Volumenrendering mit CUDA

Jörg Mensmann

Arbeitsgruppe Visualisierung und Computergrafik
http://viscg.uni-muenster.de

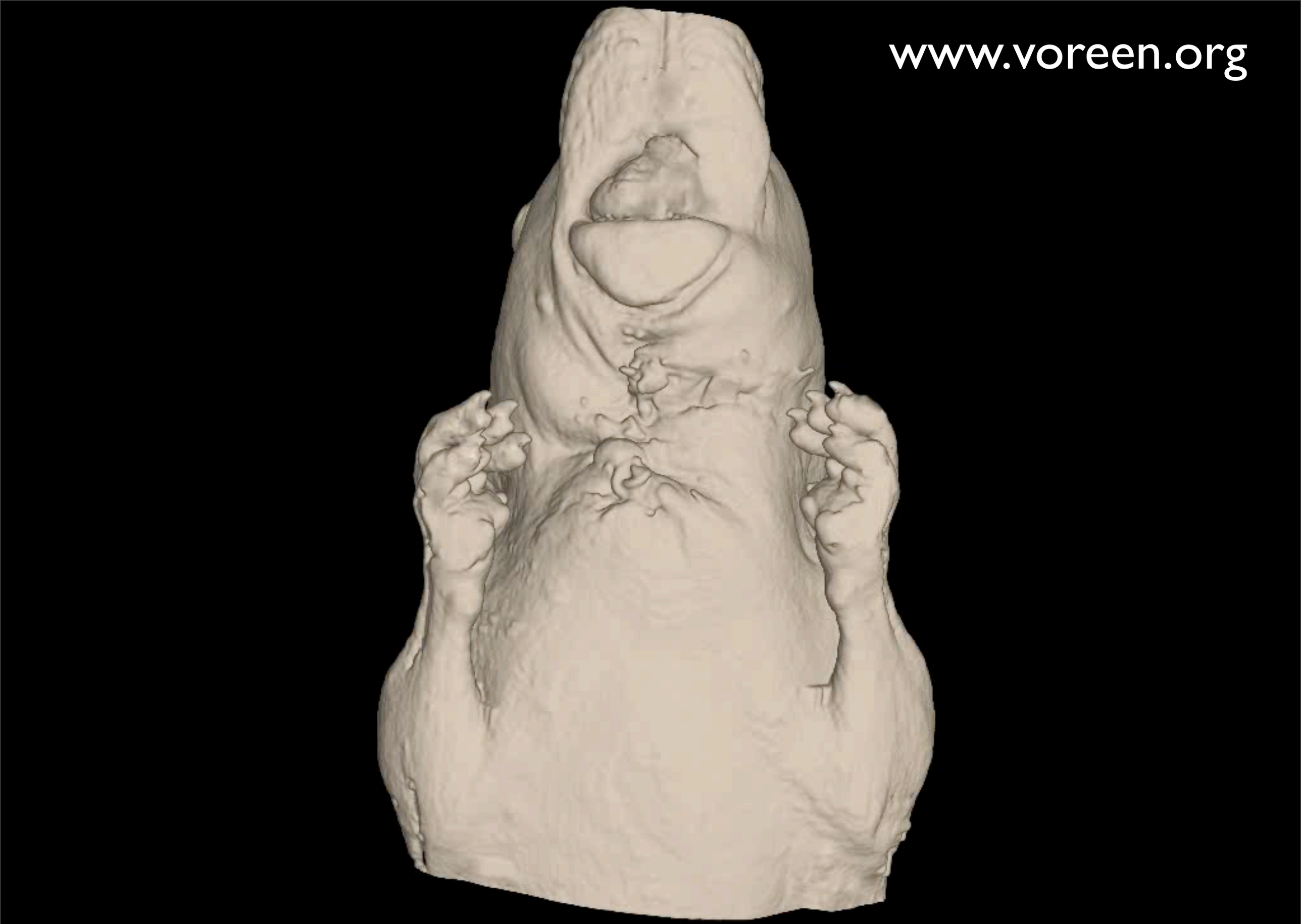
Überblick

- Volumenrendering allgemein
- Raycasting-Algorithmus
- Volumen-Raycasting mit CUDA
- Optimierung

Volumenrendering

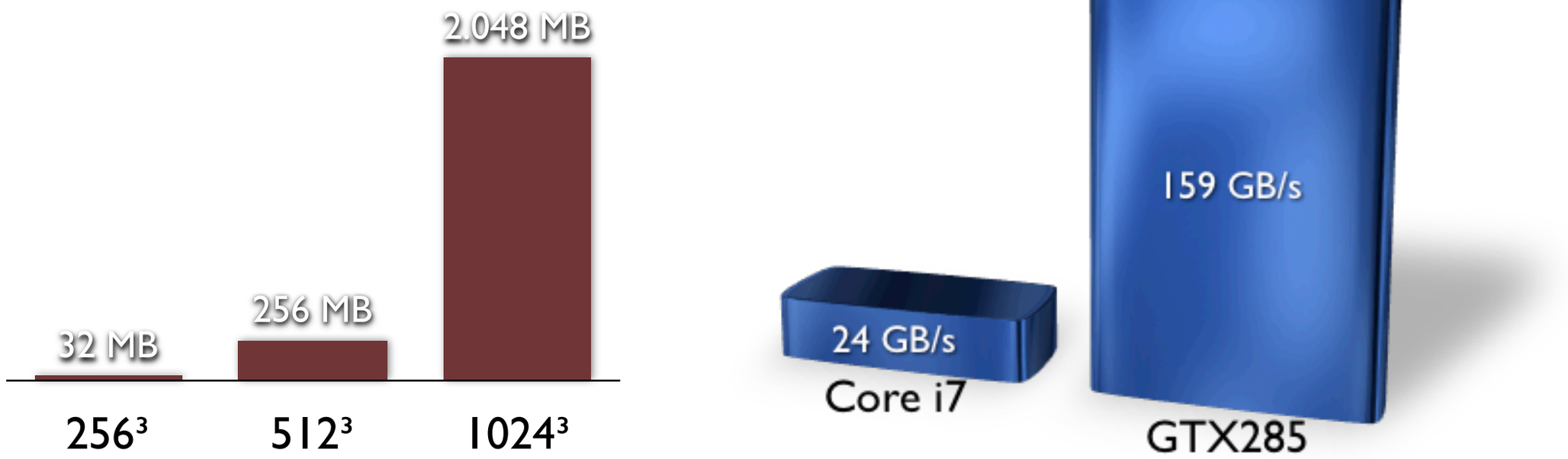
- Aufgabe: Visualisierung volumetrischer Daten
- Einsatzgebiete: Medizin, Geologie, Meterologie, Materialprüfung, ...
- Datenquellen: CT, MR, PET, Ultraschall, ...

www.voreen.org



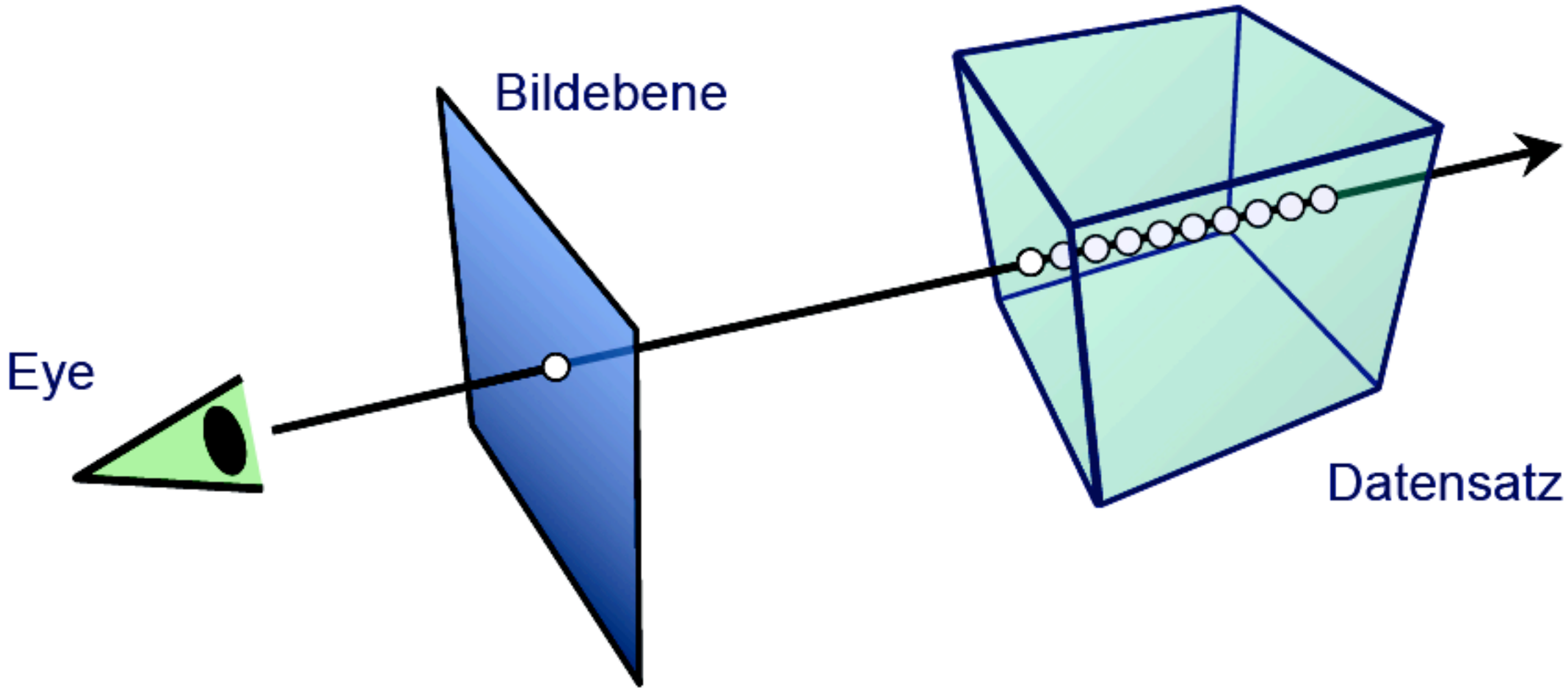
Volumenrendering

- Ziel: interaktives Rendering (>10 fps)
- Problem: große Datenmengen



- CPU-Implementierungen nicht interaktiv
- GPUs bieten höhere Speicherbandbreite

Volumen-Raycasting



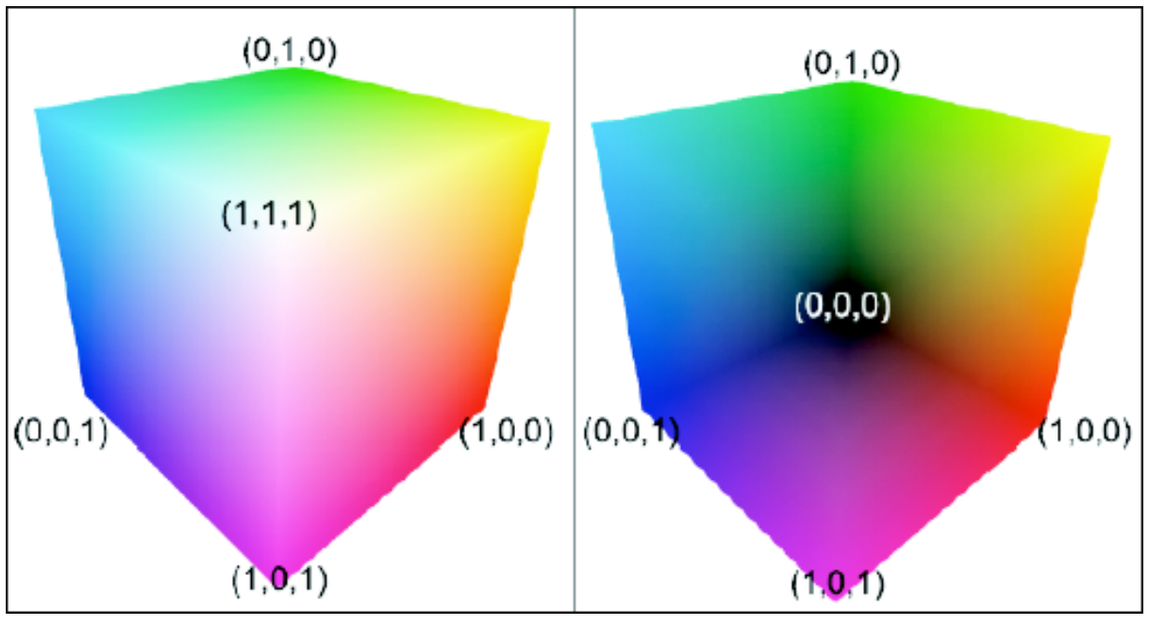
- Strahlen sind voneinander unabhängig
  ➡ Verfahren gut parallelisierbar

GPU-based Raycasting

- GPUs unterstützen Volumengrafik nicht direkt
- Ansatz: Krüger/Westermann (2003)
  - Datensatz als 3D-Textur (lineare Filterung)
  - Strahl-Traversierung im *Fragment Shader*
  - Hohe Frameraten, gute Bildqualität

GPU-based Raycasting

- Strahlen werden mittels einer *Proxy-Geometrie* spezifiziert, nicht analytisch
- Erzeugung durch Rendern eines Würfels



Raycasting mit CUDA

- Motivation:
  - Shader-Implementierung ist Umweg
    - Mehr Kontrolle über Ausführung gewünscht
    - Zusätzliche Hardware-Features nutzen
    - Bessere Dokumentation

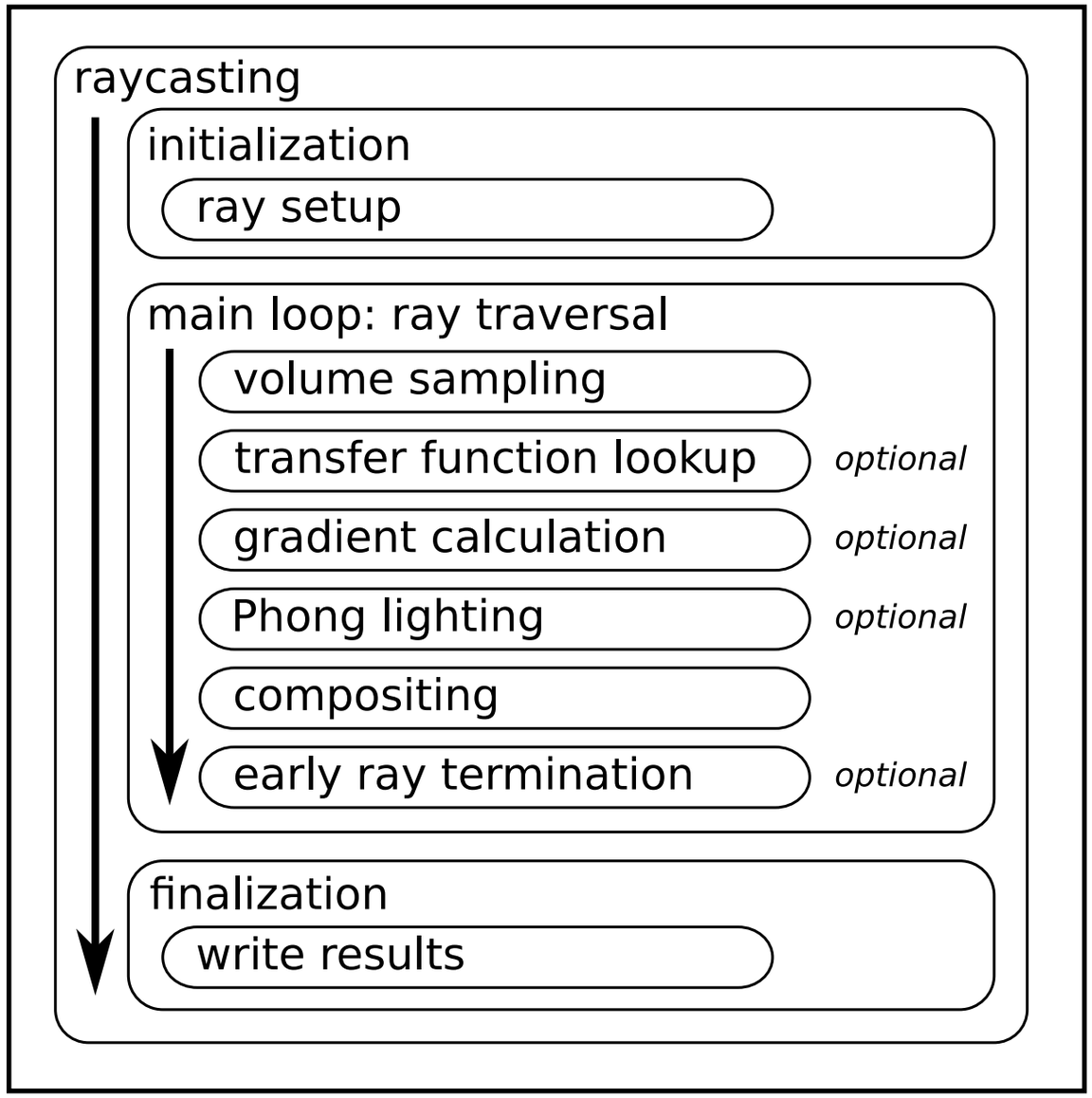
Raycasting mit CUDA

- 1:1-Umsetzung der Shader-Implementierung:
  - Thread ➡ Strahl
  - Block ➡ Bildschirmregion
  - Strahlparameter mit OpenGL erzeugen
  - Volumen-Datensatz als *CUDA-Textur*

CUDA-Texturen

- 1D/2D/3D
- lineare Interpolation (kostenlos!)
- Randbehandlung (clamp/wrap)
- Caching (optimiert für 2D-Lokalität)
- keine *Coalescing-Regeln* zu beachten

CUDA-Implementierung



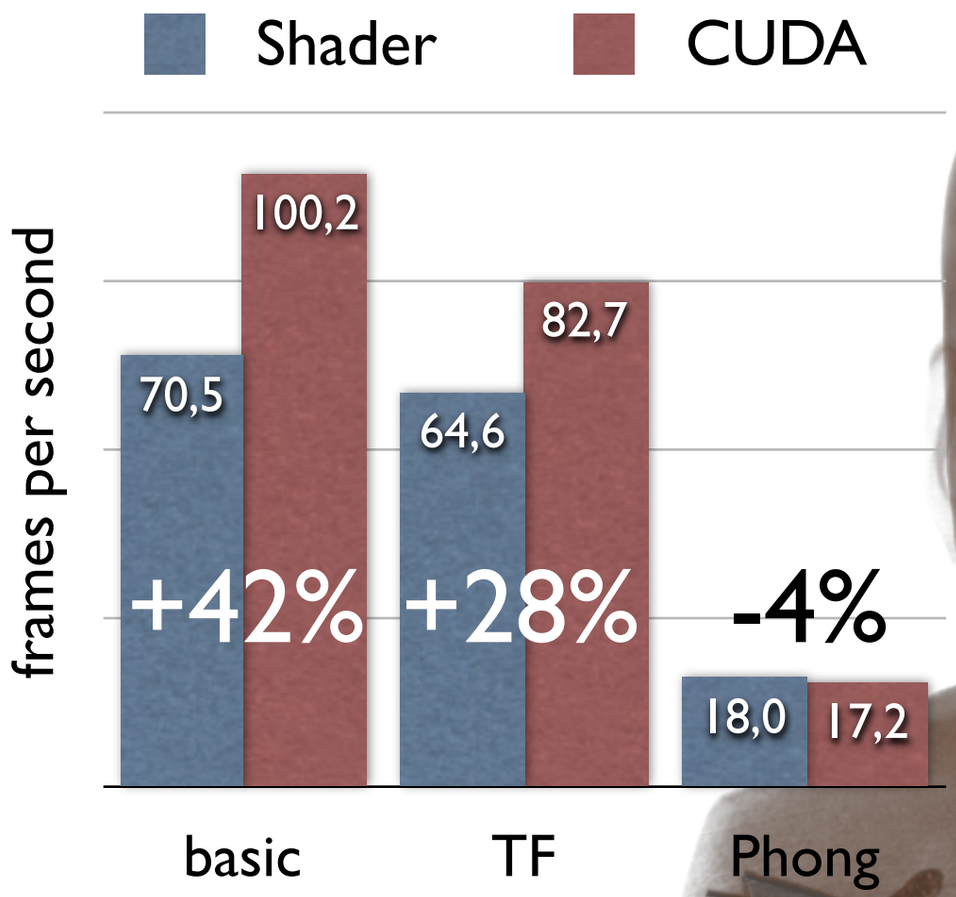
CUDA-Implementierung

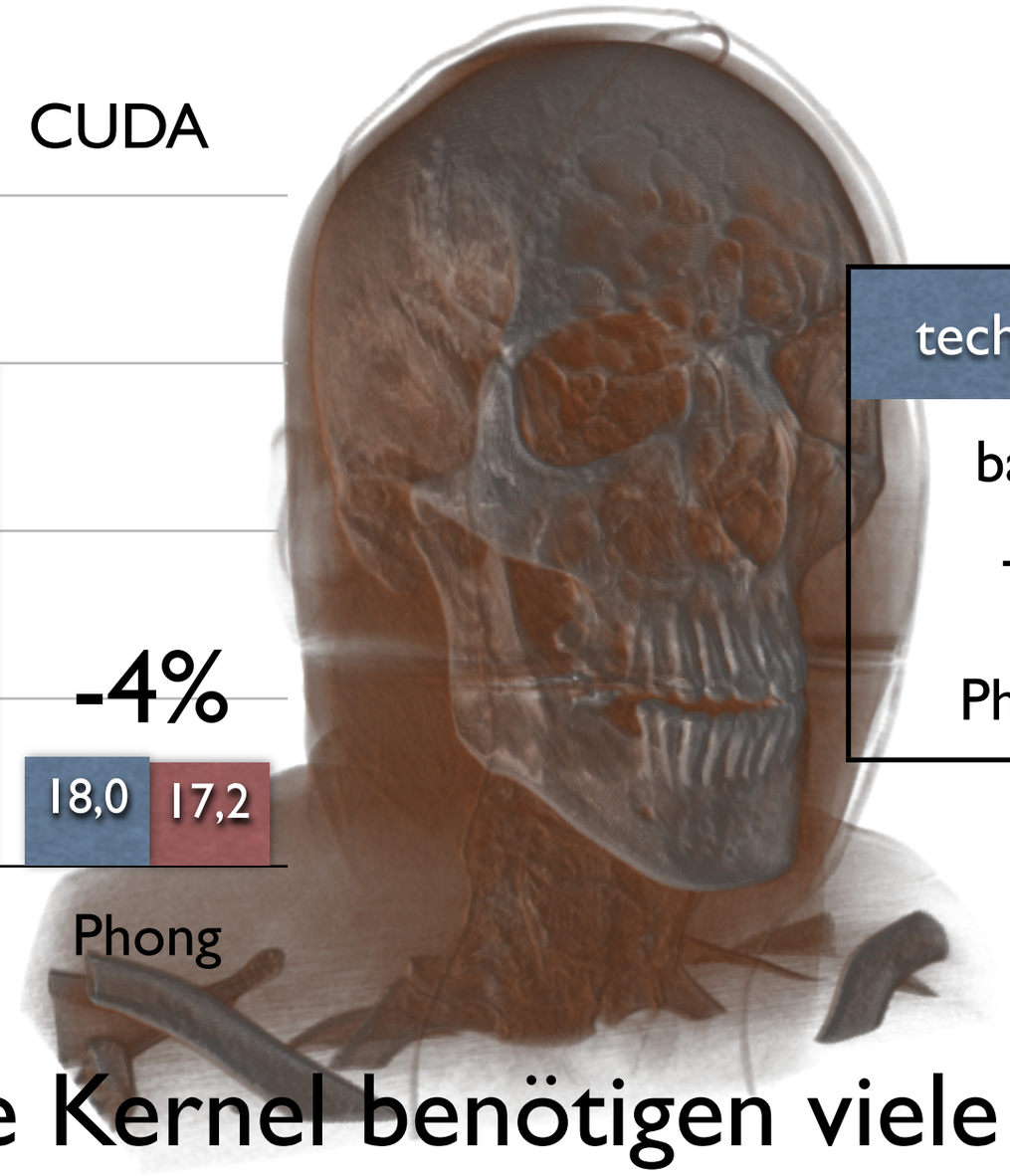
```
texture<ushort, 3, cudaReadModeNormalizedFloat> volumeTex;

void cuda_raycast_bindVolumeArray(cudaArray* array) {
    volumeTex.normalized = true;
    volumeTex.filterMode = cudaFilterModeLinear;

    volumeTex.addressMode[0] = cudaAddressModeClamp;
    volumeTex.addressMode[1] = cudaAddressModeClamp;
    volumeTex.addressMode[2] = cudaAddressModeClamp;

    channelDescVolume = cudaCreateChannelDesc<ushort>();
    cudaBindTextureToArray(volumeTex, array, channelDescVolume);
}
```

```
__global__
void simpleRaycast(float4* entryParams, float4* exitParams, float4* output,
                   uint width, uint height,
                   float qualityFactor, float3 cameraPos, float3 lightPos)
{
    uint x = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
    uint y = __umul24(blockIdx.y, blockDim.y) + threadIdx.y;

    if (x >= width || y >= height)
        return;

uint index = (__umul24(y, width) + x);

// enforce reading 4 floats although only 3 are accessed to get coalescing
volatile float4 first4 = entryParams[index];
volatile float4 last4 = exitParams[index];
float3 first = { first4.x, first4.y, first4.z };
float3 last = { last4.x, last4.y, last4.z };
```

```
while (t <= tend) {
  float3 sample = first + t * direction;
  float intensity = tex3D(volumeTex, sample.x, sample.y, sample.z);

  float3 gradient = calcGradient(sample);
  float4 color = tex1D(transferFuncTex, intensity);
  float3 shadedColor = phong(sample, color, gradient, lightPos, cameraPos);
  color.x = shadedColor.x;
  color.y = shadedColor.y;
  color.z = shadedColor.z;

  t += stepIncr;

  // perform compositing
  color.w *= qualityFactor;
  result.x = result.x + (1.0f - result.w) * color.w * color.x;
  result.y = result.y + (1.0f - result.w) * color.w * color.y;
  result.z = result.z + (1.0f - result.w) * color.w * color.z;
  result.w = result.w + (1.0f - result.w) * color.w;
}

// write output color
output[index] = result;
```

Vergleich Shader/CUDA

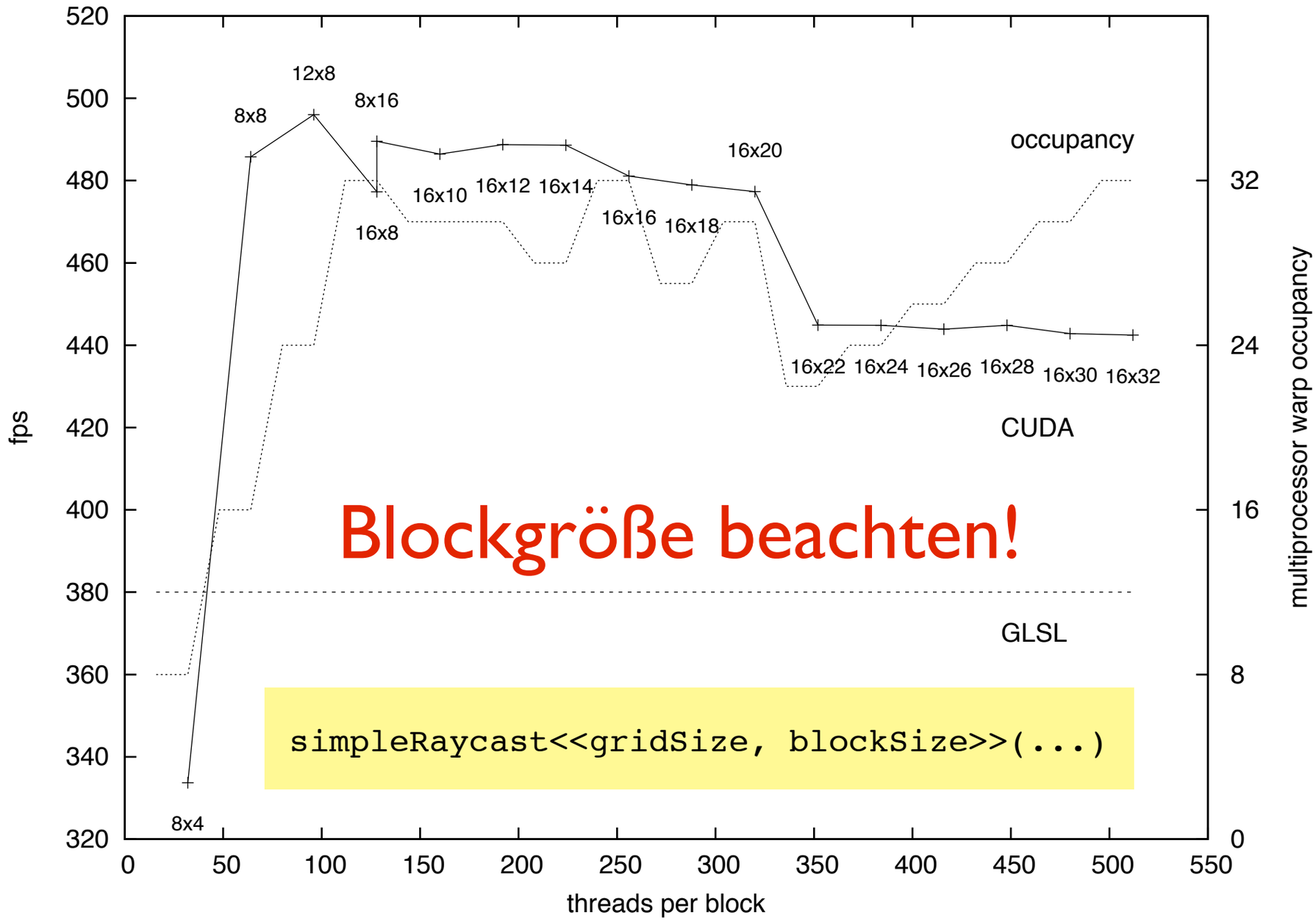




technique	fetches	registers
basic	1	15
TF	2	19
Phong	8	33

- komplexe Kernel benötigen viele Register, damit weniger Threads gleichzeitig möglich

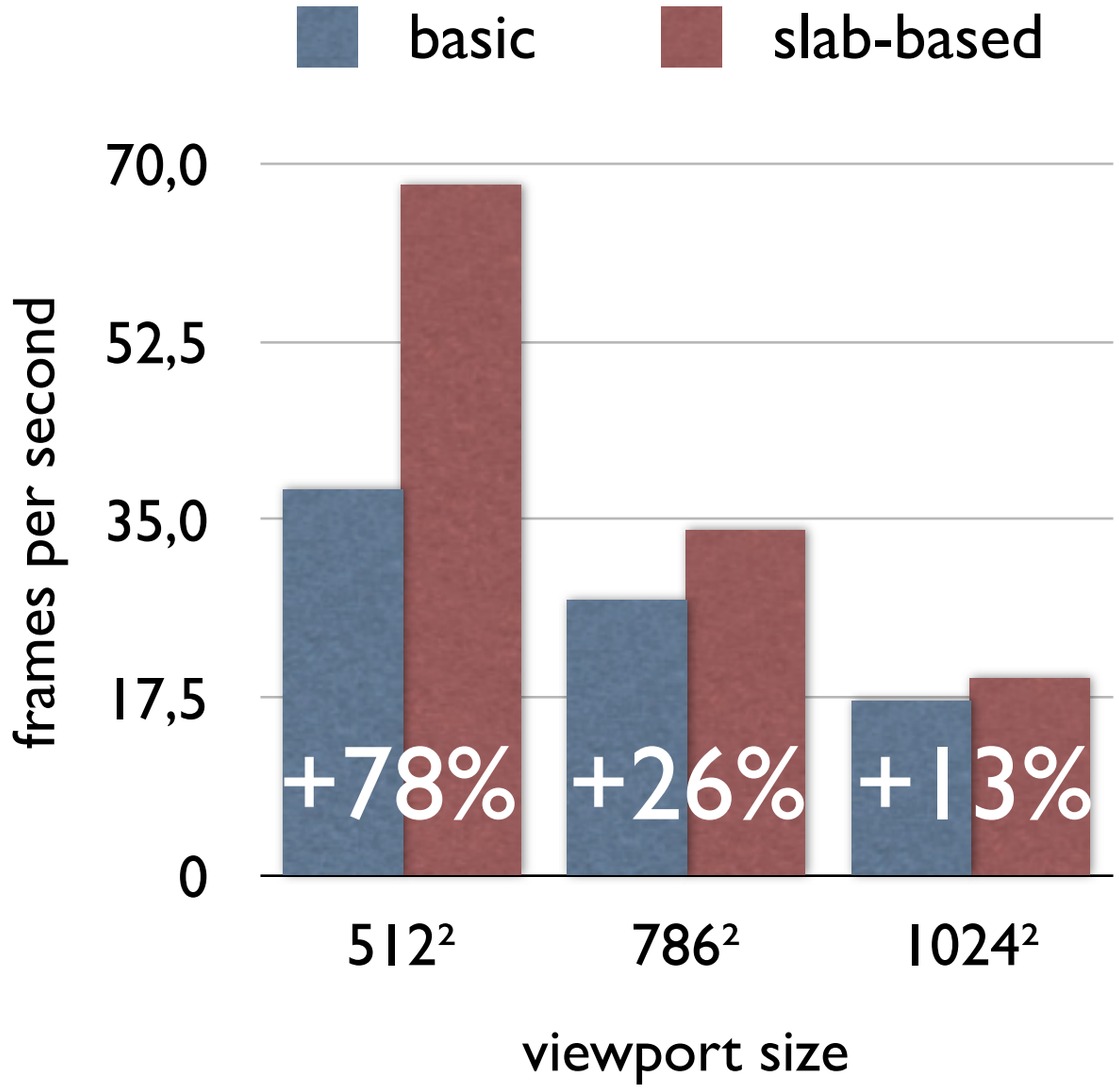
Vergleich Shader/CUDA



Raycasting beschleunigen

- Problem: auf Voxel wird mehrfach zugegriffen (Gradienten-Berechnung)
- Idee: Datensatz in *slabs* aufteilen und im Shared Memory zwischenspeichern
- nur 16 kb Shared Memory: max. 16^3 Voxel
- Overhead: Randbehandlung, Preprocessing, Koordinatensystemwechsel

Slab-based Raycasting



Optimierungs-Hinweise

- Raycasting ist Sonderfall: einfache Berechnungen, viele Speicherzugriffe
- "if" ist teuer: Branch-Coherence beachten, Präprozessor benutzen
- Kernel klein halten, große Kernel aufteilen
- Blockgrößen beachten

Fazit

- Auch vorhandene GPGPU-Verfahren können von CUDA profitieren
- Grafikhardware ist auch für nicht-Grafikanwendungen nützlich
- Aber: Verfahren müssen an die CUDA-Architektur angepasst werden