„Canny Edge Detection: Kantenerkennung“
Algorithmus mit Verwendung GPU CUDA

Inhaltsverzeichnis

1 Einführung in CANNY KANTENERKENNUNG

Der Canny-Kantendetektor bzw. Kantenerkennung ist ein Kantenerkennungsoperator, der einen mehrstufigen Algorithmus verwendet, um eine Vielzahl von Kanten in Bildern zu erkennen. Es wurde 1986 von John F. Canny entwickelt. Canny erstellte auch eine Computertheorie der Kantenerkennung, die erklärt, warum die Technik funktioniert. (Wikipedia)

Der mehrstufige Algorithmus besteht aus Umwandlung in Grauwertbild, Gauß-filter, Gradientenberechnung, Nicht-Maximum Unterdrückung, Doppelter Schwellwert-Verfahren und Kantenverfolgung durch Hysterese.

Umwandlung in Grauwertbild und Gauß-filter dienen zur Rauschensreduktion.

Als nächstes kommt Gradientenberechnung zur Erkennung der Kantenintensität und -richtung.

Danach findet Nicht-Maximum Unterdrückung statt um die Kanten auszudünnen wobei eine Unterdrückung der nicht maximalen Intensitätswerte durchgeführt wird.

Da das Ergebnis der Nicht-Maximum Unterdrückung nicht perfekt ist, einige Kanten möglicherweise nicht relevante Kanten sind und Bild damit etwas Rauchen aufweist, kommt noch der Doppelter Schwellwert-Verfahren zum Einsatz in dem zwei Schwellwerte gesetzt werden.

Als letztes kommt Kantenverfolgung durch Hysterese. Nach der Bestimmung bei Doppelter Schwellwert-Verfahren, was die starken Kanten und die schwachen Kanten sind, muss es jetzt bestimmt werden, welche schwachen Kanten als kantenrelevant übernommen oder entfernt werden sollen. Dafür sorgt Kantenverfolgung durch Hysterese.

Farbbild

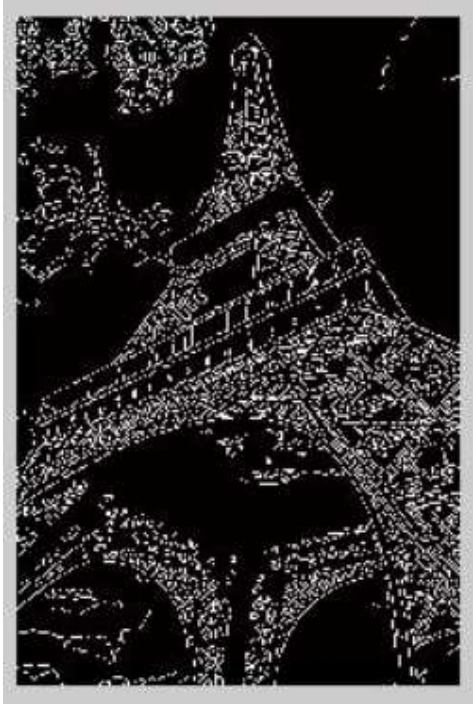


Bild nur mit Kanten

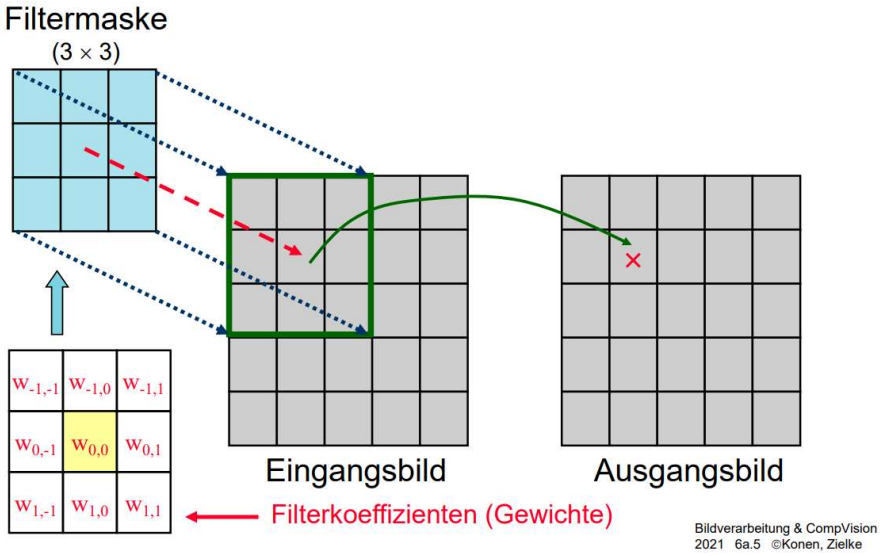
2 Grundlage

2.1 Faltung

In der Bildbearbeitung und in der Bildverarbeitung wird die diskrete Faltung eingesetzt, um entweder störende Einflüsse wie Rauschen zu beheben oder Bildinformationen wie z. B. Kanten zu extrahieren (Kantendetektion). Dabei kommen der Aufgabenstellung angepasste Faltungsmatrizen zum Einsatz, die als Operatorvorschrift für den Glättungskern zu verstehen sind. (Wikipedia)

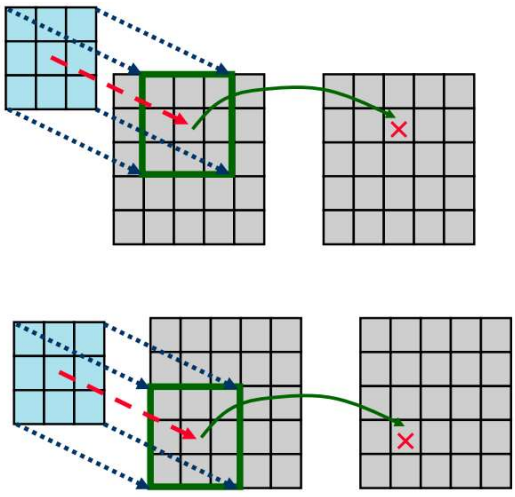
Bei der Faltung berechnet sich jeder Pixel des Ausgangsbildes als gewichtete Summe der Pixel einer Bildnachbarschaft.

Die Gewichte sind die Koeffizienten der Filtermaske (des Filterkerns).



Die Faltung des Bildes mit dem Filterkern an einer Bildposition liefert den Bildwert an der jeweils gleichen Position im Ergebnisbild.

Für die Filterung des kompletten Bildes wird die Maske sukzessiv über alle Bildpositionen geschoben.

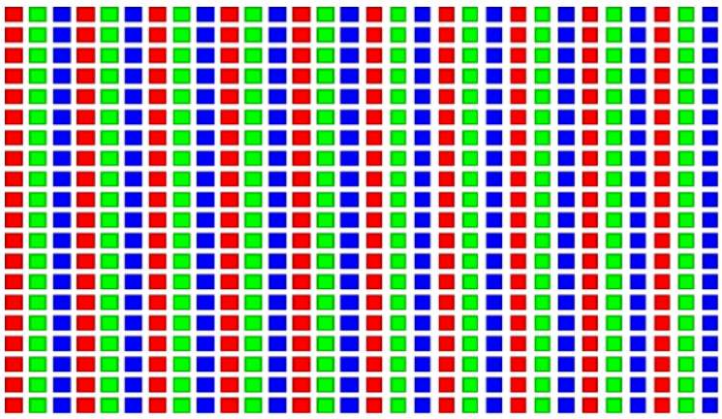


Je nach Größe der Filtermaske existiert ein Bildrand (min. 1 Pixel breit), dessen Pixel nicht normal berechnet werden können, weil ein Teil der Maske außerhalb des Eingangsbildes liegt.

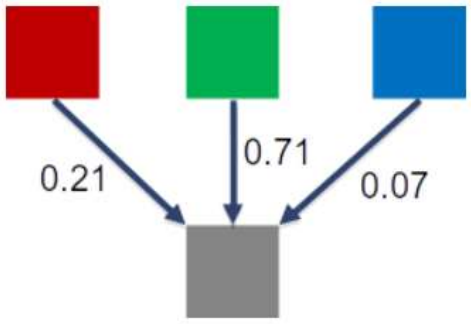
3 Umwandlung in Grauwertbild

Der Canny Kantenerkennung Algorithmus arbeitet auf zweidimensionalen Intensitätswerten. Dieser kann also auf Grauwertbildern oder einzelnen Farbkanälen Kanten erkennen, enthält aber keine Vorschrift, wie mehrere monochrome Kanäle zusammengeführt werden sollte. Jedoch üblicherweise sollten Farbbilder vor einer Kantenerkennung in ein Grauwertbild umgewandelt werden um möglichst rauschenlos und vereinfacht verarbeiten zu können.

Jeder Pixel in einem RGB(Bunt) Bild hat einen RGB Wert (Rot-Grün-Blau).
Das Format eines Bildes ist somit: (r g b) (r g b) .... (r g b):



Um in Grau Farbe zu wandeln, müssen RGB Werte in Ausmaß 21%, 71%, 7% zusammengeführt werden:



Daraus resultiert:

Farbbild

Graubild

Implementierung:

```
@cuda.jit
def cal_gray(inputImage, outputImage):
    h,w = cuda.grid(2)
    if h < inputImage.shape[0] and w < inputImage.shape[1]:
        grayScale =
        inputImage[h,w,0]*0.21
        + inputImage[h,w,1]*0.71
        + inputImage[h,w,2]*0.07

        outputImage[h,w] = grayScale
```

4 Gauß-Filter

Kanten lassen sich in einem Graustufenbild (auch Grauwertbild genannt) durch starke Schwankungen der Bildhelligkeit zwischen zwei Pixeln identifizieren. Ist ein Bild in Grauwerte bzw. Helligkeitswerte umgewandelt worden, können die Kanten demnach als Unstetigkeit dieser Grauwerte identifiziert werden. Dadurch sind Kantenerkennungen allerdings sehr empfindlich gegenüber Bildrauschen.

Es ist schwierig festzustellen, ob eine Kante aus dem eigentlichen Bildsignal oder dem zusätzlichen Bildrauschen identifiziert wurde.

Da Canny bei der Entwicklung des Kantendetektors mit der Annahme arbeitete, dass auf einem Grauwertbild neben den Kanten zusätzlich ein weißes Rauschen vorhanden ist, dient der erste Schritt des Algorithmus der Rauschunterdrückung und Glättung des Bildes. Um das Bild zu glätten und damit das Rauschen zu reduzieren, wird das Bild mit einem sogenannten Gauß-Filter gefaltet. Dazu wird die Gaußfunktion

$$G(x,y) = \frac{1}{2\pi\sigma^2}\, e^{-\frac{x^2+y^2}{2\sigma^2}}$$

auf das Bild angewendet wird.

Der darin enthaltene Parameter σ steuert den Grad der Rauschunterdrückung. Die Faltung des ursprünglichen Grauwertbilds f(x, y) zum resultierenden geglätteten Bild f′(x, y) kann also durch folgende Gleichung dargestellt werden:

$$f_1(x,y) = f(x,y) * G(x,y)$$

Daraus resultier:

Grau Bild

Bild aus Gauß-Filter

Implementierung:

```
@cuda.jit
def calc_gauss(data, gauss_filter, gauss_img):
    row, column = cuda.grid(2);

    filter_size = gauss_filter.shape[0]
    half_filter_size = int(filter_size/2)

    height = data.shape[0]
    width = data.shape[1]

    if(row < height and column < width):
        pixel_sum = 0
        counter = 0

        for ky in range(0, filter_size):
            for kx in range(0, filter_size):

                calc_row    = row    - half_filter_size + ky
                calc_column = column - half_filter_size + kx

                if( calc_row >= 0 and calc_row < height and calc_column >= 0 and calc_column < width):
                    pixel_sum += data[calc_row,calc_column] * gauss_filter[ky,kx]
                    counter += gauss_filter[ky,kx]

        gauss_img[row,column] = pixel_sum / counter
```

Jeder Pixel, der auf dem einen Thread verarbeitet wird, wird mit den umgebenden Pixeln unter Verwendung Gauß-Filter zum Unschärfewert berechnet.

Gauß-Filter basiert auf 5x5:

$$\frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

Mögliche Optimierung:

Anstatt Schleifen könnte man so viele Threads wie gesamte Anzahl der Schleifen ausführen. Dabei was die mögliche Problemstellung wäre ist wie man dafür einen Algorithmus umsetzt dass alle Threads für den einen spezifischen Pixel verarbeiten lässt genau wie die Schleifen getan haben, und dabei muss Atomare Operation verwendet werden wobei Zugriff auf Daten sequenziell geschieht das aber den Vorteil der parallele Programmierung abziehen könnte.

5 Gradientenberechnung

Der Gradientenberechnung erkennt die Kantenintensität und -richtung, indem das Bild nach X und Y bzw. Horizontal und Vertikal partiell abgeleitet und in euklidischen Betrag zusammengeführt wird.

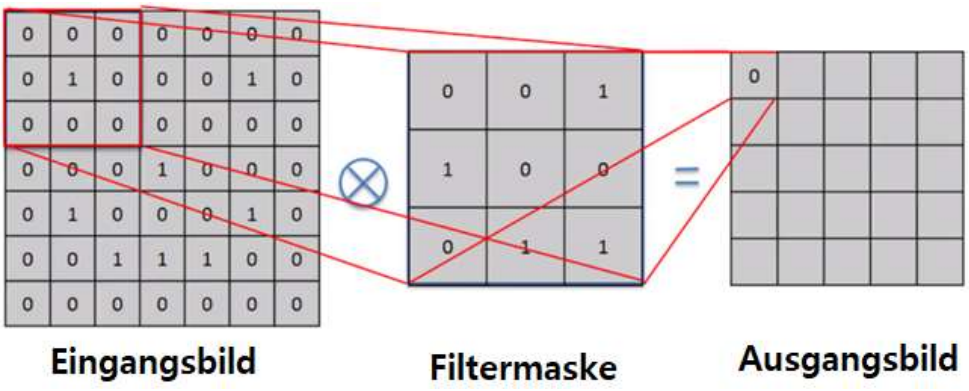
Als Ergebnis, Kanten entsprechen einer Änderung der Pixelintensität.

Um die Kanten durch Gradient zu erkennen, ist der so genannte Sobel-Filter anzuwenden, der die Intensitätsänderung in beiden Richtungen hervorhebt: horizontal (x) und vertikal (y):

$$K_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}, K_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}.$$

Sobel filters for both direction (horizontal and vertical)

Mit dem Sobel-Filter wird das Bild durch Faltung berechnet.



Bei Faltung wird jeder Pixel aus Eingangsbild mit dem Pixel auf gleicher Stelle aus Sobel-Filter multipliziert und dann die allen multiplizierten Pixel werden wiederum summiert.

Daraus ergeben sich Bild aus Gauß-Filter:

Bild aus Gauß-Filter

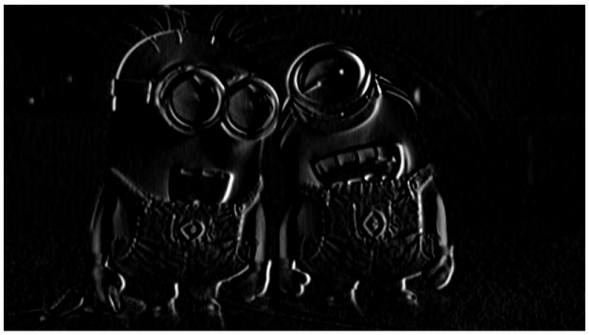


Bild aus Sobel-Kanten-x (Vertikal)

Bild aus Sobel-Kanten-y (Horizontal)

Danach werden Gradient bzw. Intensitätswert G und die Steigerung bzw. Richtung θ des Gradienten wie folgend berechnet:

$$|G| = \sqrt{I_x^2 + I_y^2},$$
$$\theta(x, y) = arctan\left(\frac{I_y}{I_x}\right)$$

Gradient intensity and Edge direction

Daraus resultiert:

Bild aus Gradient/Kanten Intensität

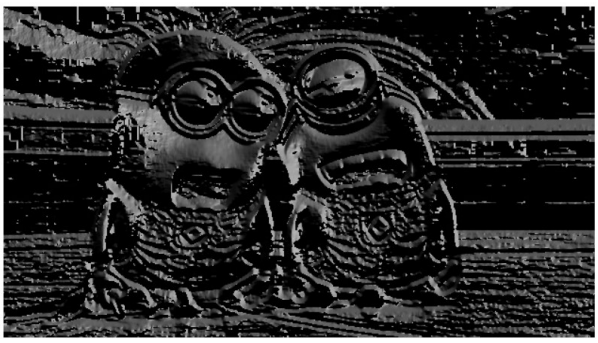


Bild aus Gradient/Kanten Richtung

Implementierung für Sobel-Filter:

```
@cuda.jit
def cal_sobel(inputImage, outputImage, filter):
    x,y = cuda.grid(2)

    if x < G_HEIGHT and y < G_WIDTH:
        convolution = 0
        for iterX in range(FILTER_SIZE_H):
            for iterY in range(FILTER_SIZE_W):
                convolution += inputImage[x+iterX][y+iterY] * filter[iterX][iterY]

        outputImage[x,y] = convolution
```

Für die Verarbeitung jeden Pixels kommt für jeweils ein Thread zu Berechnung und durch Doppel Schleifen wird die Faltung mit Sobel-Filter bzw. Maske berechnet.

Implementierung für Gradient/Kanten Intensität:

```
@cuda.jit
def cal_gradient(imgGx, imgGy, outputImage):
    x,y = cuda.grid(2)

    if x < G_HEIGHT and y < G_WIDTH:
        outputImage[x,y] = (imgGx[x,y]**2 + imgGy[x,y]**2)**0.5
```

Für jeden Pixel im resultierten Bild kommt für jeweils ein Thread zu Berechnung des Gradienten.

Implementierung für Gradient/Kanten Richtung:

```
@cuda.jit
def cal_angle(imgGx, imgGy, outputImage):
    x,y = cuda.grid(2)

    if x < G_HEIGHT and y < G_WIDTH:
        outputImage[x,y] = math.atan2(imgGy[x,y], imgGx[x,y])*180/3.14
```

Für jeden Pixel im resultierten Bild kommt für jeweils ein Thread zu Berechnung der Gradient Richtung.

Mögliche Optimierung bei Sobel-Kanten:

Anstatt Doppelschleifen könnte mit zusätzlich so vielen Thread Ausführungen wie Schleife Anzahl bessere Leistung kommen.

Aber mögliche Problemstellung wäre dass dafür Atomare Operation des Zugriffs auf Daten zum Einsatz kommen muss. Dabei ist es nicht sicher ob dies bessere Leistung als Nutzung von Doppelschleifen erbringen kann.

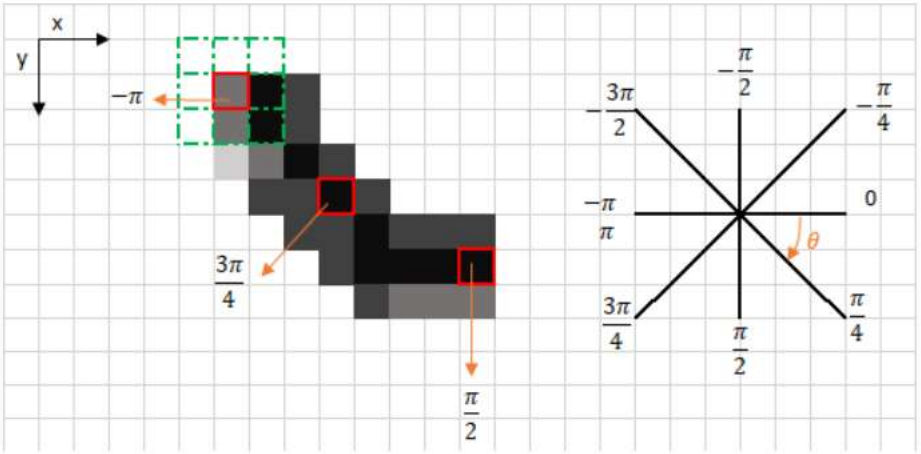
```
@cuda.jit
def cal_sobel(inputImage, outputImage, filter):
    x,y = cuda.grid(2)

    if x < G_HEIGHT and y < G_WIDTH:
        x = x - x%6
        y = y - y%6
        convolution = inputImage[x][y] * filter[x%6][y%6]
        cuda.atomic.add(outputImage, (x,y), convolution)
```

6 Nicht-Maximum Unterdrückung

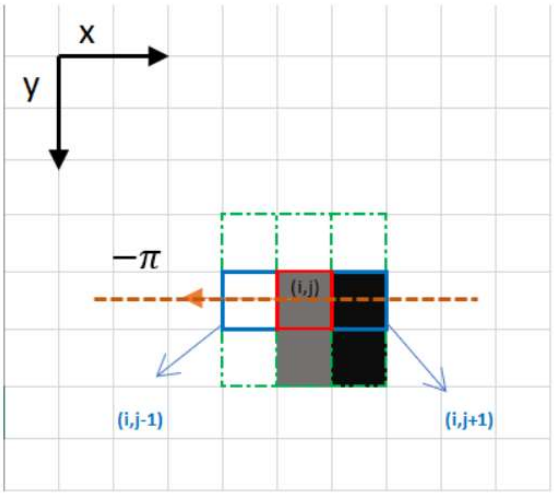
Idealerweise sollte das endgültige Bild dünne Kante haben. Daher muss Unterdrückung der nicht maximalen Pixelwerten durchgeführt werden, um die Kanten auszudünnen.

Der Algorithmus geht alle Pixel auf der Gradienten Intensitätsmatrix durch und findet Pixel mit dem nicht maximalen Wert in Kanten-Richtung.

Kanten-Richtungen sind erhältlich aus Gradient Richtung Matrix von dem vorherigen Schritt „Gradientenberechnung“.



Das rote Kästchen in der oberen linken Ecke des obigen Bildes stellt einen Intensitätspixel der Gradienten Intensitätsmatrix dar, die gerade verarbeitet wird. Die entsprechende Kantenrichtung wird durch den orangefarbenen Pfeil mit einem Winkel von Pi Bogenmaß (+/-180 Grad) dargestellt.



Die Kantenrichtung ist die orange gepunktete Linie (horizontal von links nach rechts). Zweck des Algorithmus besteht darin, zu prüfen, ob der verarbeitete Pixel mehr oder weniger intensiv ist als die benachbarten Pixel auf Kantenrichtung. Im obigen Beispiel wird der Pixel (i, j) verarbeitet und die Pixel auf Kantenrichtung werden blau hervorgehoben (i, j-1) und (i, j+1). Wenn eines dieser beiden Pixel intensiver bzw. höher ist als das verarbeitete, dann wird das verarbeitete auf 0 gesetzt (Nicht-Maximum Unterdrückung). Pixel (i, j-1) scheint intensiver zu sein, weil es weiß ist (Wert von 255). Daher wird der Intensitätswert des aktuell verarbeiteten Pixels (i, j) auf 0 gesetzt. Wenn auf Kantenrichtung keine

benachbarten Pixel mit intensiveren bzw. höheren Werten als das verarbeitete existiert, wird der Wert des aktuell verarbeiteten Pixels beibehalten.

Daraus resultiert:

Bild aus Gradient/Kanten Intensität

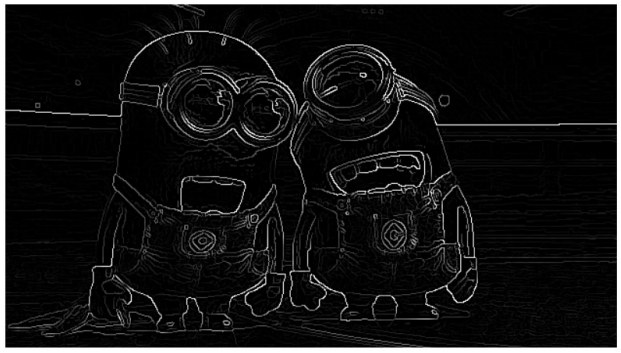


Bild aus Nicht-Maximum Unterdrückung

Implementierung für Nicht-Maximum Unterdrückung:

```
@cuda.jit
def cal_non_max_suppression(gradientImage, angleImage, outputImage):
    x,y = cuda.grid(2)
    if x < HEIGHT and y < WIDTH:
        target = gradientImage[x,y]
        angle = angleImage[x,y]

        left, right = getLeftRight(angle, x, y, gradientImage)

        if (target >= right) and (target >= left):
            outputImage[x,y] = target
        else:
            outputImage[x,y] = 0
@cuda.jit(device=True)
def getLeftRight(angle, x, y, gradientImage):
    right = None
    left = None
    #angle 0
    if (-22.5 <= angle <= 22.5) or (157.5 <= angle or angle <= -157.5):
        if y+1 < WIDTH:
            right = gradientImage[x, y+1]
        if y-1 >= 0:
            left = gradientImage[x, y-1]
    #angle 45
    elif (22.5 < angle < 67.5) or (-157.5 < angle < -112.5):
        if x+1 < HEIGHT and y+1 < WIDTH:
            right = gradientImage[x+1, y+1]
        if x-1 >= 0 and y-1 >= 0:
            left = gradientImage[x-1, y-1]
    #angle 90
    elif (67.5 <= angle <= 112.5) or (-112.5 <= angle <= -67.5):
        if x-1 >= 0:
            right = gradientImage[x-1, y]
        if x+1 < HEIGHT:
            left = gradientImage[x+1, y]
    #angle 135
    elif (112.5 < angle < 157.5) or (-67.5 < angle < -22.5):
        if x-1 >= 0 and y+1 < WIDTH:
            right = gradientImage[x-1, y+1]
        if x+1 < HEIGHT and y-1 >= 0:
            left = gradientImage[x+1, y-1]

    return left, right
```

Jeder Pixel hat 2 Hauptkriterien (Kantenrichtung und Pixelintensität (zwischen 0–255)). Basierend auf diesen Eingaben erfolgt Verfahren der Nicht-Maximum Unterdrückung:

1. Erstellen eine auf 0 initialisierte Matrix mit der gleichen Größe wie die ursprüngliche Gradienten Intensitätsmatrix.
2. Identifizieren die Kantenrichtung basierend auf dem Winkelwert aus der Gradientenrichtung bzw. Kantenrichtung Matrix.
3. Überprüfen, ob ein Pixel auf Kantenrichtung eine höhere Intensität hat als der aktuell verarbeitete Pixel.
4. Setzen den Intensitätswert des aktuell verarbeiteten Pixels auf 0 wenn ein Pixel mit einem höheren Intensitätswert auf Kantenrichtung vorhanden ist. Sonst behalten den aktuellen Intensitätswert beim aktuell verarbeiteten Pixel.

7 Doppelter Schwellwert Verfahren

Es ist festzustellen, dass das Ergebnis der Nicht-Maximum Unterdrückung nicht perfekt ist, einige Kanten möglicherweise nicht echte bzw. nicht kantenrelevant sind und das Bild etwas Rauschen aufweist. Dafür sorgt Doppelter Schwellwert Verfahren um dies zu filtern.

Doppelter Schwellwert Verfahren zielt darauf ab, drei Arten von Pixeln zu identifizieren: stark relevant, schwach relevant und nicht relevant:

Stark relevante Pixel sind die Pixel, deren Intensität so hoch ist, dass man sicher ist, dass diese zur endgültigen Kante beitragen.

Schwach relevante Pixel sind die Pixel, die einen Intensitätswert haben, der nicht ausreicht, um als stark kantenrelevant anzusehen, aber noch nicht klein genug, um als nicht relevant für die Kantenerkennung anzusehen.

Sonstige Pixel werden als nicht relevant für die Kantenerkennung festgelegt.

Doppelter Schwellwert Verfahren gilt dafür:

- Der hohe Schwellwert wird verwendet, um die stark relevanten Pixel zu identifizieren (Intensitätswert höher als der hohe Schwellwert).
- Niedriger Schwellwert wird verwendet, um die nicht relevanten Pixel zu identifizieren (Intensitätswert kleiner als der niedrige Schwellwert)
- Alle Pixel mit einem Intensitätswert zwischen beiden Schwellwerten werden als schwach relevant gehalten, und der Hysterese-Mechanismus (nächster Schritt) sorgt dafür, diese endgültig festzulegen ob sie als stark oder als nicht relevant übernommen werden.

Daraus resultiert:

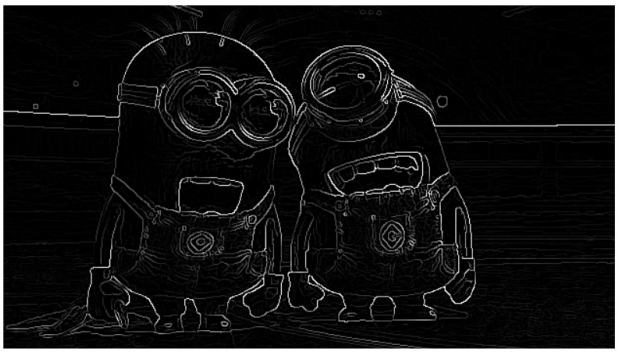


Bild aus Nicht-Maximum Unterdrückung

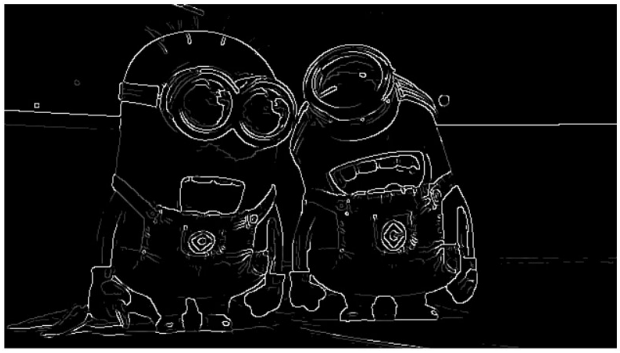


Bild aus Doppelter Schwellwert Verfahren

Implementierung für Doppelter Schwellwert Verfahren:

```
@cuda.jit
def cal_double_threshold(nonMaxSupImage, outputImage):
    x,y = cuda.grid(2)

    if x < HEIGHT and y < WIDTH:
        target = nonMaxSupImage[x,y]
        weightedTarget = nonMaxSupImage[x,y] * PIXEL_WEIGHT
        if weightedTarget > MAX:
            outputImage[x,y] = 255
        elif weightedTarget < MIN:
            outputImage[x,y] = 0
        else:
            outputImage[x,y] = target
```

Jeder Pixel wird untersucht ob größer als den hohen Schwellwert oder kleiner als niedrigen Schwellwert.

Wenn ein Pixel größer als den hohen Schwellwert ist, wird der Pixel auf maximalen Wert 255, Weiß Farbe, gesetzt.

Wenn ein Pixel kleiner als den niedrigen Schwellwert ist, wird der Pixel auf minimalen Wert 0, Schwarz Farbe, gesetzt.

Wenn weder größer als hohen Schwellwert noch kleiner als niedrigen Schwellwert, bleibt Pixel nicht geändert.

8 Kantenverfolgung durch Hysterese

Nachdem es festgelegt ist, welche Kanten stark kantenrelevant und schwach kantenrelevant sind, muss es jetzt identifiziert werden, welche schwach relevanten Kanten tatsächlich kantenrelevant sind. Dazu wird Kantenverfolgungsalgorithmus durchgeführt. Schwach relevante Kanten, die mindestens mit einer stark relevanten Kante benachbart sind, sind tatsächlich relevante Kanten. Schwach relevante Kanten, die mindestens mit einer stark relevanten Kante nicht benachbart sind, werden entfernt da sie tatsächlich nicht kantenrelevant sind.

Daraus resultiert:

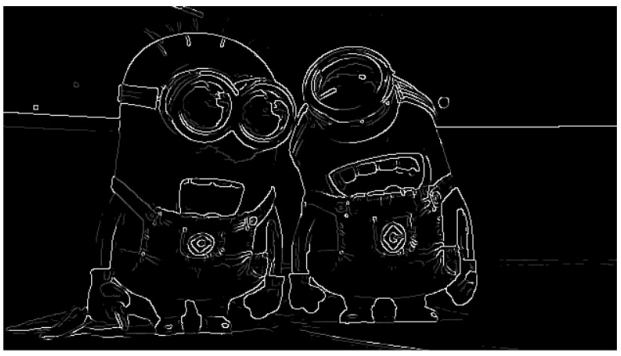


Bild aus Doppelter Schwellwert Verfahren

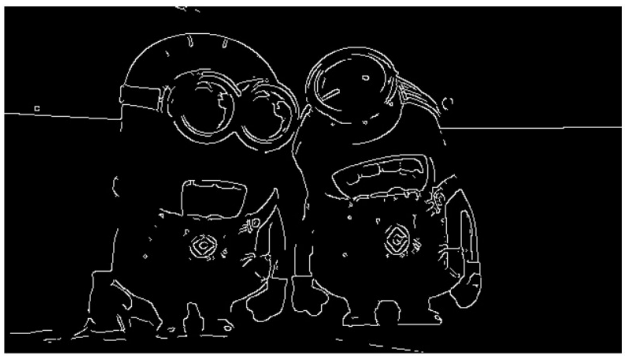


Bild aus Katenverfolgung durch Hysterese

Implementierung für Kantenrverfolgung durch Hysterese:

```
@cuda.jit
def cal_by_weak_edge_tracking_hysteresis(image):
    x,y = cuda.grid(2)
    strong = 255
    if (x < HEIGHT and y < WIDTH) and (0 < image[x,y] < strong):
        toBeFinished = False
        iterRange = cuda.local.array(3, int32)
        iterRange[0] = -1
        iterRange[1] = 0
        iterRange[2] = 1
        for iX in iterRange:
            if 0 <= x+iX < HEIGHT:
                for iY in iterRange:
                    if 0 <= y+iY < WIDTH:
                        if iX != 0 and iY != 0:
                            if image[x+iX, y+iY] == strong:
                                image[x,y] = strong
                                toBeFinished = True
                    if toBeFinished:
                        break
            if toBeFinished:
                break
```

Jeder als schwach relevant festgestellte Pixel wird untersucht, ob dies mindestens mit einer stark relevanten Kante bzw. Pixel benachbart ist.

Wenn mindestens ein stark relevanter Pixel an dem jetzt verarbeiteten Pixel benachbart ist, wird der jetzt verarbeitete Pixel auf den Intensitätswert 255, Weiß Farbe, umgesetzt.

Dieses Vorgehen kann so viel wie wollen mehrmals wiederholt werden um mehr Kanten auf hoher Präzision herauszufinden.

Danach findet Aufräumung statt, in der alle Pixel kleiner als Intensitätswert 255 auf 0 gesetzt werden.

Mögliche Optimierung:

Anstatt Schleifen könnte man so viele Threads wie gesamte Anzahl der Schleife ausführen, wobei die mögliche Problemstellung wäre wie man dafür einen Algorithmus umsetzt, dass es jeweils 8 Threads für den einen spezifischen Pixel verarbeiten lässt, genau wie die Schleife getan hat.

9 Zusammenführung aller Module: CANNY KANTENERKENNUNG

Zur Bildung des vollständigen Canny Kantenerkennung Algorithmus müssen alle Module zu Einheit integriert werden.

Das Ergebnis von vorherigem Modul dient der Eingabe für das nächste Modul.

Um Daten I/O Overhead zu reduzieren, dienen Ergebnis Daten aus GPU direkt zur Eingabe für das nächste Modul.

Implementierung:

```
if __name__ == "__main__":
    export_path = os.path.join(os.path.dirname(os.path.abspath(__file__)), "result_images")
    try:
        os.mkdir(export_path)
    except OSError as error:
        pass

    gray_img = Gray.make_gray_image(IMAGE_PATH, EXPORT_IMAGE)
    log("Gray Step is done.")
    gauss_img = gauss.make_gauss_image(gray_img, IMAGE_PATH, EXPORT_IMAGE)
    log("Gaussian Blur Step is done.")
    gradient, angle  = Gradient.make_gradient_and_angle(gauss_img, EXPORT_IMAGE)
    log("Gadient Step is done.")
    nonMaxSup_img = NonMaxSuppression.make_non_max_suppression(gradient, angle, EXPORT_IMAGE)
    log("Non Maximum Suppression Step is done.")
    doubleThreshold_img = DoubleThreshold.make_double_threshold(nonMaxSup_img, EXPORT_IMAGE)
    log("Double Threshold Step is done.")
    result = EdgeTrackingHysteresis.make_edge_tracking_hysteresis(doubleThreshold_img, EXPORT_IMAGE)
    log("Edge Tracking By Hysteresis Step is done.")

    log("Creating Canny Edge Image...")
    cannyEdgeImage = result.copy_to_host()

    imageName = os.path.join(export_path, "result_canny_edge.jpg")
    try:
        import cv2
        cv2.imwrite(imageName, cannyEdgeImage)
    except:
        image.store_img_bw(cannyEdgeImage, imageName)

    log("Canny Edge Image created: {}".format(imageName))
```

9.1 Messen der Leistung

Für ein Bild mit Größe 354 x 630 Pixel:

	Quadro M1200 4GB	Tesla V100 SXM2 32GB	OpenCV auf CPU Intel core i7-6820HQ
Umwandlung in Graubild	0,99 ms	0,25 ms	-
Gauß-Filter	2,12 ms	0,59 ms	-
Gradientenberechnung	8,16 ms	3,25 ms	-
Nicht-Maximum Unterdrückung	0,72 ms	0,26 ms	-
Doppelter Schwellwert Verfahren	0,53 ms	0,25 ms	-
Kantenverfolgung durch Hysterese	0,71 ms	0,35 ms	-
Gesamte Rechenzeit	13,23 ms	4,95 ms	16,1 ms
Sonstige Overhead	21,35 ms	13,06 ms	-
Gesamte Verarbeitungszeit	34,58 ms	18,01 ms	16,1 ms

10 Optimierung

10.1 Gradientenberechnung

Alle Teile des Algorithmus (Faltung durch Sobel-filter X/Y, Berechnung der Gradienten und der Gradientenrichtungen) sind in eine Einheit zusammengeführt.

Das heißt, jeder Pixel wird auf einmal in einem Thread durch alle Teilalgorithmen berechnet, während vor Optimierung jeder Pixel jeweils in vier Teilen des Algorithmus getrennt bei verschiedenen Threads sequentiell berechnet ist.

Implementierung:

```
@cuda.jit
def cal_gradient_n_angle(inputImage, outputGradient, outputAngle, filterX, filterY):
    x,y = cuda.grid(2)

    if x < G_HEIGHT and y < G_WIDTH:
        gX = make_convolution(x, y, inputImage, filterX)
        gY = make_convolution(x, y, inputImage, filterY)

        cal_gradient(x, y, gX, gY, outputGradient)
        cal_angle(x, y, gX, gY, outputAngle)

@cuda.jit(device=True)
def make_convolution(x, y, inputImage, filter):
    if x < G_HEIGHT and y < G_WIDTH:
        convolution = 0
        for iterX in range(FILTER_SIZE_H):
            for iterY in range(FILTER_SIZE_W):
                convolution += inputImage[x+iterX][y+iterY] * filter[iterX][iterY]

    return convolution

@cuda.jit(device=True)
def cal_gradient(x, y, gX, gY, outputImage):
    if x < G_HEIGHT and y < G_WIDTH:
        outputImage[x,y] = (gX**2 + gY**2)**0.5

@cuda.jit(device=True)
def cal_angle(x, y, gX, gY, outputImage):
    if x < G_HEIGHT and y < G_WIDTH:
        outputImage[x,y] = math.atan2(gY, gX)*180/3.14
```

Auswirkung:

Rechenzeit der Gradientenberechnung für ein Bild mit Größe 354 x 630 Pixel

	Quadro M1200 4GB	Tesla V100 SXM2 32GB
Vor Optimierung	8,16 ms	3,25 ms
Nach Optimierung	4,93 ms	1,79 ms

10.2 Doppelter Schwellwert Verfahren

Durch Beseitigung des Overhead für Ausgangsdaten reduziert sich die gesamte Verarbeitungszeit des Algorithmus.

Jetzt ist keine Allokation in GPU und im Anschluss keine Kopierung an GPU vorgenommen, das zu Übernahme der Ausgangsdaten diente.

Implementierung:

```
def make_double_threshold(nonMaxSupImage, exportImage=False):
    if exportImage:
        global EXPORT_PATH
        EXPORT_PATH = os.path.join(os.path.dirname(os.path.abspath(__file__)), "exported_images")
        try:
            os.mkdir(EXPORT_PATH)
        except OSError as error:
            pass

    global HEIGHT
    global WIDTH
    HEIGHT = nonMaxSupImage.shape[0]
    WIDTH = nonMaxSupImage.shape[1]

    threadsPerBlock = (32, 32)
    blocks = (HEIGHT//threadsPerBlock[0] + 1, WIDTH//threadsPerBlock[1] + 1)

    # Berechnung für Double Threadshold
    cal_double_threshold[blocks, threadsPerBlock](nonMaxSupImage)

    outputImage = nonMaxSupImage

    cuda.synchronize()

    if exportImage:
        imageDoubleThreashold = outputImage.copy_to_host()
        imageName = os.path.join(EXPORT_PATH, "DoubleThreashold.jpg")
        print("[INFO] Exporting image {}".format(imageName))
        try:
            import cv2
            cv2.imwrite(imageName, imageDoubleThreashold)
        except:
            image.store_img_bw(imageDoubleThreashold, imageName)

    return outputImage
```

Auswirkung:

Für ein Bild mit Größe 354 x 630 Pixel

	Quadro M1200 4GB	Tesla V100 SXM2 32GB
Gesamte sonstige Overhead	21,35 ms => 19,92 ms	13,06 ms => 11,85 ms
Gesamte Verarbeitungszeit	34,58 ms => 33,15 ms	18,01 ms => 16,8 ms

10.3 Kantenverfolgung durch Hysterese

Kantenverfolgung durch Hysterese inklusive Aufräumung Algorithmus, der nach Kantenverfolgung einmal noch ausgeführt wird, lässt sich mit Doppelter Schwellwert Verfahren integriert zusammen ausführen.

Dadurch erspart sich Rechenleistungsgebrauch zur Kernel Bereitstellung und von daher schließlich die gesamte Ausführungszeit.

Die theoretisch mögliche Gefahr ist, dass einige als schwach-relevant identifizierten Pixel nicht als kantenrelevant übernommen werden könnten, die aber eigentlich schon kantenrelevant gewesen wären.

Implementierung:

```
@cuda.jit
def cal_double_threshold_and_edge_tracking_hysteresis(nonMaxSupImage):
    x,y = cuda.grid(2)

    if x < HEIGHT and y < WIDTH:
        #Algorithmus für Doppelter Schwellwert Verfahren
        target = nonMaxSupImage[x,y]
        weightedTarget = nonMaxSupImage[x,y] * PIXEL_WEIGHT
        if weightedTarget > MAX:
            nonMaxSupImage[x,y] = 255
        elif weightedTarget < MIN:
            nonMaxSupImage[x,y] = 0
        else:
            nonMaxSupImage[x,y] = target

        #Algorithmus für Kantenverfolgung durch Hysterese mit integriertem Aufräumungsagorithmus
        cal_by_weak_edge_tracking_hysteresis(x, y, nonMaxSupImage)
```

Auswirkung:

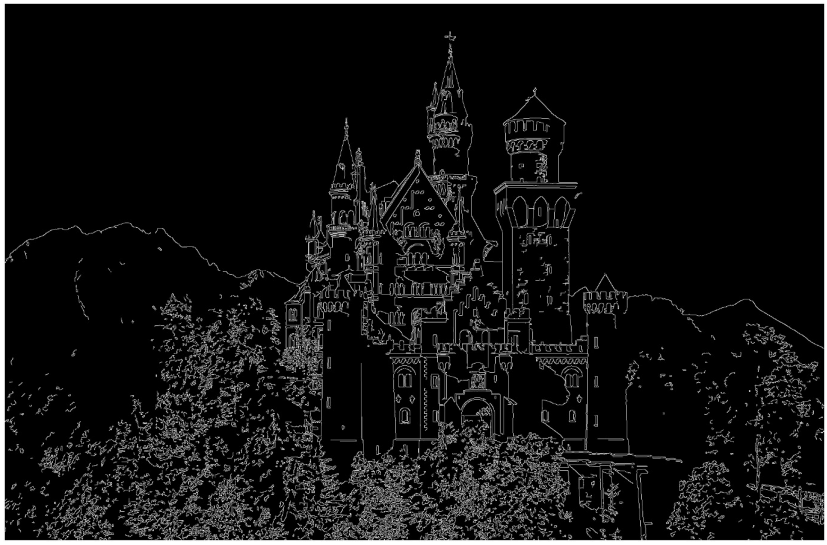
Für ein Bild mit Größe 354 x 630 Pixel

Quadro M1200 4GB	Vor Optimierung	Nach Optimierung
Doppelter Schwellwert Verfahren	0,53 ms	0,54 ms
Kantenverfolgung durch Hysterese	0,71 ms	
Total Rechenzeit	1,24 ms	0,54 ms

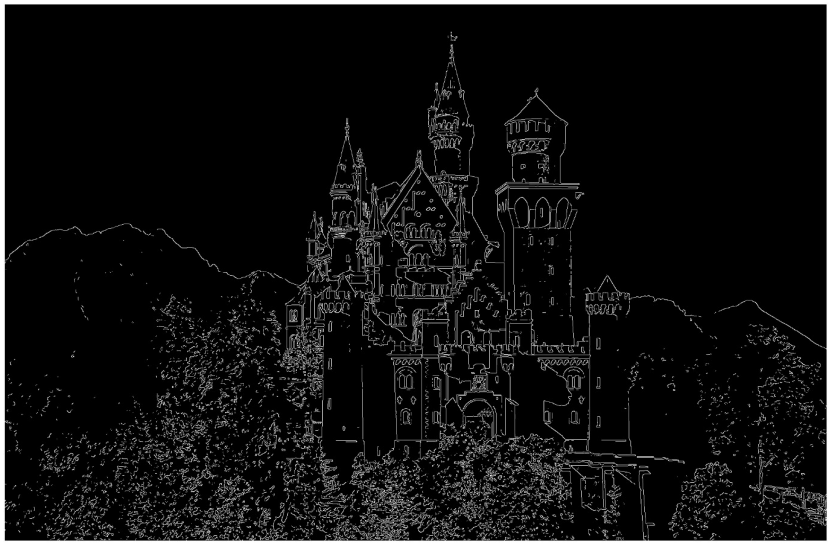
Tesla V100 SXM2 32GB	Vor Optimierung	Nach Optimierung
Doppelter Schwellwert Verfahren	0,25 ms	0,17 ms
Kantenverfolgung durch Hysterese	0,35 ms	
Total Rechenzeit	0,6 ms	0,17 ms

11 Vergleich mit Ergebnis aus OpenCV

In Kanten zu erkennendes Bild



Aus OpenCV



Aus Algorithmus mit Verwendung CUDA GPU