

# Untersuchung der Parallelisierung des Feldkamp-Davis-Kress-Algorithmus mittels CUDA

Kolloquium zum Großen Beleg

Jan Stephan

Betreuer: Prof. Dr. Wolfgang E. Nagel

Betreuer: Dr.-Ing. André Bieberle

Dr.-Ing. Guido Juckeland

Matthias Werner

# Gliederung

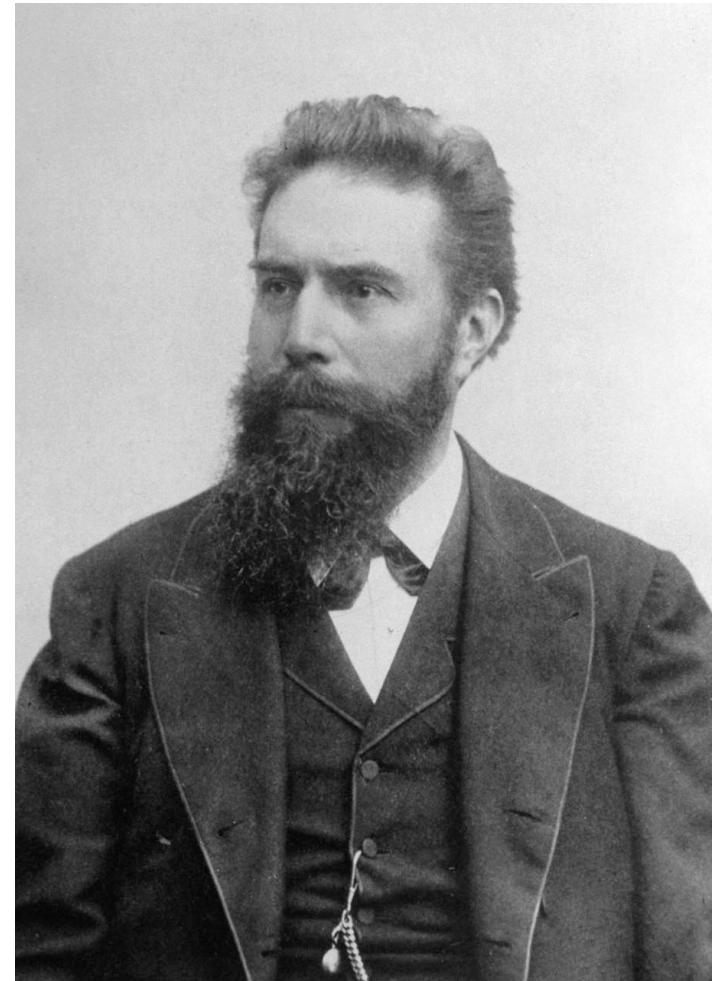
---

- Motivation
- Grundlagen der gefilterten Rückprojektion
- Implementierung
- Analyse
- Ausblick

# Motivation

---

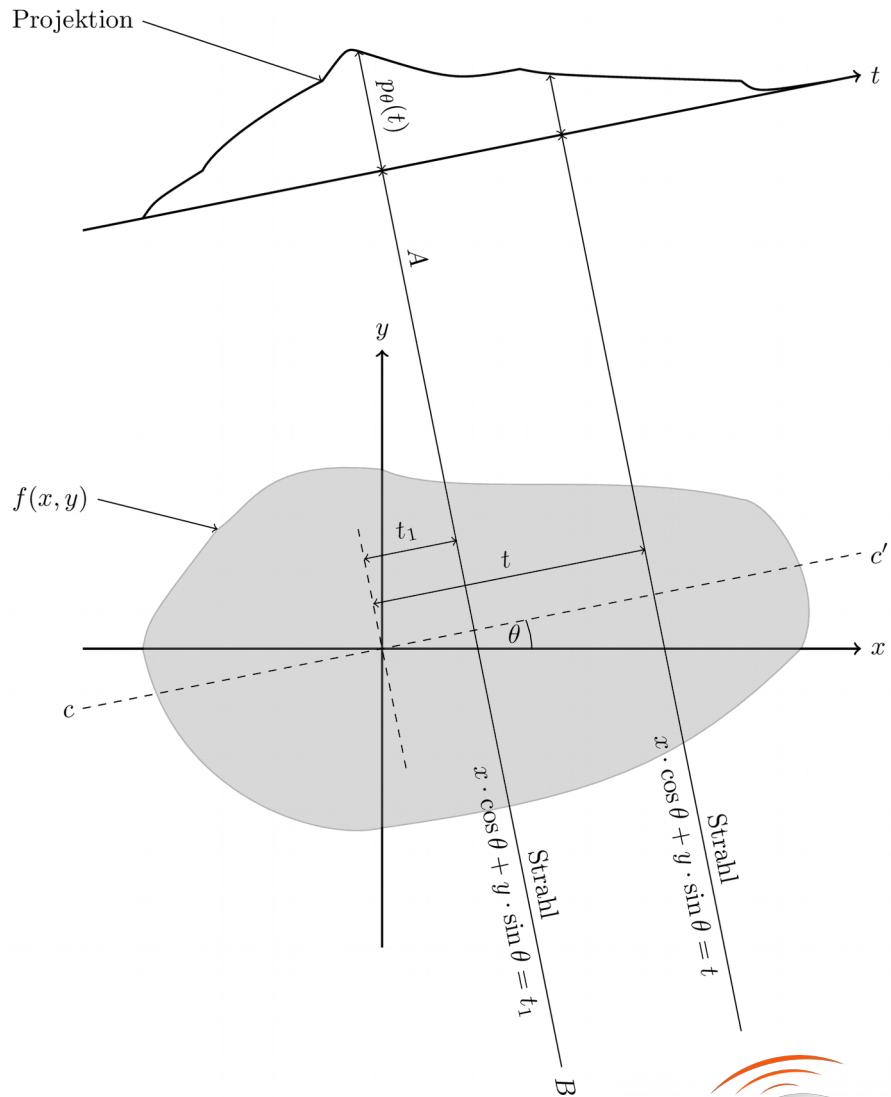
- Computertomographie: wiederholtes Anwenden des Röntgenverfahrens unter verschiedenen Winkeln
- Bedeutende nicht-invasive Analysemöglichkeit in Medizin und Materialforschung
- Kegelstrahl-CT: FDK-Algorithmus
- Große Datenmengen  
→ hoher Rechenaufwand
- GPU-Einsatz zur schnellen Verarbeitung



Wilhelm Conrad Röntgen

# Grundlagen der gefilterten Rückprojektion

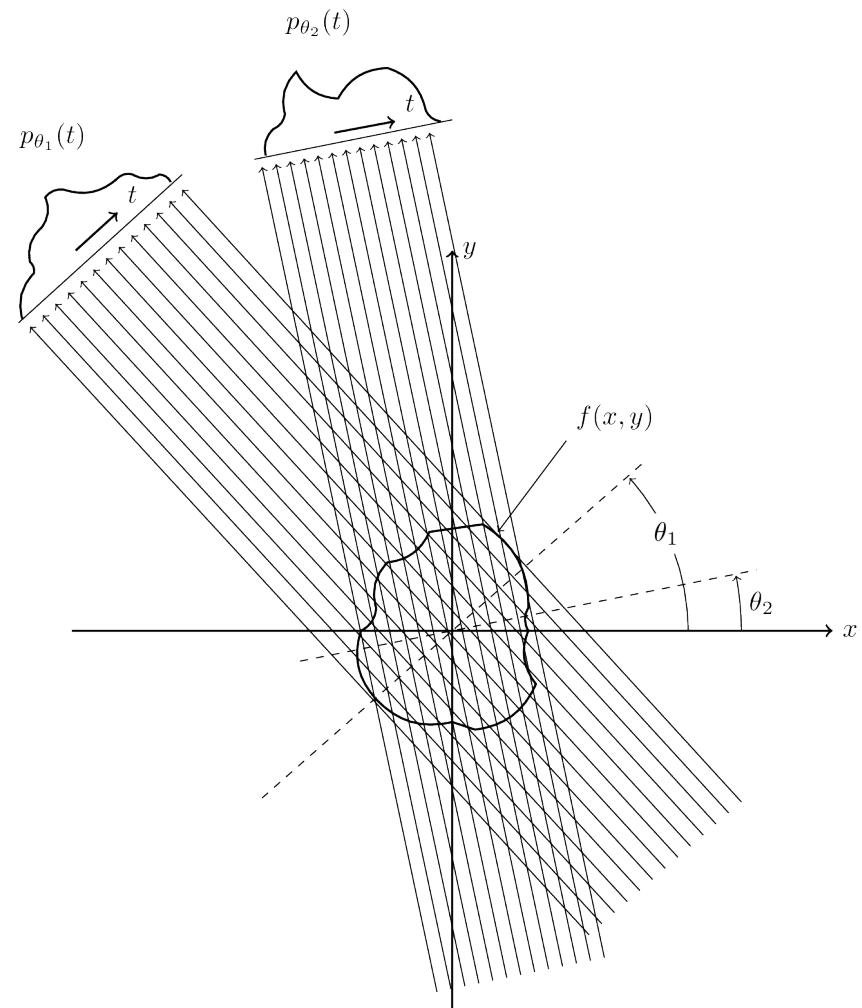
- Objekt  $f(x, y)$ : Verteilung von Abschwächungskoeffizienten
- Strahlabschwächung kann als Kurvenintegral betrachtet werden
- Umformung des Kurvenintegrals ergibt Radontransformation  $p_\theta(t)$
- Radontransformiertes Objekt: Menge aller **Projektionen**



Vorlage: [1]

# Grundlagen der gefilterten Rückprojektion

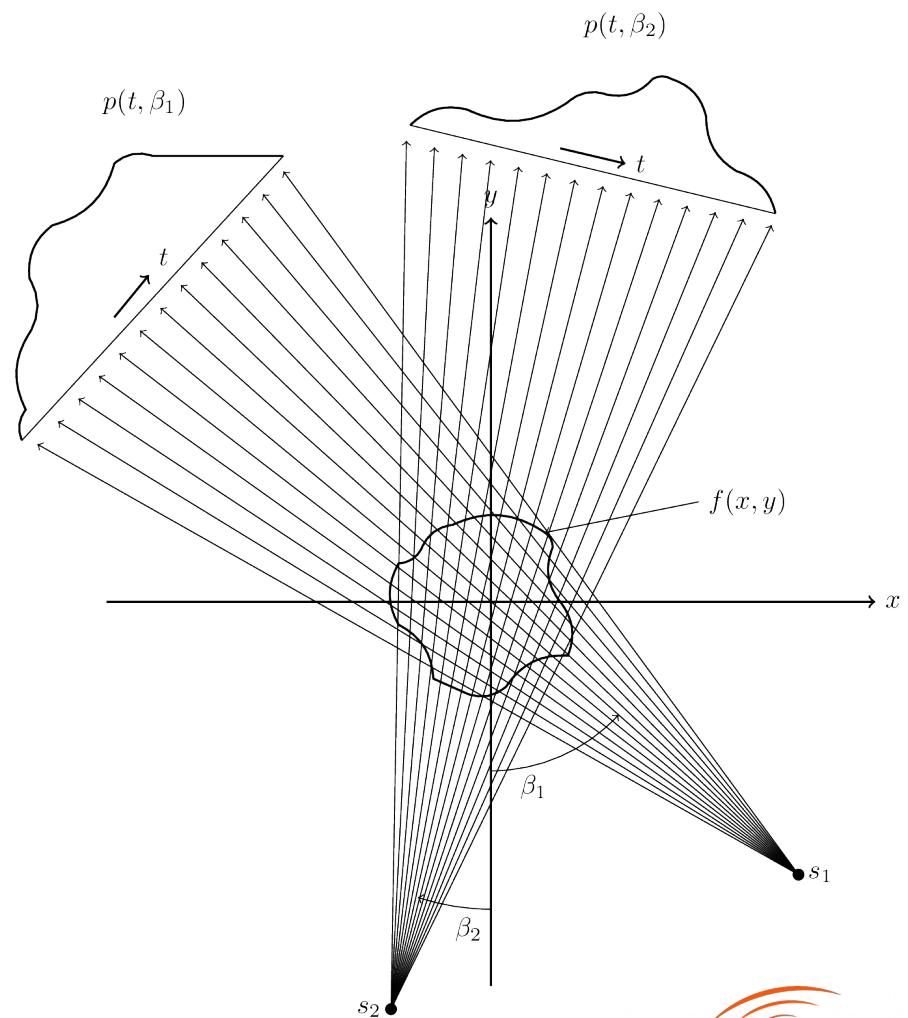
- Parallelstrahlprojektion
- Nachteil: langsam, da zusätzliche Bewegung der Quelle



# Grundlagen der gefilterten Rückprojektion



Fächerstrahlprojektion



Vorlage: [2]

# Grundlagen der gefilterten Rückprojektion

## Gefilterte Rückprojektion

- Filterung: Wichtung der Projektionen im Frequenzraum
- Rückprojektion:
  - Für jede Projektion  $p$  unter dem Winkel  $\theta$ :
    - Für jeden Punkt  $(x, y)$  im Objekt:
      - Projiziere  $(x, y)$  auf den Detektorpunkt  $t$
      - Addiere den Detektorwert am Punkt  $t$  zum vorherigen Wert des Punkts  $(x, y)$  (**Rückprojektion**)
  - Keine Abhängigkeiten zwischen einzelnen Punkten
    - *embarrassingly parallel*

# Grundlagen der gefilterten Rückprojektion

## Algorithmus der gefilterten Rückprojektion

Für alle Winkel  $\theta$ :

- 1) Projektion  $p_\theta(t)$  aufnehmen
- 2) Projektion  $p_\theta(t)$  zu  $P_\theta(w)$  fouriertransformieren
- 3) Transformierte Projektion  $P_\theta(w)$  filtern
- 4) Transformierte und gefilterte Projektion  $P_\theta(w)$  invers fouriertransformieren und zurückprojizieren

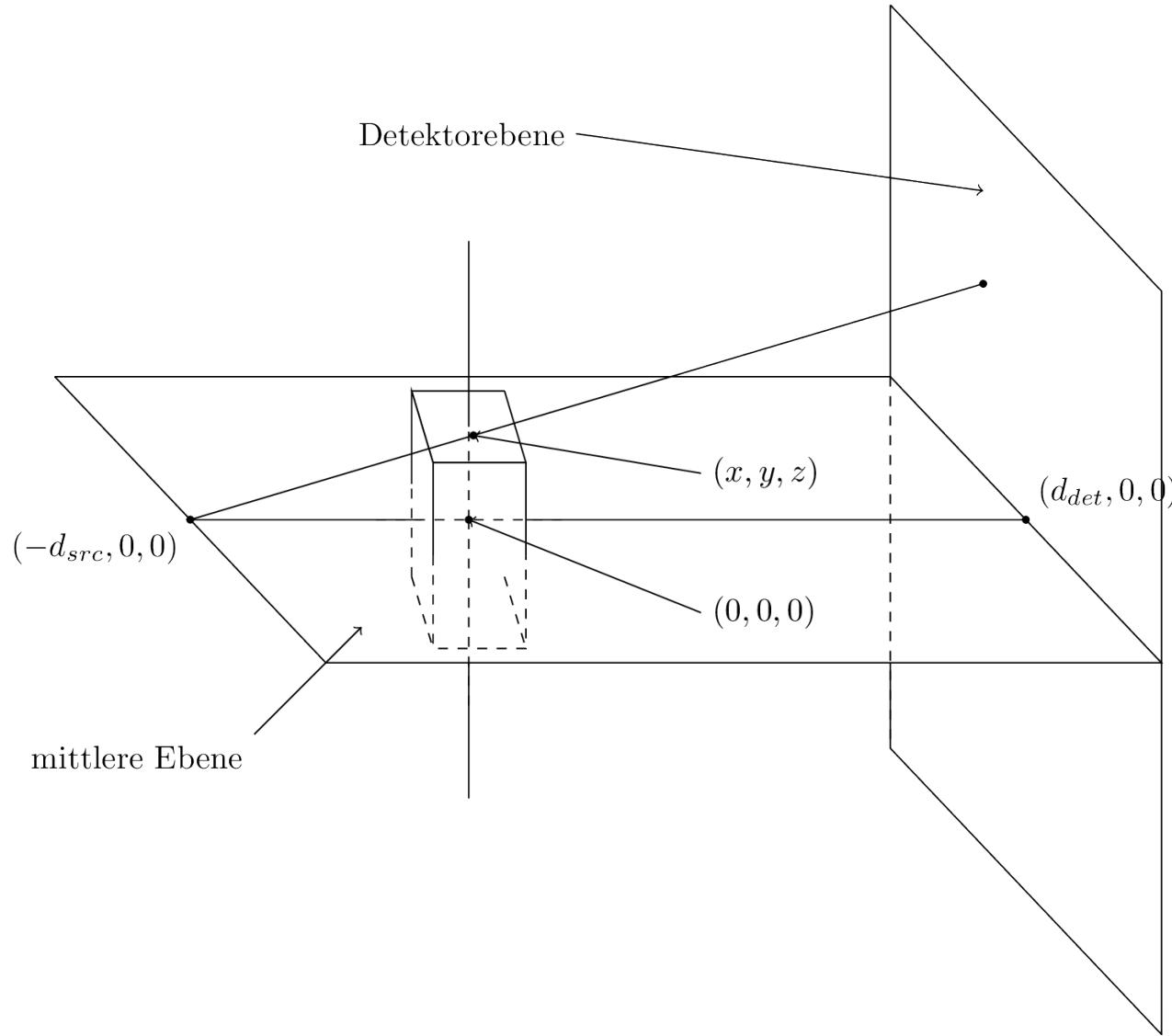
# Grundlagen der gefilterten Rückprojektion

## Feldkamp-Davis-Kress-Algorithmus

- █ bisher nur 2D-Projektionen betrachtet
- █ 3D als „Stapel“ von 2D-Rekonstruktionen → langsam
- █ Feldkamp, Davis, Kress: „echter“ 3D-Rückprojektionsalgorithmus auf Basis der Fächerstrahlprojektion [4]

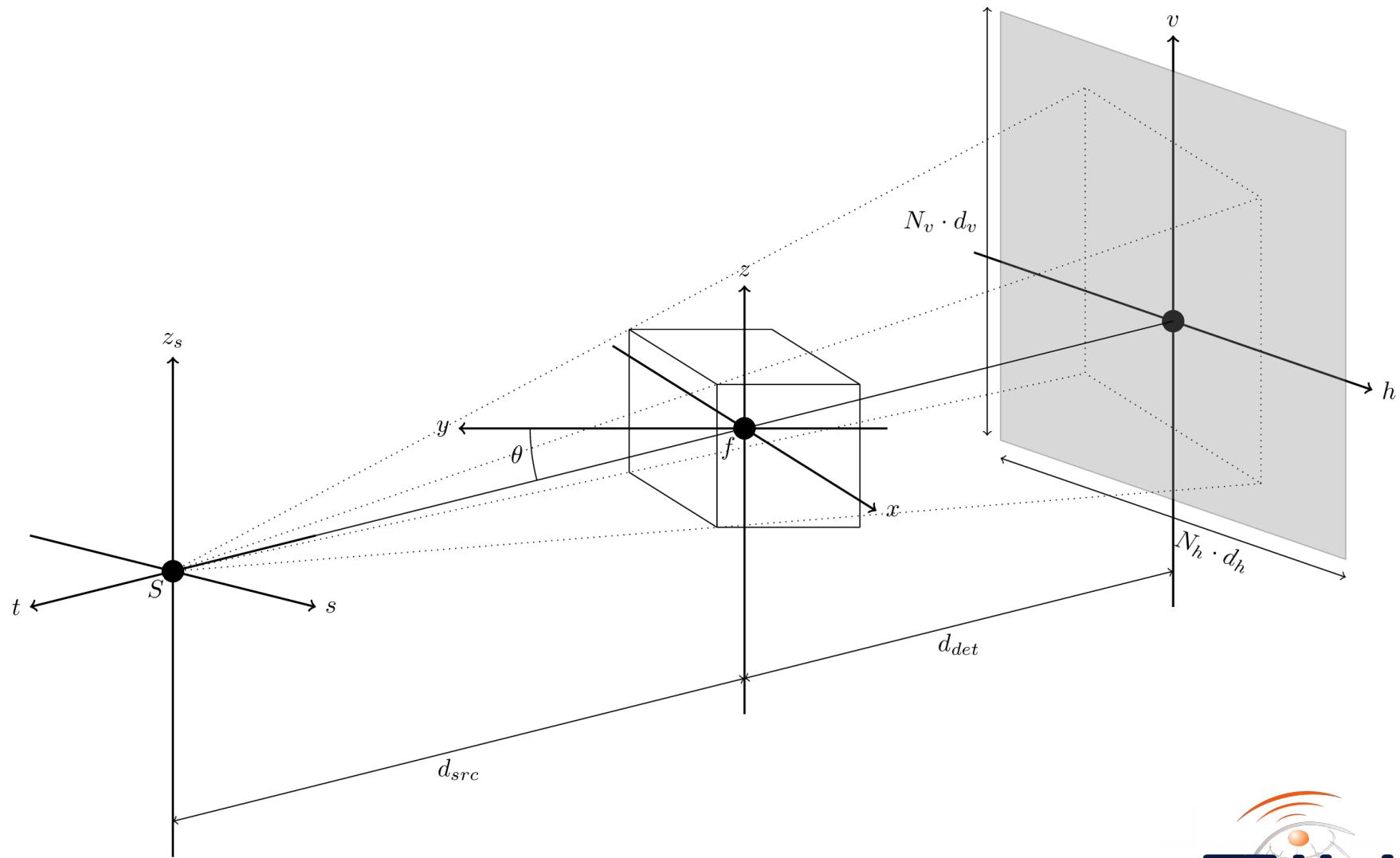
# Grundlagen der gefilterten Rückprojektion

---



Vorlage: [4]

# Grundlagen der gefilterten Rückprojektion



# Grundlagen der gefilterten Rückprojektion

## Wichtung

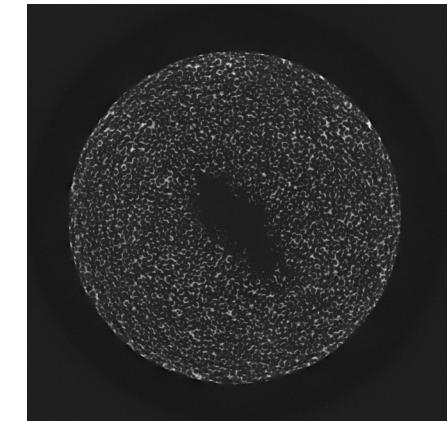
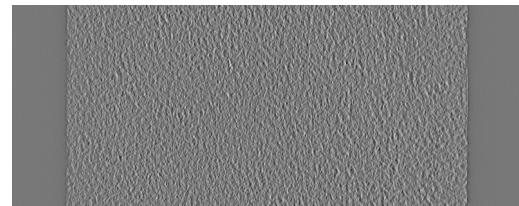
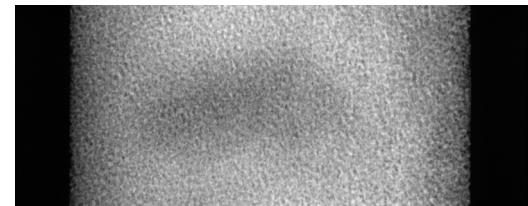
- █ Kegelstrahltomographie: Strahlen haben unterschiedlich lange Wege  
→ Wichtung der Projektion erforderlich
- █ Pixelweise Wichtung mit Wichtungsfaktor  $w_{ij}$ :

$$w_{ij} = \frac{d_{det} - d_{src}}{\sqrt{(d_{det} - d_{src})^2 + h_j^2 + v_i^2}}$$

# Grundlagen der gefilterten Rückprojektion

---

## Algorithmus



## Ziele

- Lösung des Algorithmus in sinnvoller Zeit
- Hohe Auslastung der GPU
- Geringe Wartezeiten zwischen Kernel-Aufrufen
- Effiziente Speicherzugriffe
- Bonus: auf mehreren GPUs lauffähig

## GPU-Auslastung

- Vorbedingung: wenige geteilte Ressourcen
  - Minimale Registerzahl pro Thread
  - Wenig Shared Memory pro Block

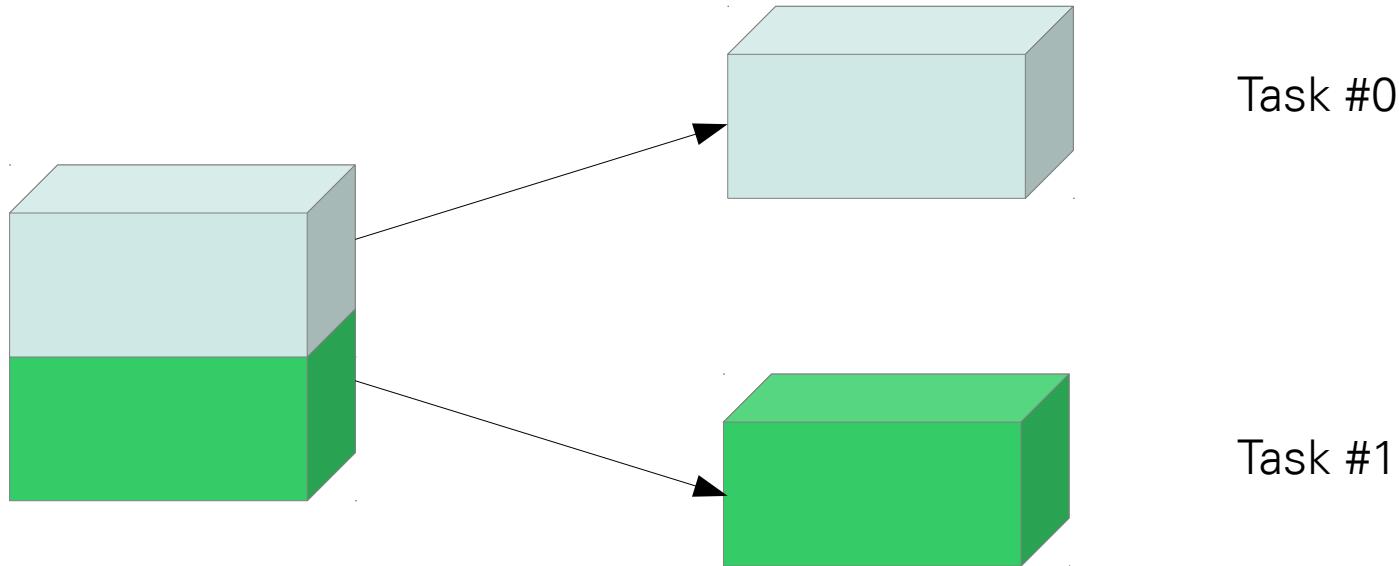
## Speicherzugriffe

- *Pitched memory* für lesenden und schreibenden Zugriff
- *Texture memory* für lesenden Zugriff
- *Constant memory* für Konstanten

# Implementierung

---

Volumenverteilung (mehrere GPUs / zu wenig GPU-Speicher)



## Geringe Wartezeiten

- Idealerweise keine Operationen außer der Rückprojektion  
→ Ausführung in eigenem Thread

# Implementierung

---

```
auto fdk(task_queue& queue, int device) → void {
    cudaSetDevice(device);

    while(!queue.empty()) {
        auto t = queue.pop(); // hole Teilvolumen-Informationen
        auto s = source(...); // konfiguriere Laden der Projektionen
        auto v = make_volume(...); // erzeuge Teilvolumen auf Device

        while(!source.drained()) {
            auto p = source.load_next(); // lade nächste Projektion
            auto d_p = copy_h2d(p); // kopiere Projektion auf Device
            weight(d_p, ...); // wiechte Projektion
            filter(d_p, ...); // filtere Projektion

            // Rückprojektion. Startet intern einen weiteren Thread
            backproject(d_p, ...);
        }
    }
}
```

## Wichtung

$$w_{ij} = \frac{d_{det} - d_{src}}{\sqrt{(d_{det} - d_{src})^2 + h_j^2 + v_i^2}}$$

- Messwertunabhängig
- Nur von festen geometrischen Parametern abhängig  
→ einmalige Generierung

## Filterung

- Verwendung von cuFFT sowie einfacher Multiplikationen

## Rückprojektion - Register

- Geringe Registerzahl für Rückprojektion erforderlich  
→ konstante Parameter im konstanten Speicher

```
__global__ void backprojection_kernel(  
    float* vol,  
    std::size_t vol_pitch,  
    cudaTextureObject_t proj,  
    float theta_sin,  
    float theta_cos)  
{  
    ...  
}
```

## Rückprojektion – *Shared memory*

- Unabhängige Voxel
  - kein Datenaustausch zwischen Threads nötig
- Eine Lese- und Schreiboperation pro Voxel
  - kein Zeitgewinn durch Einsatz des *Shared Memory*

## Rückprojektion – Wartezeiten

- Ausführung des Kernels in eigenem Thread und Stream
- Projektions-Queue: FIFO-Prinzip

## Rückprojektion – Speicherzugriffe I

- Projektionen: *Texture memory*
- Volumen: *Pitched memory*
- Konstanten: *Constant memory*

```
auto k = blockIdx.x * blockDim.x + threadIdx.x;
auto l = blockIdx.y * blockDim.y + threadIdx.y;
auto m = blockIdx.z * blockDim.z + threadIdx.z;

if((k < consts.N_x) && (l < consts.N_y) && (m < consts.N_z)) {
    auto slice_pitch = vol_pitch * consts.N_y;
    auto slice = reinterpret_cast<char*>(vol) + m * slice_pitch;
    auto row = reinterpret_cast<float*>(slice + l * vol_pitch);
    ...
}
```

## Rückprojektion – Speicherzugriffe II

- Zugriff auf *Global memory* mit hoher Latenz verbunden  
→ maskieren

```
auto old_val = row[k];
```

## Rückprojektion – Berechnung I

- Umwandlung der Array-Koordinaten in Volumenkoordinaten

```
auto x = vol_centered_coordinate(k, consts.N_x, consts.d_x);
// äquivalent für y und z
```

- Koordinaten rotieren

```
auto s = x * theta_cos + y * theta_sin;
auto t = -x * theta_sin + y * theta_cos;
```

- Rotierte Koordinaten auf Detektor projizieren

```
auto factor = consts.d_sd / (s + consts.d_so);
auto h = proj_pixel_coordinate(t * factor, consts.N_h, consts.d_h);
auto v = proj_pixel_coordinate(z * factor, consts.N_v, consts.d_v);
```

---

## Rückprojektion – Berechnung II

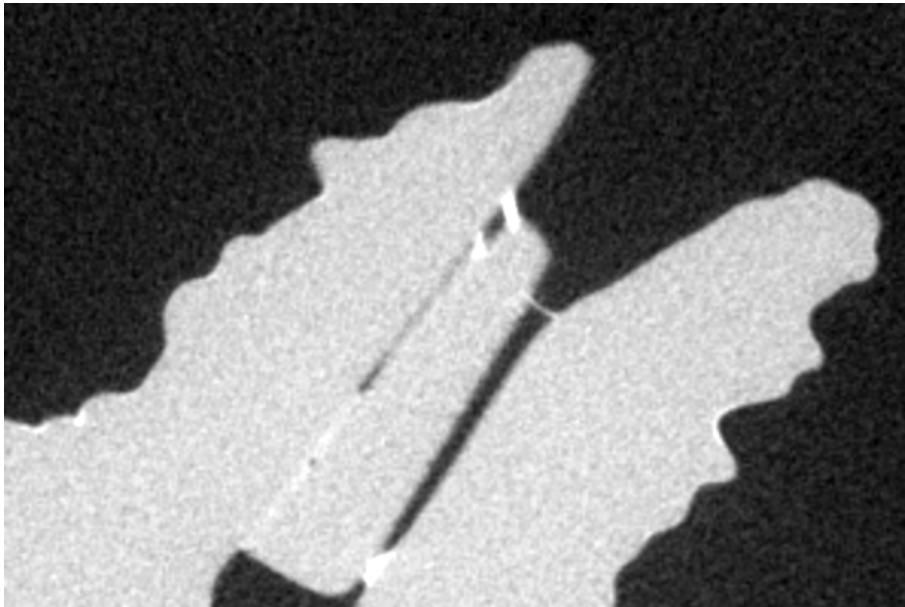
- Detektorsignal interpolieren

```
auto det = tex2D<float>(proj, h, v);
```

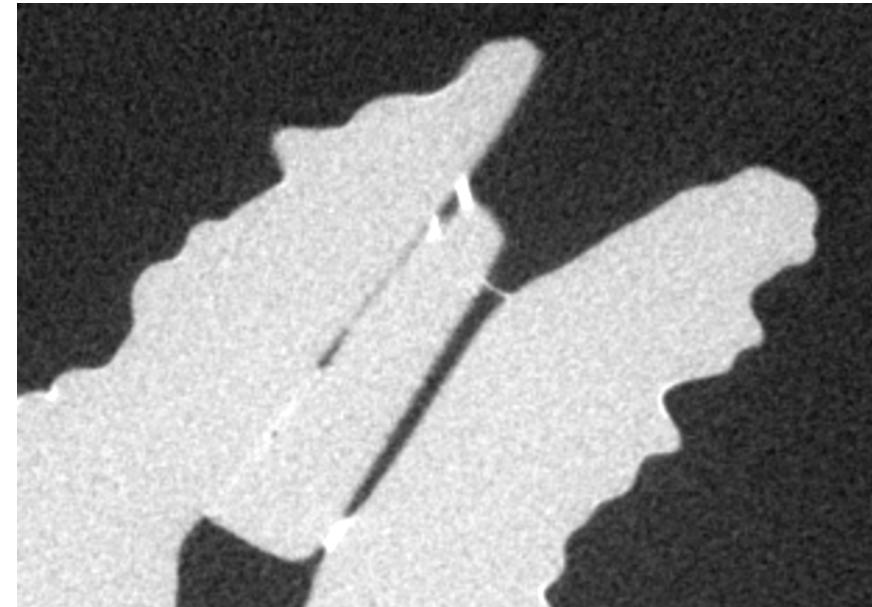
- Rückprojektion

```
auto u = -(consts.d_so / (s + consts.d_so));  
row[k] = old_val + 0.5f * det * u * u;
```

## Validierung



CPU



GPU

Vergrößerung 200%

## Messmethodik

	GTX 1080 (Standard)	Tesla K20c	Tesla K80
Taktfrequenz	1733 MHz	706 MHz	560 MHz
Speicher	8 GiB	5 GiB	12 GiB

- Volumengröße: 1070 x 1070 x 1033 Voxel (32bit float)
- fünffache Ausführung, I/O inklusive

## Berechnungsdauer

Operation	Dauer [s]	Anteil [%]
Gesamtberechnung	80,356	100
Rückprojektion	79,622	99,087
Filterung	0,631	0,786
Wichtung	0,075	0,093
memset	0,028	0,034

GPU: GTX 1080, Volumen: 1070 x 1070 x 1033, ohne I/O

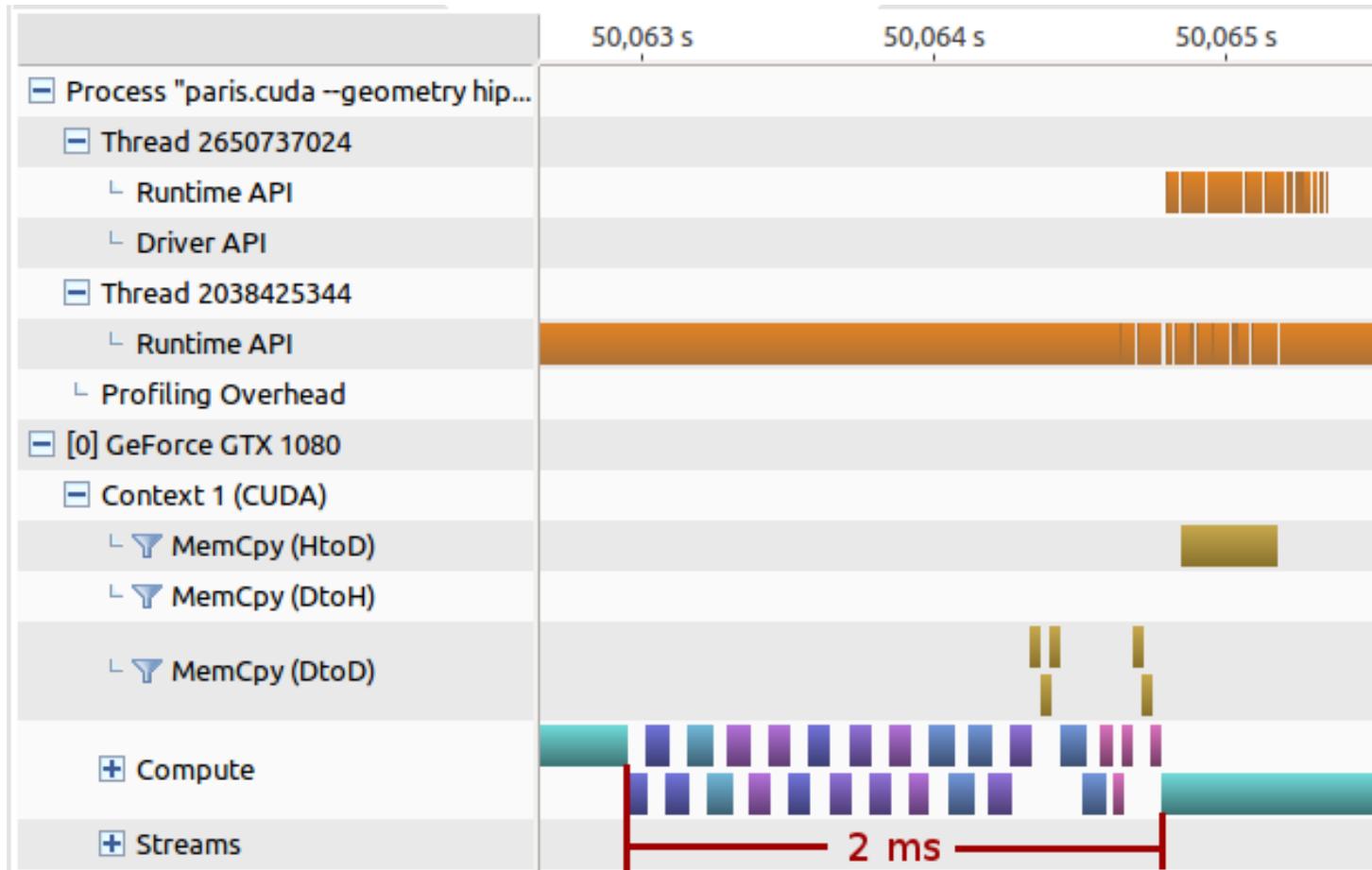
## Eigenschaften des Rückprojektionskernels

```
void paris::cuda::_GLOBAL__N__49_ttmpxft_00003e0d_00000000_7_backprojection_cpp1_ii_039ddfd4::backprojection_kernel<...
```

Start	14,86 s (14.859.887.517 ns)
End	14,916 s (14.915.728.262 ns)
Duration	55,841 ms (55.840.745 ns)
Stream	Stream 13
Grid Size	[ 67,134,517 ]
Block Size	[ 16,8,2 ]
Registers/Thread	19
Shared Memory/Block	0 B
▼ Occupancy	
Achieved	88,5%
Theoretical	100%
▼ Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B

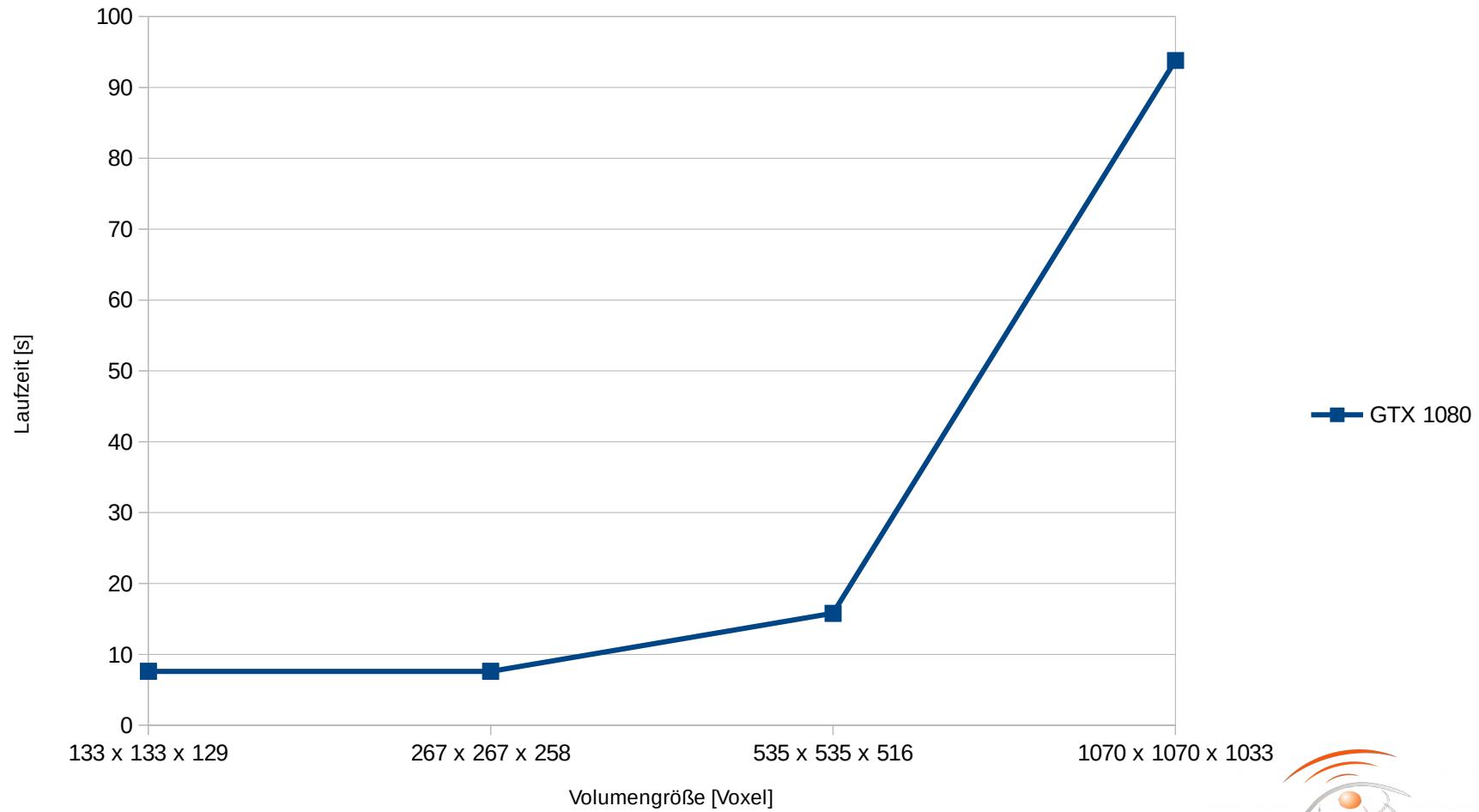
GPU: GTX 1080, Volumen: 1070 x 1070 x 1033

## Wartezeit

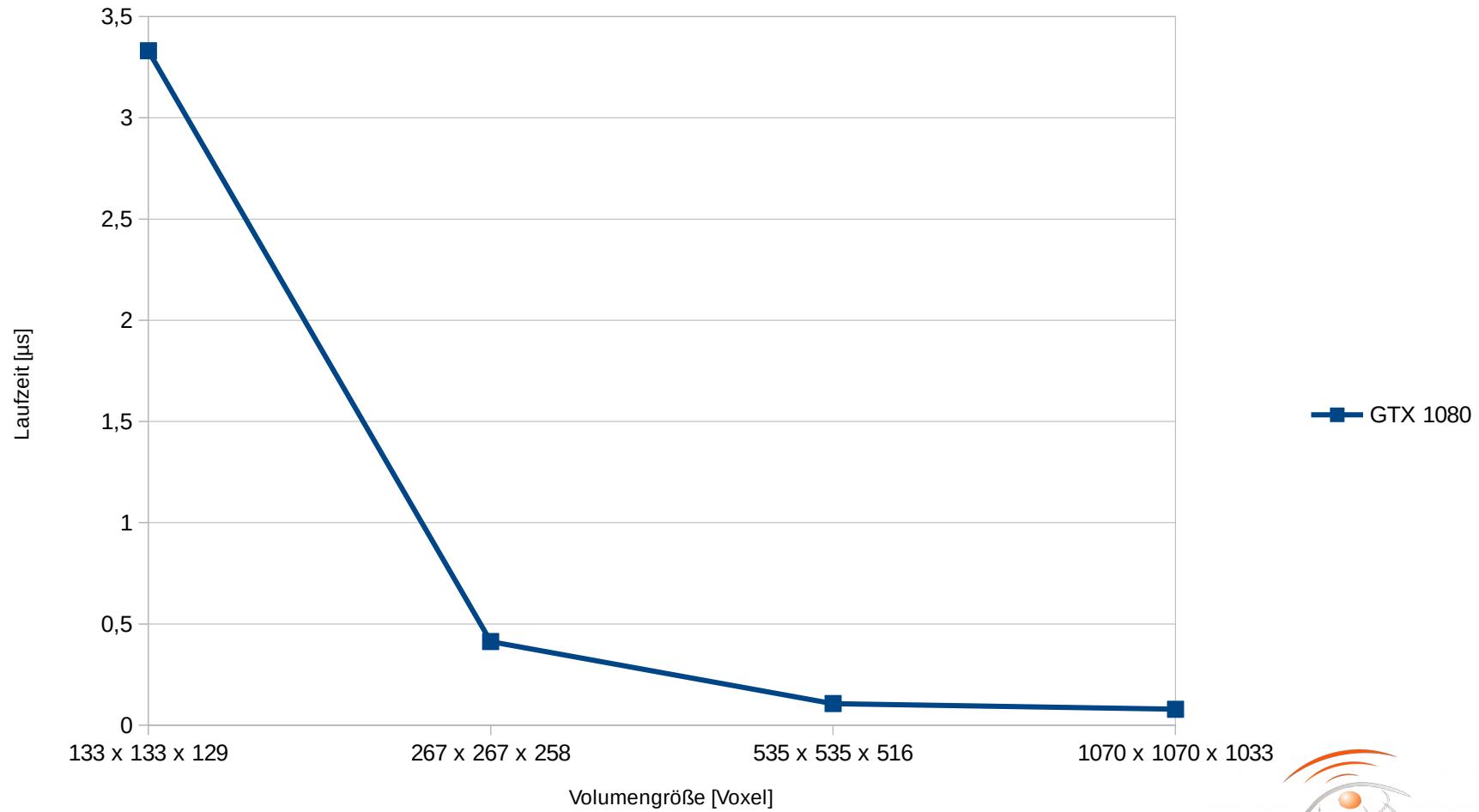


GPU: GTX 1080, Volumen: 1070 x 1070 x 1033

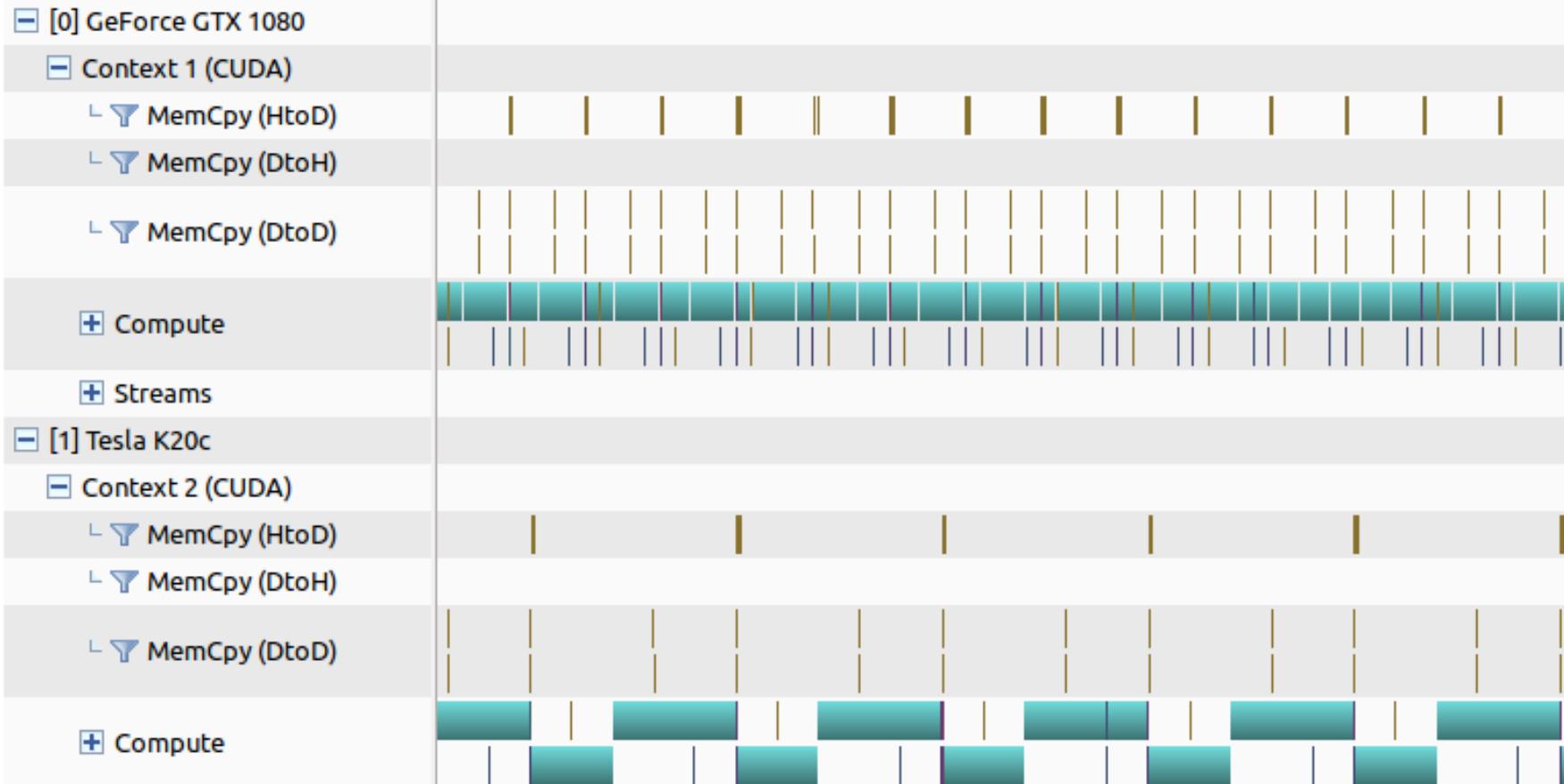
## Laufzeitverhalten – unterschiedliche Volumengrößen



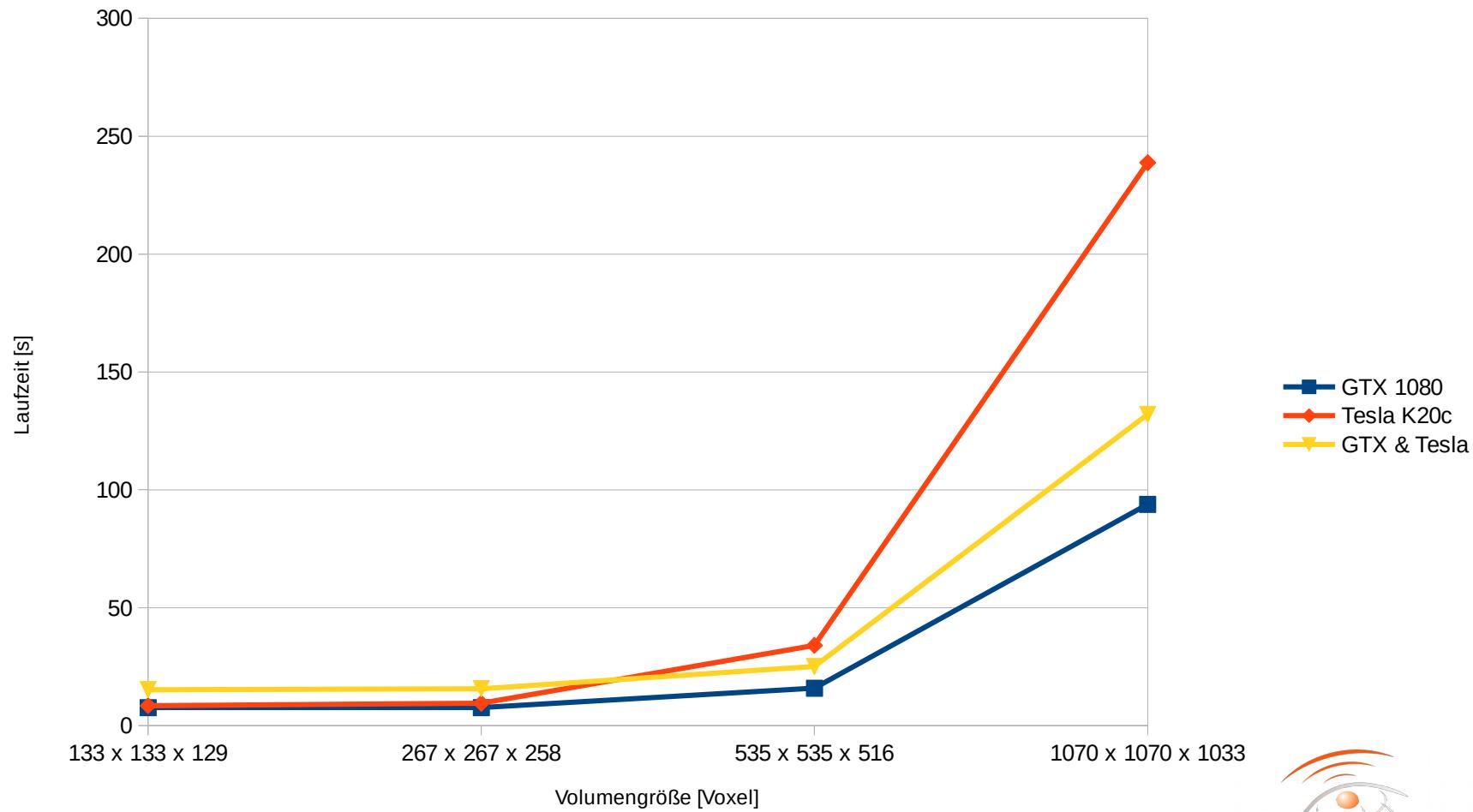
## Laufzeitverhalten – Zeit pro Voxel



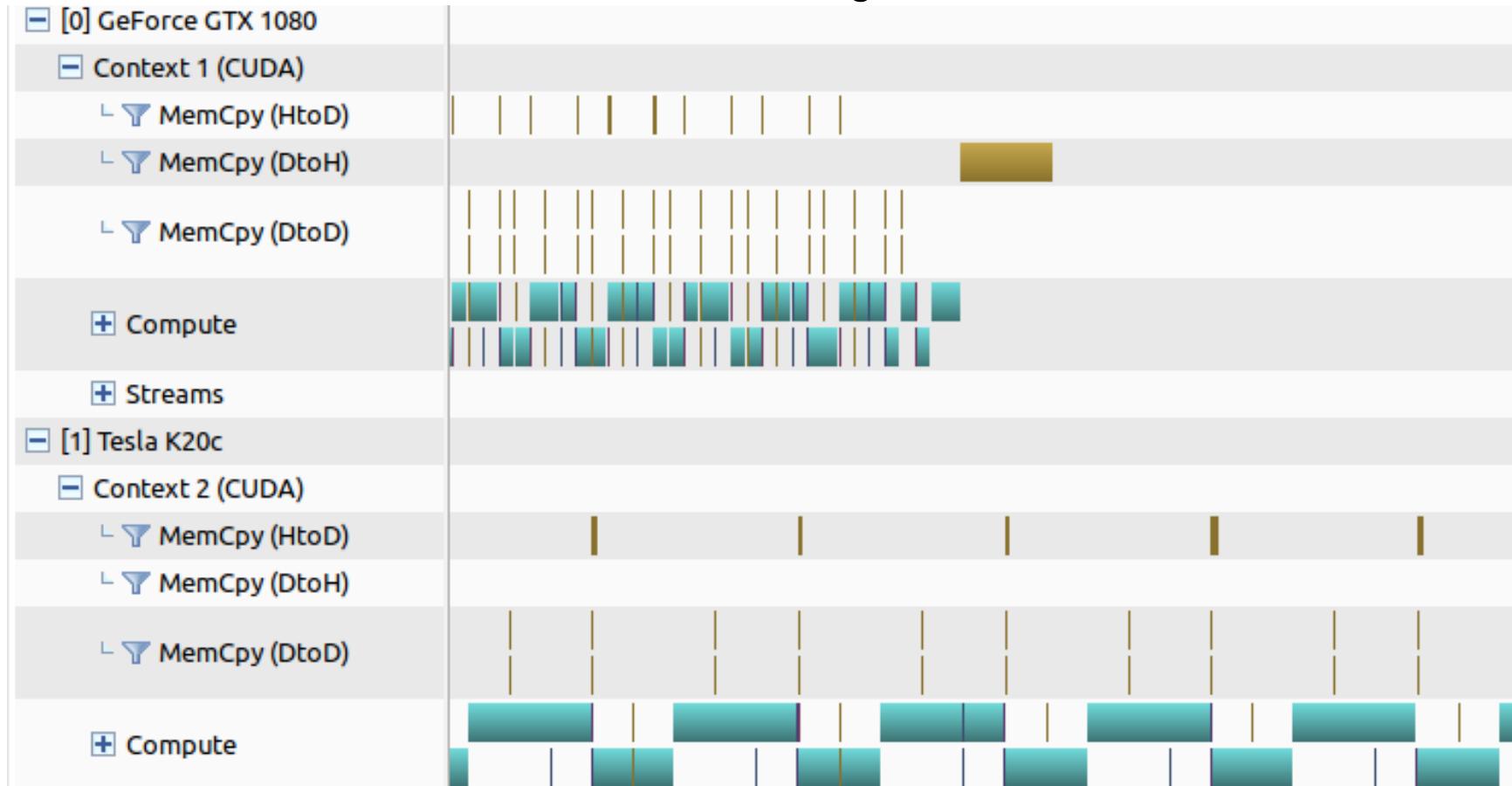
## Mehrere GPUs



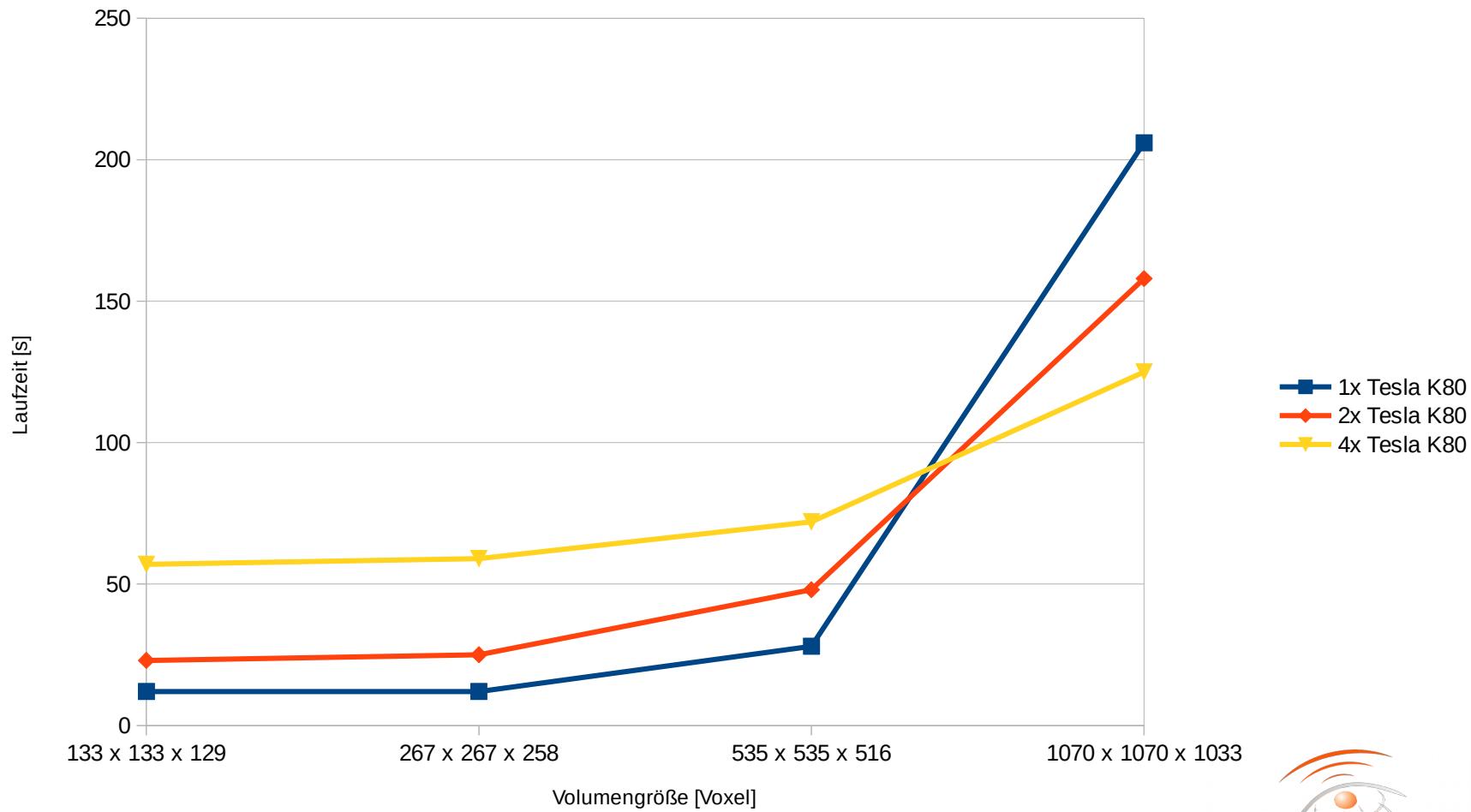
## Mehrere GPUs – Laufzeitverhalten (heterogen)



## Mehrere GPUs – Laufzeitverhalten (heterogen)



## Mehrere GPUs – Laufzeitverhalten (homogen)



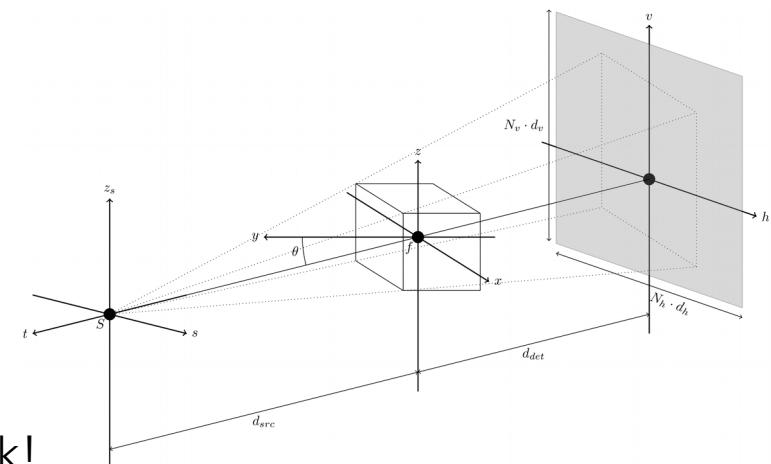
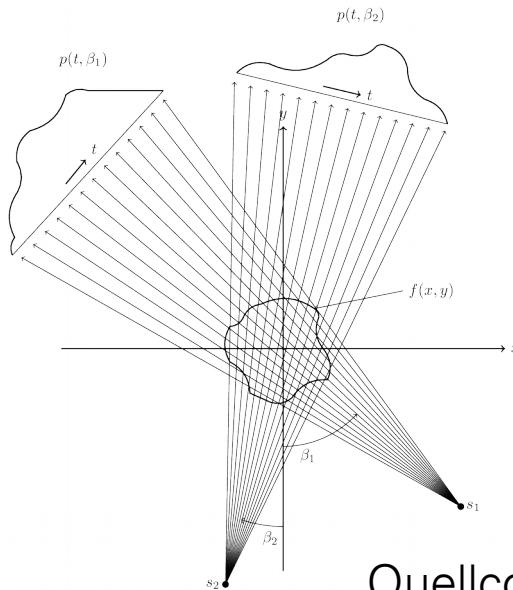
## Fazit

- Rückprojektion wird in sinnvoller Zeit gelöst
- Geringe Wartezeit zwischen Projektionen
  - noch nicht optimal
- Starke Auslastung der GPU
- Prinzipiell auf mehreren GPUs lauffähig
  - bessere Lastverteilung nötig

## Weitere Entwicklung

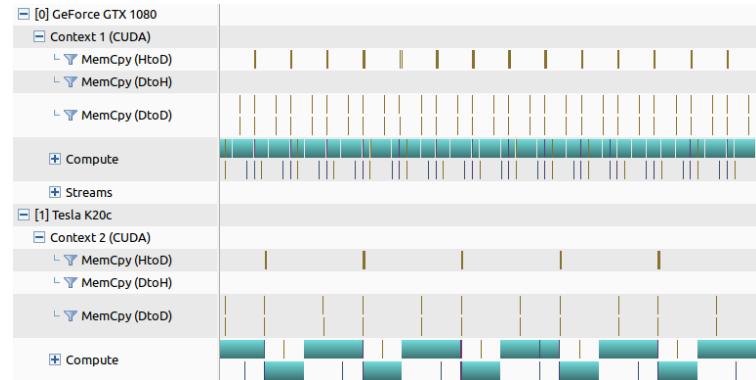
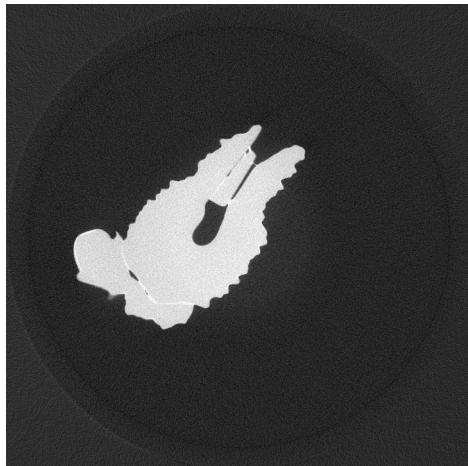
- Reduzierung des Host-Overheads
- *Machine learning* für bessere Lastverteilung
- Einbindung zusätzlicher Vor- und Nachverarbeitungsschritte
- Echtzeitrekonstruktion

# Schluss



Vielen Dank!

Quellcode: <https://github.com/HZDR-FWDF/PARIS>

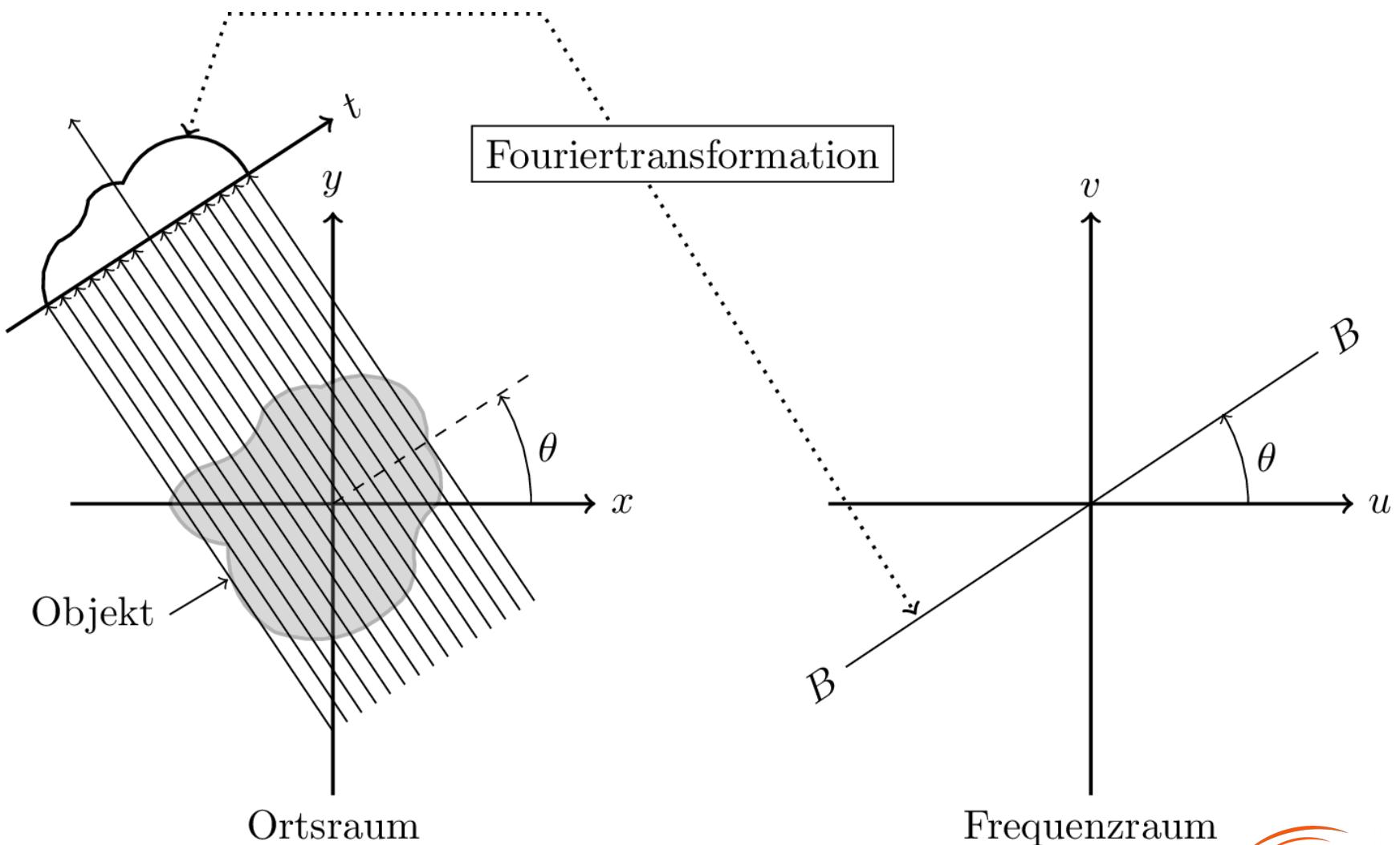


# Quellen

---

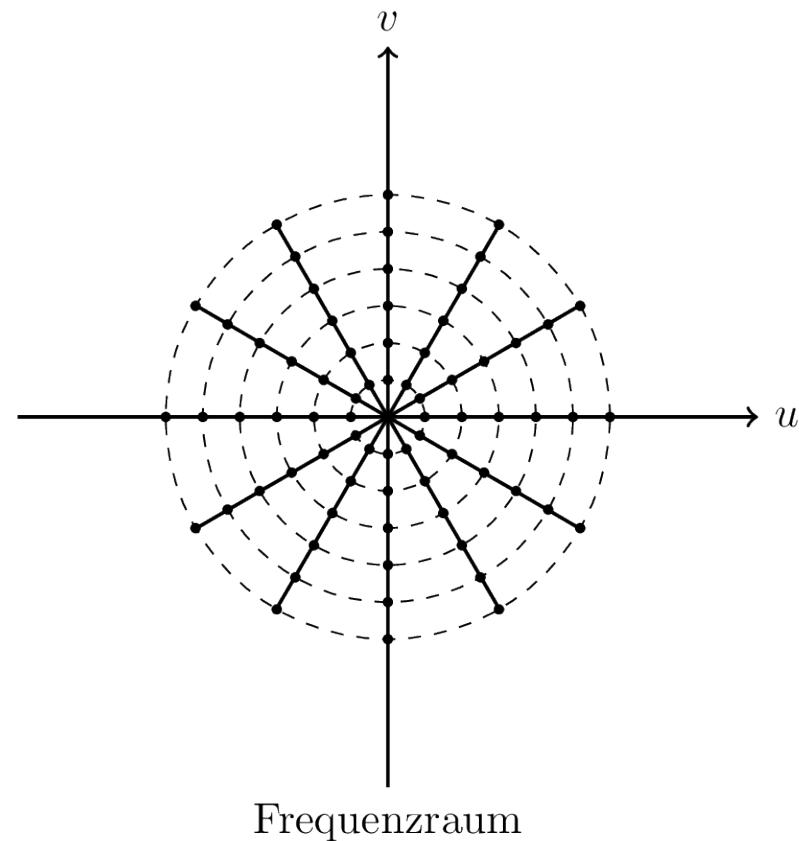
- [1] A. C. Kak: *Computerized tomography with x-ray emission and ultrasound sources*, Proceedings of the IEEE, Band 67, S. 1245 – 1272, 1979
- [2] A. Rosenfeld, A. C. Kak: *Digital Image Processing*, Band 1, 2nd Ed., Academic Press, 1982
- [3] A. C. Kak, M. Slaney: *Principles of Computerized Tomographic Imaging*, IEEE Press, 1988
- [4] L. A. Feldkamp, L. C. Davis & J. W. Kress: *Practical cone-beam algorithm*, Journal of the Optical Society of America A, Vol. 1, Issue 6, pp. 612 – 619, 1984
- [5] Holger Scherl, Benjamin Keck, Markus Kowarschik, Joachim Hornegger: *Fast GPU-Based CT Reconstruction using the Common Unified Device Architecture (CUDA)*, 2007 IEEE Nuclear Science Symposium Conference Record, S. 4464 - 4466, Oktober 2007
- [6] Xing Zhao, Jing-Jing Hu, Peng Zhang: *GPU-based 3D cone-beam CT image reconstruction for large data volume*, Journal of Biomedical Imaging, Vol. 2009, Art. 8, 2009

# Grundlagen der Computertomographie



# Grundlagen der Computertomographie

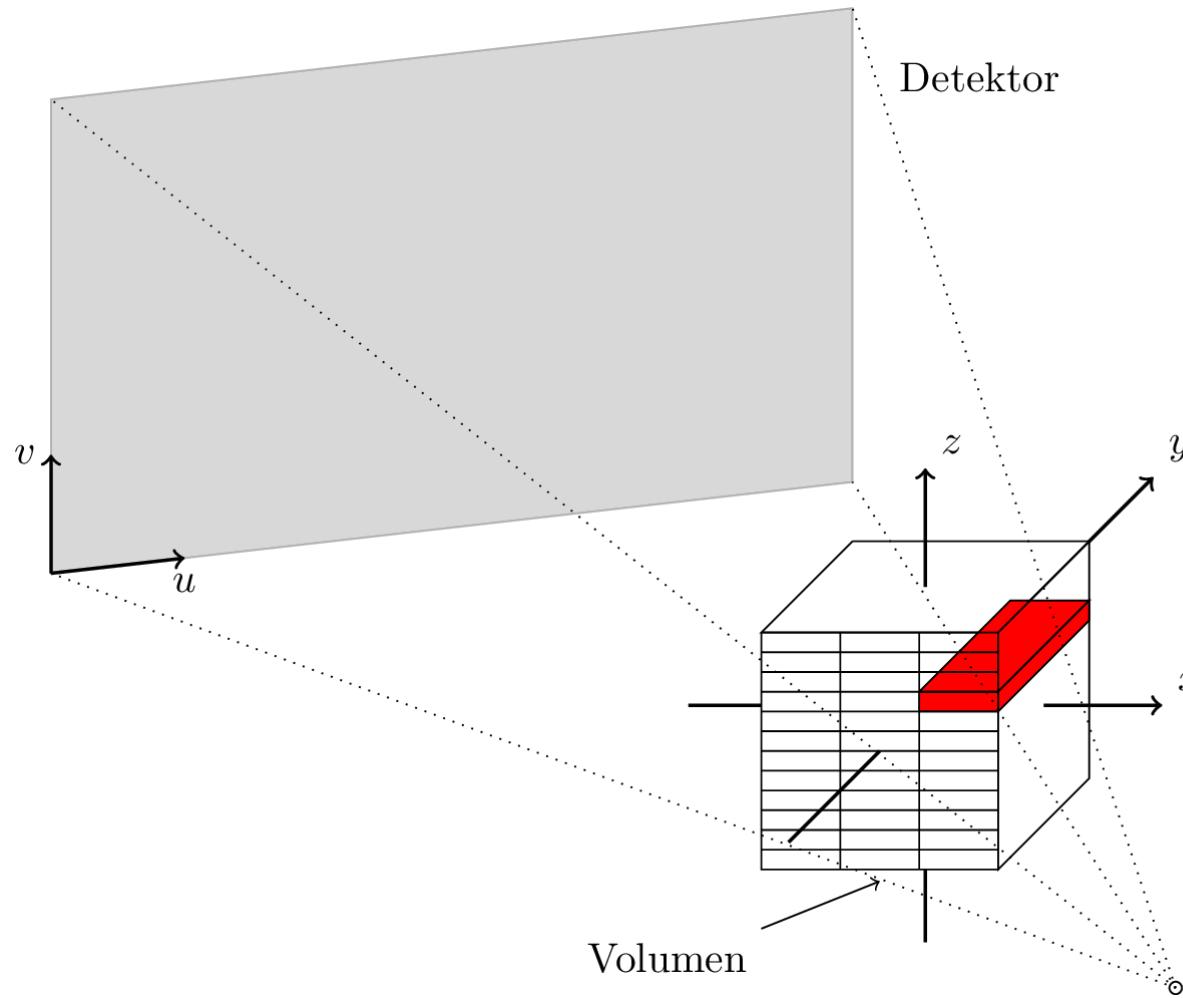
- Projektionen sind unabhängig (mit Ausnahme des Punkts  $F(0, 0)$ )
- Betrachtung aller Projektionen im Frequenzraum zu komplex
- Fouriertransformation ist eine lineare Operation → Aufsummieren im Ortsraum möglich (**Rückprojektion**)
- Filterung: Wichtung der Projektionen im Frequenzraum



Vorlage: [3], S. 59

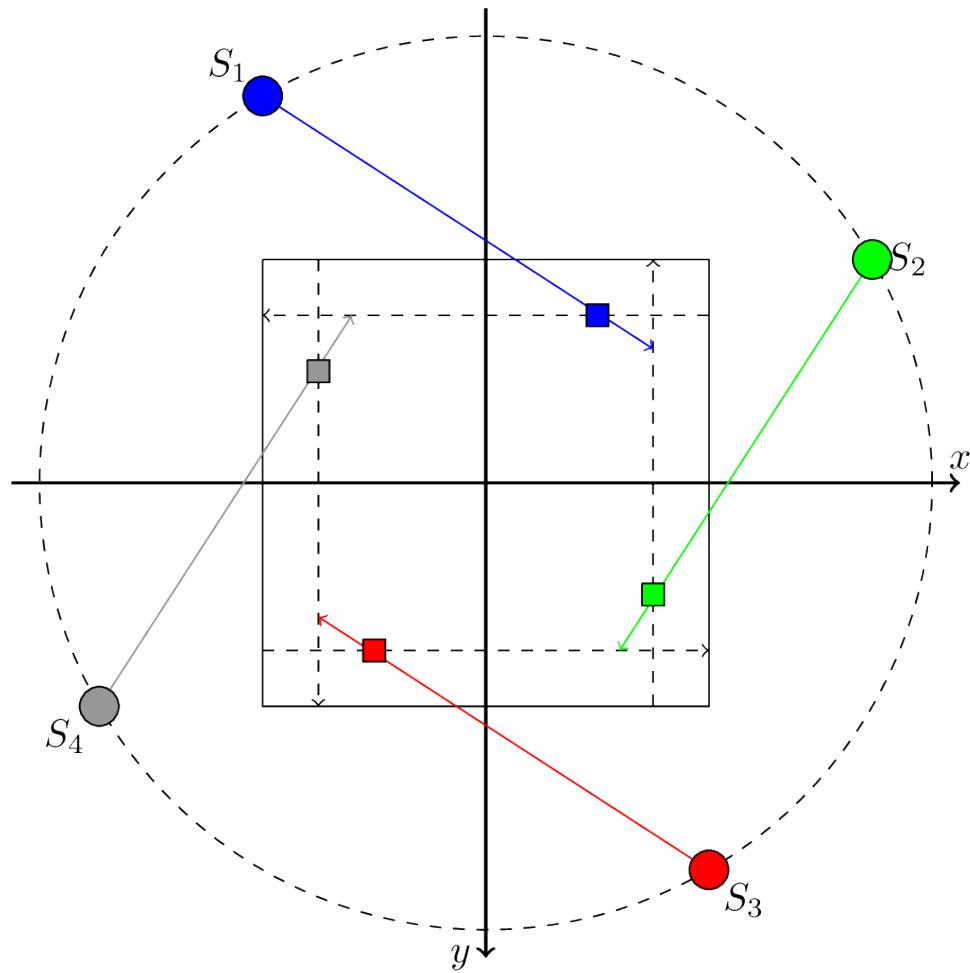
# Parallelisierung

---



Vorlage: [5]

# Parallelisierung



Vorlage: [6]