

Grundlagen von CUDA, Sprachtypische Elemente

Stefan Maskanitz

Übersicht

1. Einleitung
2. Spracheigenschaften
 - a. Threads, Blocks und Grids
 - b. Speicherorganisation
 - c. Fehlerbehandlung
 - d. Sonstiges
3. Vergleich: CPU/GPU Code

EINLEITUNG

Voraussetzungen

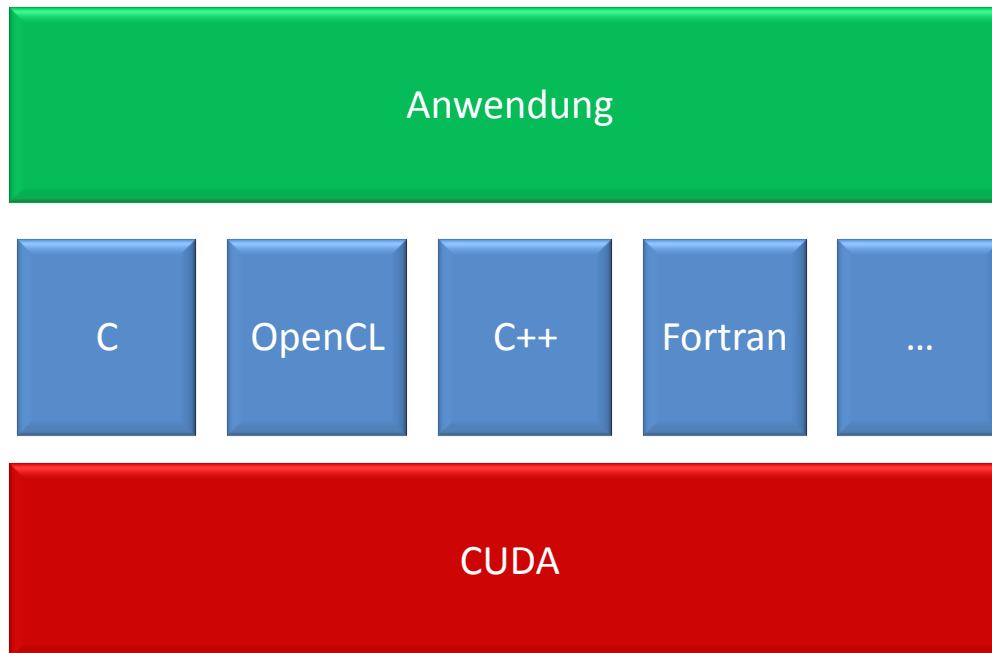
- Es wird mindestens eine CUDA fähige Nvidia Grafikkarte benötigt.
 - Dies sind alle Nvidia Modelle ab der G80 Reihe.
 - Beispiele:
 - Nvidia Geforce 8800 GTX
 - Nvidia Geforce 9600 GT
 - Nvidia Geforce 260 GTX

CUDA aktuell

- Zwei Wege, Code zu schreiben
 - C für CUDA
 - CUDA Driver API

CUDA zukünftig

- Mehrere Sprachen



C für CUDA

- Erweitert die Sprache C/C++ um einfache Sprachelemente, mit denen man Code für die Grafikkarte entwickeln kann
- Compiler: NVCC (gibt es mit dem SDK)
- Ermöglicht schnellen Einstieg

Trennung von Host und Device

- C Code wird auf dem Host (CPU) ausgeführt.
- CUDA Code wird auf einem physikalisch vom Host getrennten Device (GPU) ausgeführt.
- Host und Device verwalten ihren eigenen Speicher.
- Host muss die Device-Speicherverwaltung übernehmen
 - Hierzu gehört Allokation, Deallokation und das Kopieren von Daten von und zum Device.

CUDA Kernel

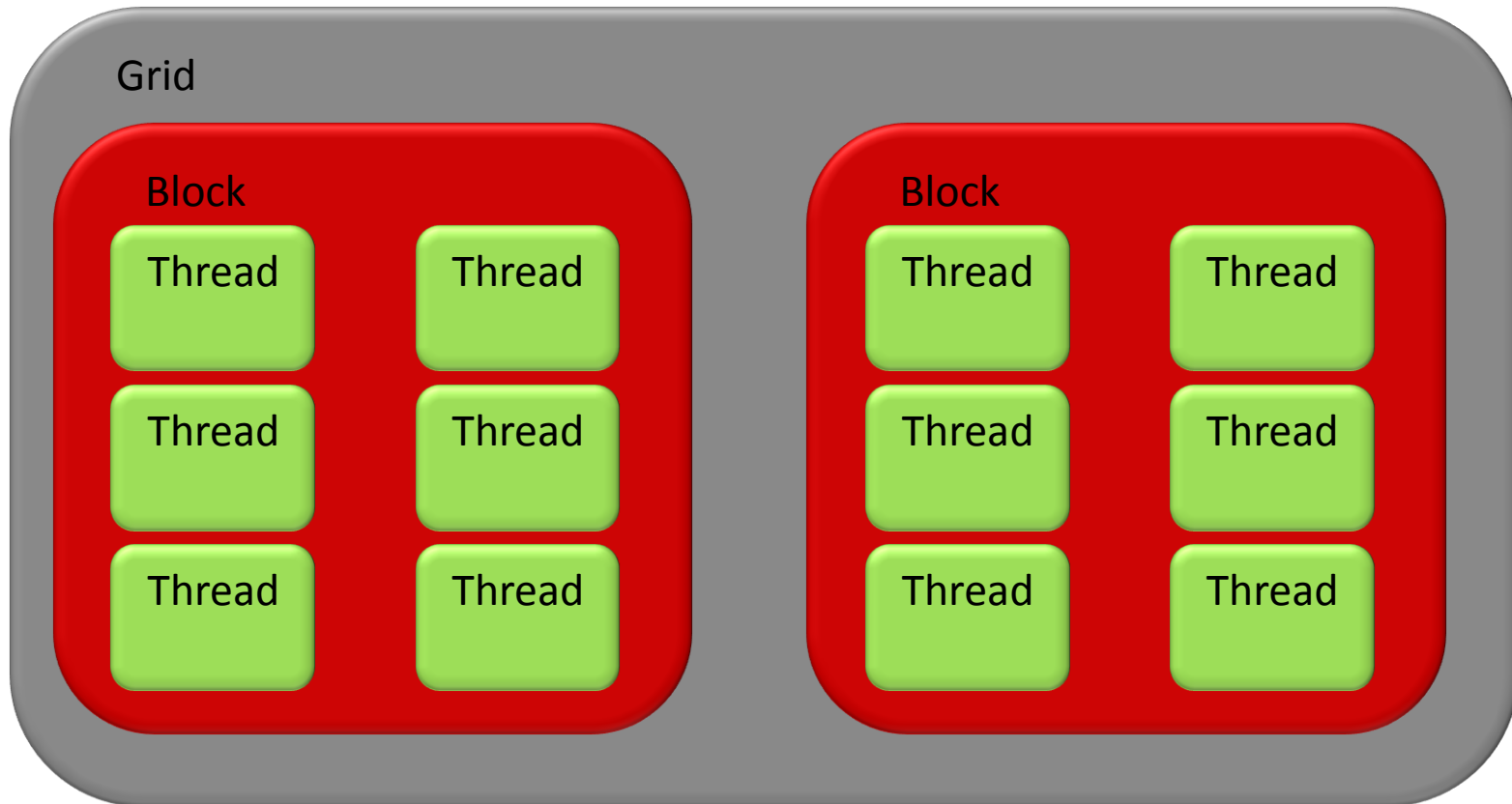
- Eine CUDA Funktion wird Kernel genannt.
- Ein Kernel muss so geschrieben sein, dass er von N verschiedenen Threads in beliebiger Reihenfolge ausgeführt werden kann. (SIMT)
- Dies ermöglicht eine einfache Skalierbarkeit
 - Neuere GPU Modelle besitzen mehr Recheneinheiten, die den selben Code ausführen.

Threads, Blocks und Grids

SPRACHEIGENSCHAFTEN

Threads, Blocks und Grids

CUDA Kernel werden von mehreren Threads ausgeführt.



Konfiguration der Grids/Blocks

- Wird im Kernelaufruf festgelegt

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
}
int main()
{
    // Kernel invocation
    // <<<dimGrid, dimBlock>>>
    VecAdd<<<1, N>>>(A, B, C);
}
```

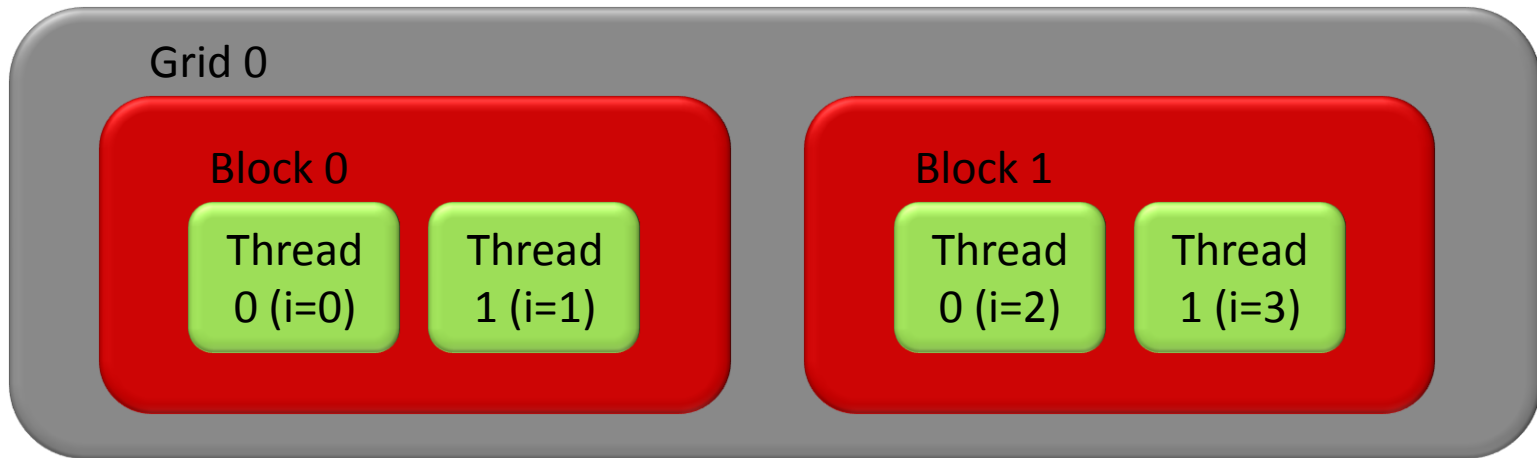
- Max. 512 Threads pro Block
 - Technische Grenze:
 - alle Threads eines Blocks sollen auf einem Prozessor ausgeführt werden
 - Prozessorspeicher begrenzt

blockIdx, blockDim, threadIdx

- Kernel besitzen Variablen mit Informationen über den Thread/Block, zu dem sie gehören

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    C[i] = A[i] + B[i];
}
int main()
{
    // Kernel invocation
    VecAdd<<<2, N/2>>>(A, B, C);
}
```

Beispiel für N = 4



```
int i = blockIdx.x * blockDim.x + threadIdx.x;
```

__syncthreads()

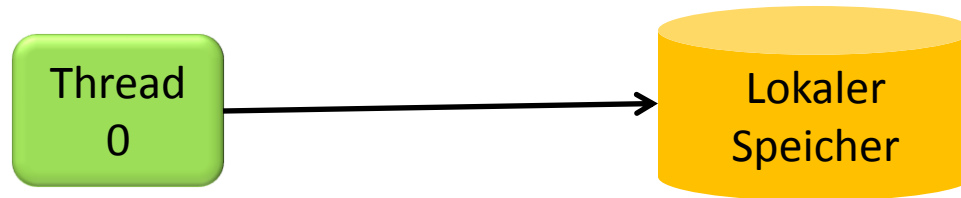
- Threads in einem Block können ihre Speicherzugriffe verwalten
 - „Barriere“, an der ein Thread wartet, bis auch alle anderen Threads aus dem gleichen Block an ihr angekommen sind.

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

Speicherorganisation

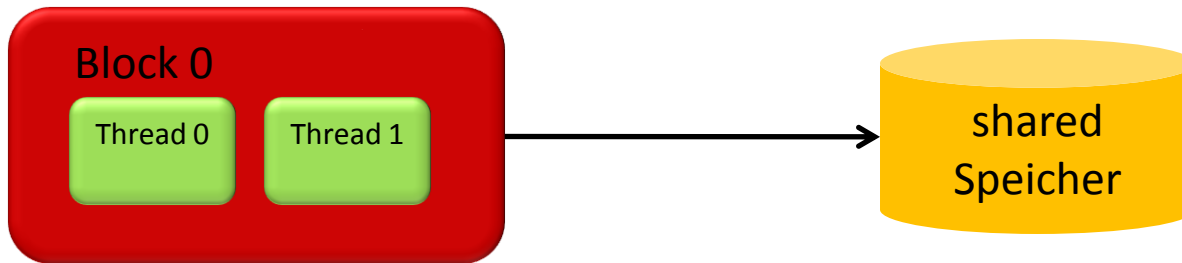
SPRACHEIGENSCHAFTEN

Lokaler Speicher



```
__global__ void foo(float* A, float* B)
{
    int counter = 0;
    // ...
}
```

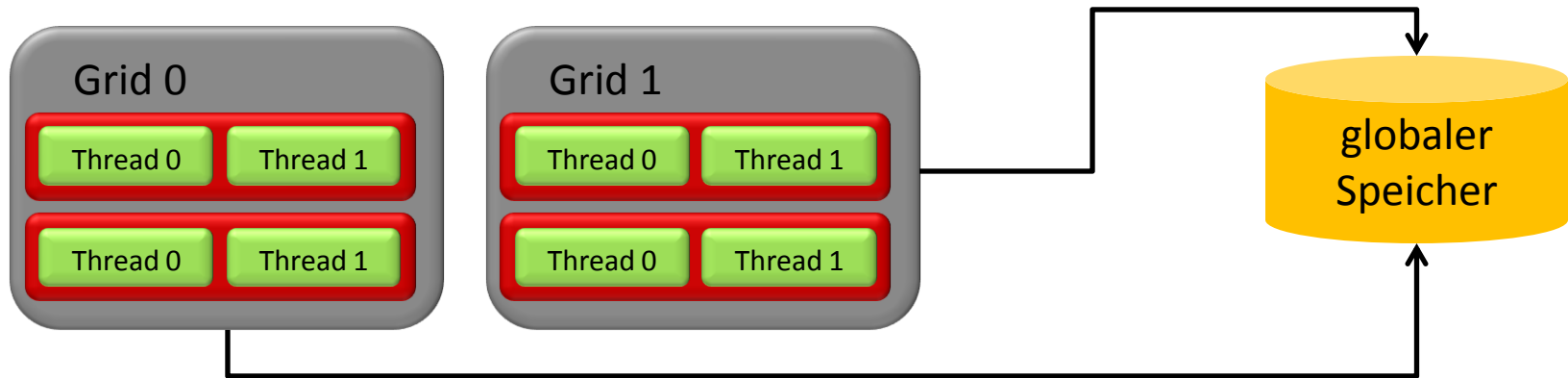
Shared Speicher



```
__global__ void foo(float* A, float* B)
{
    // ...
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    // ...
}
```

- Schnell (ähnlich Registerzugriff bei CPU Code)
- Größe begrenzt: 16 KB

Globaler Speicher



```
__global__ void VecAdd(float* A, float* B, float* C) {  
    //...  
}  
int main() {  
    // ...  
    cudaMalloc((void **) &A_d, size);  
    cudaMemcpy(A_d, A_h, size, cudaMemcpyHostToDevice);  
    VecAdd<<<2, N/2>>>(A_d, B_d, C_d);  
}
```

- Persistent über alle Kernel Aufrufe einer Anwendung

Fehlerbehandlung

SPRACHEIGENSCHAFTEN

cudaGetLastError()

- Ein Kernel gibt direkt keine Fehlermeldung aus.
- Man muss diese deshalb nach dem Aufruf mit `cudaGetLastError()` auslesen.

```
int main() {  
    // ...  
    VecAdd<<<2, N/2>>>(A_d, B_d, C_d);  
    cudaThreadSynchronize();  
    printf("%s\n", cudaGetErrorString(cudaGetLastError()));  
}
```

Device Emulation

- CUDA Code wird auf dem Host (CPU) ausgeführt
 - Debugger des Hosts kann verwendet werden (z.B. für breakpoints, ...)
 - Device Code kann um Hostfunktionen ergänzt werden (z.B. printf, ...)

Sonstiges

SPRACHEIGENSCHAFTEN

Device Funktion

- Kann zum Wiederverwenden von Code benutzt werden.
- Keine Rekursion möglich

```
__device__ int addiere(int a, int b) {  
    return a+b;  
}  
  
__global__ void foo(float* A, float* B, float* C) {  
    int a = 3;  
    int b = 4;  
    int c = addiere(a, b); // 7  
}
```

cudaThreadSynchronize()

- CUDA Kernel werden asynchron ausgeführt.
- Um auf Ergebnisse von Kernel Funktionen zu warten, kann man `cudaThreadSynchronize()` nutzen

```
int main() {  
    // ...  
    VecAdd<<<2, N/2>>>(A_d, B_d, C_d);  
    cudaThreadSynchronize();  
}
```

VERGLEICH: CPU/GPU CODE

Quadrieren

```
// CPU Code
```

```
int main() {
```

```
    int i = 0;
```

```
    float *zahlen_host;
```

```
    const int anz = 10;
```

```
    // Speicher reservieren
```

```
    zahlen_host =
```

```
    (float *)malloc( anz*sizeof(float) );
```

```
    // Zahlen quadrieren
```

```
    for ( i=0; i<anz; i++ )
```

```
        zahlen_host[i] = (float)i * (float)i;
```

```
    // Speicher freigeben
```

```
    free( zahlen_host );
```

```
}
```

```
// GPU Code
```

```
__global__ void quadrieren(float *zahlen) {
```

```
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
```

```
    // Zahl quadrieren
```

```
    zahlen[idx] = zahlen[idx] * zahlen[idx];
```

```
}
```

```
int main() {
```

```
    int i = 0;
```

```
    float *zahlen_host, *zahlen_device;
```

```
    const int anz = 10;
```

```
    // Speicher auf dem Host reservieren
```

```
    zahlen_host = (float *)malloc( anz*sizeof(float) );
```

```
    // Zahlen im Host-Speicher ablegen
```

```
    for ( i=0; i<anz; i++ ) zahlen_host[i] = (float)i;
```

```
    // Speicher auf dem Device reservieren
```

```
    cudaMalloc( (void **) &zahlen_device,  
                anz*sizeof(float) );
```

```
    // Kopieren der Zahlen zum Device
```

```
    cudaMemcpy( zahlen_device, zahlen_host,  
                anz*sizeof(float),  
                cudaMemcpyHostToDevice );
```

```
    // Quadrieren der Zahlen auf dem Device
```

```
    quadrieren<<< 2, 5 >>> ( zahlen_device );
```

```
    // Kopieren der Zahlen zum Host
```

```
    cudaMemcpy( zahlen_host, zahlen_device,  
                anz*sizeof(float),  
                cudaMemcpyDeviceToHost );
```

```
    // Speicher freigeben
```

```
    free( zahlen_host ); cudaFree( zahlen_device );
```

```
}
```