

# Parallelisierung in Computer Vision

## Grundlagen mit Implementierungsbeispielen

Christian Braun

18. Januar 2021

# Motivation

ein kurzer Überblick

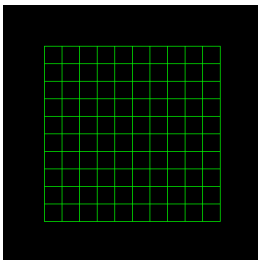
- ▶ Erste Sichtung von parallelisierbarer Lösungen
- ▶ Merkmale um die Parallelisierbarkeit einzuschätzen
- ▶ Mögliche Implementierungsbeispiele
- ▶ Probleme aus der Praxis

# Konventionen bezüglich Computer Vision

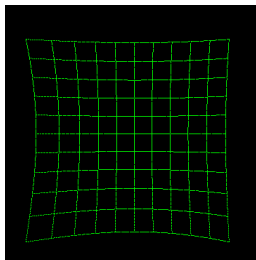
Um eine gemeinsame Grundlage zu schaffen

- ▶ Abbildungsfunktionen können ein  $n$ -Tupel auf ein  $m$ -Tupel abbilden, mit  $n, m \in \mathbb{N}$
- ▶ Ein Tupel besteht aus einer  $\text{Matrix}(c, r)$  von Pixeln, für die  $n = c * r$  gilt mit  $c, r \in \mathbb{N} \& c, r \geq 1$
- ▶ Ein Pixel ist ein 3-wertiges Skalar welches die Farben in Reihenfolge Blau, Grün und Rot darstellt (BGR Farbraum)
- ▶ Ein (Kamera-)Bild ist eine  $\text{Matrix}(c, r)$

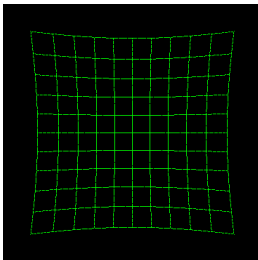
# Korrektur von Abbildungsfehlern



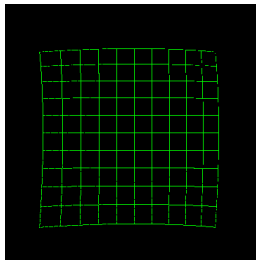
ohne Verzeichnung



volle Verzeichnung



radiale Verzeichnung



tangentielle Verzeichnung

# Unterschiedliche Arten von Parallelisierung

Bei der Parallelisierung sind zwei Begebenheiten zu unterscheiden

- ▶ Datenparallelisierung - Aufteilen einer einzelnen Aufgabe
- ▶ Aufgabenparallelisierung - Aufteilen mehrerer von einander unabhängigen Aufgaben

# Ziel der Analyse

Bei der Aufteilung einer einzelnen Aufgabe

- ▶ weniger Synchronisierungsaufwand
- ▶ Vermeidung der Notwendigkeit zur Absicherung
- ▶ Isolierung einer Aufgabe (divide and conquer)
- ▶ Kapselung für Fehlerfälle (Unhandled Exception)

# Gegenüberstellung Platzproblem

Größe und Position vor und nach der Abbildung

Übersetzung von Texten:

- ▶ Wann hört ein Satz auf?
- ▶ Wie groß ist er nach dem Übersetzen?
- ▶ Problemlösung: Tokenizer, jedoch auch eigene und neue Aufgabe

Pixelzugriff:

- ▶ Besitzt einen absehbaren Iterator
- ▶  $aktuelleSpalte + (GesamtzahlSpalten * aktuelleReihe) = wievielterPixel$
- ▶ oder auch:  $d = Image.data(); d[wievielterPixel]$

# Selbststehende Aufgabe aufteilen

## Die zu untersuchenden Eigenschaften

- ▶  $U$  ist die Menge der Ursprungspositionen im Speicher
- ▶  $Z$  ist die Menge der Zielpositionen im Speicher
- ▶  $F$  ist die Funktion, die  $U$  auf  $Z$  abbildet
- ▶ Die Größe und Position eines jeden einzelnen Elementes der Eingabemenge ist bekannt
- ▶ Die Größe und Position eines jeden einzelnen Elementes der Ausgabemenge ist bekannt
- ▶ Die Bearbeitungszeit eines Elementes



## Weitere Eigenschaften

- ▶ Ist die Positionsänderung eines Pixels injektiv, wird jeder Pixel aus  $U$  genau einem Pixel aus  $Z$  zugewiesen.
- ▶ ist die Positionsänderung eines Pixel dagegen surjektiv, kann ein Pixel aus  $Z$  durch mehrere Pixel aus  $U$  beschrieben werden.

# Zwischenerkenntnisse

Ist die Abbildungsfunktion der Position...

- ▶ injektiv, müssen nur ggfs. leere Positionen gefüllt werden.
- ▶ surjektiv, müssen Ergebnisse zur gleichen Zielposition in einem zusätzlichen Konstrukt vorgehalten werden. Werden die Eingabeelemente verändert, so müssen diese auch vor Veränderung geschützt werden, bis alles berechnet wurde (vgl. Game of Life)

# Beispiel: Skalierung eines Bildes

Halbierung des Bildes wird verwendet um in einer geringeren Datenmenge Bereiche auszuschließen

- ▶ Pixel werden in einem  $n$ -Tupel gebündelt, wobei  $n$  eine endliche Zahl ist
- ▶ Beispiel: Wird ein Bild halbiert, wird ein 4-Tupel auf ein 1-Tupel abgebildet. Dabei werden  $2 \times 2$  Pixel zu einem zusammengefasst

# Bearbeitungszeit einzelner Elemente

- ▶ Für die Bearbeitung einzelner Elemente kann eine Laufzeitanalyse durchgeführt werden
- ▶ Wenn es mehr zu bearbeitende Elemente als gleichzeitig laufende Threads existieren, so muss man sich Gedanken über die Aufteilung machen

# Amdahlsches Gesetz

## Geschwindigkeitsgewinn eines parallelisierbaren Problems

Formelzeichen	bezeichnete Größe
$T$	Gesamtlaufzeit
$t_s$	Laufzeit eines seriellen Programmabschnitts
$t_p$	Laufzeit eines parallelen Programmabschnitts
$n_p$	Anzahl der nutzbaren Prozessoren
$t_{O(np)}$	Laufzeit für Synchronisierungsaufwand
$n_s$	Speedup-Faktor

$$n_s = \frac{T}{t_s + t_{O(np)} + \frac{t_p}{n_p}} \leq \frac{T}{t_s} = \frac{T}{T - t_p}$$

# Amdahlsches Gesetz

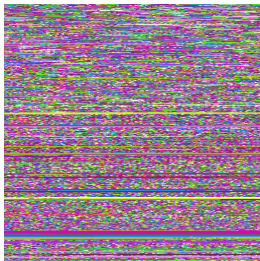
- ▶ Die Funktion hat für jede Eingabe eine obere Schranke
- ▶ Kann 50% einer Aufgabe parallel zum restlichen Teil ausgeführt werden, ist die obere Schranke 2
- ▶ Das Gesetz gibt eine schnelle Einschätzung über den Geschwindigkeitsgewinn

# Parallelisierung einer For-Schleife

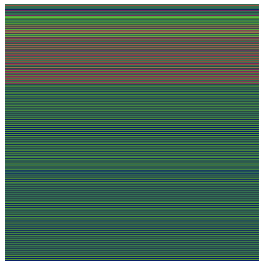
- ▶ Ein absehbarer Iterator kann in Chunks aufgeteilt werden
- ▶  $aktuelleSpalte + (GesamtzahlSpalten * aktuelleReihe) = wievielterPixel$
- ▶ Jeder Chunk bekommt einen Start- und Endindex, so wie das notwendige Inkrement
- ▶ Die entstandenen Chunks können auf einzelne Threads aufgeteilt werden

# Aufgabenteilung in Chunks

Geschwindigkeitsgewinn eines parallelisierbaren Problems



Chunkgröße 5 Pixel



Chunkgröße Anzahl der Spalten

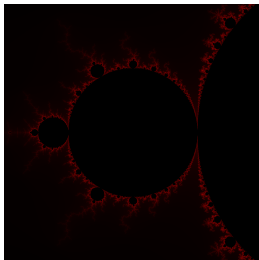


# Aufgabenteilung in Chunks

Geschwindigkeitsgewinn eines parallelisierbaren Problems

Mandelbrot mit 2048x2048 Pixeln und 4 Threads

Art	Dauer in Sekunden	Speedup
Sequentiell	40,149	-
Breite*1	13,022	3,08
Breite*51	12,126	3,31



# Threadpools

- ▶ Mehrere Threads werden in einem Objekt gekapselt
- ▶ Die Threads sind in einem wartenden Zustand, bis neue Aufgaben kommen
- ▶ Objectpooling reduziert den Initialisierungsaufwand
- ▶ Der Verwaltungsaufwand für Threads wird von der tatsächlichen Bearbeitung abgekapselt

# Taskmanager ähnliches Konstrukt

kontrolliert eine begrenzte Anzahl an Threads

- ▶ kontrolliertes Beenden des Programmes
- ▶ verhindert größere Auslastung als vorgegeben
- ▶ garantiert Bearbeitungszeit
- ▶ falls implementiert, umschalten zu sequentieller Bearbeitung (z.B. im Fehlerfall)

# Literatur

- ▶ Operating System Concepts, Abraham Silberschatz
- ▶ OpenMP Complete Specifications Version 3.1 – Jul 2011
- ▶ Mikroprozessortechnik, Klaus Wüst
- ▶ Design Patterns, Erich Gamma
- ▶ C++ Primer, Stanley B.Lippman