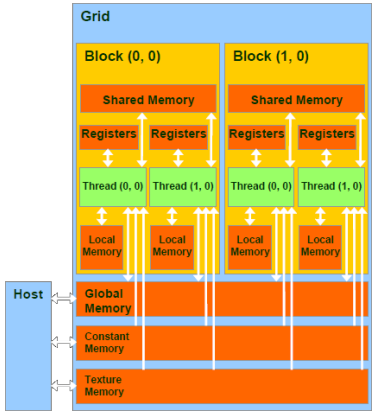


(Quelle: NVIDIA 2008)

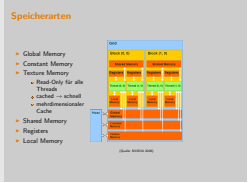


Speicherarten

- Global Memory
- Constant Memory
  - Read-Only für alle Threads
  - cached → schnell
- Texture Memory
- Shared Memory
- Registers
- Local Memory

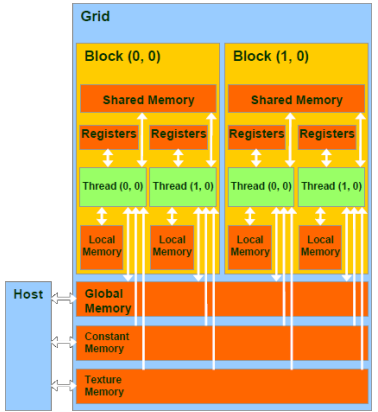


(Quelle: NVIDIA 2008)

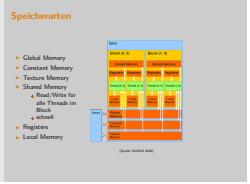


Speicherarten

- Global Memory
- Constant Memory
- Texture Memory
  - Read-Only für alle Threads
  - cached → schnell
  - mehrdimensionaler Cache
- Shared Memory
- Registers
- Local Memory

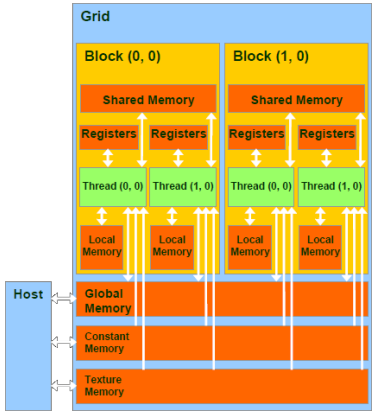


(Quelle: NVIDIA 2008)

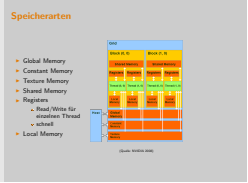


Speicherarten

- Global Memory
- Constant Memory
- Texture Memory
- Shared Memory
  - Read/Write für alle Threads im Block
  - schnell
- Registers
- Local Memory

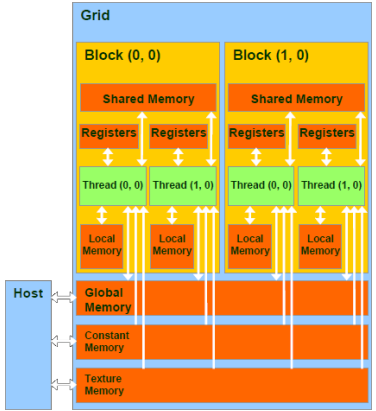


(Quelle: NVIDIA 2008)

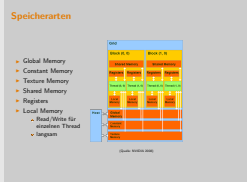


Speicherarten

- Global Memory
- Constant Memory
- Texture Memory
- Shared Memory
- Registers
  - Read/Write für einzelnen Thread
  - schnell
- Local Memory

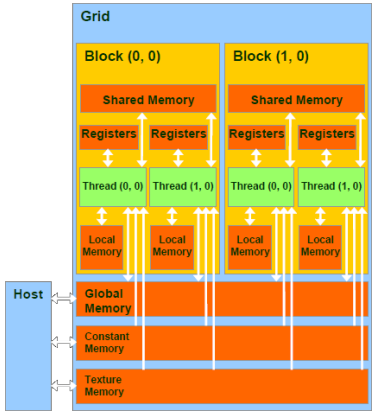


(Quelle: NVIDIA 2008)



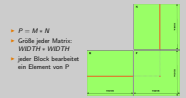
Speicherarten

- Global Memory
- Constant Memory
- Texture Memory
- Shared Memory
- Registers
- Local Memory
  - Read/Write für einzelnen Thread
  - langsam



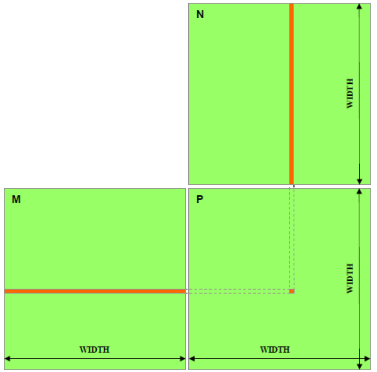
(Quelle: NVIDIA 2008)

Beispiel: Matrizenmultiplikation



Beispiel: Matrizenmultiplikation

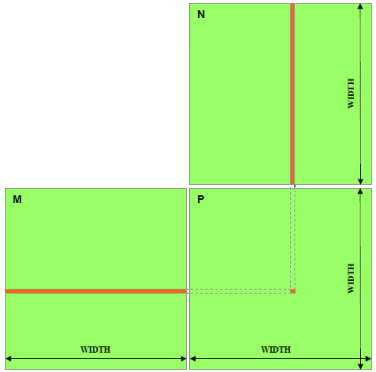
- $P = M * N$
- Größe jeder Matrix: $WIDTH * WIDTH$
- jeder Block bearbeitet ein Element von P



Beispiel: Matrizenmultiplikation
Problem:
Viele (langsame) Zugriffe auf dieselben Speicherbereiche!

Beispiel: Matrizenmultiplikation

- $P = M * N$
- Größe jeder Matrix: $WIDTH * WIDTH$
- jeder Block bearbeitet ein Element von P

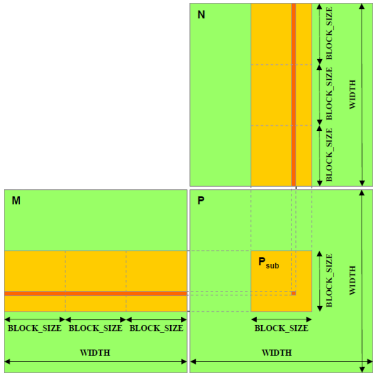


Problem:

Viele (langsame) Zugriffe auf dieselben Speicherbereiche!

Beispiel: Matrizenmultiplikation
Besser:
Aufteilen der Matrix unter Ausnutzung des (schnellen) Shared Memory!

Beispiel: Matrizenmultiplikation



Besser:

Aufteilen der Matrix unter Ausnutzung des (schnellen) Shared Memory!

Wichtige Funktionen und Variablen

- blockIdx.x und blockIdx.y identifizieren Block
- threadIdx.x und threadIdx.y identifizieren Thread innerhalb seines Blocks
- void __syncthreads(void) synchronisiert alle Threads eines Blocks

Wichtige Funktionen und Variablen

- `blockIdx.x` und `blockIdx.y` identifizieren Block
- `threadIdx.x` und `threadIdx.y` identifizieren Thread innerhalb seines Blocks
- `void __syncthreads(void)` synchronisiert alle Threads eines Blocks

Quellen

- **NVIDIA CUDA Library Documentation** http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/online/index.html
- **CUDA: A New Architecture for Computing on the GPU** von NVIDIA (2006) http://people.maths.ox.ac.uk/gilesm/hpc/NVIDIA/Cuda2.pdf
- **NVIDIA's Next Generation CUDA Compute Architecture: Fermi** von NVIDIA (2006) http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf
- **Tutorial CUDA** von Cyril Zeller (2008) http://people.maths.ox.ac.uk/gilesm/hpc/NVIDIA/NVIDIA_CUDA_Tutorial_No_NDA_Apr08.pdf

Quellen

- **NVIDIA CUDA Library Documentation** http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/online/index.html
- **CUDA: A New Architecture for Computing on the GPU** von NVIDIA (2006) http://people.maths.ox.ac.uk/~gilesm/hpc/NVIDIA/Cuda2.pdf
- **NVIDIA's Next Generation CUDA Compute Architecture: Fermi** von NVIDIA (2006) http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf
- **Tutorial CUDA** von Cyril Zeller (2008) http://people.maths.ox.ac.uk/~gilesm/hpc/NVIDIA/NVIDIA_CUDA_Tutorial_No_NDA_Apr08.pdf

Vielen Dank für die Aufmerksamkeit.

Fragen?

Vielen Dank für die Aufmerksamkeit.

Fragen?