Implementierung des Sobelfilters

in C und Assembler

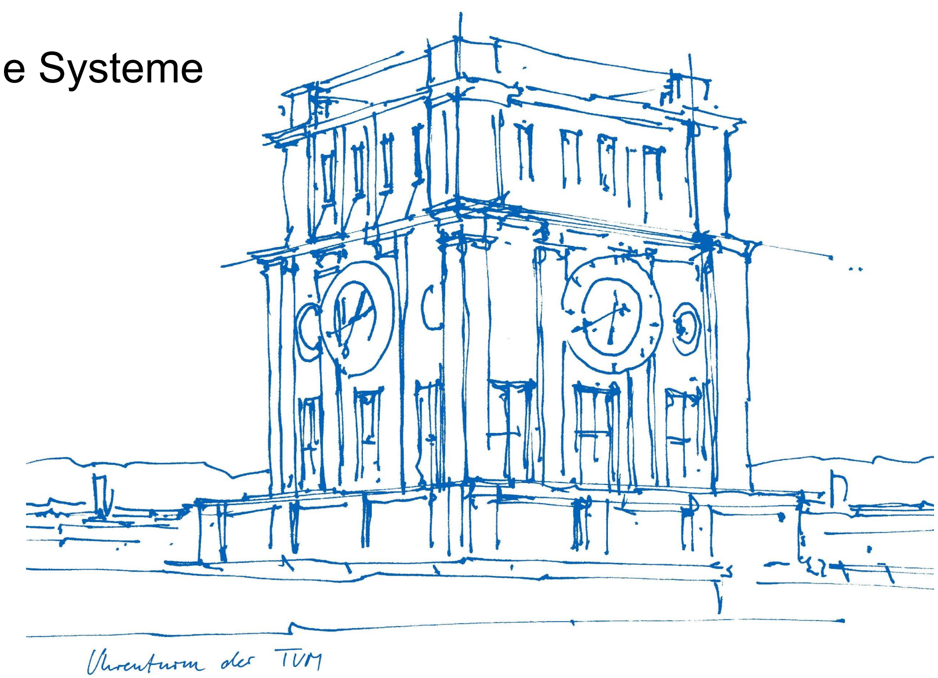
Manuel Schnaus, Leonhard Chen und Stefan Gruber

Technische Universität München

Fakultät für Informatik

Lehrstuhl für Rechnerarchitektur und Parallele Systeme

München, der 03. März 2020



Praktikum Aspekte der Systemnahen Programmierung

Einleitung

- Sobel-Filter hebt Umrisse im Bild hervor
- Filtert horizontal und vertikal und fügt Ausgabe-Bilder zusammen
- Definition des Sobel-Filters:

1. $A^v = M^v * E$

2. $A^{v,F}_{(x,y)} = \sum_{i=-k}^{k} \sum_{j=-k}^{k} M^v_{(k+i,k+j)} \cdot E^F_{(x+i,y+j)}$

3. $p_{(x,y)} = (R_{(x,y)}, G_{(x,y)}, B_{(x,y)})$

4. $G^F_{(x,y)} = \sqrt{(A^{v,F}_{(x,y)})^2 + (A^{h,F}_{(x,y)})^2}$

Lösungsansatz

Algorithm 1: Mathematischer Filter-Algorithmus

Data: Eingabebild $E \in \mathbb{R}^{\mathrm{b} \times \mathrm{h} \times 3}$, Ausgabebild $G \in \mathbb{R}^{\mathrm{b} \times \mathrm{h} \times 3}$, Breite des Bildes $b \in \mathbb{N}$, Höhe des Bildes $h \in \mathbb{N}$

$$M^h = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix}, M^v = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

$y = 1$

while $y < h - 1$ **do**

$x = 1$

while $x < b - 1$ **do**

$$A^v_{(x,y)} = \sum_{i=-1}^{1} \sum_{j=-1}^{1} M^v_{(1+i,1+j)} \cdot E_{(x+i,y+j)}$$

$$A^h_{(x,y)} = \sum_{i=-1}^{1} \sum_{j=-1}^{1} M^h_{(1+i,1+j)} \cdot E_{(x+i,y+j)}$$

$$G_{(x,y)} = \sqrt{(A^v_{(x,y)})^2 + (A^h_{(x,y)})^2}$$

$x = x + 1$

$y = y + 1$

Lösungsansatz

$E_{(x-1,y-1)}$	$E_{(x,y-1)}$	$E_{(x+1,y-1)}$
$E_{(x-1,y)}$	$E_{(x,y)}$	$E_{(x+1,y)}$
$E_{(x-1,y+1)}$	$E_{(x,y+1)}$	$E_{(x+1,y+1)}$

$$A^{v}_{(x,y)} = E_{(x-1,y-1)}$$

Lösungsansatz

$E_{(x-1,y-1)}$	$E_{(x,y-1)}$	$E_{(x+1,y-1)}$
$E_{(x-1,y)}$	$E_{(x,y)}$	$E_{(x+1,y)}$
$E_{(x-1,y+1)}$	$E_{(x,y+1)}$	$E_{(x+1,y+1)}$

$$A^{v}_{(x,y)} = E_{(x-1,y-1)} + 2\,E_{(x,y-1)}$$

Lösungsansatz

$E_{(x-1,y-1)}$	$E_{(x,y-1)}$	$E_{(x+1,y-1)}$
$E_{(x-1,y)}$	$E_{(x,y)}$	$E_{(x+1,y)}$
$E_{(x-1,y+1)}$	$E_{(x,y+1)}$	$E_{(x+1,y+1)}$

$$A^{v}_{(x,y)} = E_{(x-1,y-1)} + 2\,E_{(x,y-1)} + E_{(x+1\ y-1)}$$

Lösungsansatz

$E_{(x-1,y-1)}$	$E_{(x,y-1)}$	$E_{(x+1,y-1)}$
$E_{(x-1,y)}$	$E_{(x,y)}$	$E_{(x+1,y)}$
$E_{(x-1,y+1)}$	$E_{(x,y+1)}$	$E_{(x+1,y+1)}$

$$A^{v}_{(x,y)} = E_{(x-1,y-1)} + 2\,E_{(x,y-1)} + E_{(x,+1\;y-1)} - E_{(x-1,y+1)}$$

Lösungsansatz

$E_{(x-1,y-1)}$	$E_{(x,y-1)}$	$E_{(x+1,y-1)}$
$E_{(x-1,y)}$	$E_{(x,y)}$	$E_{(x+1,y)}$
$E_{(x-1,y+1)}$	$E_{(x,y+1)}$	$E_{(x+1,y+1)}$

$$A^{v}_{(x,y)} = E_{(x-1,y-1)} + 2\,E_{(x,y-1)} + E_{(x,+1\;y-1)} - E_{(x-1,y+1)} - 2E_{(x\;y+1)}$$

Lösungsansatz

$E_{(x-1,y-1)}$	$E_{(x,y-1)}$	$E_{(x+1,y-1)}$
$E_{(x-1,y)}$	$E_{(x,y)}$	$E_{(x+1,y)}$
$E_{(x-1,y+1)}$	$E_{(x,y+1)}$	$E_{(x+1,y+1)}$

$$A^{v}_{(x,y)} = E_{(x-1,y-1)} + 2\,E_{(x,y-1)} + E_{(x,+1\;y-1)} - E_{(x-1,y+1)} - 2E_{(x\;y+1)} - E_{(x+1,y+1)}$$

Lösungsansatz

- Ersetzen der Matrix-Zugriffe durch eine einheitliche Summe

$$A^{v}_{(x,y)} = E_{(x-1,y-1)} + 2\,E_{(x,y-1)} + E_{(x,+1\;y-1)} - E_{(x-1,y+1)} - 2E_{(x\;y+1)} - E_{(x+1,y+1)}$$

und

$$A^{h}_{(x,y)} = E_{(x-1,y-1)} - \;E_{(x,+1\;y-1)} + 2E_{(x-1\;,y)} - 2E_{(x+1,y)} + E_{(x-1,y)} - E_{(x+1,y+1)}$$

- **Vorteile**: Keine Speicherzugriffe auf die Falungsmatrizen, keine mit 0 multiplizierte Elemente berücksichtigt

- Vier der sechs Elemente werden sowohl für die Berechnung von $A^{v}_{(x,y)}$ als auch $A^{h}_{(x,y)}$ benötigt

➔ Gleichzeitige Berechnung beider Werte zur Einsparung von Load-Operationen

Lösungsansatz

- Verarbeitung von Bildern der Farbtiefe 24bpp → 8 Bit pro Farbkanal eines Pixels
- Die einzelnen Farbkanäle eines Pixels im Ausgabe-Bild sind nicht voneinander abhängig
- Schleife über die Farbkanäle des Bildes statt über die Pixel → auf jedes Byte des Eingabe-Arrays wird separat die Formel angewandt
- Umsetzung in einer einfachen Schleife über die Speicheradresse des Input-Arrays mit einem Counter über der Breite

Lösungsansatz

Algorithm 2: Optimierter Filter-Algorithmus

Data: Eingabebild E[], Ausgabebild G[], Breite des Bildes b, Höhe des Bildes h

$E_{max} = b \cdot h \cdot 3 - b \cdot 6 - 6 + E$

$w = 0$

while $E < E_{max}$ **do**

  if $w = 3 \cdot (b - 2)$ **then**

    $E = E + 6$

    $G = G + 6$

    $w = 0$

    continue

  $A^{v,F} = E[0] + 2 \cdot E[3] + E[6] - E[6 \cdot b] - 2 \cdot E[6 \cdot b + 3] - E[6 \cdot b + 6]$

  $A^{h,F} = E[0] - E[6] + 2 \cdot E[3 \cdot b] - 2 \cdot E[3 \cdot b + 6] + E[6 \cdot b] - E[6 \cdot b + 6]$

  $G^F = \sqrt{(A^{v,F})^2 + (A^{h,F})^2}$

  $G[0] = G^F$

  $G = G + 1$

  $E = E + 1$

  $w = w + 1$

Lösungsansatz

- Ausführung der Rechnung erfolgt mit einem Offset von einem Byte in jedem Schleifendurchlauf gleich und unabhängig voneinander

➔ SIMD Optimierung bietet sich an

- 16 Farbkanäle können in ein SIMD-Register geladen werden
- **Schwierigkeit**: Aufgrund der Additionen und Multiplikationen reichen 8-Bit Werte nicht
- **Lösung**: Erweiterung der 8-Bit Lanes auf 16-Bit Lanes mit dem Befehl `uxtl`/`uxtl2`
- Verrechnung der mit Zwei multiplizierten Werte nach der Erweiterung mit `ushll`
- Nur noch 8 Farbkanäle werden gleichzeitig berechnet

Lösungsansatz

$$G^F_{(x,y)} = \sqrt{(A^v_{(x,y)})^2 + (A^h_{(x,y)})^2}$$

- Vor der Quadrierung der Werte $A^v_{(x,y)}$ und $A^h_{(x,y)}$ Aufteilung der Register auf jeweils zwei Register mit 32-Bit Lanes

➔ Verhinderung von Overflows

- Vor der Berechnung der Wurzel: Konvertierung aller Lanes zu Floats mit `ucvtf`, um die Wurzel-Instruction fsqrt auf allen Lanes zu ermöglichen

- Anschließend: Cast zu Integer mit der Instruction `fcvtzu`

Lösungsansatz

- 16-Bit Werte müssen zu 8-Bit Werten umgewandelt werden → Beschränkung der Lanes auf einen Maximalwert von 255

- Beispiel: Registers R = [0100 1001], welches 2 Lanes mit jeweils 4 Bit besitzt und auf vier beschränkt werden soll

- Das Register Q wird auf allen Lanes mit dem Maximalwert beschrieben:
  ➔ Q = [0010 0010]

- Erstellen einer Mask M über die Instruction `cmhi` mit dem Ergebnis-Register R und Q als Parameter

- M ist auf allen Lanes bitweise auf 1 gesetzt, wo R größer als der Wert von Q ist
  ➔ M = [0000 1111]

Lösungsansatz

R = [0100 1001], Q = [0010 0010], M = [0000 1111], !M = [1111 0000]

- Durch den bitweisen Operator & auf R und !M werden alle Lanes auf 0 gesetzt, die größer als der Maximalwert (im Beispiel: 4) sind
  ➔ R = R & !M = [0100 1001] & [1111 0000] = [0100 0000]

- In die auf 0 gesetzten Register muss nun der Maximalwert geschrieben werden
  ➔ Q = Q & M = [0010 0010] & [0000 1111] = [0000 0010]

- Anwendung des bitweisen Operator | auf R und Q
  ➔ R = R | Q = [0100 0000] | [0000 0010] = [0100 0010]

Lösungsansatz

- Weil alle Pixel in einem 3x3-Bereich um den zu berechnenden Pixel benötigt werden, sind Randpixel nicht definiert
- Verwendung eines Paddings, in welchem die Randpixel nach außen gespiegelt werden
- Algorithmus wird auf ein in jede Richtung um ein Pixel erweitertes Bild ausgeführt

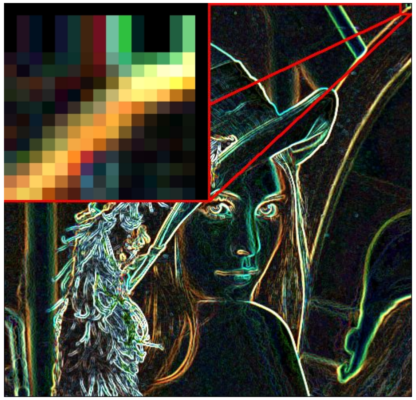
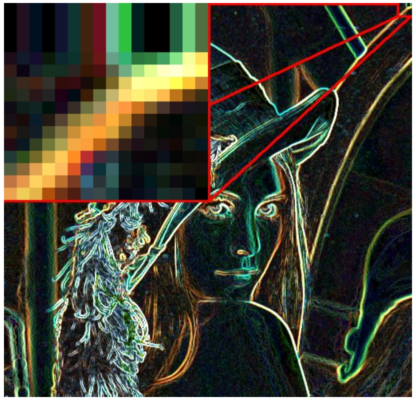
○	○	○
○	$E_{(0,0)}$	$E_{(1,0)}$
○	$E_{(0,1)}$	$E_{(1,1)}$

$\longrightarrow$

$E_{(0,0)}$	$E_{(0,0)}$	$E_{(1,0)}$
$E_{(0,0)}$	$E_{(0,0)}$	$E_{(1,0)}$
$E_{(0,1)}$	$E_{(0,1)}$	$E_{(1,1)}$

Genauigkeit

- Unterschiedlicher Genauigkeiten abhängig von Randbehandlung
- Resultate:

Genauigkeit

$$A^{v,F}_{(x,y)} = \sum_{i=-k}^{k} \sum_{j=-k}^{k} M^{v}_{(k+i,k+j)} \cdot E_{(x+i,y+j)} \qquad G^{F}_{(x,y)} = \sqrt{(A^{v,F}_{(x,y)})^2 + (A^{h,F}_{(x,y)})^2}$$

- Vertikale/Horizontale Komponente A des Ausgabe-Bildes kann größer als 255 sein
- A^2 kann in den 16-Bit Lanes der Summe in SIMD Overflow verursachen
- Laufzeit einsparen durch Akzeptieren das A selten Overflows verursacht
- Kernalgorithmus größenordnungsmäßig in $O(n^2)$
- Mit Mirrored-Padding m + (n - 2) zusätzliche Durchläufe

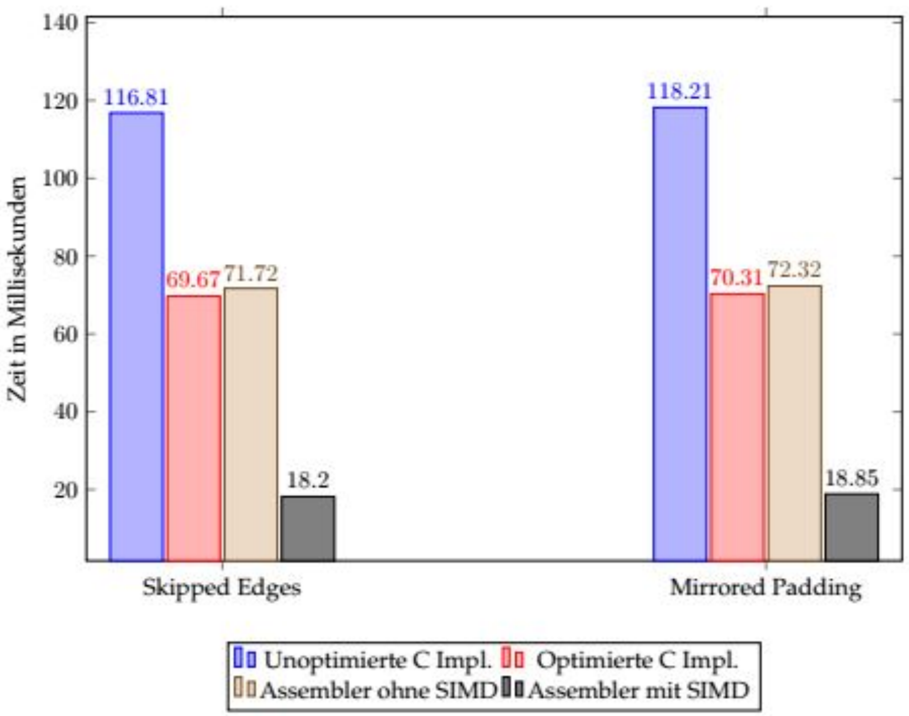
Performanzanalyse

- Vergleich der Assembler-Implementierung zur C-Implementierung

- 8 verschiedene Modi durch 3 Parameter:
  - **Kompilierstufe:** Kompiliert mit Optimierungsstufe *O2* oder *O3*
  - **Randstrategie:** *Skipped Edges* – oder *Mirrored Padding* – Strategie
  - **Implementierung:** Standard-Implementierung in C, C-Implementierung mit verbessertem Algorithmus, Standard-Assembler-Implementierung oder Assembler mit SIMD-Operationen

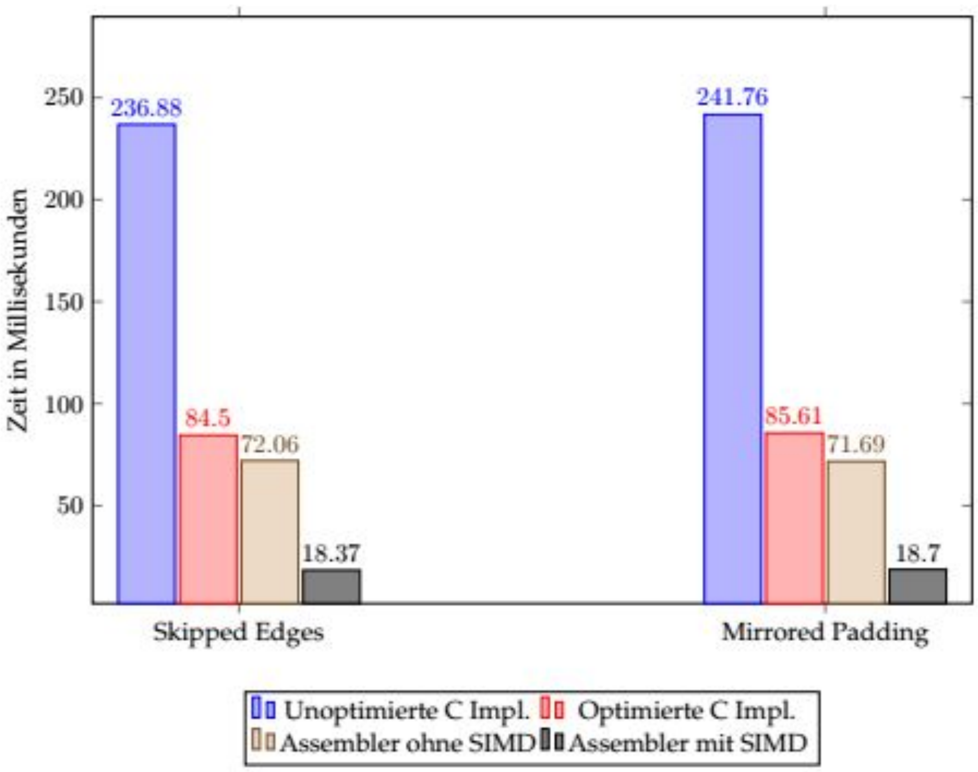
Performanzanalyse

- Kompiliert mit *O3*
- Kaum Unterschied zwischen den Randstrategien
- Extreme Differenz zwischen unoptimierter C-Implementierung und Assembler mit SIMD
- Wenig Differenz zwischen optimierter C-Implementierung und normaler Assembler-Implementierung
- Optimierte C-Implementierung schneller als Assembler-Implementierung ohne SIMD



Performanzanalyse

- Kompiliert mit *O2*
- Erneut kaum Unterschied zwischen den Randstrategien
- Noch größere Differenz zwischen unoptimierter C-Implementierung und Assembler mit SIMD
- Größere Differenz zwischen normaler Assembler- und optimierter C-Implementierung
- Assembler immer schneller als C



Zusammenfassung und Ausblick

- Mögliche Laufzeit-Verbesserung durch Berechnen der Quadrate in 16 Bit Lanes anstatt 32 Bit Lanes, wobei mit Qualitätsverlust des Ergebnisses zu rechnen ist

- Marginale Verbesserungsmöglichkeit durch Integration der vertikalen Spiegelung der *Mirrored Padding* - Strategie in den Kernalgorithmus, zur Vermeidung redundanter Ladebefehle

- Extreme Performanzgewinne durch Nutzung vektorisierter Berechnungen in Assembler mit Hilfe von SIMD-Operationen
  - ➜ Möglicherweise häufig lohnenswert, gewisse Berechnungen daher in Assembler / Inline-Assembler durchzuführen