Parallelrechner Projekt “Histogram Equalization”

1 Grundlagen

1.1 Histograme

Ein Histogram h*(I)* eines Graustufenbildes ist eine Darstellung der Häufigkeit mit dem die Intensitätswerten bzw. Graustufen N_0 bis N_n auf dem Bild vorkommen. In diesem Projekt wird von 8-bit Pixeln ausgegangen, so dass Histograme einen Definitionsbereich von *0* bis *255* aufweisen.

Histogram-Operationen ermöglichen den Kontrast eines Bildes zu verbessern. Die erste dieser Operationen ist die Normalisierung, wodurch das Histogram “ausgedehnt und verchoben wird” [1] um die gesamte Breite der verfügbaren Grauwerten auszunutzen. Gegeben die minimalen und maximalen im Originalbild vorhandenen Graustufen O_{min} und O_{max}, die minimalen und maximalen verfügbare Intensitäten N_{min} und N_{max} können die normalisierte Grauwerte wie folgt berechnet werden:

$$I_{neu} = (I_{alt} - N_{min}) \cdot \left(\frac{N_{max} - N_{min}}{O_{max} - O_{min}}\right) + N_{min} \quad (1)$$

Unter Annahme von 8-bit Pixeln, also $N_{min} = 0$ und $N_{max} = 255$ wird die Berechnungsvorschrift zu:

$$I_{neu} = \frac{I_{alt} \cdot 255}{O_{max} - O_{min}} \quad (2)$$

ABBILDUNG 1: GVP-IMAGE + HISTOGRAM

Ein weiteres Verfahren zur verbesserung des Kontrastes ist die Histogramequalisierung, dessen Ziel ist, das Histogram zu glätten, so dass im Idealfall alle Grauwerte gleich häufig vorkommen. Somit werden “einige dunkle Werte heller gemacht, und einige helle Werte verdunkelt, während immernoch die gesamte Breite an Intensitäten genutzt wird” [2] . Als Abbildung zwischen den alten und den neuen Pixelwerten wird das kumulative Histogram (“cumulative distribution funtion, CDF”) verwendet:

$$c(I) = \frac{1}{M} \cdot \sum_{i=0}^{I} h(i) \quad (3)$$

wo *M* Gesamtzahl der Pixel im Bild ist. Da die CDF einen Werteberich zwischen 0 und 1 aufweist, musst sie auf dem Interval $[N_{min}, N_{nax}]$ (hier [0, 255]) skaliert werden:

$$I_{neu} = 255 \cdot c(I_{alt}) \quad (4)$$

ABBILDUNG 2: GVP-IMAGE + Histogram

1.2 Farbbilder und Farbräume

Bisher sind Graustufenbilder angenommen worden, also Bilder mit einem einzigen Kanal. Um die Normalisierung und Equalisierung auf Farbbilder (also mit 3 Kanälen) anzuwenden, wäre ein naheliegender Ansatz, die einzelnen Kanäle des RGB-Bilds jeweils zu normalisieren/equalisieren,

Dies würde allerdings zu einer Änderung des Farbabgleichs, also des Verhältnis der Farben im Bild zueinander (“color balance” [3]) führen. Dies kann durch eine Konvertierung des RGB-Bilds in einem anderen Farbraum gelingen, der die Helligkeit getrennt von der Farbe darstellt.

Beliebt ist der HSV-Farbraum (Hue, Saturation, Value), dass im Gegensatz zum eher Maschinen-orientierten RGB veruscht, die Farbkomponenten intuitver und verständlicher für die menschliche Vorstellung zu gestalten [1]. Hier dienen die H- und S-Komponenten (Farbton und Sätigung) zur Darstellung der Farben, während die V-Komponente die Information über die Helligkeit enthält, diese wäre für unsere zwecke besonders interessant.

Ohne in die Details der Herleitung einzugehen, stellt z.B. die OpenCV-Dokumentation eine Berechnungsvorschrift für die RGB zu HSV-Transformation zur Verfügung [4]:

$$V \leftarrow max(R,G,B) \quad (5)$$

$$S \leftarrow \begin{cases} \frac{V - min(R,G,B)}{V} & if\ V \neq 0 \\ 0 & else \end{cases} \quad (6)$$

$$H \leftarrow \begin{cases} \frac{60-(G-B)}{(V-min(R,G,B))} & if\ V = R \\ \frac{120+60(B-R)}{(V-min(R,G,B))} & if\ V = G \\ \frac{240+60(R-G)}{(V-min(R,G,B))} & if\ V = B \end{cases} \quad (7)$$

$$if\ H < 0 : H \leftarrow H + 360$$

$$mit\ H \in [0,360], S\ und\ V \in [0,1]$$

Auf dem ersten Blick ist es auffälig, dass diese Berechnungen einige sequentiellen Schritten beinhaltet, die der parallelen Durchführung im Wege stehen. Einerseits die Maximum- und Minimumsuche, auch wenn sie sich auf lediglich drei Werte beziehen. Für die Cuda-Implementierung sind aber die if-else Bedingungen problematischer, da sie zur Thread-Divergenz führen können. Es wäre schwer zu gewährleisten, dass alle Threads in einem Warp den gleichen Zweig der if-else-Verzweigung nehmen. Ist das nicht der Fall, werden die “if” und “else”-blocks nacheinander ausgeführt, dass zu einer weiteren Serialisierung der Berechnung beitragen würde [5].

Eine geeignete Alternative wäre ein Farbraum, der die Helligkeit getrennt von der Farbe darstellt aber gleichzeitig eine für Parallelisierung günstigere Konvertierung vom und in den RGB-Farbraum aufweist. Diese Voraussetzung erfüllen die YUV und YCbCr Farbräume, entwickelt jeweils für die analoge und digitale Bildübertragung [2]. YUV-/YcrCb- Bilder besitzen ein “luma”-Kanal (Y), dass die Helligkeitsinformation enthält, also das Bild als reines Graustufenbild. Die UV- bzw. CbCr- Kanäle tragen die Farbinformation (“chrominance”), “die Differenz zwischen der jeweiligen Farbe[blau und rot] und Weiß bei gleicher Helligkeit ” [[1], S. 579] .

Neben der Art und Weise, wie die UV- bwz. CrCb-Kanäle aus RGB berechnet werden, unterscheidet sich YcrCb von YUV in seiner größerer Eignung für die 8-bit Bilddarstellung.

Die Umrechnung $RGB \rightarrow YCbCr$ erfolgt durch die Gleichung:

$$YCbCr = M \cdot RGB + C \quad (8)$$

Mit den 3x1 Vektoren $YCrCb$ und RGB, der Konversionsmatrix

$$M = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.5 \\ 0.5 & -0.419 & 0.081 \end{pmatrix} \text{ mit Inverse } M = \begin{pmatrix} 1 & 0 & 1.402 \\ 1 & -0.344 & -0.714 \\ 1 & 1.772 & 0 \end{pmatrix}$$

und dem Korrekturvektor C

$$C = \begin{matrix} 0 \\ 128 \\ 128 \end{matrix}$$

bzw. die Rückkonvertierung $YCbCr \rightarrow RGB$:

$$RGB = M^{-1} \cdot (YCbCr - K) \quad (9)$$

Matrixprodukte, Vektorsumen und vor allem die Abwesenheit von if-else Verzweigungen machen diese Umrechnung wesentlich günstiger mit Hinblick auf Parallelisierung.

2 Implementierung (Code-Dokumentation)

Anhand der Aufgabenstellung und der Vorüberlegungen von vorherigen Abschnitt wird im folgenden die Implementierung der Farbtransformation sowie der Histogramnormalisierung und -equalisierung zuerst sequentiell in C++ und dann parallel in CudaC++ vorgestellt.

Auf die Verwendung externer Bibliotheken zum Einlesen/Speichern von Bildern wurde verzichtet und stattdessen die benötigten Funktionen in einer eigenen Klasse implementiert. Das Projekt gliedert sich grundsätzlichs in zwei Klassen:

2.1 Image-Klasse

Die ***Image*** Klasse dient zum Laden, Speichern und Zugreifen auf Bilder in den netpbm-Formate .pbm, .pgm, .ppm. Dabei gilt die Einschränkung, dass es von 8-bits pro Pixel pro Kanal (also 24 bits für Farbpixels) ausgegangen wird.

Eigenschaften

rows, cols, channels	Anzahl der Spalten, Zeilen und Kanäle
numValues	Anzahl der Stufen einer einzelnen Farbe/Farbkanals (i.d.R. 256)
colosSpace	Farbraum (RGB, HSV, YUV(eigentlich YcrCb)) bzw. Graustufenbild wenn nur ein Kanal vorhanden
Host_pixels, dev_pixels	Zeiger auf eindimensionalen byte-array mit den Pixelwerten im host-speicher und nur wenn CUDA-Methoden aufgerufen, im Speicher vom CUDA-Gerät
type (enum fileType)	Gibt das Format (.pbm, .pgm, .ppm oder nicht-Zugelassen) des Bildes an, sowei ob sich bei diesem um eine binär oder ASCII-Datei handelt
src	Datei-Pointer zur Qelldatei, aus dem das Bild ausgelesen werden soll

Konstruktoren

Image(string path)	Erstellt ein Image-Objekt aus der unter *path* gespeicherten Datei
Image(int rows, int cols, colorSpace cs, int numColors, fileType type)	Erstellt ein Image-Objekt mit den eingegebenen Eigenschaften, alle Pixel werden zu 0 gesetzt

Methoden

Get-Methoden	Get-Methoden für Anzahl der Zeilen, Spalten, Dateiformat, Farbstufen, Farbraum sowie Pixel-Zeiger (für host- und CUDA-Gerät)
getChannel(int c),	Liefert das *c*-te Bildkanal als Graustufenbild (.pgm Format)
setChannel(Image channel, int c)	Ersetzt die Pixel-Werte vom *c*-te Kanal mit dem Werten von *channel*, vorausgesetzt *channel* ist ein Graustufenbild der gleichen Dimensionen wie das Zielobjekt
load(string path)	Überschreibt das Image-Objekt mit der unter *path* gespeicherten Datei
save(string path)	Speichert das Bild unter den eigegebenen Dateipfad (Format wird automatisch eingestellt)
Farbraumkonvertierung	Siehe unten

Die Image-Klasse stellt zwei Farbraumkonvertierungen zur Verfügung:

- ***color2gvp*** : Farbbild zu Graustufenbild (Format und Kanalanzahl bleiben dabei unverändert). Für YCbCr Bilder reicht aus, das „Luma“-Kanal in die andere beide Kanäle zu Kopieren. Für RGB-Bilder wird zunächst das Y-Kanal berechnet und dieses in alle drei Kanäle eingesetzt
- ***rgb2yuv*** und ***yuv2rgb*** : RGB zu YCbCr bzw. YCbCr zu RGB entsprechend des Gleichungssystems aus dem Abschnitt ***1.2 Farbbilder und Farbräume.*** Zusätzlich werden alle Werte auf dem Bereich [0,255] eingeschränt („clamping“).

- Zusätzlich wurde eine RGB zu HSV (***rgb2hsv)*** Konvertierung mit der Berechnungsvorschrift aus [4] zum Testen implementiert, hat aber für die Durchführung der Histogramoperationen keine Bedeutung.

Die Farbkonvertierungen können sowohl auf der *host*-Maschine als auch auf dem CUDA-Gerät ausgeführt werden (die Methoden sind durch entsprechenden Präfix *host_* bzw. *dev_* gekennzeichnet). Zur Parallelisierung wurden Grid-Stride-Loops verwendet, so dass jeder Thread das Gleichungsystem höchstens n [1]mal ausführt.

Die RGB -YCbCr -Konvertierung wurde zusätzlich mit *pinned-memory* sowie *unified-memory* implementiert.

2.2 Histogram-Klasse

Die *Histogram*-Klasse ist für das Erstellen und Verarbeiten von Histograme zuständig. Die Einschränkung auf 8-bits pro Kanal pro Pixel gilt weiterhin, somit kann davon ausgegangen werden, dass Histograme höchstens 256 einträge haben werden.

Eigenschaften

numValues	Anzahl der Histogrameinträge, i.d.R. 256
Host_values, dev_values	Zeiger auf int-array mit den Histogrameinträgen in *host*- bzw. CUDA-Gerät-Speicher
host_valuesCumulative, dev_valuesCumulative	Zeiger auf double-array mit den Einträgen des kumulativen Histograms in *host*- bzw. CUDA-Gerät-Speicher
host_lookUpTable, dev_lookUpTable	Zeiger auf byte-array mit den neuen Graustufen ($I_{neu,}$ siehe ***1.1 Hitsograme***) nach Normalisierung bzw. Equalisierung des Histograms
dev_pixels	Zeiger auf den byte-array mit den Pixelwerten im CUDA-Gerät-Speicher
MinValue, maxValue	Höchster und niedrigster Pixelwert im Bild, benötigt für die Normlisierung

Konstruktor

Histogram(Image& src, int host=0)	Erstellt das Histogram des eingegebenen Bildes; der *host*-Parameter bestimmt ob das Histogram auf dem CUDA-Gerät (0) oder auf der *host* Maschine konstruiert wird

Methoden

Neben einer Methode zur Darstellung des zuletz gespeicherten Histograms auf einem Stream und einer *save*-methode zum Speichern des Histograms und des kumulativen Histograms auf einer Textdatei bilden die *getHistogram-* , *normalize-* und *equalize*-Methoden den Hauptbestandteil der Histogram-Klasse. Analog zu den Farbraumkonvertieungen können diese Methoden sowohl auf dem *host*-Rechner als auch auf dem CUDA-Gerät ausgeführt werden (*host_* bzw. *dev_* -Präfix):

getHistogram*:* die serielle Konstruktion des Histograms bzw. kumulativen Histograms wird durch jeweils eine *for*-Schleife bewerkstellig. Zur Verallgemeinerung wird dafür nurr das erste Bildkanal betrachtet, Farbbilder werden zunächst vom RGB in den YcbCr-Farbraum konvertiert.

Die parallele Histogram-Konstruktion erfolgt mithilfe von shared-memory und atomic-Operationen in zwei Schritten (Kernels), nach dem in [6] vorgestellten Algorithmus:

1. Im ersten *Kernel* wird die Histogram-Konstruktion zwischen allen Blocks aufgeteilt, so dass jeder Block ein Teilhsitogram aus nur m Pixels[2] erstellt. Die blockweise Einteilung des Bilds ermöglicht,

---

1 $n = \frac{Zeilen \cdot Spalten}{Blockgröße \cdot Gridgröße}$

2 $m = \frac{Zeilen \cdot Spalten}{Gridgröße}$

die Teilhistograme im geteilten Speicher (*shared memory*) zu speichern und erst am Ende des *Kernels* im globalen Speicher zu laden.

2. Im zweiten *Kernel* werden die Teilhistograme zusammegebracht. Jeder Thread ist für ein Histogrameintrag/Graustuge zuständig und „sammelt“ die Werte dieses Eintrags über alle teilhsitograme, um sie dann im globalen Speicher zu laden.

Die große Herausforderung stellt allerdings die parallele Berechnung des kumulativen Histograms dar. Allgemeinbetrachtet handelt es sich hier um eine Präfixsumme. Ein Ansatz sowie Code-Beispiele zur Lösung dieses Problems stellt [7] vor. Mit wenigen Anpassungen kann damit die Berechnung des kumulativen Histograms erfolgen.

Die Präfixsumme einer Liste von *k*-Zahlen (*k* ist eine zweier Potenz) kann in zwei Schritten berechnet werden:

1. Up-Sweep: die Elemente der Liste werden iterativ paarweise addiert. In der ersten Iteration finden *k/2* Additionen statt und alle Elementen mit ungeradem Index werden durch die Summe mit ihrem Vorgängerelement ersetzt. Bei der nächsten Iteration werden die Summen aus den vorherigen Schritt miteinaner addiert (es finden also $k/2^2$ Summen statt) usw. bis zur *ld(k)*-te Iteration. Am Ende dieses Schrittes steht die Summe aller Elementen an der letzten Stelle der Liste (siehe Abb. 1).

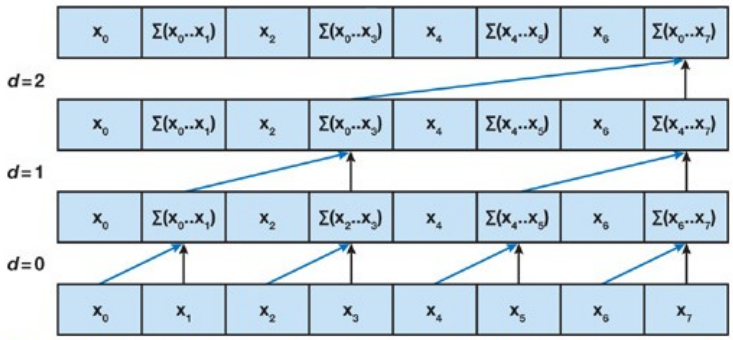

Abbildung 1: *Up-Sweep Schritt des Algorithmus zur Berechnung der Präfixsumme [7]*

2. Down-Sweep: Zur Beginn dieses Schrittes wird die Gesamtsumme im letzten Element durch *0* ersetzt, und die im vorherigen Schritt entstandene Baumstruktur von oben nach unten durchgegangen. „Dabei leitet jeder Knoten die *0* auf sein linkestes Kind und die Summe seines Wertes und des Wertes seines linken Kindes zu seinem rechten Kind“ [7]. Nach *ld(k)* Iterationen liegt die Präfixsumme der ersten *k-1* Elementen vor, mit 0 an der ersten stelle (siehe Abb. 2). Zum Schluss müssen alle Elemente um eine Stelle nach links verschoben worden, und an letzter Stelle die Summe aller Elementen aus dem Schritt 1 eingesetzt.

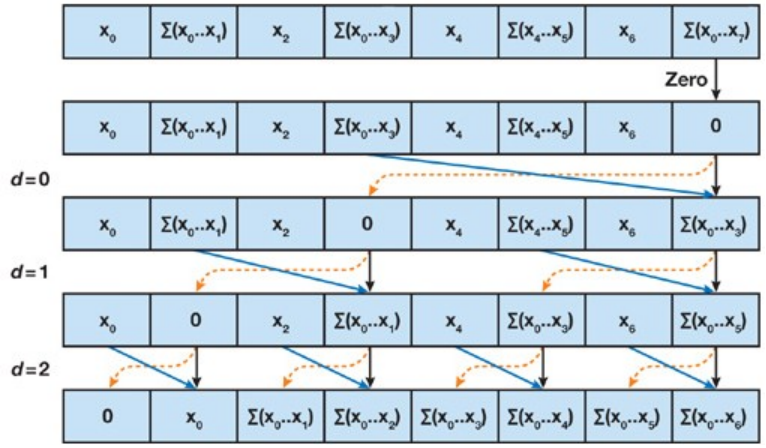

Abbildung 2: *Down-Sweep Schritt der parallelen Präfixsumme aus [7]. Man Merke, nach der letzten Iteration muss die Summe um ein Element nach links gerückt werden*

Dieser Vorgang kann in mehreren Teilsummen aufgeteilt werden, die von jeweils einem Block auf geteilten-Speicher ausgeführt werden. Die Teilsummen werden in einem Hilfsfeld eingetragen und von dem wiederum die Präfixsumme gebildet. Diese dient als Puffer, um aus den Werten der Teilsummen die Gesamtsumme herzuleiten (siehe Abb. 3).

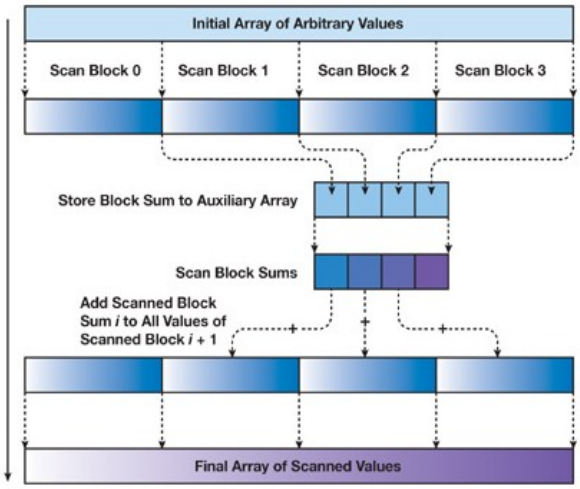

Abbildung 3: *Erweiterung der parallelen Präfixsumme auf Felder beliebiger Elementenanzahl [7]*

Normalize und equalize: Zur Normalisierung und Equalisierung des Histograms wird für jede Graustufe I_{alt} die neue, I_{neu}, gemäß Gleichung (2) bzw. (4) berechnet, womit jeweils eine *LookUp*-Tabelle entsteht.

Für beide Klassen gilt, dass die *dev*-Methoden nicht die parallele Implementierung an sich, sondern lediglich die Speicherzuweisung, Kernel Aufruf und die Messung der Ausführungszeit mittels CudaEvents beinhalten. Die Kernel-Implementierung erfolgt separat auf der entsprechenden .cu Datei.

3 Benchmarking

Zu Benchmarking wurden 21 Farbbilder und 21 Graustufenbilder verschiedener Dimensionen ausgewertet. Einerseits wurde die Korrekte Implementierung der Bildverarbeitung getestet (*EqualizationTest.cu*) und die Histogramoperationen mit Farbraumkonvertierung sowie mit getrennter Equalisierung der R-,G- und B-Kanäle durchgeführt (siehe ordner Bechmark).Anschließend wurde die Performance der Cuda-Implementierung anhand von zwei Parametern untersucht:

- Speedup: Geschwindigkeitsvorteil der CUDA-Implementierung gegen die serielle Variante auf der CPU bei gleicher Last. Der Benchmark wurde auf einer Nvidia GeForce 840M mit 3 Multiprozessoren, eine Obergrenze von 2048 Threads pro Multiprozessor und einer theoretischen Speicher Bandbreite von 14,4 Gbits/s. Zum Vergleich wurde eine Intel i7 5500U-CPU eingesetzt. Die Ergebnisse sind auf Abb. 4 dargestellt
- Außerdem wurde die Ausführungszeit der Benchmarks für verschiedenen Kernel-Konfigurationen untersucht (siehe Abb. 5). Um einer kompletten Hardware-Auslastung näher zu kommen wurden die Berechnungen mit der maximal erlaubten Anzahl an Threads (6144) durchgeführt.
- Zusätzlich wurde der Einfluss von *pinned-memory* und *unified-memory* beobachtet, dies führte allerdings zur wirderwarteten Ergebnissen, da die Ausführungszeit der Kernel stieg (besonders dramatisch im Fall der *unified memory*)

Die CUDA-Variante ist in der Regel 15 bis 20 mal schneller als die seriellen Algorithmen auf der CPU, dabei tragen die paralleliserten Algorithmen der Histogram-Klasse (und ihre Ausführung auf *shared-memory*) stärker als die „brute force“- Parallelisierung der Farbtransformationen zum Geschwindigkeitsgewinn bei.

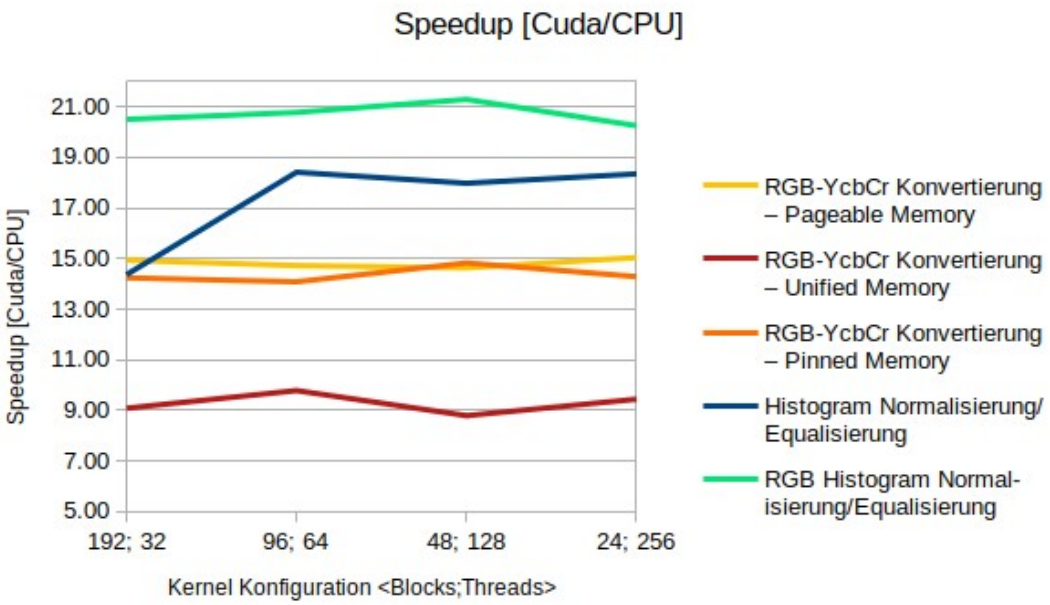

Abbildung 4: *Geschwindigkeitsvorteil der Parallelen CUDA-Ausführung gegenüber die seriellen Varianten auf der CPU*

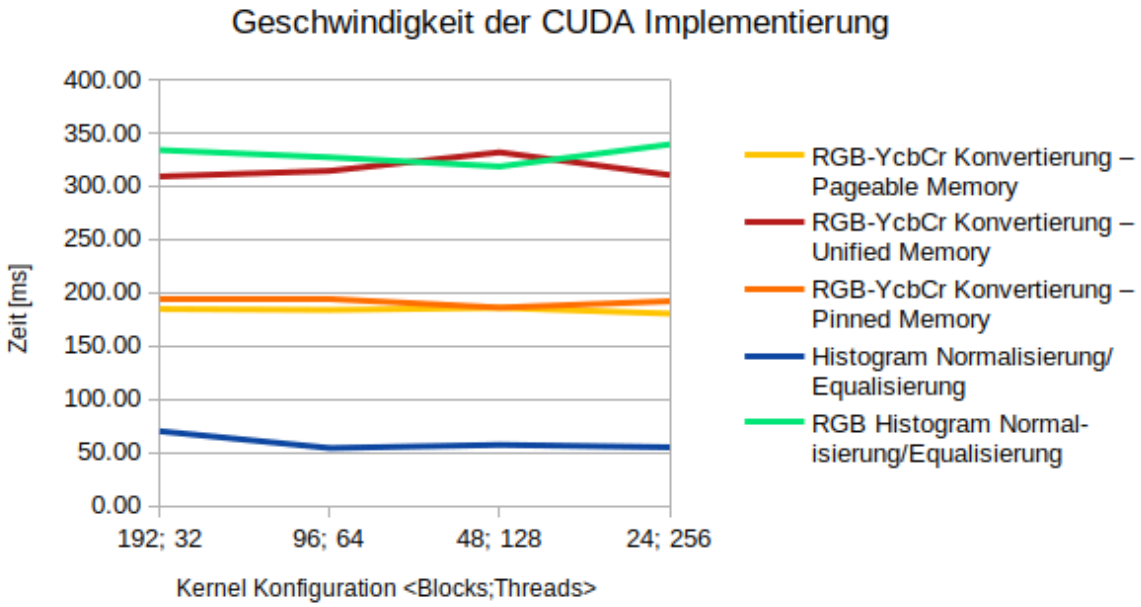

Abbildung 5: *Geschwindigkeit der Cuda-Implementierung bei verschiedenen Kernel-Konfigurationen*

Quellenverzeichnis

1: Mark S. Nixon, Alberto S. Aguado, Feature Extraction & Image Processing for Computer Vision, 2012
2: Richard Szeliski, Computer Vision: Algorithms and Applications, 2011
3: Histogram equalization, , https://en.wikipedia.org/wiki/Histogram_equalization
4: RGB ↔ HSV, 2020, https://docs.opencv.org/3.4/de/d25/imgproc_color_conversions.html
5: Introduction to GPGPU and CUDA Programming: Thread Divergence, 2013,
https://cvw.cac.cornell.edu/gpu/thread_div
6: GPU Pro Tip: Fast Histograms Using Shared Atomics on Maxwell, 2015,
https://developer.nvidia.com/blog/gpu-pro-tip-fast-histograms-using-shared-atomics-maxwell/
7: Nvidia, Kurt Akeley, Huber Nguyen, GPU Gems 3: Programming Techniques for High-Performance Graphics and General-Purpose Computation, 2007