Westfälische
Wilhelms-Universität
Münster

Fachbereich Mathematik und Informatik
Arbeitsgruppe Parallele und Verteilte Systeme
Prof. Dr. Sergei Gorlatch

Parallelisierung eines Bildrekonstruktionsalgorithmus mit CUDA

Dominique Meiländer

d.meil@uni-muenster.de

Motivation

Aufgabenstellung:

- Bildrekonstruktion aus PET Daten
  - Algorithmus (LM OSEM) wird am UKM verwendet
  - Große Datenmengen $\Rightarrow$ mehrere Stunden Rekonstruktionszeit
- Parallelisierung auf modernen Grafikkarten
  - Programmierung mit *CUDA*

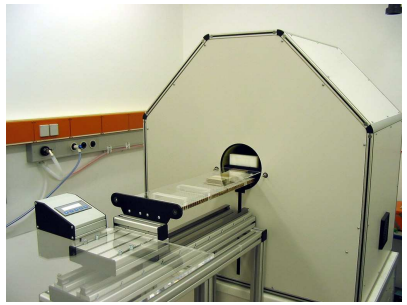
Ziel:

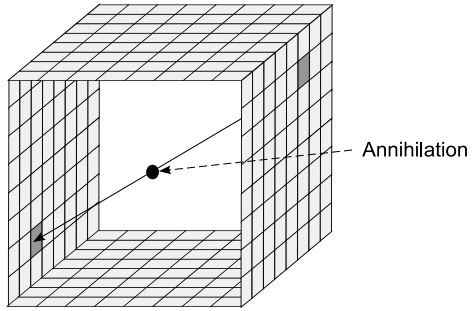
- Wie lässt sich der LM OSEM Algorithmus auf der GPU umsetzen?
- Wie sind die Ergebnisse (bezüglich Laufzeit, Skalierbarkeit, Programmierung)?

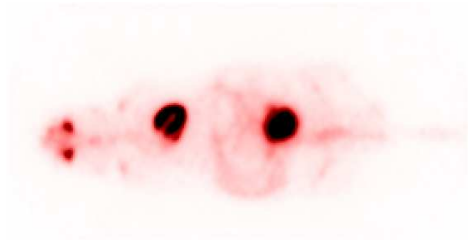
Zunächst: Was ist PET?

PET Verfahren

- PET (Positronen-Emissions-Tomographie)
- Bildgebendes Verfahren aus der Nuklearmedizin (u.a. zur Krebsdiagnose)
- Lokalisation einer radioaktive Substanz in einem Körper
- Kollision von Positron und Elektron $\Rightarrow$ Zwei Gammastrahlen (Annihilation)
- Detektoren registrieren Gammastrahlen (*Event*) $\Rightarrow$ Kollision auf Linie zwischen Detektoren
- Bildrekonstruktion: Events $\longrightarrow$ 3D-Bild
- Rekonstruktionsalgorithmus: *LM OSEM Algorithmus*





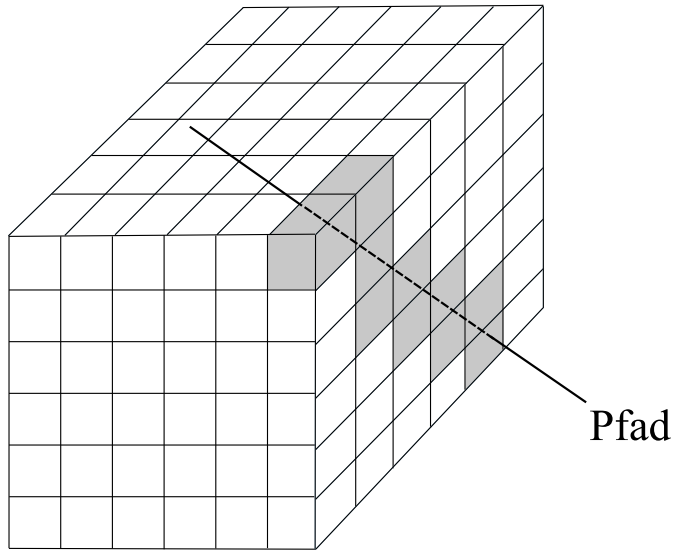


LM OSEM Algorithmus

- LM OSEM (List-Mode Ordered Subset Expectation Maximization)
- Algorithmus: $f_{l+1} = f_l c_l; \quad c_l = \frac{1}{A^t 1} \sum_{i \in S_l} (A_i)^t \frac{1}{A_i f_l}$
- Eingabe: m PET Events $\Rightarrow$ s *Subsets*
- Ausgabe: 3D-Rekonstruktionsbild f
- Berechnung der Linie (*Pfad*) eines Events durch Rekonstruktionsbild (Siddon-Algorithmus)
- A_i: Schnittlänge von Pfad zu Event i mit Voxeln von f (Im Code: path)

```
for each (subset l) {
  for each (event i ∈ S_l) {
    c_{l,i} = (A_i)^t 1/(A_i f_l)
    c_l += c_{l,i}; }
  f_{l+1} = f_l c_l; }
```

LM OSEM Pseudocode



Sequenzieller Programmcode

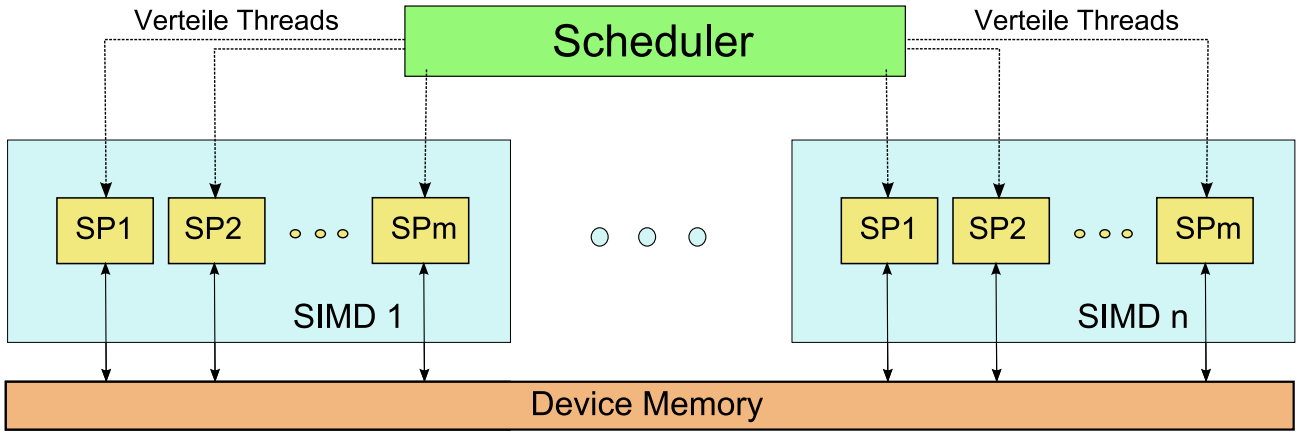
$$f_{l+1} = f_l c_l; \quad c_l = \frac{1}{A^t 1} \sum_{i \in S_l} (A_i)^t \frac{1}{A_i f_l}$$

```
for (int l = 0; l < subsets; l++) {
  [...] /* Events einlesen */
  /* Projektion */
  for (int i = 0; i < subset_size; i++) {
    /* Berechne A_i */
    path = compute_path(i);
    /* Berechne c_l */
    for (int j = 0; path[j] != -1; j++)
      c_l[path[j].coord] += c * path[j].length; /* c = A_i f_l */
  }
  /* Berechne f_{l+1} (Update) */
  for (int k = 0 ; k < image_size; k++) {
    if (c_l[k] > 0.0)  f[k] *= c_l[k];  } }
```

Sequenzieller Programmcode des LM OSEM Algorithmus

Architektur: GPU (Graphics Processing Unit)

- 1 - 16 SIMD Multiprozessoren
- 8 SPs (Stream Processors) pro Multiprozessor
- Globaler (bis zu 4 GB) und lokaler Speicher (256 KB)
- Anbindung über PCI Express Bus
- Automatisches Threadmanagement über Scheduler



- Programmierung: *NVIDIA CUDA* (Compute Unified Device Architekture)

Programmierung der GPU mit CUDA

- GPU Threads werden zu Blöcken gruppiert und auf Multiprozessoren verteilt
- Programmierer konfiguriert Blockgröße und -anzahl

```
kernelFunktion<<<NUM_BLOCKS, BLOCK_SZ>>>();
```

- Scheduler verteilt Thread-Blöcke transparent
- Koordination anhand vom Scheduler vergebener IDs

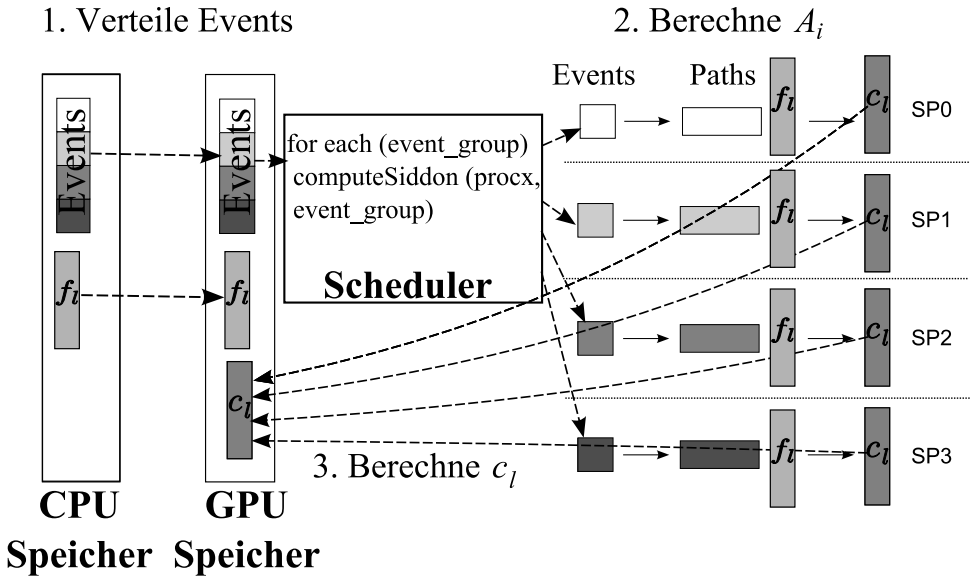
```
int *data; // Zu verarbeitendes Datenfeld
int idx = threadIdx + blockIdx * BLOCK_SZ;
data[idx]++;
```

- Keine atomaren Funktionen für Gleitkommazahlen

GPU Implementierung der Projektion

Vorgehen:

1. Lade Events in GPU Speicher; Scheduler verteilt Threads auf SPs
2. Für jedes Event i: Berechne A_i
3. Berechne c_l nicht atomar



GPU Implementierung der Projektion

```
/* Lade Events und Rekonstruktionsbild in GPU Speicher */
cuda_memcpy(events, c_l);

/* Projektion */
__global_projektion<<<NUM_BLOCKS, BLOCK_SZ>>>(events, c_l);
```

CPU Programmcode zum Laden der Events und Berechnung der Projektion (main-Methode)

```
/* m Anzahl Events; p Anzahl Threads; threadIdx ThreadID */
void __global_projektion(Event *events, float *c_l) {
  int evPerThread = m/p;
  for (int i = 0; i < evPerThread; i++) {
    // Berechne A_i
    path = compute_path(events[threadIdx * evPerThread + i]);
    // Berechne c_l
    for each (pathElt p ∈ path)
      c_l += c * p.length; /* c = A_i f_l */
  }
}
```

SP Programmcode zur Berechnung der Projektion (Kernel)

Laufzeiten

GPU (NVIDIA GeForce 9800 GX2):

- 16 Multiprozessoren $\Rightarrow$ 128 SPs
- Prozessortakt: 1.5 GHz
- 512 MB Globaler Speicher
- 64 GB/s on-chip Bandbreite

Q9550 Quad-Core-Prozessor:

- 4 Prozessorkerne
- Prozessortakt: 2.83 GHz
- 11 GB/s Memory Bus Bandbreite

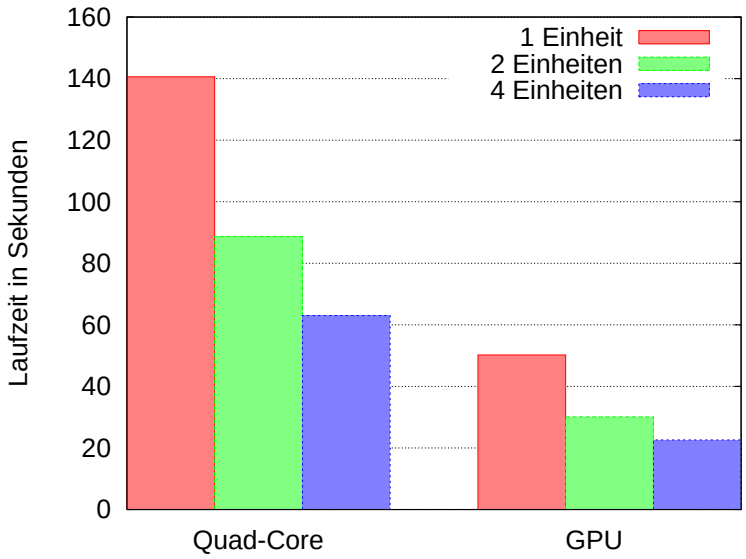
Laufzeiten

GPU (NVIDIA GeForce 9800 GX2):

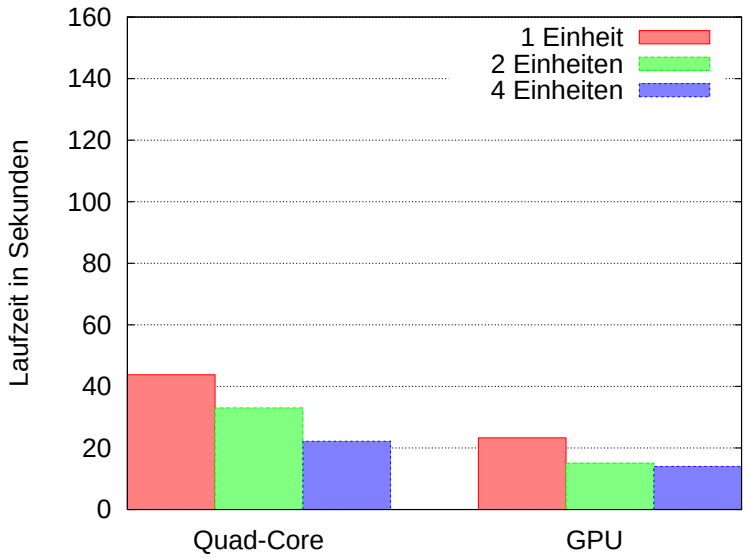
- 16 Multiprozessoren $\Rightarrow$ 128 SPs
- Prozessortakt: 1.5 GHz
- 512 MB Globaler Speicher
- 64 GB/s on-chip Bandbreite

Q9550 Quad-Core-Prozessor:

- 4 Prozessorkerne
- Prozessortakt: 2.83 GHz
- 11 GB/s Memory Bus Bandbreite



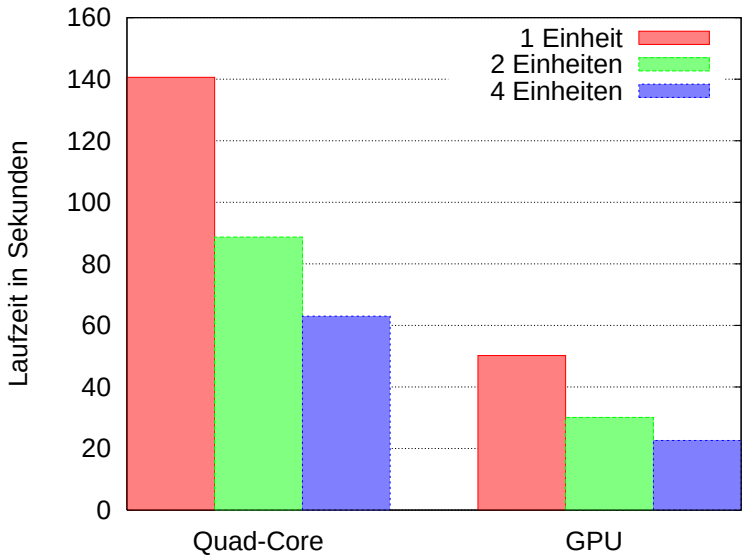
Gesamtlaufzeit bei Auflösung
N=$150 \times 150 \times 280$



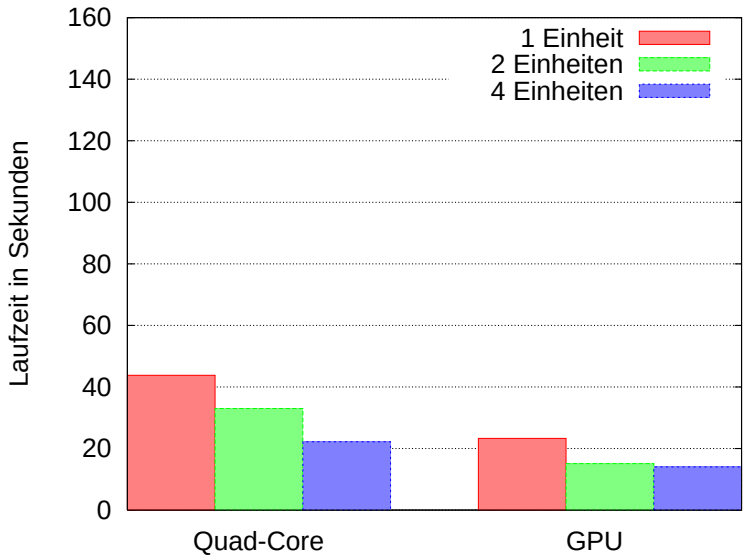
Gesamtlaufzeit bei Auflösung
N=$75 \times 75 \times 140$

Laufzeiten

- Laufzeit sequenzieller Programmcode: 140.6 s
- Implementierung auf 4 GPUs: Speedup um Faktor 6.2 im Vergleich zu sequenziellem Code
- Laufzeit auf 4 GPUs um Faktor 2.8 schneller als auf 4 Kernen des Quad-Core

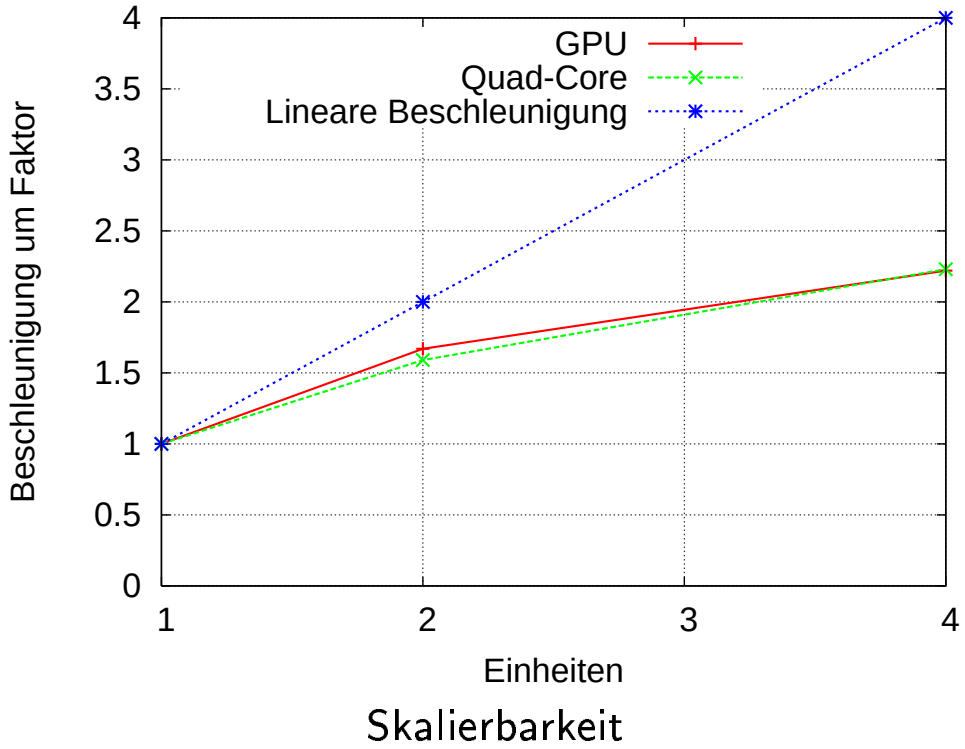


Gesamtlaufzeit bei Auflösung
N=$150 \times 150 \times 280$



Gesamtlaufzeit bei Auflösung
N=$75 \times 75 \times 140$

Skalierbarkeit



Skalierbarkeit

- GPU: Speedup um Faktor 1.3 - 1.7
- Quad-Core: Speedup um Faktor 1.4 - 1.6

Zusammenfassung

Performance

Algorithmus auf 4 GPUs um Faktor 2.8 schneller als auf Quad-Core

Skalierbarkeit

GPUs skalieren ähnlich gut wie Quad-Core

Programmier-Komfort

OpenMP Implementierung nur 3 LoC mehr als sequenzieller Code
⇒ Quad-Core über OpenMP einfacher zu programmieren als GPU über CUDA

Kosteneffektivität

Workstation mit high-end CPU und low-cost GPU ungefähr gleich teuer, wie Workstation mit medium-cost CPU und 4 high-end GPUs
⇒ GPU kostengünstiger, aufgrund besserer Performance

Zusammenfassung

Performance

Algorithmus auf 4 GPUs um Faktor 2.8 schneller als auf Quad-Core

Skalierbarkeit

GPUs skalieren ähnlich gut wie Quad-Core

Programmier-Komfort

OpenMP Implementierung nur 3 LoC mehr als sequenzieller Code
$\Rightarrow$ Quad-Core über OpenMP einfacher zu programmieren als GPU über CUDA

Kosteneffektivität

Workstation mit high-end CPU und low-cost GPU ungefähr gleich teuer, wie Workstation mit medium-cost CPU und 4 high-end GPUs
$\Rightarrow$ GPU kostengünstiger, aufgrund besserer Performance

Danke für Ihre Aufmerksamkeit!