Praxiseinheit: „Realisierung einer hardwarebeschleunigten Disparitätenberechnung zur automatischen Auswertung von Stereobildern“

Christian Werner

Institut für Betriebssysteme und Rechnerverbund
TU Braunschweig

31.10., 01.11. und 28.11.2015

Gliederung

Motivation

- Bisher in der Vorlesung:
  Wie strukturiert man große, verteilte Anwendungen?
  - inhärent parallel
  - Trennung zwischen: UI, Logik und Daten
    ($\rightarrow$ 3- bzw. 4-Schicht-Modell)
- Nun:
  Wie optimiere ich die Performance, wenn es viel zu rechnen gibt?
  - Parallelität vs. schnelle sequentielle Ausführung
  - CPU vs. GPU
  - Berechnung lokal vs. remote
  - Kopplung mehrere Recheneinheiten in einem Verbund
- Schwerpunkt:
  „General Purpose“ Berechnungen auf GPUs (GPGPU)

Motivation

- Bisher in der Vorlesung:
  Wie strukturiert man große, verteilte Anwendungen?
  - inhärent parallel
  - Trennung zwischen: UI, Logik und Daten
    ($\rightarrow$ 3- bzw. 4-Schicht-Modell)
- Nun:
  Wie optimiere ich die Performance, wenn es viel zu rechnen gibt?
  - Parallelität vs. schnelle sequentielle Ausführung
  - CPU vs. GPU
  - Berechnung lokal vs. remote
  - Kopplung mehrere Recheneinheiten in einem Verbund
- Schwerpunkt:
  „General Purpose“ Berechnungen auf GPUs (GPGPU)

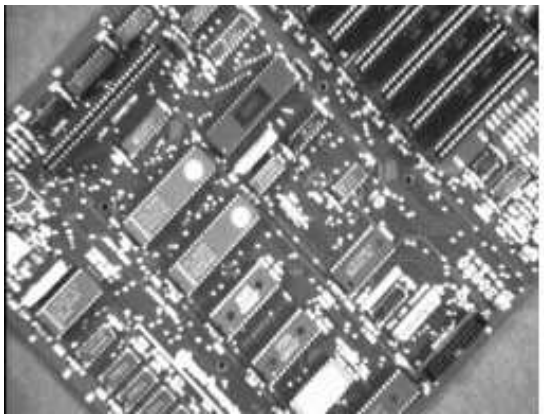
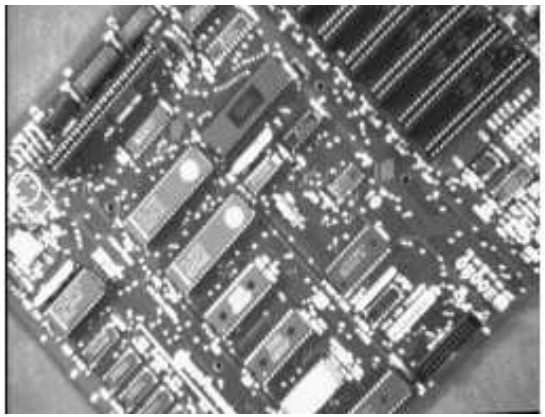
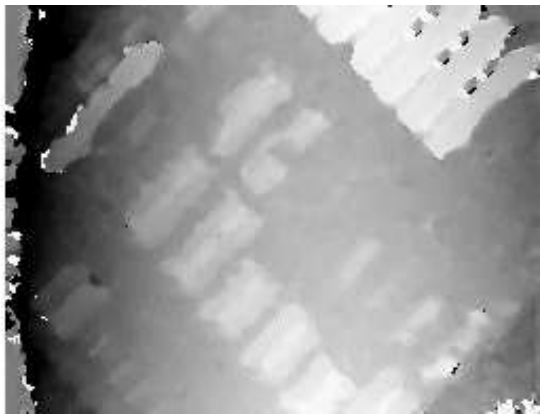
Motivation

- Bisher in der Vorlesung:
  Wie strukturiert man große, verteilte Anwendungen?
  - inhärent parallel
  - Trennung zwischen: UI, Logik und Daten
    ($\rightarrow$ 3- bzw. 4-Schicht-Modell)
- Nun:
  Wie optimiere ich die Performance, wenn es viel zu rechnen gibt?
  - Parallelität vs. schnelle sequentielle Ausführung
  - CPU vs. GPU
  - Berechnung lokal vs. remote
  - Kopplung mehrere Recheneinheiten in einem Verbund
- Schwerpunkt:
  „General Purpose“ Berechnungen auf GPUs (GPGPU)

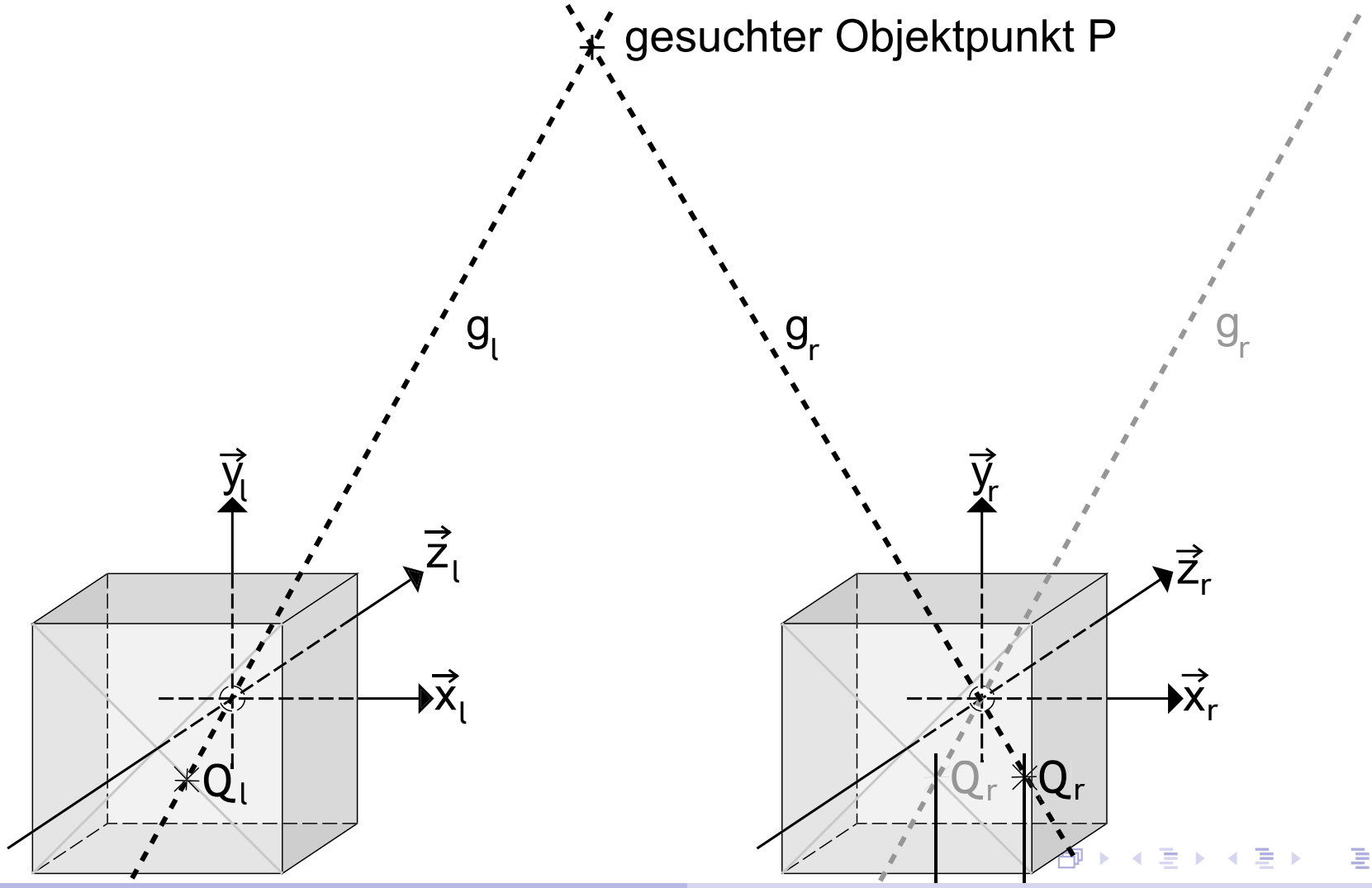
Motivation

Aber kann man die Rechenleistung auch für andere Zwecke einsetzen?

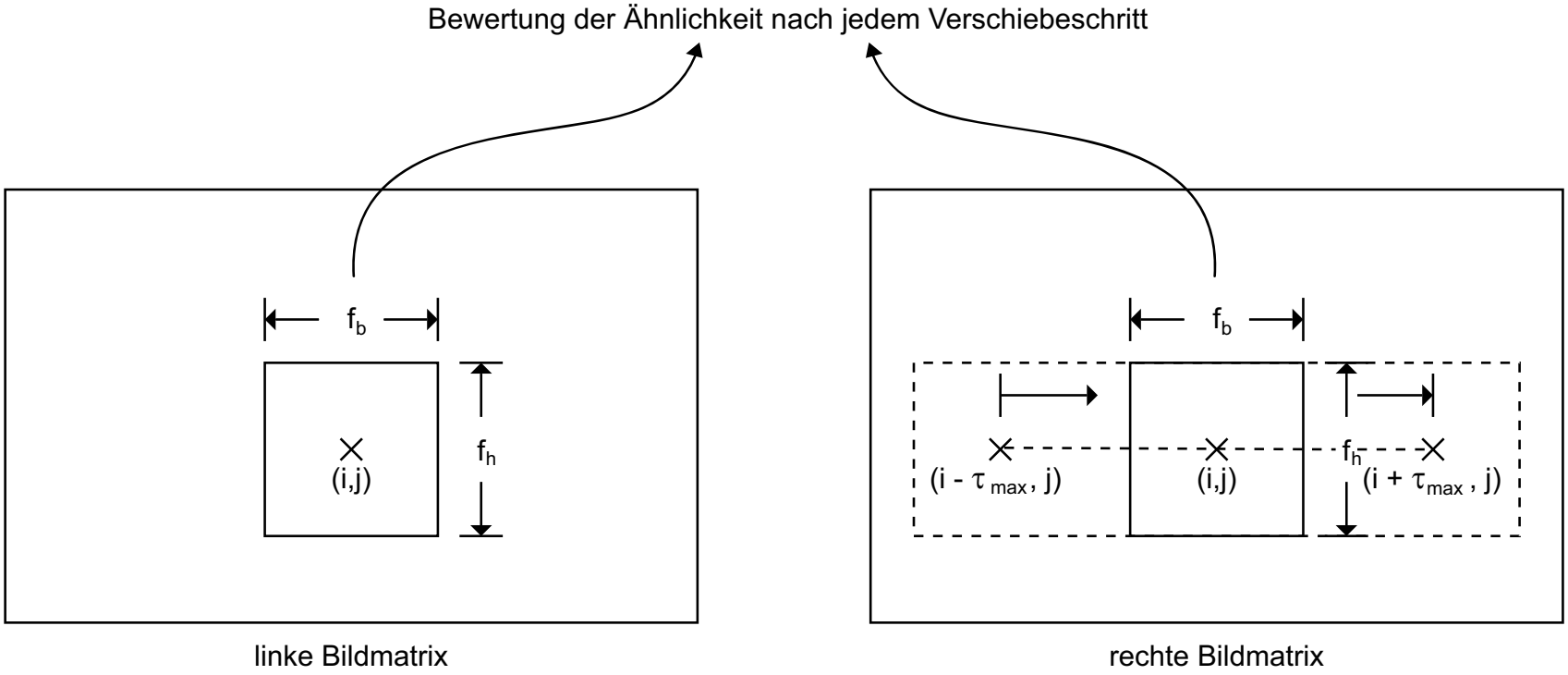
Stereobilder und Disparitätenmatrix

Rauminformationen aus Disparitätenmatrix



(Flächenbasierte) Berechnung der Disparitätenmatrix



Bewertungsfunktionen

$$f_1(i,j,\tau) = \frac{\sum_{k=-f_b/2}^{f_b/2} \sum_{l=-f_h/2}^{f_h/2} [P_l(i+k,j+l) - P_r(i+k+\tau,j+l)]^2}{\sqrt{\sum_{k=-f_b/2}^{f_b/2} \sum_{l=-f_h/2}^{f_h/2} P_l(i+k,j+l)^2 \cdot \sum_{k=-f_b/2}^{f_b/2} \sum_{l=-f_h/2}^{f_h/2} P_r(i+k+\tau,j+l)^2}}$$

$$f_2(i,j,\tau) = \sum_{k=-f_b/2}^{f_b/2} \sum_{l=-f_h/2}^{f_h/2} [P_l(i+k,j+l) - P_r(i+k+\tau,j+l)]^2$$

$$f_3(i,j,\tau) = \sum_{k=-f_b/2}^{f_b/2} \sum_{l=-f_h/2}^{f_h/2} |P_l(i+k,j+l) - P_r(i+k+\tau,j+l)|$$

Was ist CUDA?

- **C**ompute **U**nified **D**evice **A**rchitecture
- CUDA ist eine Bibliothek, die es erlaubt, auf NVIDIA GPUs eigenen Code auszuführen
- CUDA abstrahiert die Hardware so weit, dass man auf der GPU (nahezu) normal aussehenden C-Code ausführen kann
- Neben der nativen C-Bibliothek gibt es inzwischen auch Schnittstellen zu Matlab, Java und anderen Programmiersprachen

Aufgabenstellung

1. Implementieren Sie einen CUDA-Kernel, der mit Hilfe der Bewertungsfunktion $f_3(i, j, \tau)$ die Disparitätenmatrix für die Pixel eines Stereobildpaares mit einem möglichst höhen Parallelitätsgrad berechnet.
2. Binden Sie Ihren Kernel in die Software *StereoLab* ein und untersuchen Sie das Laufzeitverhalten im Vergleich zur sequentiellen Ausführung auf der Host-CPU.
3. Verwendem Sie einen Web-Service, der Ihren CUDA-Kernel kapselt und starten Sie eine lokale Instanz des Services. Untersuchen Sie das Laufzeitverhalten.
4. Implementieren Sie eine Client/Server-Variante auf Basis von TCP-Sockets. Untersuchen Sie das Laufzeitverhalten.
5. „Publizieren“ Sie Ihre Ergebnisse. So knapp wie möglich; min. 2, max. 4 Seiten.

Aufgabenstellung

1. Implementieren Sie einen CUDA-Kernel, der mit Hilfe der Bewertungsfunktion $f_3(i, j, \tau)$ die Disparitätenmatrix für die Pixel eines Stereobildpaares mit einem möglichst höhen Parallelitätsgrad berechnet.
2. Binden Sie Ihren Kernel in die Software *StereoLab* ein und untersuchen Sie das Laufzeitverhalten im Vergleich zur sequentiellen Ausführung auf der Host-CPU.
3. Verwendem Sie einen Web-Service, der Ihren CUDA-Kernel kapselt und starten Sie eine lokale Instanz des Services. Untersuchen Sie das Laufzeitverhalten.
4. Implementieren Sie eine Client/Server-Variante auf Basis von TCP-Sockets. Untersuchen Sie das Laufzeitverhalten.
5. „Publizieren“ Sie Ihre Ergebnisse. So knapp wie möglich; min. 2, max. 4 Seiten.

Aufgabenstellung

1. Implementieren Sie einen CUDA-Kernel, der mit Hilfe der Bewertungsfunktion $f_3(i, j, \tau)$ die Disparitätenmatrix für die Pixel eines Stereobildpaares mit einem möglichst höhen Parallelitätsgrad berechnet.
2. Binden Sie Ihren Kernel in die Software *StereoLab* ein und untersuchen Sie das Laufzeitverhalten im Vergleich zur sequentiellen Ausführung auf der Host-CPU.
3. Verwendem Sie einen Web-Service, der Ihren CUDA-Kernel kapselt und starten Sie eine lokale Instanz des Services. Untersuchen Sie das Laufzeitverhalten.
4. Implementieren Sie eine Client/Server-Variante auf Basis von TCP-Sockets. Untersuchen Sie das Laufzeitverhalten.
5. „Publizieren“ Sie Ihre Ergebnisse. So knapp wie möglich; min. 2, max. 4 Seiten.

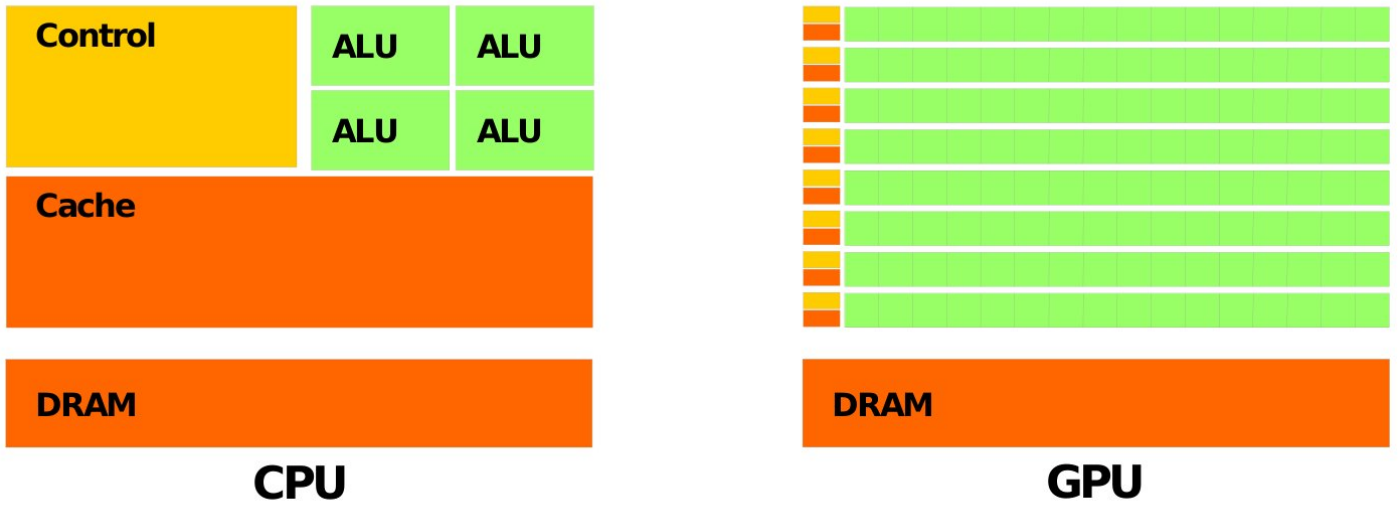
Aufgabenstellung

1. Implementieren Sie einen CUDA-Kernel, der mit Hilfe der Bewertungsfunktion $f_3(i, j, \tau)$ die Disparitätenmatrix für die Pixel eines Stereobildpaares mit einem möglichst höhen Parallelitätsgrad berechnet.
2. Binden Sie Ihren Kernel in die Software *StereoLab* ein und untersuchen Sie das Laufzeitverhalten im Vergleich zur sequentiellen Ausführung auf der Host-CPU.
3. Verwendem Sie einen Web-Service, der Ihren CUDA-Kernel kapselt und starten Sie eine lokale Instanz des Services. Untersuchen Sie das Laufzeitverhalten.
4. Implementieren Sie eine Client/Server-Variante auf Basis von TCP-Sockets. Untersuchen Sie das Laufzeitverhalten.
5. „Publizieren“ Sie Ihre Ergebnisse. So knapp wie möglich; min. 2, max. 4 Seiten.

Aufgabenstellung

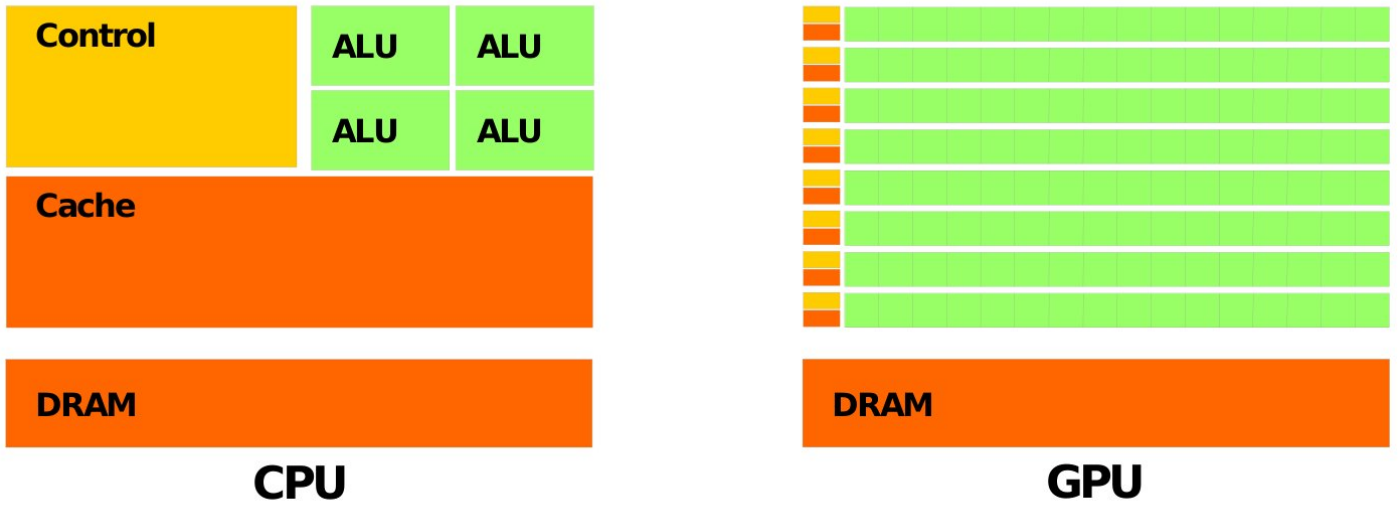
1. Implementieren Sie einen CUDA-Kernel, der mit Hilfe der Bewertungsfunktion $f_3(i, j, \tau)$ die Disparitätenmatrix für die Pixel eines Stereobildpaares mit einem möglichst höhen Parallelitätsgrad berechnet.
2. Binden Sie Ihren Kernel in die Software *StereoLab* ein und untersuchen Sie das Laufzeitverhalten im Vergleich zur sequentiellen Ausführung auf der Host-CPU.
3. Verwendem Sie einen Web-Service, der Ihren CUDA-Kernel kapselt und starten Sie eine lokale Instanz des Services. Untersuchen Sie das Laufzeitverhalten.
4. Implementieren Sie eine Client/Server-Variante auf Basis von TCP-Sockets. Untersuchen Sie das Laufzeitverhalten.
5. „Publizieren“ Sie Ihre Ergebnisse. So knapp wie möglich; min. 2, max. 4 Seiten.

Aufbau einer GPU



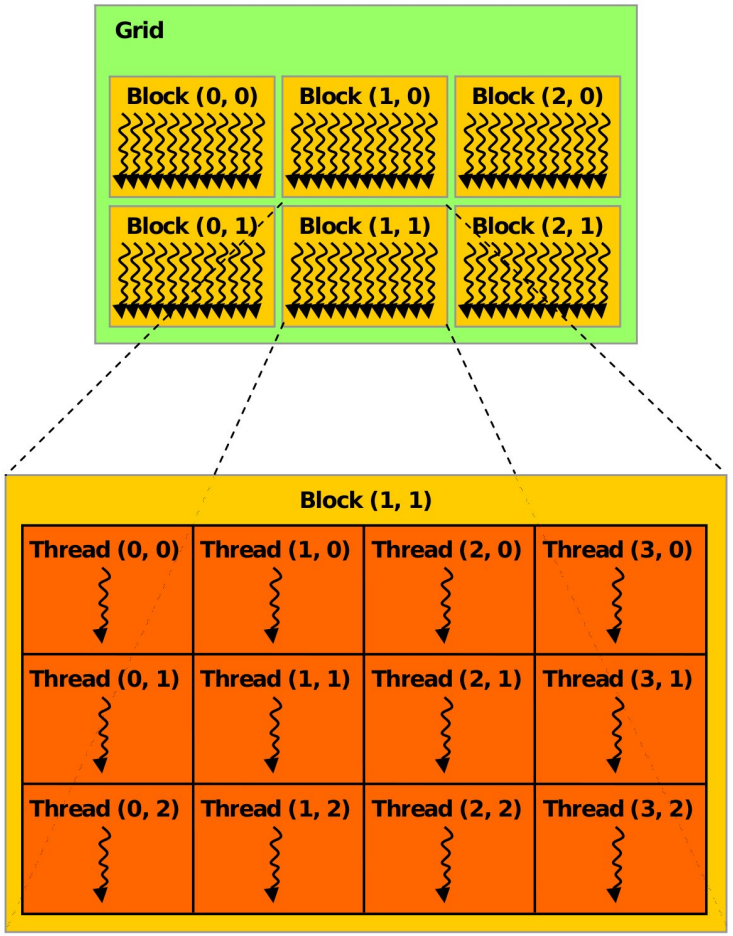
- Im Unterschied zu CPUs gibt es bei GPUs nur kleine Caches und keine komplexe Logik um den Programmablauf vorherzusagen
- Aber: Auf GPUs gibt es *viele* parallele Recheneinheiten

Aufbau einer GPU



- Im Unterschied zu CPUs gibt es bei GPUs nur kleine Caches und keine komplexe Logik um den Programmablauf vorherzusagen
- Aber: Auf GPUs gibt es *viele* parallele Recheneinheiten

Aufbau einer GPU



Aufbau einer GPU

- Eine aktuelle GPU besitzt mehrere Stream-Multiprozessoren (SM).
- Jeder davon besitzt mehrere Stream-Prozessoren (SP, alias Unified Shader oder Unified Streaming Processor)
- Jeder Stream-Prozessor kann pro Instruktion mehrere Datensätze verarbeiten (SIMD: single instruction, multiple data)
- Aktuelle NVIDIA-GPUs enthalten sehr viele Stream-Prozessoren
- im Pool: Kepler-Architektur, GK104, 1.152 SP verteilt auf 6 SM

Speicher der GPU

Local memory

Speicher pro Thread, z.B. für lokale Variablen

Shared memory

Speicher pro Block, nur Threads im Block haben Zugriff

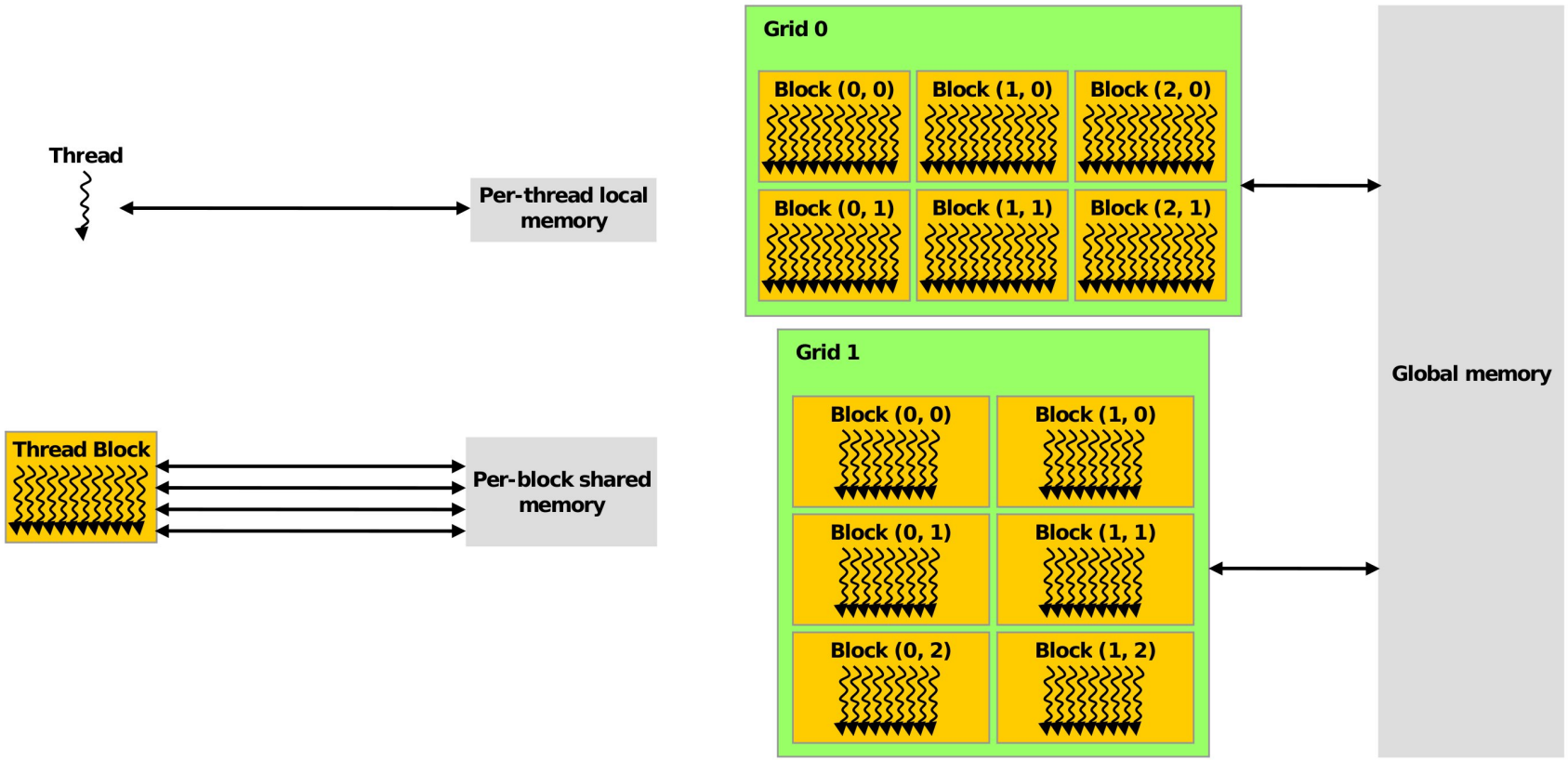
Global memory

Globaler Speicher, alle Threads haben darauf Zugriff (am langsamsten)

Texture memory

Globaler Read-only Speicher

Speicher der GPU



Aufruf eines Kernels

```
1  // Kernel definition
2  __global__ void VecAdd(float* A, float* B, float* C)
3  {
4  }
5
6  int main()
7  {
8      // Kernel invocation
9      VecAdd<<<1, N>>>(A, B, C);
10     // Note: A,B,C: pointers to CUDA global memory!
11 }
```

Aufruf eines Kernels

- `nvcc`, ein von NVIDIA gestellter spezieller (Pre)-Compiler übersetzt .cu Dateien (C-Dateien mit zusätzlichen Annotationen).
- Mit `__global__` annotierte Funktionen werden auf der GPU ausgeführt.
- `FunctionName<<<...>>>()` ist der Syntax zum Ausführen der Funktion auf der GPU.

Thread-Index

```
1  // Kernel definition
2  __global__ void MatAdd(float A[N][N], float B[N][N],
3                         float C[N][N])
4  {
5      int i = threadIdx.x;
6      int j = threadIdx.y;
7      C[i][j] = A[i][j] + B[i][j];
8  }
9
10 int main()
11 {
12     // Kernel invocation
13     dim3 dimBlock(N, N);
14     MatAdd<<<1, dimBlock>>>(A, B, C);
15 }
```

Thread-Index

- In `FunctionName<<<dimGrid, dimBlock>>>()` geben dimGrid und dimBlock an, auf wievielen GPU-Blöcken (dimGrid) und auf wievielen Threads pro Block der Kernel ausgeführt wird
- Innerhalb des Kernels kann dann mittels threadIdx und blockIdx auf den Index des entsprechenden Threads und Blocks zugegriffen werden (globale Variablen)
- Über threadIdx und blockIdx kann man innerhalb des Kernels erkennen, in welchem Thread und Block man sich befindet, um unterschiedliche Berechnungen durchführen zu können (z.B. andere Abschnitte eines Bildes)

Berechnungen auf mehrere Blöcke verteilen

```
1  // Kernel definition
2  __global__ void MatAdd(float A[N][N], float B[N][N],
3                         float C[N][N]) {
4      int i = blockIdx.x * blockDim.x + threadIdx.x;
5      int j = blockIdx.y * blockDim.y + threadIdx.y;
6      if (i < N && j < N)
7          C[i][j] = A[i][j] + B[i][j];
8  }
9
10 int main() {
11     // Kernel invocation
12     dim3 dimBlock(16, 16);
13     dim3 dimGrid((N + dimBlock.x - 1) / dimBlock.x,
14                  (N + dimBlock.y - 1) / dimBlock.y);
15     MatAdd<<<dimGrid, dimBlock>>>(A, B, C);
16 }
```

Berechnungen auf mehrere Blöcke verteilen

- Hier werden pro Block immer genau $16 * 16 = 256$ Threads eingesetzt
- Dies ist notwendig, da die eingesetzten Grafikprozessoren Beschränkungen setzen (typische GPUs können zurzeit maximal 512 Threads pro Block ausführen)
- Um am Ende das ganze zweidimensionale Array abzudecken wird die Dimension des Grids dann dynamisch aus der Größe des Arrays und der Größe des Blocks berechnet
- `dimBlock` darf nie mehr Threads angeben als die GPU pro Block ausführen kann
- `dimGrid` kann dagegen mit der Größe der zu verarbeitenden Daten skalieren. Die GPU führt so viele Blöcke wie möglich parallel aus, der Rest sequenziell

Speicher allokieren

```
1  // Allocate vectors in device memory
2  float *d_A, *d_B, *d_C;
3  cudaMalloc((void**)&d_A, N * sizeof(float));
4  cudaMalloc((void**)&d_B, N * sizeof(float));
5  cudaMalloc((void**)&d_C, N * sizeof(float));
6
7  // [...]
8
9  // free memory
10 cudaFree(d_A);
11 cudaFree(d_B);
12 cudaFree(d_C);
```

Speicher allokieren

- Globaler Speicher in der GPU wird wie in C üblich allokiert und freigegeben, nur mit speziellen Methoden
- Der resultierende Pointer darf nicht in Host-Code benutzt werden, da auf den GPU-Speicher nicht direkt zugegriffen werden kann!

Zwischen Host-Speicher und CUDA-Speicher kopieren

```
1
2  // Copy vectors from host memory to device memory
3  // h_A and h_B are input vectors stored in host memory
4  cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
5  cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
6
7  // [execute kernel]
8
9  // Copy result from device memory to host memory
10 // h_C contains the result in host memory
11 cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
```

Zwischen Host-Speicher und CUDA-Speicher kopieren

- Mittels `cudaMemcpy` können zwischen Host- und CUDA-Speicher Daten ausgetauscht werden

Zum Weiterlesen...

- CUDA C Programming Guide (`http://docs.nvidia.com/cuda/cuda-c-programming-guide/`)
- Beispiel-Styles für Publikation (`http://www.ieee.org/publications_standards/publications/authors/authors_journals.html`)