Spezialarchitekturen I: GPGPU

Architektur, Programmierung und Anwendungen

Mario Kicherer

Seminar „High-performance reconfigurable computing“
Institut für Technische Informatik (ITEC)
Universität Karlsruhe (TH)

29. Februar 2008

Inhaltsverzeichnis

Was bedeutet GPGPU?

GPGPU ist eine Abkürzung für „General purpose computations on GPUs“ und steht für die Nutzung von GPUs zur Lösung von allgemeinen mathematischen Problemen.

GPUs früher und heute

- Früher
  - Einfache Framebuffer bzw. Speicher mit Digital-Analog-Wandler
  - Mit den Jahren zunehmend mehr Logik in den GPUs
- Heute
  - Hoch entwickelte massiv-parallele Streamprozessoren

Aufbau einer GPU I

- Bis ein fertiges Bild auf dem Monitor erscheint müssen mehrere Stufen einer Pipeline durchlaufen werden. Die wichtigsten Stufen einzelnd erklärt:

  Input Die Übergabe der Daten erfolgt meist über eine Grafikbibliothek wie OpenGL oder DirectX. Diese abstrahieren die modellspezifischen Protokolle der Grafikchips.

  Transform&Light In dieser Stufe werden die Eckpunkte der einzelnen Modelle in ein sogenanntes Weltkoordinatensystem eingefügt und deren Beleuchtung bestimmt.

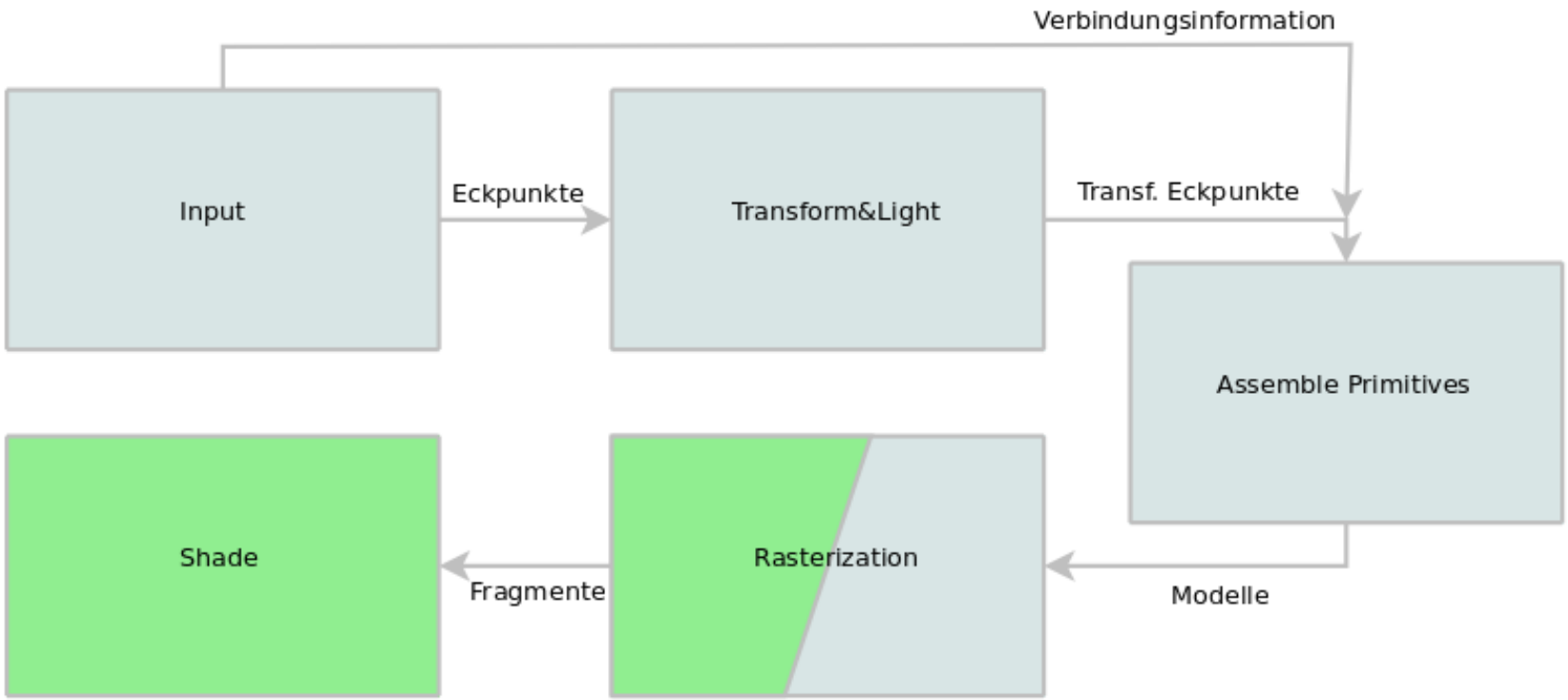
Aufbau einer GPU II

Assemble primitives In dieser Stufe werden die einzelnen Eckpunkte wieder zu grösseren Einheiten wie Dreiecken zusammengefügt.

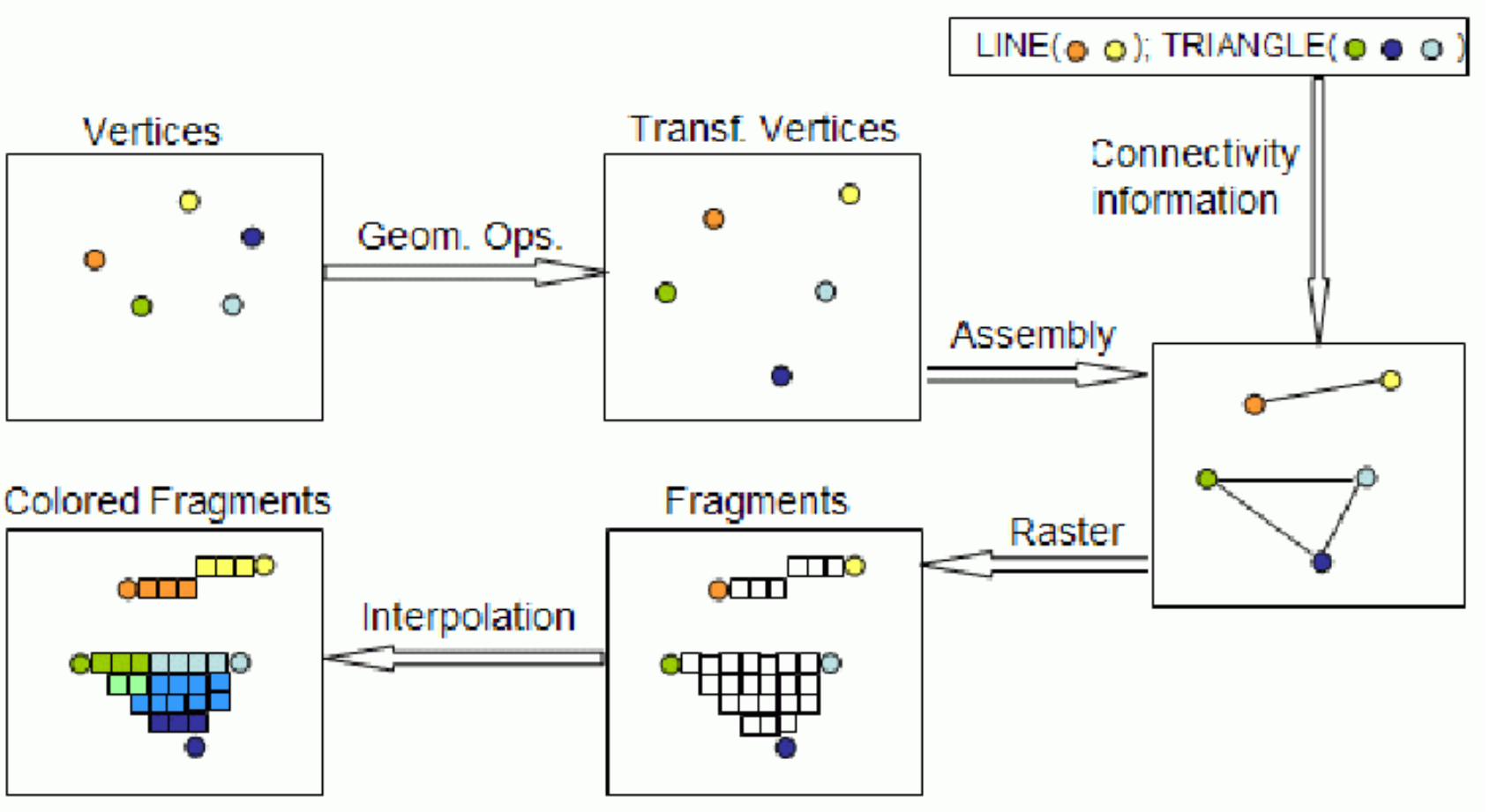
Rasterization Von den durch Vektoren beschriebenen Modellen wird nun ein 2-dimensionales Abbild berechnet. Dieses kann man sich als Prototyp des späteren Bilds auf dem Monitor vorstellen.

Shading Für jedes Prototyp-Pixel wird nun u.a. mittels der Texturdaten die endgültige Farbe berechnet.

Schematischer Aufbau der GPU Pipeline



Beispielhafter Ablauf der GPU Pipelinestufen



Für GPGPU nutzbare Strukturen

Vertex processor

- Entstanden aus der „Transform&Light“-Stufe
- Voll programmierbar (SIMD wie auch MIMD)
- Die Position der aktuellen Dateneinheit im Speicher kann verändert werden („Scatter“- Prinzip).
- Es können keine anderen Dateneinheiten außer der aktuellen gelesen werden

Für GPGPU nutzbare Strukturen

Geometry processor

- Entstanden aus der „Assemble Primitives“-Stufe
- Spezifiziert seit DirectX-10
- Ist flexibler als der Vertex processor und kann auch neue Daten hinzufügen bzw. entfernen.

Für GPGPU nutzbare Strukturen

Fragment processor

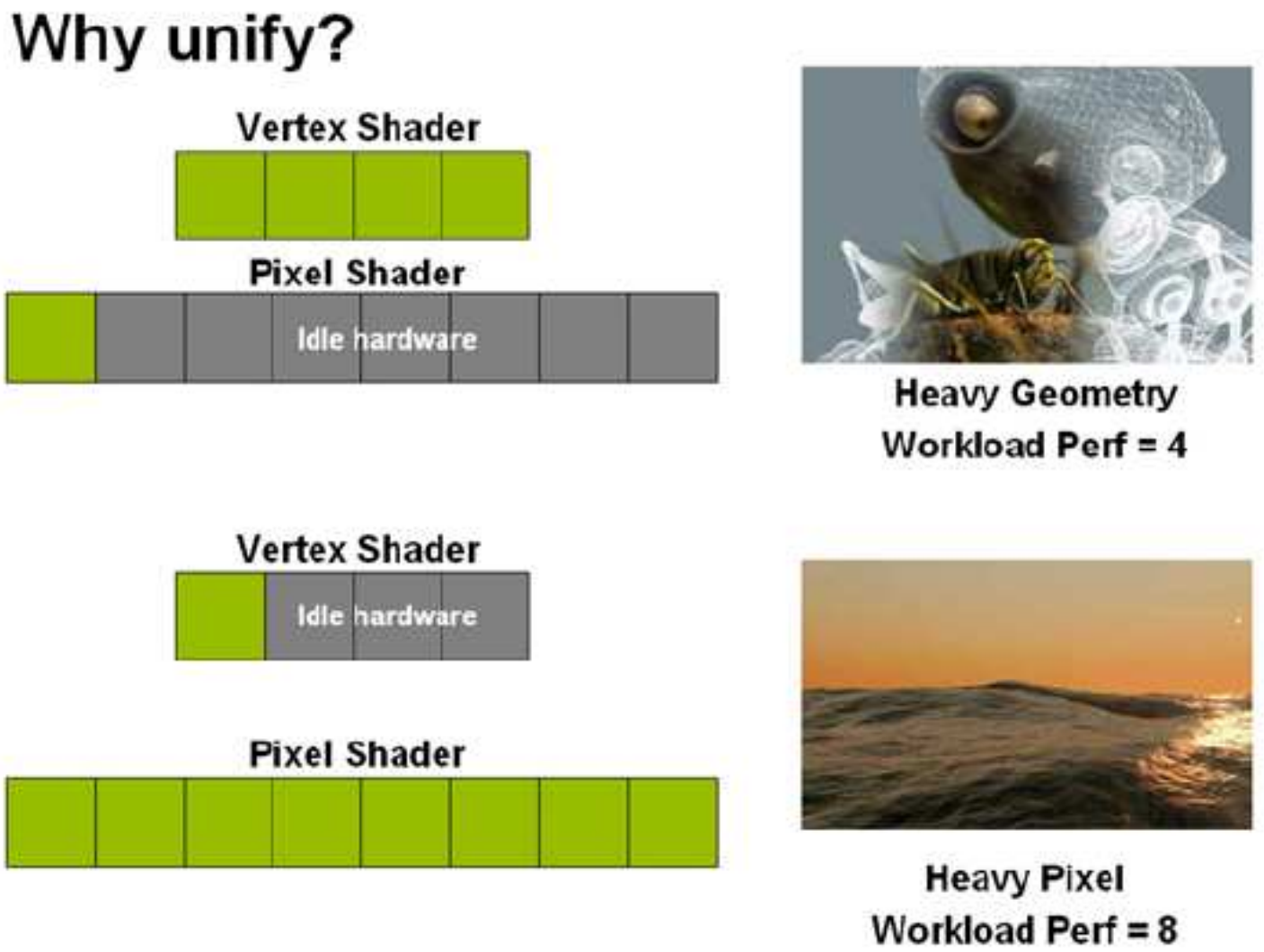
- Entstanden aus der Shader-Stufe
- Volle Programmierbarkeit (nur SIMD)
- Es kann von beliebigen Speicheradressen gelesen werden („Gather“ Prinzip).
- Die Speicherstelle zur Ausgabe kann nicht frei gewählt werden.

Für GPGPU nutzbare Strukturen

NEU: Unified Shaders

- Erstmals auf dem PC-Markt in einer GPU von Nvidia
- Können zur Laufzeit individuell entweder als Vertex-, Geometry- oder Fragment-processor benutzt werden.
- Austattung der aktuellen GPUs:
  - Nvidia G92: 128 Unified shaders
  - ATI RV670: 320 Unified shaders

Für GPGPU nutzbare Strukturen



Generelle Einschränkungen

- Für einige wissenschaftliche Anwendungen war der GPGPU-Ansatz bisher nur eingeschränkt eine Option, da - wenn überhaupt - nur Datentypen mit einfacher Genauigkeit verarbeitet werden konnten.
- Erst mit der neusten Generation (DirectX-10.1) sind die GPUs durch die Einführung von 64-bit-Datentypen auch für diese Bereiche interessant geworden.
- Die einzelnen Prozessoren unterliegen immernoch den Regeln der Pipeline, d.h. jede Stufe hat festgelegte Ein- und Ausgabeparameter.
- Vor der Shader-Version-4 war die Anzahl der Instruktionen pro Kernel begrenzt.

Motivation

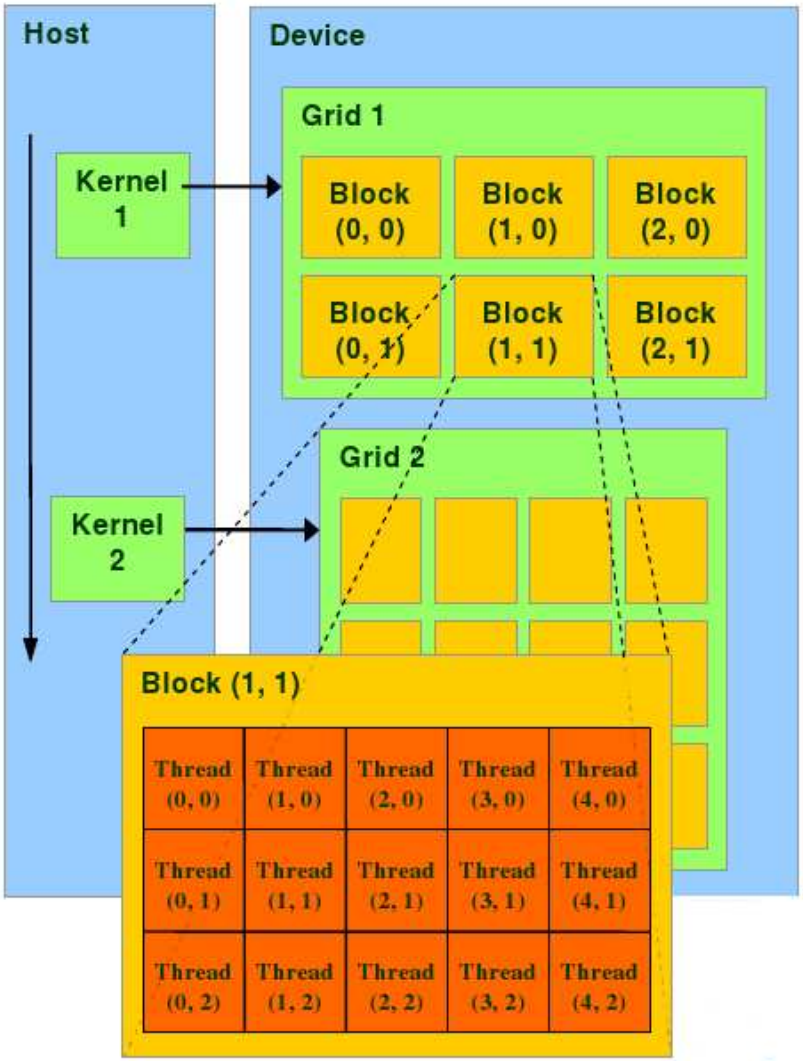
- Konventionelle Shadersprachen (wie Cg, HLSL und GLSL) für Grafikverarbeitung ausgelegt.
- Es wurden Lösungen entwickelt, die dem Programmierer das Schreiben von GPGPU-Andwendungen erleichtern, z.B. CUDA, CTM, Brook und RapidMind.

Nvidia CUDA

- Entwickelt von Nvidia, veröffentlicht im Februar 2007
- C mit so genannten parallelen Extensions
- Unterstützt ab Geforce-8-Serie
- Abstrahiert die vorhandenen Streamprozessoren in einem System

Schematische Organisation

- Thread IDs werden in mehrdimensionalen Blöcken organisiert
- Threads innerhalb eines Blocks können auf einen gemeinsamen Speicherbereich zugreifen und Semaphoren nutzen



Codebeispiel I

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition

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

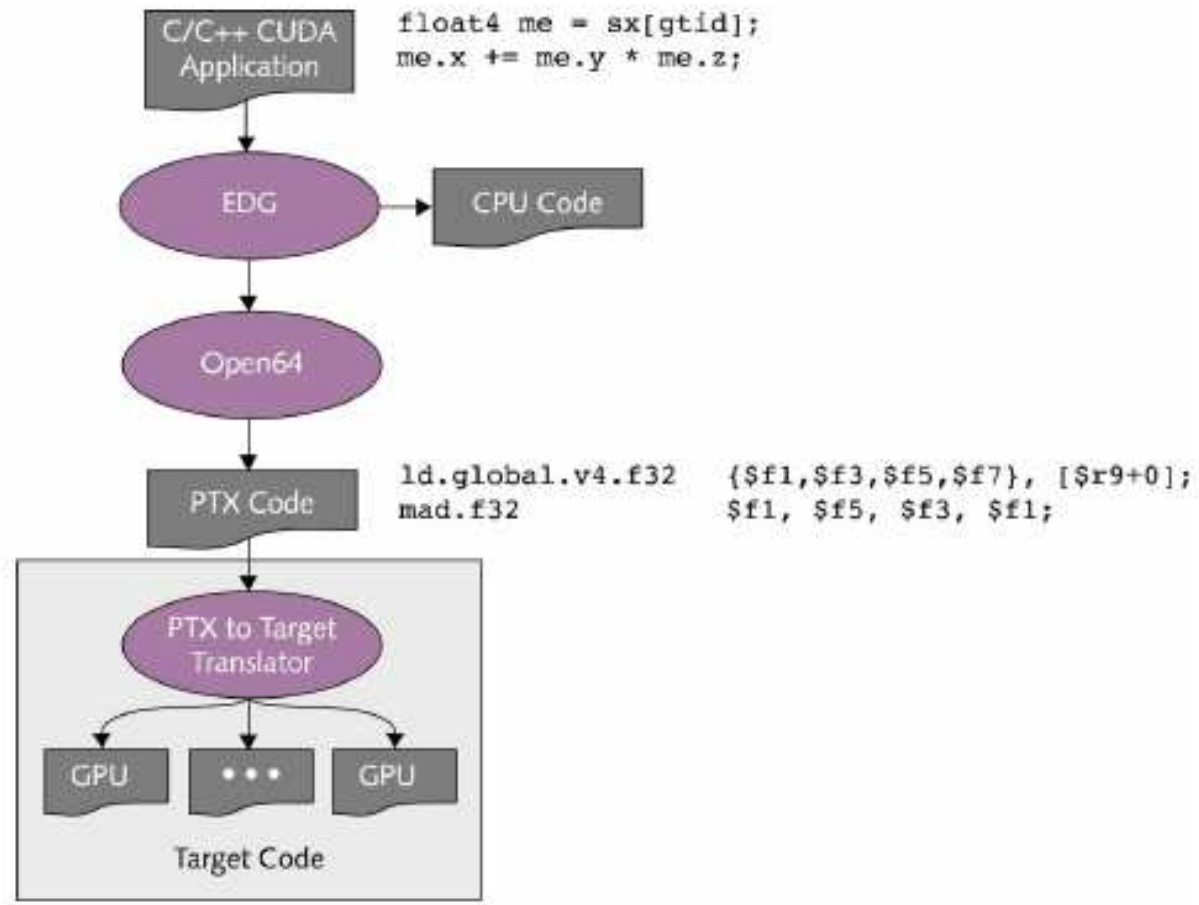
Codebeispiel II

```
// allocate host (CPU) memory
float* h_A = (float*) malloc(N * sizeof(float));
float* h_B = (float*) malloc(N * sizeof(float));
… initalize h_A and h_B …

// allocate device (GPU) memory
float* d_A, d_B, d_C;
cudaMalloc( (void**) &d_A, N * sizeof(float));
cudaMalloc( (void**) &d_B, N * sizeof(float));
cudaMalloc( (void**) &d_C, N * sizeof(float));

// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float),
  cudaMemcpyHostToDevice) );
cudaMemcpy( d_B, h_B, N * sizeof(float),
  cudaMemcpyHostToDevice) );
```

Codebeispiel III

```
__global__
void vecAdd(float* A, float* B, float* C);

// From host, kernel launch is like function call
// Execute kernel on N/256 blocks of 256 threads
vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
```

Code-Übersetzung



Folding@Home

- Projekt der Universität von Stanford
- Distributed-Computing-Network zur Simulation von Proteinfaltungen
- Erforschung der Veränderung des Verhaltens von Proteinen bei einer Faltung

Motivation

- Die Aminosäuresequenz eines Proteins muss eine bestimmte räumliche Struktur durch Faltung erreichen um korrekt zu funktionieren.
- Die Ursachen für Alzheimer, BSE, CJD (Creutzfeldt-Jakob-Krankheit), ALS (Amyotrophe Lateralsklerose) oder Parkinson werden inzwischen auf die Missfaltung von Proteinen zurückgeführt.

Problematik

- Gängige Computer können die ersten Nanosekunden einer Faltung in absehbarer Zeit simulieren.
- Gesamter Ablauf einer korrekten Faltung kann mehrere Mikro- bis Millisekunden dauern.

Leistungsvergleich

OS Type	Current TFLOPS*	Active CPUs	Total CPUs	MFLOPS/Unit
Windows	175	183878	1905327	0.95
Mac OS X/PPC	7	9315	110710	0.75
Mac OS X/Intel	21	6700	35228	3.13
Linux	41	23899	267532	1.72
GPU	37	634	5029	58.36
Playstation	818	32991	412446	24.79
Total	1099	257417	2736272	4.27

*TFLOPS is the actual teraflops from the software cores, not the peak values from CPU/GPU/PS3 specs

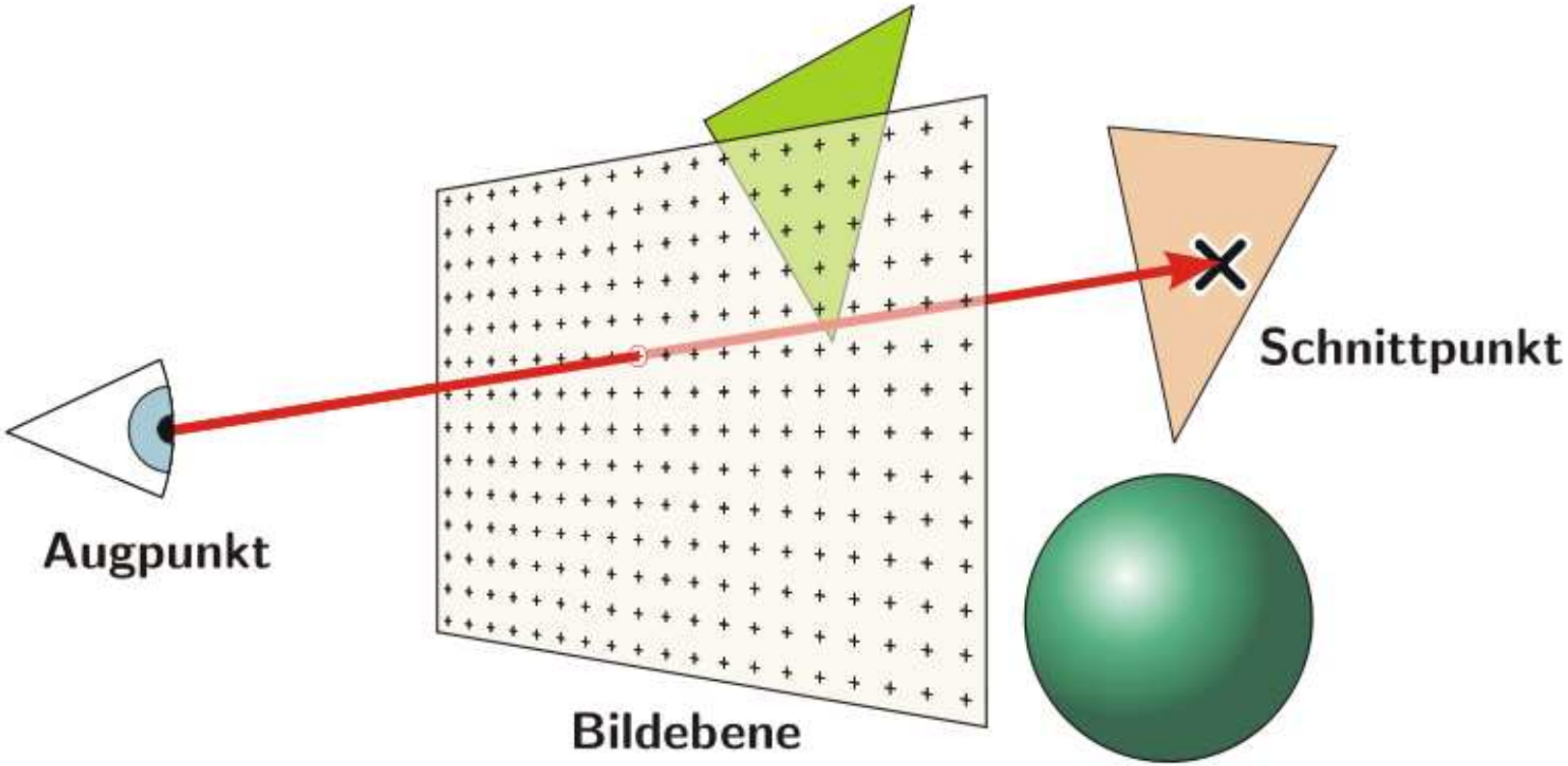
Stand 21.01.2008, Quelle http://folding.stanford.edu

Raytracing

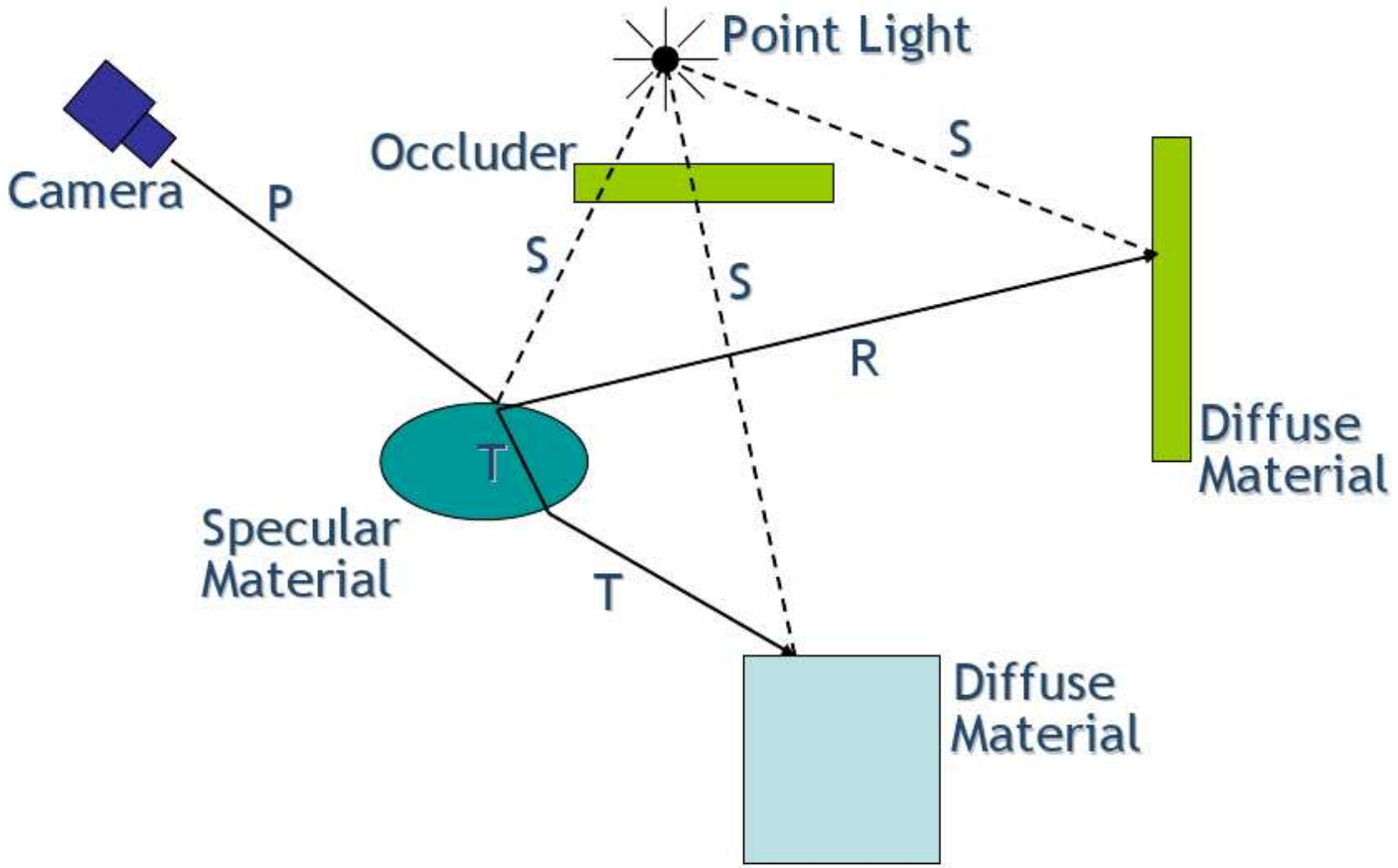
- Verfahren zur Berechnung von 2-dimensionalen Bildern aus 3-dimensionalen Objektbeschreibungen
- Realistische Berechnung einer Szene nahezu unmöglich
- Verlustbehaftete Optimierungen wurden entwickelt um den Vorgang zu beschleunigen.

Beispielszene

Funktionsweise Raycasting



Funktionsweise Raytracing



Resultate

Bei sogenannten hybriden Verfahren, bei denen die Berechnung zwischen CPU und GPU aufgeteilt wird, konnten bei den Raytracing-Methoden eine durschnittliche Verdoppelung der Geschwindigkeit festgestellt werden, bei der Raycasting-Methode sogar eine Verbesserung um den Faktor 10.

Fragen?

Referenzen I


David Luebke, NVIDIA
General-purpose computation on graphics hardware
Supercomputing 2006.


John Owens, UC Davis
Data-parallel Algorithms and Data Structures
Supercomputing 2007.


Mike Houston, Stanford University
High Level Languages for GPUs - Overview“,
SIGGRAPH 2007.


Mark Harris, NVIDIA
Introduction to CUDA
SIGGRAPH 2007.

Referenzen II


Tom R. Halfhill, www.MPRonline.com
Parallel processing with CUDA


GPGPU.org
http://www.gpgpu.org


Stanford University
Folding@Home
http://folding.stanford.edu


Philippe C.D. Robert, Severin Schoepke and Hanspeter Bieri
Hybrid Ray Tracing - Ray Tracing Using GPU-Accelerated Image-Space Methods
Institute of Computer Science and Applied Mathematics, University of Bern