

Echtzeit-Hintergrundadaption für 3D-Personentracking

Studienarbeit
von

Fang Li

An der Fakultät für Informatik
Institut für Anthropomatik
Group On Human Motion Analysis

Erstgutachter: Prof. Dr.-Ing. Rainer Stiefelhagen
Betreuer: Dipl. Inf. Tobias Feldmann

21. Juni 2010 – 21. September 2010

Fang Li
Insterburgerstrasse 2
76139 Karlsruhe

Hiermit erkläre ich an Eides statt, dass ich die von mir vorgelegte Arbeit selbstständig verfasst habe, dass ich die verwendeten Quellen, Internet-Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Karlsruhe, den 21. September 2010

(Unterschrift)

Fang Li

Inhaltsverzeichnis

Abbildungsverzeichnis	5
1 Einführung	9
1.1 Motivation, Zielsetzung und Beitrag der Arbeit	9
1.2 Gliederung der Arbeit	10
2 Stand der Technik	11
2.1 Satz von Bayes	11
2.2 Segmentierung im Bereich Bildverarbeitung	12
2.2.1 Verfahren	13
2.2.2 Optimierung der Segmentierung	13
2.3 GPU Berechnung	13
2.3.1 Anwendungsgebiete	14
2.3.2 Vorteile	14
2.3.3 Nachteile	14
2.3.4 Wahrscheinlichkeit auf GPU berechnen	15
2.4 Vorarbeiten zur Vorder-/Hintergrund-Segmentierung	15
2.4.1 Hintergrundmodell	17
2.4.2 Verarbeitung Illuminanz und Chrominanz	18
2.4.3 Vordergrundmodell	18
3 Grundlagen	21
3.1 Likelihood und Normalverteilung	21
3.2 GMM (Momentenmethode)	22
3.3 Farbräume	22
3.4 OpenCV	24
3.5 Compute Unified Device Architecture (CUDA)	24
4 Umsetzung	27
4.1 Analysen der Algorithmen	27
4.2 Initialisierung	27
4.3 CUDA Modulentwicklung	29
4.3.1 Kernel	29
4.3.2 Thread Hierarchie	31
4.3.3 Speicherzugriff	32
4.4 Modul-Integration	32
5 Experimente und Auswertungen	35

5.1	Geschwindigkeitsmessung zur Datenübertragung zwischen CPU und GPU . . .	35
5.2	Zeitmessung zur Arithmetik der Gleitkommazahl-Matrizen	36
5.3	Zeitmessung zum Modul Gauss-Mischverteilung im Hintergrundmodell	38
5.3.1	Zeitmessung zum Prototyp des CUDA-Moduls	38
5.3.2	Zeitmessung zum integrierten Modul	39
5.3.3	Offene Fragen und ein kurzer Ausblick über weitere Optimierung . . .	42
6	Zusammenfassung	43

A Beispielanwendung

45

Literatur

47

Abbildungsverzeichnis

1.1	Unterschiedliche Strukturen zwischen CPU und GPU Quelle: [CUDAGuide] . . .	10
2.1	Segmentiertes Beispielbild Quelle: [HornKorsche04]	13
2.2	Floating-Point Operationen pro Sekunde und Bandbreite des Speichers für CPU und GPU Quelle: [CUDAGuide]	15
2.3	Ablauf eines Segmentierungsschrittes Quelle: [Feldmann2009AFS]	16
2.4	Modellierung und Segmentierung des Hintergrundes Quelle: HumanEva	17
3.1	Dichtfunktion der Normalverteilungen $\mathcal{N}(0,1)$ (blau), $\mathcal{N}(0,2)$ (grün) und $\mathcal{N}(-1,2)$ (rot) Quelle: [WikiGauss]	22
3.2	Darstellung der Farbräume RGB und HSV Quelle: [Latsch08]	23
3.3	YUV-Farbemodell, Y-Wert=0,5 Quelle: [WikiYUV]	24
3.4	CUDA Verarbeitungsablauf Quelle: [WikiCUDA]	25
4.1	CUDA Thread Hierarchie Quelle: [CUDAGuide]	28
4.2	CUDA-SDK	29
4.3	Pixelweise Berechnung auf CPU	33
4.4	Gaussmixture auf GPU (3 Komponenten)	34
5.1	Zeitmessung zur Bildübertragung(NV Geforce 9800GT mit 512MB)	36
5.2	Matrizen Multiplikation: Quellmatrizen A, B; Zielmatrix C	37
5.3	Zeitmessung zur Arithmetik der Gleitkommazahl-Matrizen	37
5.4	Gauss-Mischverteilung auf GPU	38
5.5	Zeitmessung des CPU- und GPU-Moduls im Debug-Modus	40
5.6	Zeitmessung des CPU- und GPU-Moduls im Release-Modus	41

Kapitel 1

Einführung

1.1 Motivation, Zielsetzung und Beitrag der Arbeit

Die Arbeit ist auf die Forschung zur makellosen Menschenbewegungsanalyse mit einem Multi-Kamerasystem im Bereich Bildverarbeitung konzentriert. Hierfür werden Videos eines Sportlers aus mehreren Perspektiven aufgenommen. Danach wird die Analyse ausgeführt, um die Pose anhand der Bilddaten möglichst genau zu schätzen. Hierin liegt ein grundlegendes Problem: Wie kann man den Sportler aus dem Hintergrund finden und heraustrennen? Das Ziel ist es, die Vordergrundobjekte in einer Bildsequenz zu finden. Für die Segmentierung ist die Erfassung im Hintergrund und Vordergrund erforderlich. Die gegenwärtige Segmentierung sind meistens durch bestimmte Bedingungen beschränkt. Zum Beispiel werden die Videos nur im statischen Labor(feststehender Hintergrund) mit kontrollierter Beleuchtung aufgenommen. In dieser Arbeit wird Multiview Silhouetten Fusion angewandt, damit kann die Segmentierung in instabiler Umwelt ausgeführt werden. Die Auswirkung der Beleuchtung zum Objekt wird als ein anderes Element nachgedacht.

Um eine bessere 3D-Rekonstruktion mit weniger Störungen zu erfüllen, wird die Informationen der 3D-Szene integriert werden. Wegen der relativ aufwändigen Berechnung läuft das Verfahren nach der Integration langsam. Das Ziel dieser Arbeit ist um eine Lösung mit guter Qualität und Geschwindigkeit der Segmentierung zu finden.

In der Forschung zu 3D-Personentracking über ein Mehrkamerasystem, sollen Massenbilder in Echtzeit parallel bearbeitet werden. Hauptsächlich in dem Bereich Hintergrundadaption ist die pixelbasierte Bilderbverarbeitung ein Prozess, der parallel und in Echtzeit verläuft. Jedes Pixel der Bilder ist unabhängig von den anderen und kann als ein Thread in einem parallelen Prozess angenommen werden. Die gesuchte Optimierung stammt aus der GPU-Technologie. Die Grafikkarte übernimmt die rechenintensiven Aufgaben der Bildverarbeitung. Dadurch wird der Hauptprozessor (CPU) entlastet. D.h. durch die Umsetzung der Algorithmen laufen die Verfahren deutlich schneller auf GPU im Vergleich zu dem gängigen Verfahren auf CPU. Der grundsätzliche Strukturunterschied zwischen GPU und CPU wird in der Abbildung 1.1 dargestellt. Mehrprozessorsysteme wären eine alternative Lösung. Wegen geringerer Kosten und Komplexitäten der Wartung und Optimierung wird hier nur die GPU-Berechnung angewandt.

Daher beschäftigt sich die Arbeit praktisch mit den Algorithmen der Echtzeit-



Abbildung 1.1: Unterschiedliche Strukturen zwischen CPU und GPU Quelle: [CUDAGuide]

Hintergrundadaption. Mit einem SDK für GPU-Programmierung werden die angepassten Berechnungen auf GPU umgesetzt. Die C-Codes (CPU-Codes) der Algorithmen werden durch GPU-Codes mit der Unterstützung einer speziellen Bibliothek ersetzt. Nach der Code-Integration wird die Belastung des Systems optimiert. CPU sendet die Befehle und Quellbilder zum GPU. GPU führt die Befehle aus, bearbeitet die Bilder und schickt die Ergebnisse zurück. Die meisten Berechnungsaufgaben werden auf Grafikkarte ausgeführt. Die Leistung der Bilderverarbeitung in Echtzeit wird sich stark erhöhen. Außerdem sollen die Häufigkeiten der Übertragungen zwischen CPU und GPU minimiert werden. Grafikkarten haben üblicherweise wenig Speicherraum. Deswegen wird der Zustand der Speicherverteilung in der Berechnung überwacht, wenn sehr viele Bilder gleichzeitig bearbeitet werden. Ein Speicherfehlerüberprüfungsverfahren wird in dem ganzen Prozess angewandt.

1.2 Gliederung der Arbeit

Diese Arbeit ist in fünf Kapitel untergliedert. Sie beginnt mit der Einführung in Kapitel 1. Die Motivation und Zielsetzung der Arbeit werden erklärt. In Kapitel 2 gibt es eine Übersicht über den aktuellen Stand der Technologien moderner GPU bzw. der Likelihood-Berechnung auf GPU. Des weiteren wird die relative Untersuchung von Lars Dießelberg eingebracht, die Algorithmen der Segmentierung auf CPU anwendet. Im dritten Kapitel werden die benötigten Grundlagen über Likelihood, Normalverteilung und CUDA Technologie dargestellt. Kapitel 4 enthält die Modellierung des adaptierten und integrierten Verfahrens. Das ehemalige CPU-Code wird durch GPU-Code ersetzt. Die pixelbasierte Berechnung wird auf GPU verschoben. Anschließend findet sich in Kapitel 5 ein ausführlicher Überblick über alle Ergebnisse der Tests, die mit der Implementierung auf GPU bzw. auf CPU durchgeführt werden. Danach folgt die Zusammenfassung und abschließende Beurteilung dieser Arbeit.

Kapitel 2

Stand der Technik

In diesem Kapitel werden die Theoreme und späteren verwendeten Technologien und deren heutiger Stand, die zur Realisierung dieses Programms notwendig waren, vorgestellt.

2.1 Satz von Bayes

Das Bayestheorem ist der theoretische Grundstein dieser Arbeit. Es ist ausführlich in [Bayestheorem] erklärt. Der Satz, der in der Wahrscheinlichkeitsrechnung wichtige Bedeutung hat, ist nach Thomas Bayes(1702-1761) benannt. Er war ein englischer Mathematiker, welcher seine Forschung auf die Berechnung der binomischen Distribution konzentrierte. Das Theorem kam zuerst in dem Artikel *An Essay towards solving a Problem in the Doctrine of Chances* im Jahr 1763 vor, der von seinem Freund Richard Price bearbeitet und dargelegt wurde.

Das Bayestheorem zeigt, wie man mit bedingten Wahrscheinlichkeiten rechnet. Der Satz geht von der Ausnahme aus, dass verschiedene Attribute von Klassen durch ihre gemeinsame Wahrscheinlichkeitsverteilungsfunktion zusammenhängen und unterschiedliche Klassen sich in ihrer Wahrscheinlichkeitsverteilungsfunktion unterscheiden. Dies ist durch die folgende Formel dargestellt:

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)} \quad (2.1)$$

Hierbei sind A und B zwei Ereignisse. $P(A)$ und $P(B)$ sind die A-priori-Wahrscheinlichkeit für A und B , $P(B) > 0$. $P(B|A)$ ist die Wahrscheinlichkeit für B unter der Bedingung, dass A vorkommt. Eine A-priori-Wahrscheinlichkeitseinschätzung wird anhand des Lernverfahrens verändert, und in eine A-posteriori-Verteilung überführt. Unbekannte Parameter werden geschätzt. Dafür werden Konfidenzregionen festgelegt und wird die Prüfung von Hypothesen für die Parameter abgeleitet. In der Bayes-Statistik wird die Wahrscheinlichkeit nicht nur für zufällige Ereignisse, sondern ganz allgemein für Aussagen eingeführt. Mittels Bayestheorem können Beweise für verschiedene Hypothesen gewichtet werden. Es ergeben sich folgende Eigenschaften Bayes'scher Lernmethoden [Andreas06]:

- "Jedes Trainingsbeispiel kann die Wahrscheinlichkeit für die Korrektheit einer Hypothese schrittweise beeinflussen. Somit sind Bayes'sche Lernmethoden flexibler als Methoden, die Hypothesen eliminieren, die für ein Beispiel nicht stimmen.
- Hintergrundwissen kann mit beobachteten Daten kombiniert werden, um die Wahrscheinlichkeit für eine Hypothese festzulegen.
- Hintergrundwissen in Bayes'schem Lernen ist nicht nur die A-priori-Wahrscheinlichkeit für jede Hypothese, sondern auch eine Wahrscheinlichkeitsverteilung für die beobachteten Daten für jede mögliche Hypothese.
- Bayes'sche Methoden können Hypothesen anpassen, die Wahrscheinlichkeitsvorhersagen angeben.
- Neuklassifizierung geschieht durch die nach Zutreffwahrscheinlichkeit gewichtete kombinierte Voraussage mehrerer Hypothesen.
- Wenn Bayes'sche Algorithmen nicht praktisch anwendbar sind, können sie trotzdem als Standard für die optimale Entscheidungsfindung herangezogen werden, gegen den die verwendeten Algorithmen gemessen werden können."

Das Bayestheorem und seine Erweiterungen spielen eine große Rolle in verschiedenen wissenschaftlichen Gebieten, besonders in der Bildverarbeitung. Zum Beispiel kann ein unbekanntes Idealbild X aus Datensatz Y (gestörte Version von X) rekonstruiert werden. Weitere Anwendungsbeispiele sind:

- Segmentierung
- Kantenbestimmung
- Tomographie (3D aus 2D)
- Bewegungsanalyse
- Transformationsfeld zwischen zwei Bildern

Im Multi-Kamerasystem befinden sich viele unsichere Elemente, zum Beispiel zufälliges Rauschen, defekte Pixel und zu dunkle oder zu helle Bereiche auf einem Bild. Um eine unbekannt gute Segmentierung aus den aufgenommenen Bildern unter bestimmten Bedingungen zu realisieren, wird das Bayestheorem in dieser Arbeit verwendet. Damit werden die diverse Fehler aufgehoben, und wird die Bildrekonstruktion aus verrauschten oder anders gestörten Daten festgestellt.

2.2 Segmentierung im Bereich Bildverarbeitung

In der Computerversion ist Segmentierung ein pixelbasierter Prozess, womit digitale Bilder in mehrere Segmente partitioniert werden. Die Segmentierung ist der erste Schritt der Bildanalyse. Das Ziel ist durch die Veränderung der Bilddarstellung um die Bildanalysen zu vereinfachen. Mit anderen Worten ist die Segmentierung eine Reduktion von Bilddaten zur besseren Strukturierung. Eine Beispielgrafik folgt zur Veranschaulichung:



Abbildung 2.1: Segmentiertes Beispielbild Quelle: [HornKorsche04]

2.2.1 Verfahren

Es sind viele Verfahren zur Segmentierung bekannt. Grundsätzlich werden sie oft in folgender Gruppen eingeteilt.

- Pixelorientierte Verfahren
- Kantenorientierte Verfahren
- Regionenorientierte Verfahren

Es gibt noch weitere Verfahrensansätze:

- Modellbasierte Verfahren
- Texturorientierte Verfahren

In dieser Arbeit gehört die Segmentierung zu den pixelorientierten Verfahren. Jedes Pixel ist unabhängig von den anderen und wird einzeln betrachtet in der Entscheidung, zu welchem Segment es gehört.

2.2.2 Optimierung der Segmentierung

Optimierung ist oft erforderlich für eine Segmentierung, da mehrere Elemente wie zum Beispiel Bildqualität, Algorithmus und gewählten Parametern die Qualität der Segmentierung beeinflussen können. Im Multi-Kamerasystem sollte man einen besseren Algorithmus wählen, indem der Vordergrund und Hintergrund von der aufgenommenen Bilder in Echtzeit und effizient segmentiert werden.

2.3 GPU Berechnung

Moderne Grafikkarten sind mit GPU (Graphics Processing Unit) eingebaut. Ein GPU ist ein allgemeiner SIMD (Single Instruction, Multiple Data) paralleler Prozessor, der für wissenschaftliche Anwendungen als Streamprozessor rechenintensive funktioniert, hauptsächlich in Aufgabe der 2D- und 3D-Computergrafik. GPU Transistoren lassen sich für den Aufbau der

Arithmetik einsetzen. Im November 2006 hat Nvidia CUDA als API für wissenschaftliche Berechnungen freigegeben. Mit einem SDK und API kann ein angepasstes Programm basierend auf C Sprache auf Geforce 8 Series GPU ausführen. AMD hat auch eine ähnliche Technologie FireStream entwickelt. Nach ein paar Jahren Entwicklung, ist der GPU ein flexibler und stark leistungsfähiger Prozessor geworden. GPU-Programmierer können zur Zeit die verschiedene Programmiersprachen z.B C, C++, Java usw. benutzen, um eigene Programme zu entwickeln. Außerdem können heutige GPUs nicht nur mit einfacher Genauigkeit sonder auch doppelter Genauigkeit rechnen.

2.3.1 Anwendungsgebiete

Die Hardware-Unterschiede zwischen GPU und CPU entscheiden die jeweiligen Anwendungsbereiche. GPU Berechnung wird heutzutage in folgenden Bereichen eingesetzt:

- Parallele Berechnung des Rechners
- Segmentierung - 2D und 3D
- Matlab Beschleunigung
- Digitale Image/Video Bearbeitung
- Wissenschaftliche Berechnung
- Schnelle Fourier-Transformation(FFT)
- USW.

2.3.2 Vorteile

Da sich die GPU auf allgemeine Berechnungen spezialisiert, liegt der Vorteil der Verwendung der GPU gegenüber CPU nicht nur in der höheren Rechenleistung sondern auch in der höheren Speicherbandbreite. Ein aktuelles Beispiel-Modell ist Nvidia GeForce GTX 480. Diese GPU kann um 1344.96 GFLOPS [WikiGPGPU] laufen. Seine Datenübertragungsrate zum Grafikspeicher beträgt 177.4 GB/s. Dem gegenüber steht der Intel Core i7-980X 3.33GHz mit nur 130.72 GFLOPS [WikiGPGPU] für Arithmetik. Die zugehörige maximale Datenrate (im Triple-Channel-Modus) beträgt nur 38.4 GB/s [WikiGPGPU]. Die folgenden Diagramme stellen die Abwicklung dar.

Ein weiterer Vorteile ist der geringe Preis im Vergleich zu ähnlich schnellen anderen Lösungen. Tatsächlich kann man die Grafikkarte fast auf jedem PC finden.

2.3.3 Nachteile

Die SDKs von verschiedenen Herstellern sind unterschiedlich (Befehle, Grammatik usw.). Das bedeutet, für eine spezielle Aufgabe auf einer speziellen Grafikkarte ist ein spezielles Konzept nötig. Im Vergleich zu gängiger Programmierung auf CPUs ist der Entwicklungsprozess relativ aufwändig.

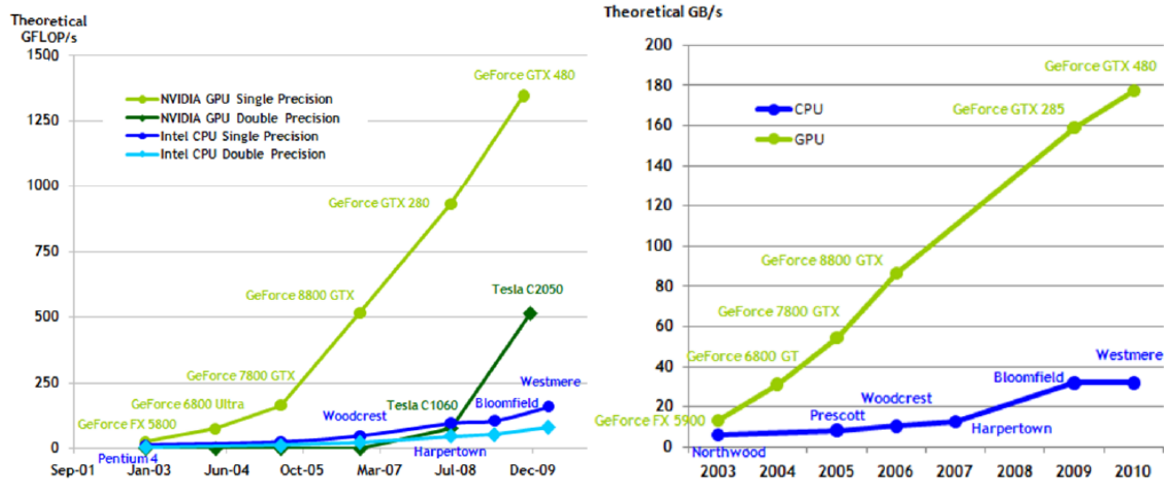


Abbildung 2.2: Floating-Point Operationen pro Sekunde und Bandbreite des Speichers für CPU und GPU Quelle: [CUDAGuide]

Im Gegensatz zum Hauptspeicher eines Computers hat eine Grafikkarte üblicherweise relativ weniger Speicherraum. Shared-Memory-Technologie ist oft erforderlich, um mehr Speicher zu erhalten. Außerdem soll die Grafikkarte den Speicherstatus über spezielle Fehlerkorrekturverfahren überwachen, indem die Ergebnisse der Berechnung sichergestellt werden.

Die Leistung der GPUs ist auch noch abhängig von der Busbandbreite zwischen CPU und GPU. Die Datenkopie Zwischen GPU- und CPU-Speicher ist eine relativ langsame Operation. Diese soll in der GPU-Programmierung minimiert werden.

2.3.4 Wahrscheinlichkeit auf GPU berechnen

In den letzten Jahren wurde GPU-Berechnung auf verschiedenen Bereiche angewandt. Im Gebiet Wahrscheinlichkeitsberechnung, können die Algorithmen durch GPU-Berechnung beschleunigt werden. Patrick C. und Pierre Dumouchel [Patrick 08] haben Akustik-Wahrscheinlichkeitsberechnung auf GPU ausgeführt. Claus L, Giorgio P. und Alois K. [Claus 08] haben die pixelbasierte Segmentierung eines Partikelfilters durch GPU-Berechnungen beschleunigt. Die oben vorgestellten Implementationen zeigen die sehr effizienten Leistungen gegenüber denen auf CPU. Die Ergebnisse der Forschungen motivieren die Anwendung der GPU-Berechnung im 3D-Personentracking.

2.4 Vorarbeiten zur Vorder-/Hintergrund-Segmentierung

Tobias Feldmann, Lars Dießelberg und Annika Wörner haben ein Verfahren für die Vorder-/Hintergrundtrennung mittels einer Silhouettenbasierten 3D-Rekonstruktion der Szene in einem Multi-Kamerasystem entwickelt [Feldmann2009AFS]. Die Segmentierung hat die Fähigkeit, sich an nicht nur statischen sondern auch dynamischen Hintergrund anzupassen. Außerdem erwies sich auch eine Erkennung von Schatten und Aufhellungen. Jeder Pixel aus dem aufgenommenen Bild wird unabhängig von den anderen aufgenommen und verarbeitet. Durch

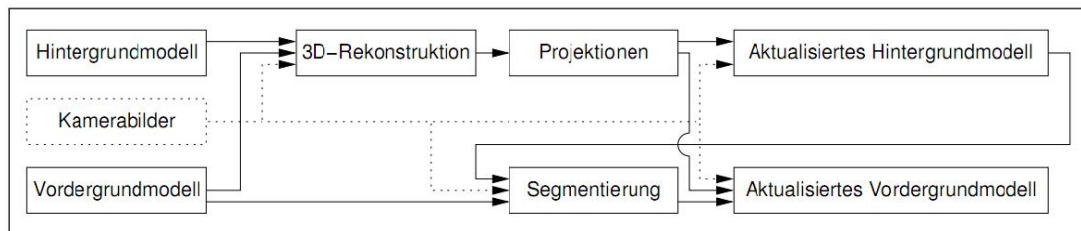


Abbildung 2.3: Ablauf eines Segmentierungsschrittes Quelle: [Feldmann2009AFS]

eine Integration der 3D-Szene Informationen wurde eine bessere Segmentierung mit geringen falschen Silhouette-Klassifizierungen erreicht. Der Ablauf des Ansatzes lässt sich in folgenden Schritten zerlegen:

1. Lade Bilder des nächsten Zeitschrittes
2. 3D-Rekonstruktion mittels des Belegungsgitters
3. Projektion des Belegungsgitters zur Erstellung der Lernmaske
4. Anpassung des Hintergrundmodells mit Szenenprojektion als Lernvektor
5. Segmentierung der Bilder nach aktualisiertem Modell
6. Berechne Statistiken des Vordergrunds zum Einsatz im nächsten Zeitschritt

Der Prozess wird anschaulich durch das Diagramm (Abbildung 2.3) dargestellt. Zuerst werden die fortlaufenden Bilder aufgeladen und zum YUV-Farbraum konvertiert. Dann werden die Wahrscheinlichkeiten des Vordergrunds von allen Kameras in den Prozess der 3D-Rekonstruktion integriert. Damit können die Fehler der Silhouetten durch die Berechnung der Belegungswahrscheinlichkeiten der Voxel korrigiert werden. Nach der Integration wird die Belegungswahrscheinlichkeiten wieder in die verschiedenen Kameraperspektiven projiziert. Dann wird jeder Voxel iteriert. Alle Wahrscheinlichkeiten werden in die Maske projiziert. Das Ergebnis ist eine probabilistische Vordergrundmaske. Im nächsten Schritt wird das Hintergrundmodell durch die konvertierte Vordergrundmaske aktualisiert. Danach wird die Segmentierung des Vordergrunds mit den aktuellen Bildern, aktueller Vordergrund-GMM und aktualisierter Hintergrund-GMM ausgeführt. Zum Abschluss wird die Vordergrund-GMM mit den aktuellen Bildern, der Wahrscheinlichkeitsprojektion und der Segmentierung gelernt. Der Effekt wird in Abbildung 2.4 dargestellt.

Das Vordergrund-/Hintergrund-Lernverfahren kann in drei Teile zerlegt werden. Danach folgt eine kurze Erklärung der einzelnen Schritte.

1. Hintergrund-Lernverfahren
2. Verarbeitung Illuminanz und Chrominanz
3. Vordergrund-Lernverfahren

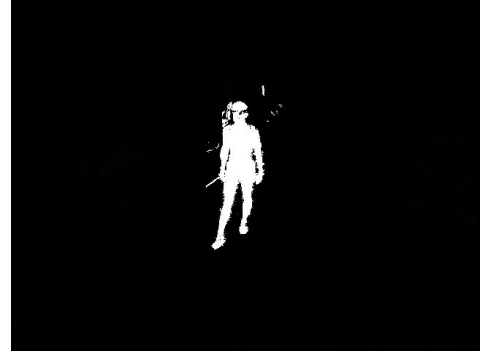


Abbildung 2.4: Modellierung und Segmentierung des Hintergrundes Quelle: HumanEva

2.4.1 Hintergrundmodell

In Hintergrundmodell wird jeder Pixel durch eine Gauss-Mischverteilung mit bis zu K Komponenten modelliert. Die maximale Anzahl der Komponenten K ist ein Parameter des Algorithmus und nimmt üblicherweise kleine Werte, normalerweise zwischen 3 und 5 an. Jeder Bereich des Farbraums wird durch einen Gaussglocke dargestellt. Die Gewichtung einer Komponente ist proportional zu der Häufigkeit, in welcher die Farbe des Hintergrundes in den Bereich der Komponente fällt. Die Hintergrundlikelihood wird durch die folgende parametrische Funktion dargestellt:

$$p(c_t | F_t = 0) = \sum_{k=1}^K \omega_t^k \prod_{d=1}^3 \eta(c_t^d, \mu_t^{k,d}, \Sigma_t^{k,d}) \quad (2.2)$$

mit der Dichtfunktion der Gaussverteilung:

$$\eta(x, \mu, \Sigma) = \frac{1}{\sqrt{2\pi\Sigma}} \exp\left(-\frac{1}{2} \frac{(x - \mu)^2}{\Sigma}\right) \quad (2.3)$$

C_t sei der Farbwert des Pixels im Bild t der Sequenz. Um die Berechnung zu vereinfachen, werden die Farbkanäle als stochastisch unabhängig betrachtet. ω_t^k sei das Gewicht der k -ten Gausskomponenten. $\mu_t^{k,d}$ und $\Sigma_t^{k,d}$ seien Mittelwert und Varianz des d -ten Farbkanals mit $d \in 1, 2, 3$. Für die Berechnung der Gewichte, Mittelwerte und Varianzen wird ein gleitendes Fenster über die letzten L Zeitschritte eingesetzt. Durch Vergleich der Farbe-Distribution um Zeitpunkt t und $t + 1$, wird das Pixel als *bg* oder *fg* klassifiziert.

Im Bild t wird eine GMM mit einer Reihe von Koeffizienten $(\omega_t^k, \mu_t^{k,d}, \Sigma_t^{k,d})$ aus bekannten Hintergrundfarbwerten c_t gewonnen. Wenn im Bild $t + 1$ eine neuer Farbwerte c_{t+1} als Hintergrund klassifiziert wird, soll diese Farbe in der GMM integriert werden. Damit kann das Modell aktualisiert werden. Deshalb wird zunächst unter den vorhandenen Komponenten eine zum Farbwert passende gesucht. Wenn sich keine passende Komponente befindet, wird eine neue Komponente mit $c_{t+1}, \omega_{init}, \Sigma_{init}$ initialisiert. Wird hingegen eine passende gefunden, soll die Komponente mit dem vorhandenen Farbwerte aktualisiert werden.

Die Berechnung der Gauss-Mischverteilung für jedes Pixel ist aufwendiger im Multi-Kamerasystem, da der rechenintensive Prozess hoch parallel und in Echtzeit durchgeführt

wird. Um eine bessere Leistung zu erzielen, wird hier eine optimale Lösung verwendet. Eine Technik des Grafikprozessors für Berechnungen wird für das Hintergrundmodell eingesetzt, um die Bildverarbeitung zu beschleunigen, und CPU zu entlasten.

2.4.2 Verarbeitung Illuminanz und Chrominanz

Im Multi-Kamerasystem sind Illuminanz und Chrominanz ein Problem für die Segmentierung. Das Lernverfahren dazu ist als eine Erweiterung des Hintergrundmodells dargestellt:

$$p(c_t|F_t = 0) = \underbrace{\frac{1}{2} \sum_{k=1}^K \omega_t^k \Pi_{d=1}^3 \eta(c_t^d, \mu_t^{k,d}, \Sigma_t^{k,d})}_{\text{Hintergrundlikelihood}} + \underbrace{\frac{1}{2} p(c_t|S_t = 1)}_{\text{Schatten / Aufhellungen-Likelihood}} \quad (2.4)$$

Hierbei steht $S_t \in \{0, 1\}$ für Schatten oder Aufhellungen. Die Menge der Schatten und der Aufhellungen des Hintergrunds wird als gleichverteilt angenommen. Weiterhin berechnet $p(c_t|S_t = 1)$ sich aus:

$$p(c_t|S_t = 1) = \sum_{k=1}^K \omega_t^k p(c_t|S_t^k = 1) \quad (2.5)$$

Angenommen, (Y_B, U_B, V_B) ist eine Farbe eines Pixels aus dem Hintergrund. Ein neuer Parameter λ für die Rate der Luminanz ist durch $\lambda = \frac{Y_t}{Y_B} = \frac{c_t^1}{\mu_t^{k,1}}$ definiert. Zusätzlich bestehen die Schatten- und Aufhellungsschwellwerte mit $\tau_S < 1$ und $\tau_H > 1$. Wenn $\tau_S \leq \lambda \leq \tau_H$, wird die Farbe c_t als Schatten oder Aufhellung klassifiziert. Wenn im Gegensatz λ nicht in den oben genannten Bereich fällt, soll die Dichtfunktion $p(c_t|S_t = 1)$ Null sein.

2.4.3 Vordergrundmodell

Am Anfang ist nichts über die Objekte des Vordergrunds bekannt. Durch den Vergleich mit dem Hintergrund verändert sich die Farb-Distribution des Vordergrunds wegen der Bewegungen oder variierender Objekte sehr schnell. Deswegen ist eine pixelweise Verarbeitung nicht mehr geeignet. Stattdessen wird das Bild in größere Blöcke zerlegt. Eine Gauss-Mischverteilung wird anhand des Histogramms geschätzt. Im Lernverfahren des Vordergrunds wird nur die Pixelfarbe gelernt. Die Pixelposition wird nicht berücksichtigt. Die Farbverteilung wird aus bekanntem und neuem Vordergrund aufgebaut. Der bekannte Vordergrund wird als Gauss-Mischverteilung mit k-Means modelliert. Der k-Means Algorithmus besteht aus folgenden Schritten:

1. Zufällige Wahl von k Clusterzentren
2. Ordne einen Farbwert dem Cluster zu, dessen Zentrum am nächsten liegt
3. Bestimme neue Clusterzentren über Mittelwertbildung
4. Neuverteilung der Farbwerte entsprechend Schritt 2
5. Iteriere Schritte 3 und 4 bis sich die Zuordnung nicht mehr ändert

Die Farbe von neuen Objekten wird als gleich verteilt angenommen. $U(c)$ steht für die Farbdistribution als Uniform. Zusätzlich wird eine neue Wahrscheinlichkeitsfunktion P_{NF} (neuer Vordergrund) eingeführt. Die Wahrscheinlichkeitsdichtfunktion des Vordergrunds wird folgendermaßen definiert:

$$p(c|F = 1) = P_{NF}U(c) + (1 - P_{NF}) \sum_{k=1}^K \omega^k \eta(c, \mu^k, \Sigma^k) \quad (2.6)$$

Kapitel 3

Grundlagen

In diesem Kapitel werden die Grundlagen, die für diese Arbeit notwendig sind, vorgestellt.

3.1 Likelihood und Normalverteilung

In der Bilderdatenschätzung wird die Likelihood-Funktion benötigt. Dabei handelt es sich um eine mathematische Funktion, die im Rahmen der Maximum-Likelihood-Methode verwendet wird, um Parameter einer Dichte- bzw. Wahrscheinlichkeitsfunktion zu schätzen. Hierbei ist X eine Zufallsvariable mit zugehöriger Dichte- bzw. Wahrscheinlichkeitsfunktion $f(x, \theta)$. Es bezeichne θ einen unbekannt Parameter. Sind nun x_1, x_2, \dots, x_n verschiedene Realisationen dieser Zufallsvariablen, so erhält man die Likelihood-Funktion dieser Stichprobe als

$$L(\theta) = f(x_1; \theta) \cdot f(x_2; \theta) \cdot \dots \cdot f(x_n; \theta) \quad (3.1)$$

Die Likelihood-Funktion ist also stets eine Funktion, die nur abhängig von dem unbekannt Parameter θ ist. Deswegen kann die Berechnung durch GPU parallelisiert werden.

Hier ist ein Beispiel der Likelihood-Funktion [WikiGauss]: Die Normal- oder Gauß-Verteilung (nach Carl Friedrich Gauß) $\mathcal{N}(\mu, \sigma^2)$ mit Mittelwert μ und Varianz σ^2 besitzt die Wahrscheinlichkeitsdichte

$$f(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right) \quad (3.2)$$

Sind nun x_1, \dots, x_n Realisationen einer Normalverteilung $\mathcal{N}(m, s^2)$ mit unbekanntem Mittelwert und Varianz, so erhält man die entsprechende Likelihood-Funktion mit $\theta = (m, s)$ zu

$$L(m, s^2) = \prod_{i=1}^n f(x_i; m, s^2) = \left(\frac{1}{2\pi s^2}\right)^{n/2} \exp\left(-\frac{\sum_{i=1}^n (x_i - m)^2}{2s^2}\right) \quad (3.3)$$

Die Normalverteilung ist eine sehr wichtige kontinuierliche Wahrscheinlichkeitsverteilung in vieler natur-, wirtschafts- und ingenieurwissenschaftlichen Gebieten. In der Dichtfunktion der Normalverteilung, steht der Mittelwert μ für den Erwartungswert. Die Varianz σ^2 , als Standardabweichung genannt, beschreibt die Breite der Normalverteilung. Die Abbildung 3.1 zeigt die Dichtfunktionen einiger Normalverteilungen.

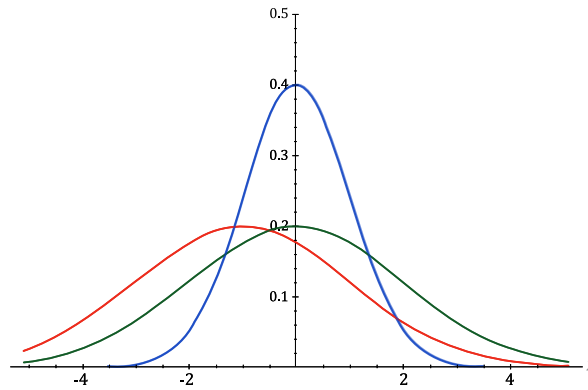


Abbildung 3.1: Dichtfunktion der Normalverteilungen $\mathcal{N}(0,1)$ (blau), $\mathcal{N}(0,2)$ (grün) und $\mathcal{N}(-1,2)$ (rot) Quelle: [WikiGauss]

3.2 GMM (Momentenmethode)

Die Momentenmethode ist ein Schätzverfahren in der schließenden Statistik. Damit kann der Parameter einer Grundgesamtheitsverteilung anhand einer Stichprobe geschätzt werden. GMM ist eine sehr allgemeine statistische Methode, die von Lars Peter Hansen [Schlittgen2000] entwickelt wurde. Die Vorgehensweise wird durch zwei Schritte dargestellt.

1. Die Parameter θ_i der theoretischen Verteilung werden als Funktionen der Momente m_k angegeben:

$$\theta_i = h_i(m_1, m_2, \dots, m_n) \quad (3.4)$$

2. Die Momentenschätzer $\hat{\theta}_i$ für die einzelnen Parameter werden berechnet, indem in den obigen Gleichungen die empirischen Momente $\hat{m}_k = \frac{1}{n} \sum_{i=1}^n x_i^k$ für die Momente m_k eingesetzt werden:

$$\hat{\theta}_i = h_i\left(\frac{1}{n} \sum_{i=1}^n x_i^1, \frac{1}{n} \sum_{i=1}^n x_i^2, \dots, \frac{1}{n} \sum_{i=1}^n x_i^n\right) \quad (3.5)$$

Ein einzelner Parameter kann direkt errechnet werden. Bei mehreren zu schätzenden Parametern ergibt sich häufig ein Gleichungssystem, das bezüglich der unbekannt Parameter aufgelöst werden kann. Durch Einsetzen der Werte einer Stichprobe erhält man dann Werte für die Parameter der theoretischen Verteilung.

3.3 Farbräume

Ein Farbraum wird als Messraum für die einheitliche visuelle Wahrnehmung 'Farbe' definiert. Es existieren unterschiedliche Farbräume, die eine große Rolle in verschiedenen Anwendungsbereichen spielen. Zur Zeit gibt es 30 bis 40 Farbräume. Diese unterscheiden sich durch den beabsichtigten Einsatzbereich. Die Auswahl eines geeigneten Farbraums entscheidet die Qualität einer farbbasierten Segmentierung.

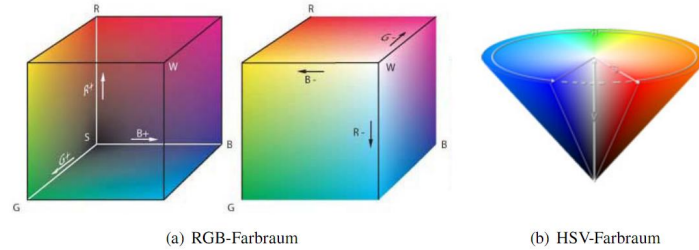


Abbildung 3.2: Darstellung der Farbräume RGB und HSV Quelle: [Latsch08]

Beim RGB-Farbraum entsteht die Farbe durch Mischung der einzelnen Farbkanäle (Rot, Grün und Blau). Der RGB-Farbraum wird für farbdarstellende Systeme eingesetzt. In dieser Arbeit ist die wichtigste Eigenschaft für eine Trennung nicht die Chrominanz sondern die Luminanz, da die Luminanz eines Objektes stark von der Einstellung der Beleuchtung abhängig ist und sich häufig verändert, während die Chrominanz fast invariant bleibt. Deswegen ist der RGB-Farbraum hier nicht geeignet.

Beim HSV-Farbraum wird die Farbe mit Hilfe des Farbtons (Hue), der Farbsättigung (Saturation) und der Helligkeit (Value) definiert. Er ist wie der RGB-Farbraum ein wichtiger Farbraum in der digitalen Bildverarbeitung. Im Gegensatz zum RGB-Farbraum ist er relativ einfach zu berechnen und in Farbauswahldialogen besser angepasst. In dieser Arbeit verändern sich die Farbe des Objekts und Hintergrunds nicht häufig. Das bedeutet, dass bei festgestelltem Farbtonkanal und variierendem Helligkeitskanal nur die wahrgenommene Helligkeit verändert wird. Andererseits sind die Nachteile auch deutlich. In der Umrechnung zwischen RGB- und HSV- Farbraum, wenn die Farben sehr dunkel sind, können kleine Änderungen an den RGB-Werten zu sehr großen Änderungen des Farbtons und/oder der Sättigung führen. Die Auswirkung ist starkes Kamerarauschen in dunklen Bereichen einer Szene.

Um diesen Fall zu vermeiden, wurde schließlich der YUV-Farbraum ausgewählt. Die Umrechnung eines RGB-Farbwertes wird durch folgende Gleichungen festgelegt:

$$\begin{aligned}
 Y &= 0.299R + 0.587G + 0.114B \\
 U &= 0.713(R - Y) + \Delta \\
 V &= 0.564(B - Y) + \Delta
 \end{aligned}
 \tag{3.6}$$

Hier $\Delta = 128$ bei 8-Bit-Kanälen. Die Konvertierung wurde in *OpenCV* in der Methode *cvCvtColor* mit Transformationsmodus *CV_RGB2YCrCb* implementiert. Ein Faktor λ wird noch angewandt, um die Helligkeit eines Pixels zu skalieren, also $(R', G', B') = \lambda(R, G, B)$. Die Transformation wird durch folgende Formeln angegeben:

$$\begin{aligned}
 Y' &= \lambda Y \\
 U' &= \lambda(U - \Delta) + \Delta \\
 V' &= \lambda(V - \Delta) + \Delta
 \end{aligned}
 \tag{3.7}$$

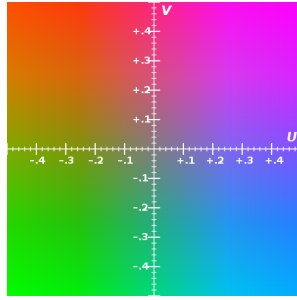


Abbildung 3.3: YUV-Farbmodell, Y-Wert=0,5 Quelle: [WikiYUV]

Durch die proportionale Berechnung wird die geringe Farbauflösung von dunklen Bildbereichen direkt im Farbraum wiedergegeben. Abbildung 3.3 ist ein Beispiel für ein YUV-Farbmodell:

3.4 OpenCV

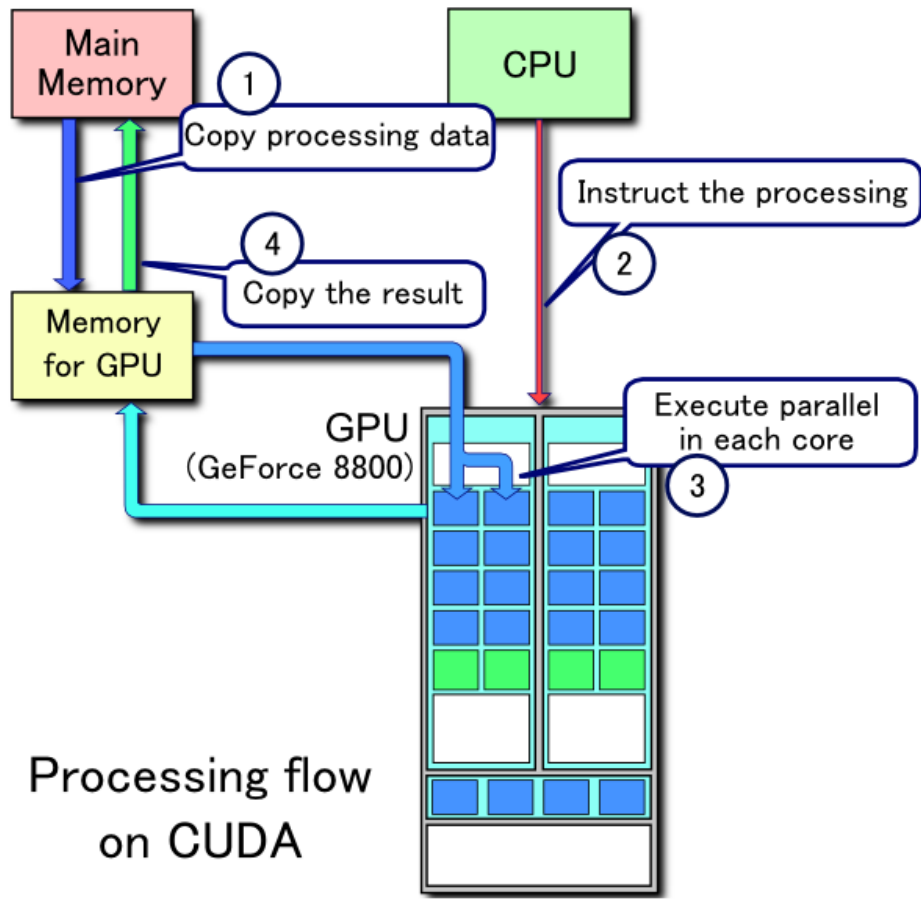
OpenCV ist eine plattformunabhängige Programmiersbibliothek, die von Intel entwickelt wurde. Die aktuelle stabile Version ist 2.1 vom April 2010. Die Bibliothek ist hauptsächlich in den Programmiersprachen C und C++ und dafür geschrieben. Open im Name bedeutet dass sie quelloffen unter BSD-Lizenz ist. Das CV steht für Computer Vision. OpenCV enthält eine große Menge der Algorithmen für Bildverarbeitung und maschinelles Sehen. Dies spielt eine wichtige Rolle in der Programmierung dieser Arbeit.

3.5 Compute Unified Device Architecture (CUDA)

Die angewandte Technologie zur Beschleunigung der Bildverarbeitung in dieser Arbeit ist CUDA (Compute Unified Device Architecture). Dies ist eine von Nvidia entwickelte Technik zur Beschleunigung wissenschaftlicher und technischer Berechnungen durch Einbeziehung der Grafikkarte in die Berechnungen. Die CUDA kann mit einer Grafikkarte ab der Geforce 8-Serie, Quadro FX 5600 und Tesla Linien eingesetzt werden. Im Juni 2010 wurde von Nvidia die CUDA-Version 3.1 veröffentlicht. Die Anwendungen werden durch die Programmiersprache C mit Nvidia CUDA-SDK ausgeführt. Es existieren aber auch Wrapper für die Programmiersprachen Python, Java, Fortran und .NET bzw. Anbindungen an Matlab[WikiCUDA]. Der Ablauf der CUDA-Programmierung wird durch folgendes Diagramm 3.4 dargestellt.

In einem CUDA-Programm wird eine Funktion, die für GPU ausgeführt wird, *Kernel* genannt. Der Kernel wird auf GPU als viele unterschiedliche Threads mehrmals ausgeführt. Beide *Host* (CPU) und *Device* (GPU) verwalten eigene Speicher. Datenkopie dazwischen ist häufig. Ein CUDA-Programm wird durch folgende Schritte in ein großes Projekt integriert.

- Kernel Code auf *.cu* Datei schreiben und mit NVCC kompilieren
- CPU-Aufrufer in Header Datei speichern
- In C++ Quellecode Header Datei "#include"



Processing flow on CUDA

Abbildung 3.4: CUDA Verarbeitungsablauf Quelle: [WikiCUDA]

- System korrekt modifizieren und aufbauen

Kapitel 4

Umsetzung

Im Folgenden wird beschrieben, wie die Segmentierung beschleunigt werden kann. Beginnend mit der Analyse der vorhandenen angewandten Algorithmen des Verfahrens zur Vorder-/Hintergrundtrennung, in der alle Teile und ihr Zusammenspiel kurz vorgestellt werden. Die optimierbaren Teile sollen für GPU-Beschleunigung festgestellt werden, danach folgen Details zu Aufbau und Integration der neuen Module.

4.1 Analysen der Algorithmen

Von der Vorarbeit in Kapitel 2 werden die Algorithmen der Segmentierung für Multi-Kamerasystem übernommen. Um die aufwändigen intensiven Berechnungen zu beschleunigen, sollen die angepassten Teile, die parallel-laufend und unabhängig voneinander sind, aus den Algorithmen auf der GPU umgesetzt werden. Außerdem soll exzessives Kopieren zwischen der GPU und der CPU vermieden werden.

Im Hintergrundmodell wird jedes Bild durch eine Gauss-Mischverteilung mit diagonalen Kovarianzmatrizen berechnet. Ein Modell wird für jedes Pixel verwendet. Deshalb kann jeder Prozess als ein Thread mit gleichem Berechnungsgewicht in der GPU ausgeführt werden. Im Vordergrundmodell wurde k-Means Algorithmus als Clustering-Algorithmus für Farbschätzung verwendet. Jedes Bild wird nicht pixelweise sondern blockweise bearbeitet. Im Vergleich zum Hintergrundmodell ist das Optimierungsrezept für das Vordergrundmodell komplexer (eine Dimension mehr). Aus diesem Grund werden die Algorithmen des Hintergrundmodells hauptsächlich auf der GPU umgesetzt. Durch die Auswertung des umgebauten Hintergrundmodells entscheidet sich, ob es für das Hintergrundmodell-mit-Schatten-und-Aufhellungen als Grundlage dienen kann, da dieses die Erweiterung des Hintergrundmodells darstellt. Die zwei Modelle, Vordergrundmodell und Hintergrundmodell-mit-Schatten-und-Aufhellungen, würden dann in der Zukunft nach vertiefenden Optimierungen auf der GPU ausgeführt werden.

4.2 Initialisierung

Die Gauss-Mischverteilung ist der Kern der rechenintensiven Aufgaben im Hintergrundmodell. Die Aufgabe ist eine hohe Belastung für die CPU. Jeder Farbwert eines Pixels in VGA-Bildern

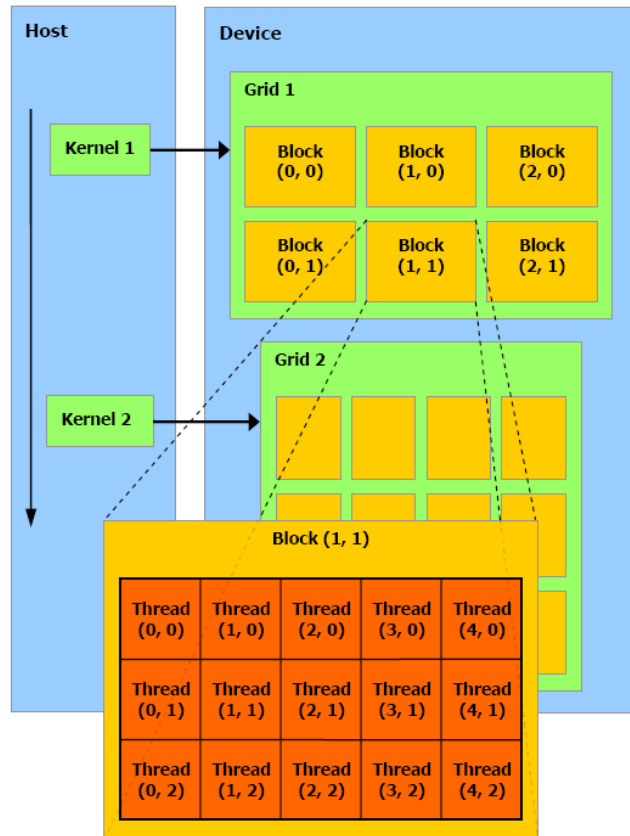


Abbildung 4.1: CUDA Thread Hierarchie Quelle: [CUDAGuide]

(640x480) wird als Gleitkommazahl durch die Normalverteilungsfunktion berechnet. Zunächst muss ein aufgenommenes Bild zu einer Gleitkommazahl-Matrix (640x480) konvertiert werden. Mit Unterstützung der CUDA-Technologie wird jede Gleitkommazahl als ein Thread auf der GPU parallel und mehrmals ausgeführt. D.h. Gauss-Mischverteilung für jedes Pixel wird als *Kernel* Funktion auf der GPU implementiert. Die Hierarchie der GPU steht in Abbildung 4.1. Das Berechnungsgitter (Grid) ist aus Thread-Blöcken zusammengesetzt. Die Kapazitäten des Gitters sind abhängig vom GPU Speicher. Hier wird das CUDA-Programm auf einer Nvidia GeForce 9800 Grafikkarte mit 512MB Speicher eingesetzt. Die Gitter und Blöcke können 1, 2, oder 3-dimensional sein. Im Multi-Kamerasystem werden 2D-Bilder von jeder Kamera verarbeitet. Deshalb wird hier ein 2D-Entwurf angewandt.

Das Programm wurde hauptsächlich in C-Sprache mit OpenCV und CUDA-SDK geschrieben. Die Abbildung 4.2 zeigt die Struktur des CUDA-SDK. C-Code und CUDA-Code sind durch grüne bzw. blaue Blöcke dargestellt. Normale Bibliotheken sind die Standard-Bibliotheken von C, z.B. *math.h*, FFT, BLAS usw. Die werden mittels der Standard-Header-Datei verfügbar gemacht. Der CPU Host-Code wird in C-Sprache geschrieben. Host-Code ist unabhängig von NVCC. Dieser enthält die Befehle, welche die GPU und GPU-Speicher kontrollieren. Die CUDA-Kernels werden gemeinsam von C und CUDA Code geschrieben und als .cu Datei gespeichert. Die parallelen Berechnungen (meistens mathematische Funktionen) werden darin ausgeführt. Das CUDA-Projekt wird durch NVCC-Compiler kompiliert. Der NVCC-Compiler bearbeitet den CPU-Code mit Host-Compiler und `#include's` bzw. seinen zugehörigen Links.

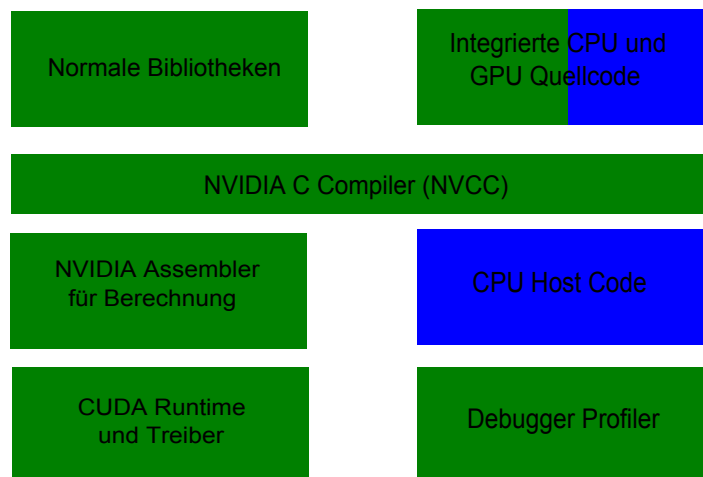


Abbildung 4.2: CUDA-SDK

Zusätzlich enthält CUDA-SDK noch die zugehörigen Assembler, Treiber und Debugger Profiler. Die Programmierungswerkzeuge werden nach der Installation verfügbar sein. Der Test- und Beispiel-Code in CUDA-SDK sind auch ausführbar. Die Programmbibliothek OpenCV wird hier als eine Brücke eingesetzt. Auf die Farbwerte der Bilder wurde als Gleitkommazahlen durch OpenCV zugegriffen. Der Prozess kann durch den folgende Fluss dargestellt werden.

$$C \text{ Code} \Leftrightarrow \text{OpenCV} \Leftrightarrow \text{CUDA} \quad (4.1)$$

4.3 CUDA Modulentwicklung

Für die CUDA Integration werden sechs Module beim Bedarf entwickelt. Damit würden die Methoden für verschiedene Fälle im Hintergrundmodell flexibel aufgerufen werden können. Diese werden auf .cu Datei geschrieben und gespeichert. Die Module bestehen aus den folgenden Teilen. Die entsprechenden Beschreibungen stehen im anschließenden Unterabschnitt.

- *Modul 1: GaussArray.cu*
- *Modul 2: GaussSingleParameter.cu*
- *Modul 3: GaussMultiParameter.cu*
- *Modul 4: GaussMixture.cu*
- *Modul 5: probGPU.cu*
- *Modul 6: Measure.cu*

4.3.1 Kernel

Kernel, C Funktion in CUDA benannt, ist durch eine Deklaration `_global_` definiert. Dies wird N -mal in N unterschiedlichen parallelen CUDA-Threads auf der GPU implementiert.

Jeder Thread hat eine eindeutige Thread-ID. Mit der eingebaute Variable *threadIdx*, kann auf die Thread-ID im Kernel zugegriffen werden. Für einen gegebenen Kernel wird eine neue Konfiguration-Syntax <<< ... >>> mit bestimmter Thread-Anzahl verwendet. Liste 4.1 und 4.2 zeigen ein Beispiel für Matrizenaddition in C-Code und CUDA-Code. Die geschachtelte Schleife wird durch ein eingeschlossenes Gitter ersetzt. In den oberen Modulen wird die Funktion jedes Pixels im Kernel angelegt.

Listing 4.1: addMatrix C-Code

```

void addMat(float* A, float *B, float* C, int width, int height)
{
    int index;
    for ( int i = 0; i < width; ++i )
        for ( int j = 0; j < height; ++j ) {
            index = i + j*width;
            c[index] = a[index] + b[index];
        }
}

int main() {
    addMat( A, B, C, width, height );
}

```

Listing 4.2: addMatrix CUDA-Code

```

// Kernel berechnen
__global__ void addMat(const float *A, const float *B, float *C,
const int width, const int height)
{
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int j = blockIdx.y * blockDim.y + threadIdx.y;
    int index = i + j*width;
    if(i < width && j < height)
        C[index] = A[index] + B[index];
}

void addMatGPU (const float *A, const float *B, float *C,
const int width, const int height)
{ //Kernel aufrufen
    dim3 dimBlock (blocksize, blocksize);
    dim3 dimGrid (width/dimBlock.x+(width%dimBlock.x == 0?0:1),
                  height/dimBlock.y+(height%dimBlock.y == 0?0:1));
    addMat<<<<dimGrid, dimBlock>>> (AD, BD, CD, width, height);
}

```

Für Modul 1 werden alle Elementen (Typ: Gleitkommazahl) von einer Datenreihe mit bestimmter Länge durch eine Normalverteilung parallel berechnet. Jedes Element wird als ein selbstständiger Thread im Kernel bearbeitet. Die Parameter, die Standardabweichung σ und der Erwartungswert μ sind konstant. Im Modul 2 wird ein Quellenbild mit konstanten Parametern σ und μ durch eine Normalverteilung konvertiert. Jedes Pixel des Quellenbilds wird als

ein Thread im Kernel parallel berechnet. Modul 3 ist ähnlich wie Modul 2. Die Parameter σ und μ der Gauss-Mischverteilung werden statt Konstanten als zwei Gleitkommazahl-Matrizen Σ und M verwendet. Die Größe der Parametermatrizen und des Quellenbilds müssen identisch sein. Damit kann ein Pixel aus dem Quellenbild mit den Parametern in gleicher Position auf dem Bild Σ und M berechnet werden. Tatsächlich werden insgesamt 307200 Threads (640×480 Pixels) mit gleichem Gewichte im Kernel ausgeführt.

Modul 4 ist eine Erweiterung von Modul 3. Die Parametermatrizen Σ und M werden in zwei Reihen angesetzt. Die Anzahl der Matrizen in der jeweiligen Reihe ist durch die Komponentenanzahl N genannt. Außerdem besitzt jede Komponente ein unterschiedliches Gewicht ω . Diese Gewichte werden als eine Reihe der Gleitkommazahl-Matrizen mit Anzahl N , sowie Σ und M dargestellt. Das Quellenbild wird mit der drei Gruppen N -mal berechnet. Somit beträgt die Anzahl der Threads für ein Einzelkanal-Quellenbild $307200 \times N$ im Kernel. Für die Grafikkarte Nvidia GeForce 9800 GT mit 512MB Speicher beträgt der Maximalwert N ca. 70. Zum Schluss wird die gewichtete Summe als Endergebnis zum Zielbild gespeichert.

Im Modul 5 steht die Arithmetik-Berechnung für Gleitkommazahlen im Kernel. Darin wird jeder Pixelwert von zwei Quellenbildern berechnet und in ein Zielbild Punkt-zu-Punkt angelegt und gespeichert. Es gibt keine Berechnung im Modul 6, deshalb wird kein Kernel dazu definiert. Die GPU führt nur die Befehle von der CPU aus. Die Übertragungszeit für die Bilder mit verschiedenen Größen (1×1 bis 1000×1000 Pixels) wird gemessen.

4.3.2 Thread Hierarchie

threadIdx ist ein Vektor, der aus drei Komponenten besteht. Somit können Threads durch 1D-, 2D- oder 3D-Thread-Index identifiziert werden. Der entsprechende 1D-, 2D- oder 3D- Thread-Block wird daraus zusammengesetzt. Durch dieser Organisation können Vektor, Matrix oder Volumen indexiert und berechnet werden.

Der Index eines Threads ist stark abhängig von seiner ID. Für einen 1D- Block ist der Index identisch mit dem Thread; für einen 2D- Block mit der Größe (D_x, D_y) , ist die ID des Threads mit Index (x, y) $(x + yD_x)$; für einen 3D- Block mit der Größe (D_x, D_y, D_z) , lautet die ID des Threads mit Index (x, y, z) $(x + yD_x + zD_xD_y)$. In Modul 1 für ein 1D- Array wird einfach ein 1D- Index verwendet. In Module 2, 3, 4, 5 ist der Input ein VGA-Bild mit 640×480 Pixels. Somit sind die Indices des Gitters 2-dimensional.

Die Anzahl des Threads pro Block ist beschränkt, weil sich alle Threads in einem Block im gleichen Prozessor befinden müssen, und begrenzten Speicherraum des Prozessors gemeinsam nutzen müssen. Auf den aktuellen GPUs kann ein Thread-Block bis 1024 Threads beinhalten. Allerdings kann ein Kernel durch mehrere Threads-Blöcke mit gleicher Größe implementiert werden. Es bedeutet:

$$\text{Summe der Threads} = (\text{Threadanzahl per Block}) \times (\text{Blockanzahl}) \quad (4.2)$$

Wie in Abbildung 4.1 werden Thread-Blöcke in einem 1D- oder 2D- Gitter (grid) organisiert. Anzahl der Threads pro Block und Anzahl der Blöcke werden durch Syntax `<<< ... >>>` festgestellt. Der Typ kann *int* oder *dim3* sein. Hier wird der Typ des Parameters als *dim3* dargestellt. Wenn die Anzahl zu klein ist, wird die Leistung wegen zu wenig Threads reduziert. Auf der anderen Seite ist für eine größere Anzahl eine aktuelle leistungsstarke Hardware

notwendig. Damit das Programm bei dieser Arbeit eine gute Kompatibilität besitzt und der Datendurchsatz des Speichers völlig verwendet wird, wird die Anzahl der Threads pro Block (blocksize) auf 512 festgelegt, was der maximalen Anzahl an auszuführenden Threads pro Block für die hier verwendete GeForce 9800 GT entspricht. Ein VGA-Bild als Gitter mit 640×480 Pixeln ist dafür blockweise trennbar. Trotzdem wurde ein Befehl für die Lösung einer untrennbaren Threadanzahl hinzugefügt. Wenn die Threadanzahl pro Gitter nicht durch 512 (blocksize) trennbar ist, wird die Anzahl der Blöcke automatisch um 1 erhöht.

4.3.3 Speicherzugriff

Das CUDA Programm nimmt an, dass ein System aus *Host* und *Device* besteht. Die beide haben ihre eigenen getrennten Speicher (CPU-Speicher u. GPU-Speicher). Kernel kann den GPU-Speicher nicht operieren. Deshalb werden während der Laufzeit einige Funktionen für *allocate*, *deallocate*, *copy* und *transfer* (Host<->Device) angeboten. Typischerweise wird *cudaMalloc* für die Speicherzuordnung, *cudaFree* für die Speicher Freigabe, und *cudaMemcpy* für die Datenübertragung zwischen der CPU und GPU verwendet. Daten aus dem Speicher zu holen ist sehr aufwendig. Deshalb wird in jedem Modul zuerst *cudaMalloc* auf dem GPU-Speicher mit bestimmter Größe ausgeführt. Dann werden die Quelldaten, wenn möglich nur einmal, auf die GPU kopiert, um den Speicher-Datendurchsatz völlig zu verwenden. Dazu wird noch ein Befehl *safeAlloc* für den Speichercheck eingesetzt. Alle Module werden nach dem Ablauf *anlegen* \rightarrow *benutzen* \rightarrow *leeren* implementiert. Wenn kein Speicherraum mehr verfügbar ist, hört das Programm auf und eine Fehlermeldung erscheint.

4.4 Modul-Integration

In der Vorarbeit zum Lernverfahren des Hintergrundmodells, kann der Input entweder Einzel-, oder ein Multikanal Bild sein. Wegen der Anwendung des YUV-Farbraums wird üblicherweise ein Bild mit drei Gleitkommakanälen eingesetzt. Für den Pixelzugriff auf das Multikanal Bild in OpenCV, werden die Elemente als 3D mit der Variable i (Zeile), j (Spalte), k (Kanal) indiziert. Ein Aufbau für 2D-Bilder mit 3D-Index in der CUDA-Vorgehensweise ist relativ aufwendig und komplex. Deshalb wird am Anfang das Quellbild durch den OpenCV-Befehl *CvSplit* in drei Einzelkanal Bilder getrennt. Die drei Bilder werden nach der Gauss-Mischverteilung berechnet und könnten dann durch den OpenCV-Befehl *CvMerge* wieder vereinigt werden. Jedoch wird in dieser Arbeit das vereinigte Zielbild als die Summe der Produkte aus jeder Komponente dargestellt.

In der Abbildung 4.3 liegt die Laufrichtung des vorhandenen Speicherlayouts von der Vorarbeit [Feldmann2009AFS] vor. Ein Pixel des Zielbildes wird durch den gesamten Block in dieser Abbildung dargestellt. Der Prozess wird pixelweise als 1D-Index im Hauptspeicher ausgeführt. Für den Pixelwert des Punkts (x, y) auf das Quellbild, werden der Wert von jedem Kanal des Bildes, das zugehörige Gewicht für jede Komponente, der Erwartungswert μ sowie die Standardabweichung σ des Quellenpixels entnommen. Danach wird die Berechnung der Dichtfunktion des Hintergrundmodells punktweise sequenziell ausgeführt.

$$p(c_t | F_t = 0) = \sum_{k=1}^K \omega_t^k \prod_{d=1}^3 \eta(c_t^d, \mu_t^{k,d}, \Sigma_t^{k,d}) \quad (4.3)$$

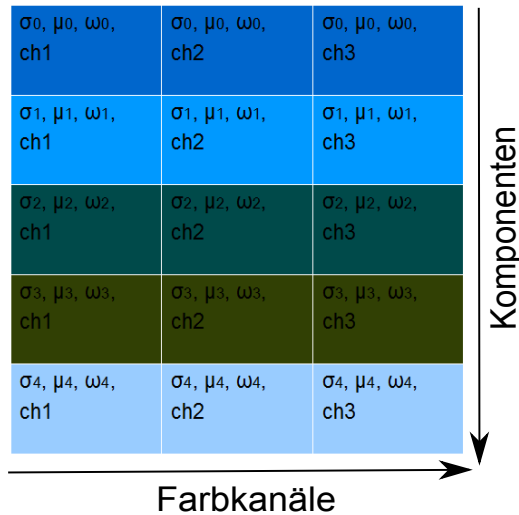


Abbildung 4.3: Pixelweise Berechnung auf CPU

Im Anschluss wird das Ergebnis auf dem Zielbild gespeichert. Der Prozess läuft sehr langsam, da er sehr rechenintensiv ist und die CPU voll ausgelastet.

Die Abbildung 4.4 zeigt die optimierte Laufausrichtung der CUDA-Vorgehensweise. Der Berechnungsprozess wird von sequenziell nach parallel umgewandelt. Die vorhandenen Pixelwerte werden als virtuelle Bilder zusammengefasst und in zwei Dimensionen indexiert. Nach der Initialisierung werden alle Bilder, die berechnet werden sollen, einmalig auf den GPU-Speicher kopiert. Die obere Funktion wird auf der GPU realisiert. Die Berechnung jedes Pixels auf den Bildern wird als ein Thread parallelweise ausgeführt.

Der Ablauf des Moduls lässt sich von links nach rechts in drei Schritten zerlegen. Im ersten Schritt wird das Quellbild, ein Multikanal Gleitkommabild, durch den OpenCV-Befehl *CvSplit* in drei Einzelkanal Bilder getrennt. Im folgenden Schritt werden die drei Einzelkanal Bilder mit den zugehörigen Parameter Bildern (die Erwartungswerte und die Standardabweichungen) nach der Normalverteilung multipliziert. Das Ergebnis wird anschließend mit dem jeweils zugehörigen Gewicht der Komponente multipliziert und auf dem Zielbild gespeichert. Wenn es mehrere Komponenten gibt, werden die Ergebnisse der Komponenten addiert und die Summe auf dem Zielbild gespeichert.

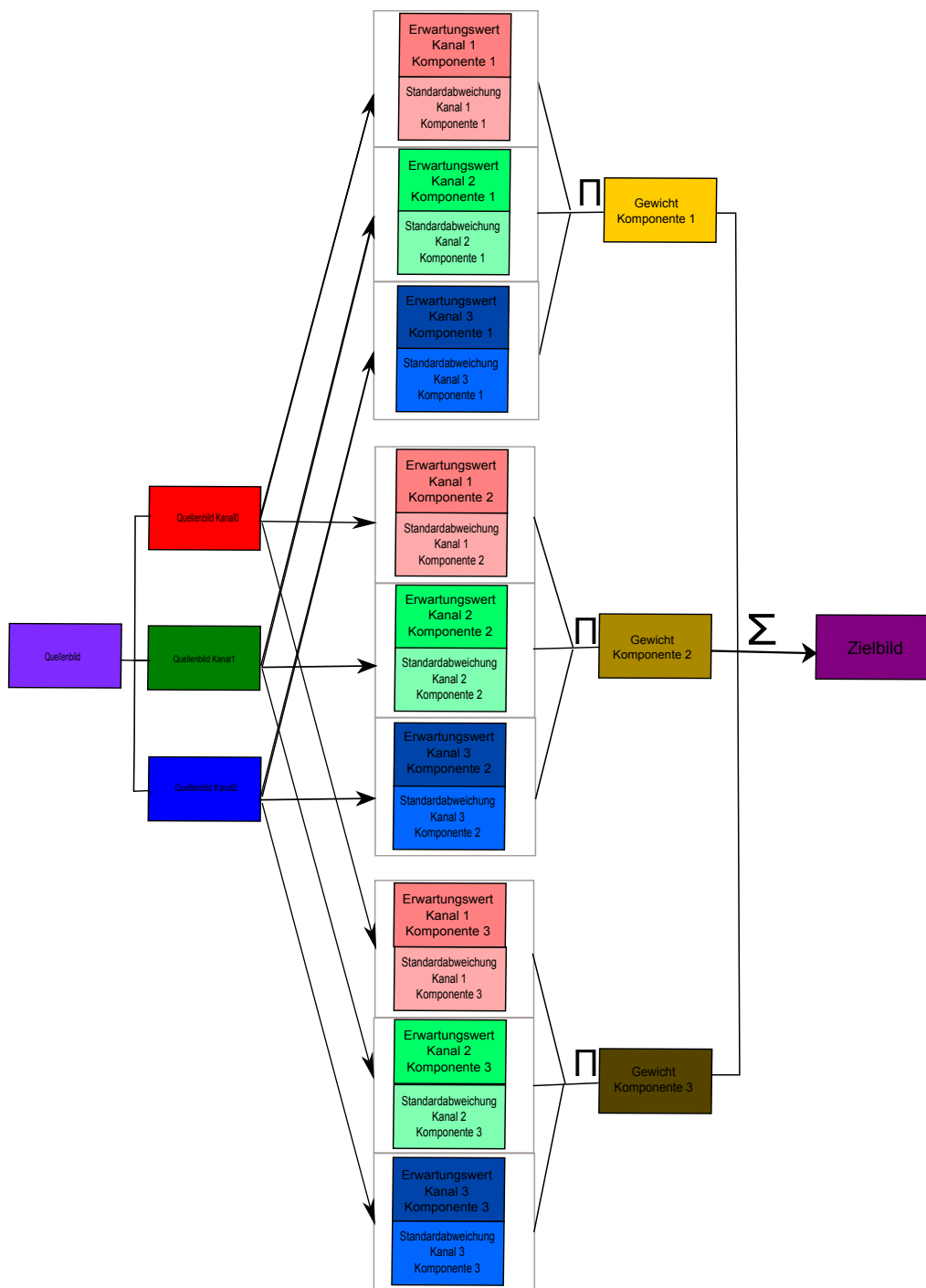


Abbildung 4.4: Gaussmixture auf GPU (3 Komponenten)

Kapitel 5

Experimente und Auswertungen

Die Schwerpunkte der Experimente liegen auf einigen Laufzeitmessungen und einer Beurteilung der Leistung des umgesetzten Moduls aus zusammengefassten Ergebnissen. Anschließend folgen ein kurzer Ausblick über weitere Optimierung und verbleibende offene Fragen. Im ersten Unterabschnitt wird eine Zeitmessung für die Datenübertragung zwischen CPU und GPU beschrieben. Danach wird eine Leistungsmessung auf GPU für die Arithmetik der Gleitkommazahl-Matrizen vorgestellt. Im nächsten Unterabschnitt wird eine Evaluierung für die Leistung des integrierten CUDA-Moduls vorgenommen. Alle Versuche werden auf einem 2.83GHz Intel Core(TM)2 Quad Q9550 mit 8GB Arbeitsspeicher und einer NVidia Geforce 9800GT Grafikkarte mit 512MB Speicher durchgeführt. Die angewandten Betriebssysteme sind Windows 7 64Bit und Ubuntu 10.04LTS.

5.1 Geschwindigkeitsmessung zur Datenübertragung zwischen CPU und GPU

Die Datenübertragung zwischen CPU und GPU ist eine relativ langsame Operation. Sie ist abhängig von der verwendeten Hardware. Ein CUDA-Programm soll die Anzahl der Operationen praktisch minimieren. Deswegen wird versucht so viel wie möglich auf der GPU zu implementieren. Dies bedeutet, dass Datenstrukturen auf dem GPU Speicher erzeugt, ausgeführt und gelöscht werden. Nur die Berechnungsergebnisse werden zurück zum *Host* geführt. Außerdem sollen viele kleine Übertragungen zu einer großen Übertragung zusammengefasst werden. Für die CUDA-Optimierung sollte die Übertragungsleistung zwischen *Host* und *Device* bekannt sein. Aus diesem Grund wird eine Messung zur Übertragungsgeschwindigkeit ausgeführt. In diesem Programm (Modul 6: Measure.cu) werden die Serienbilder mit verschiedener Größen (1×1 bis 1024×1024 Pixel) zum GPU-Speicher kopiert. Die entsprechende Übertragungszeit jedes Bildes wird sequenziell gemessen. Der Befehl `cudaMemcpy()` wird typischerweise für die Übertragung verwendet. Die Messergebnisse sind durch die Abbildung 5.1 als Kurve dargestellt.

Abbildung 5.1 zeigt, ist die Eingabe kleiner als ca. 325×325 Pixel, ist die Geschwindigkeit relativ stabil. Der Datenfluss ist kleiner als der Durchsatz. Vergrößern sich die Bilder, wird die Übertragungszeit linear dazu länger, bis zu einer Bildgröße von 745×745 Pixel. Danach steigt die Zeitkurve zwar noch linear, aber sehr viel stärker. Die Schnittstellen der Grafikkarte

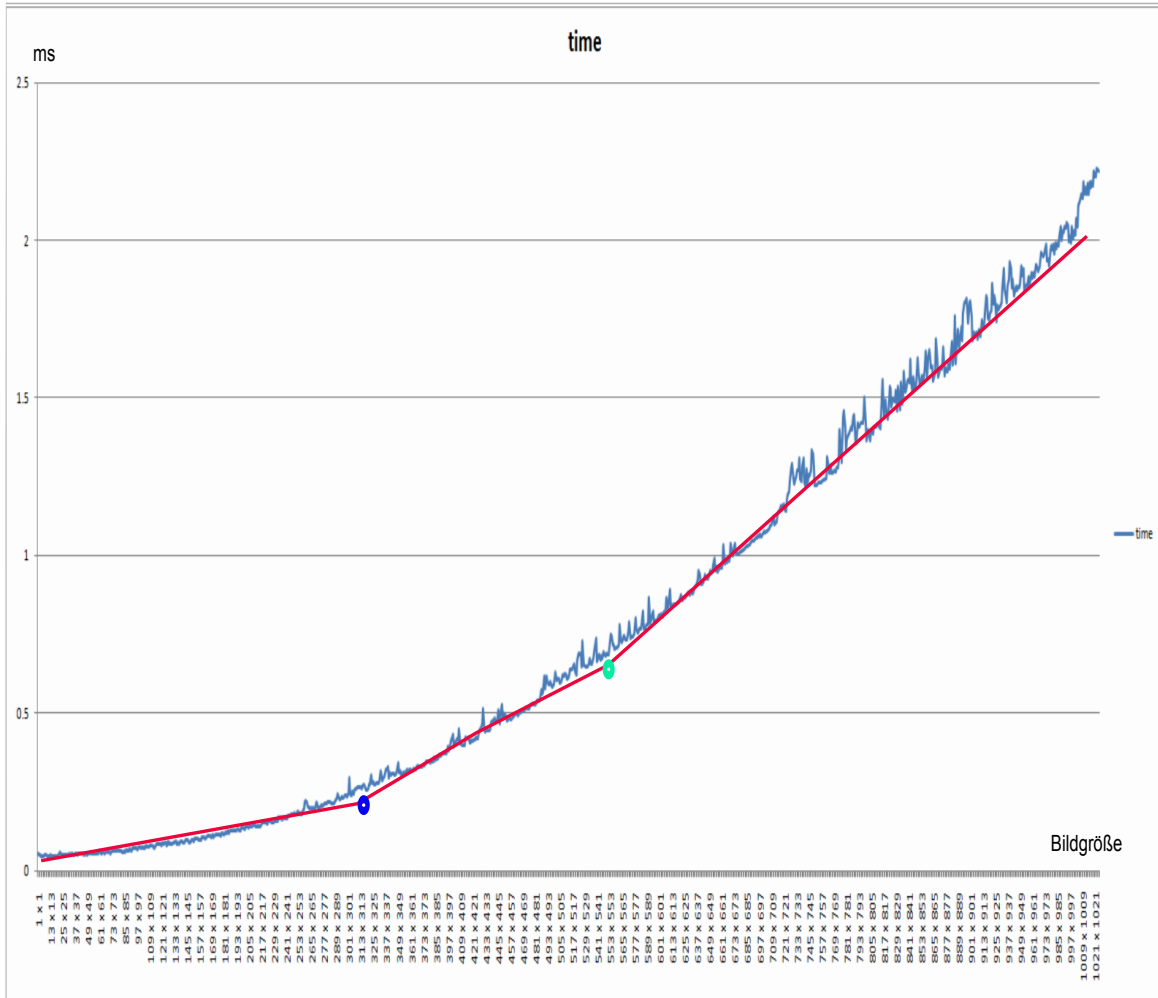


Abbildung 5.1: Zeitmessung zur Bildübertragung(NV Geforce 9800GT mit 512MB)

zum System werden voll ausgelastet. Jedes Systemereignis könnte sich auf die Geschwindigkeit auswirken, da sich die Peaks ab einer Größe von 745×745 Pixel häufen. Dies bedeutet, dass das CUDA-Programm nicht mit anderen Programmen parallel implementiert werden sollte.

5.2 Zeitmessung zur Arithmetik der Gleitkommazahl-Matrizen

Nach der Geschwindigkeitsmessung folgt ein Vergleich zur Berechnungsleistung zwischen CPU und GPU. Zwei quadratische Gleitkommazahl-Matrizen A und B werden durch klassischen C-Code bzw. CUDA-Code multipliziert. Die Matrix C ist die Resultatmatrix. Die Ausführungszeit der GPU enthält noch die Versand- und Empfangszeit.

Die Matrizen-Multiplikation auf der Programmiersprache C wird durch zwei geschlossene Schleifen von den zwei Achsen ausgeführt. Im CUDA-Code werden die Indizes der Blöcke und Threads zu Matrizen-Indizes konvertiert. Damit werden alle Elemente der Matrizen als Threads unabhängig voneinander im Kernel parallelweise ausgeführt. Das Ergebnis jedes Threads wird in die entsprechende Position auf der Matrix C gespeichert. Die Strategie

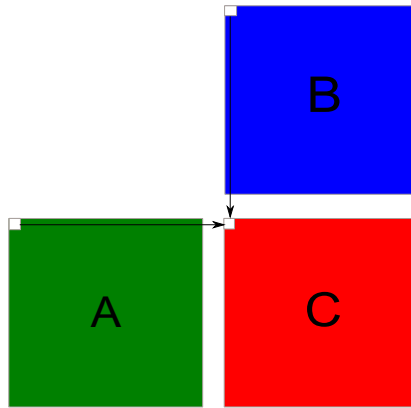


Abbildung 5.2: Matrizen Multiplikation: Quellmatrizen A, B; Zielmatrix C

des Indexes ist in Abbildung 5.2 dargestellt. Zum Beispiel werden für einen leichten Thread im Kernel, die Pixelwerte aus Position (0, 0) aus den Quellenmatrizen A und B genommen und multipliziert. Das Ergebnis wird auf die Position (0, 0) auf der Zielmatrix C abgelegt.

In der Abbildung 5.3 liegt die Berechnungszeit für die Matrizen-Multiplikation auf der CPU bzw. GPU. Auf der linken Seite steht die Tabelle der Testergebnisse. Das entsprechende Diagramm befindet sich rechts. Die Zeitachse wird logarithmisch skaliert. Aus dem Vergleich der Ergebnisse ist ersichtlich, dass durch die Umwandlung auf der GPU die Berechnung maximal ungefähr 40-fach beschleunigt werden kann. Für die Berechnung von kleinen Bildern, kleiner als 256×256 Pixel, wird der Unterschied der Leistungen der CPU bzw. GPU nicht ersichtlich. Zurzeit spielt die Übertragungszeit noch eine wichtige Rolle in der GPU-Ausführungszeit. Die Moderne CPU kann diese nicht rechenintensive Aufgabe noch gut bewältigen. Je größer die Bilder sind, desto deutlicher ist der Vorteil der GPU-Berechnung, obwohl die Datenübertragung aufwendig ist. Ohne Optimierung kann das CUDA-Programm bei 4096×4096 Pixel schon ungefähr 40-fach beschleunigt werden.

Größe(Pixel)	CPU(ms)	GPU(ms)
256*256	34	11
512*512	274	80
1024*1024	5661	573
2048*2048	107243	4531
4096*4096	1174086	36670

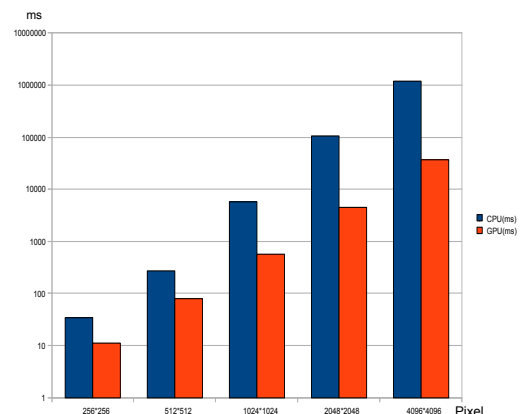


Abbildung 5.3: Zeitmessung zur Arithmetik der Gleitkommazahl-Matrizen

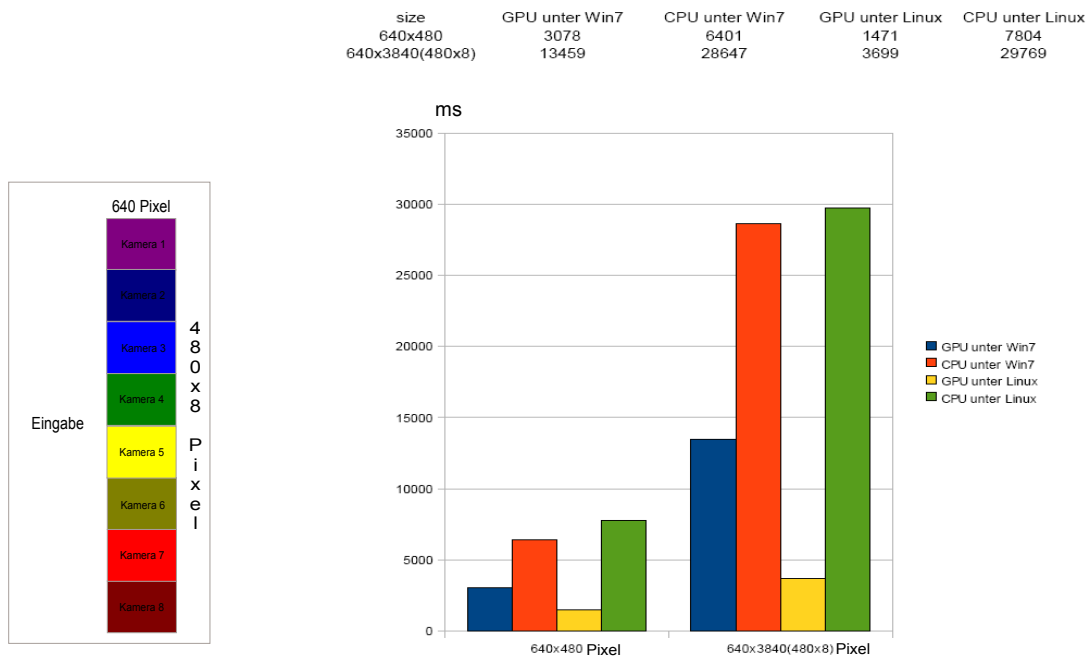


Abbildung 5.4: Gauss-Mischverteilung auf GPU

5.3 Zeitmessung zum Modul Gauss-Mischverteilung im Hintergrundmodell

Dieser Teil ist der Kern der Arbeit und wird in zwei weitere Unterabschnitte gegliedert. Der erste Teil ist die Beschreibung und Auswertung des CUDA-Prototyps für die Gauss-Mischverteilung. Der zweite Teil ist die Vorstellung der benutzten Datensätzen und die Bewertungsmethodik zur quantitativen Evaluierung des integrierten Moduls.

5.3.1 Zeitmessung zum Prototyp des CUDA-Moduls

In dieser Arbeit besteht das Multi-Kamerasystem aus 8 VGA-Kameras. Somit wird ein Berechnungstest für die Gauss-Mischverteilung aus 8 VGA-Bildern ($8 \times 640 \times 480$ Pixel) auf der CPU bzw. GPU ausgeführt. Die Eingangsbilder werden durch die Abbildung 5.4 links im GPU-Speicher dargestellt. Weiterhin werden die Eingangsbilder im Kernel durch die Gauss-Mischverteilung berechnet. Damit die Ergebnisse präzise sind, werden die 4 Berechnungen 1000mal implementiert. Die Ergebnisse der Zeitmessung liegen auf der Abbildung 5.4 rechts. Die Berechnung des Einzelbilds wird als Referenz implementiert. Die Laufzeit unter Windows und Linux sind unterschiedlich. Die Leistung unter Linux ist geringfügig besser. Im Vergleich zu Windows, kann das CUDA-Programm unter Linux mehr verfügbare Ressource von der GPU erwerben.

Der Test zeigt, dass unter beiden Systemen die Ausführungszeiten linear zur Bildgröße ansteigen. Wenn der Input ein VGA-Bild (640×480 Pixel) ist, wird die Berechnung auf der GPU zwischen 3- bis zu 5-fach beschleunigt. Da die Berechnung der Normalverteilung viel aufwendiger als die allgemeine Arithmetik ist, wird der Vorteil der Berechnungsfähigkeit auf

der GPU deutlicher beim Bearbeiten der Normalverteilung der Gleitkommazahl-Matrizen, als beim Bearbeiten der allgemeinen Arithmetik der Matrizen. Beim Test aus dem letzten Unterabschnitt zeigt sich jedoch lediglich ein Zeitvorteil von 2- bis 3 fach für die Berechnung von kleinen Bilder (256×256 bis 512×512 Pixel).

Wenn ein Quellbild dem 8fachen eines VGA-Bilds entspricht, benötigt die CPU unter Windows bzw. Linux fast 30ms um es zu berechnen. Auf der GPU dauert die Berechnung unter Linux nur gut 3ms. Die Berechnung eines Bildes mit der 8-fachen Größe eines VGA Bildes auf der GPU benötigt die doppelte Zeit, während sie bei der Implementierung auf der CPU ungefähr die vierfache Zeit braucht. Ein potenzieller Grund ist, dass Windows 7 im Vergleich zu Linux selbst mehr Ressourcen von der GPU verwendet. Ein Beispiel ist die Benutzeroberfläche Aero von Windows 7. Sie benötigt mindestens 32 Megabyte Grafikkartenspeicher. Außerdem ist das CUDA-Programm unter Windows optimierbarer, besonders im Bereich Speicherzuordnung und gemeinsam genutzter Speicher.

5.3.2 Zeitmessung zum integrierten Modul

Im Vordergrund befindet sich ein Sportler in einem vorhandenen Videodatensatz **HumanEva**. Die Anzahl der Komponenten der Hintergrundmischverteilung K wird auf 5 begrenzt. Die anderen Parameter werden auf die Werte aus den Vorabexperimenten eingestellt, die sich schon im Ablauf der Implementierung als generell gut geeignet herausgestellt haben. Eine kleine Veränderung der Werte wird nicht zu grundsätzlich anderen Ergebnissen führen. Alle nötigen Werte sind der Konfigurationsdatei aus dem Anhang A zu entnehmen.

Der Ablauf des Verfahrens lässt sich auf dem Videodatensatz **HumanEva** anwenden. Der Datensatz wird mit 8 Kameras unter relativ komplexen Bedingungen aufgenommen. Danach werden die zu segmentierenden Bilder für die jeweilige Kamera frameweise bearbeitet. Die segmentierte Bildsequenz wird in einem Ordner **shapes** gespeichert, der nicht vorhanden ist, und vom Programm angelegt wird.

Die Ergebnisse der Zeitmessung sind in Abbildung 5.5 und 5.6 zu sehen. 10 sequenzielle Bilder (Frame1 - Frame10) werden zufällig aus den zu segmentierenden Bildern ausgewählt. Anschließend werden diese Bilder aus Kamera 1 bis 7 (Cam1 - Cam7) durch das CUDA-Modul bzw. das Modul der Vorarbeit segmentiert. Nach einer 1000-maligen Laufanzahl, werden die Mittelwerte als endgültige Ergebnisse dargestellt. Im Debug-Modus ist die Optimierung offensichtlich, da durch CUDA-Umsetzung die Segmentierung durchschnittlich 6- bis 8 fach beschleunigt werden kann. Für jeweils jede Kamera ist die Szene unterschiedlich. Deshalb ist die Leistung der Segmentierung auch jeweils unterschiedlich. In der Abbildung 5.5, wenn der Hintergrund nicht komplex ist, dauert die Ausführungszeit des CUDA-Moduls pro Bild weniger als 500ms. Ansonsten braucht kein Bild mehr als 1000ms um segmentiert zu werden. Im Gegenzug dazu dauert die Ausführungszeit der Vorarbeit durchschnittlich gut 2500ms bis zu 3500ms. Die Ergebnisse in der Abbildung 5.6 zeigen, dass die Ausführungszeit des CUDA-Moduls im Release-Modus nicht optimiert wird und fast identisch bleibt. Im Gegenteil wird die Leistung des Moduls der Vorarbeit im Release-Modus stark erhöht. Es funktioniert relativ besser als das CUDA-Modul. Die Ausführungszeit pro Bild dauert durchschnittlich weniger als 600ms. Im Release-Modus wird der C-Code gründlich optimiert. Der CUDA-Code wird auf die GPU implementiert, und kann nicht optimiert werden. Außerdem ist die Leistung der Messung stark abhängig von der Hardware und des Betriebssystems.

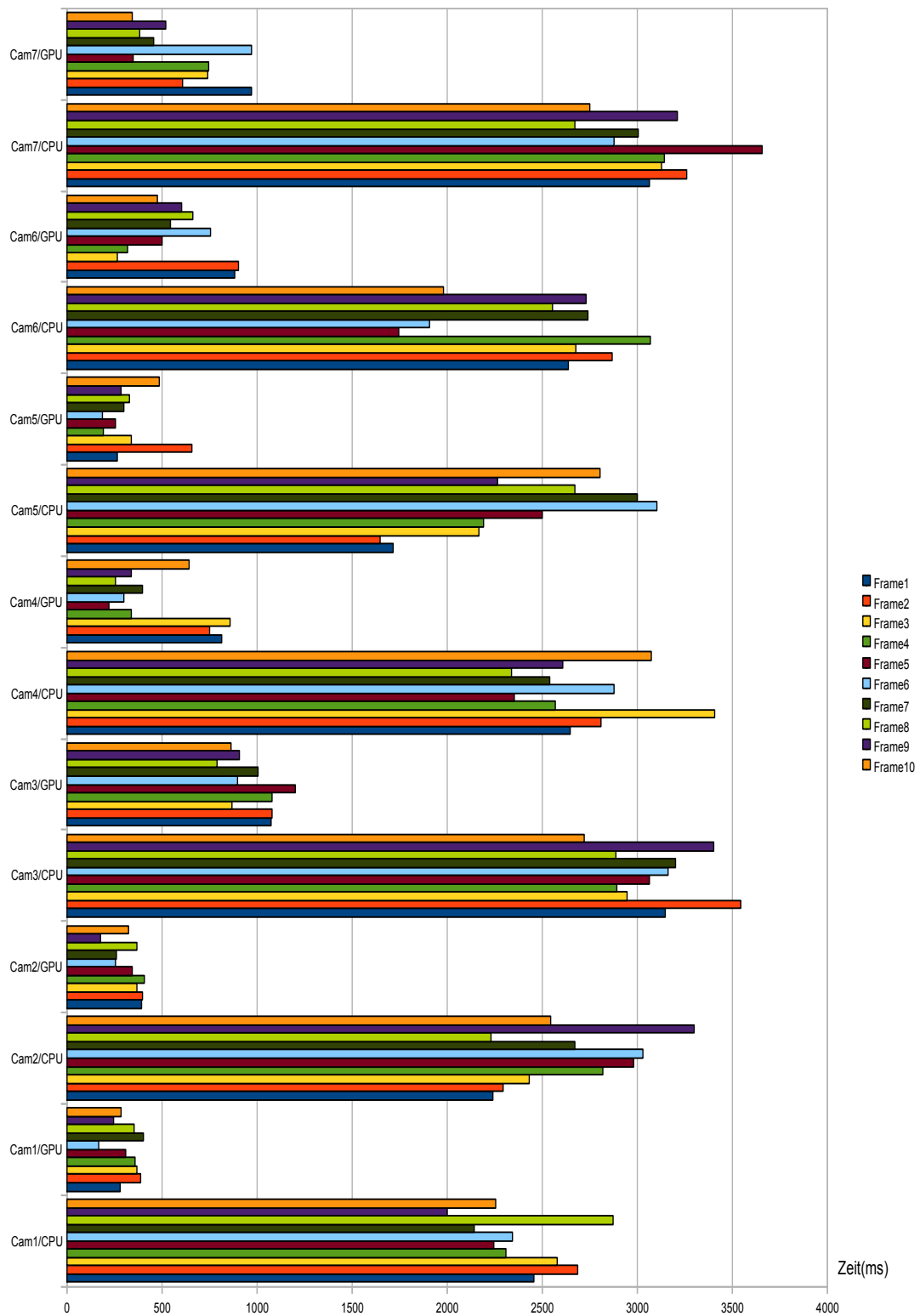


Abbildung 5.5: Zeitmessung des CPU- und GPU-Moduls im Debug-Modus

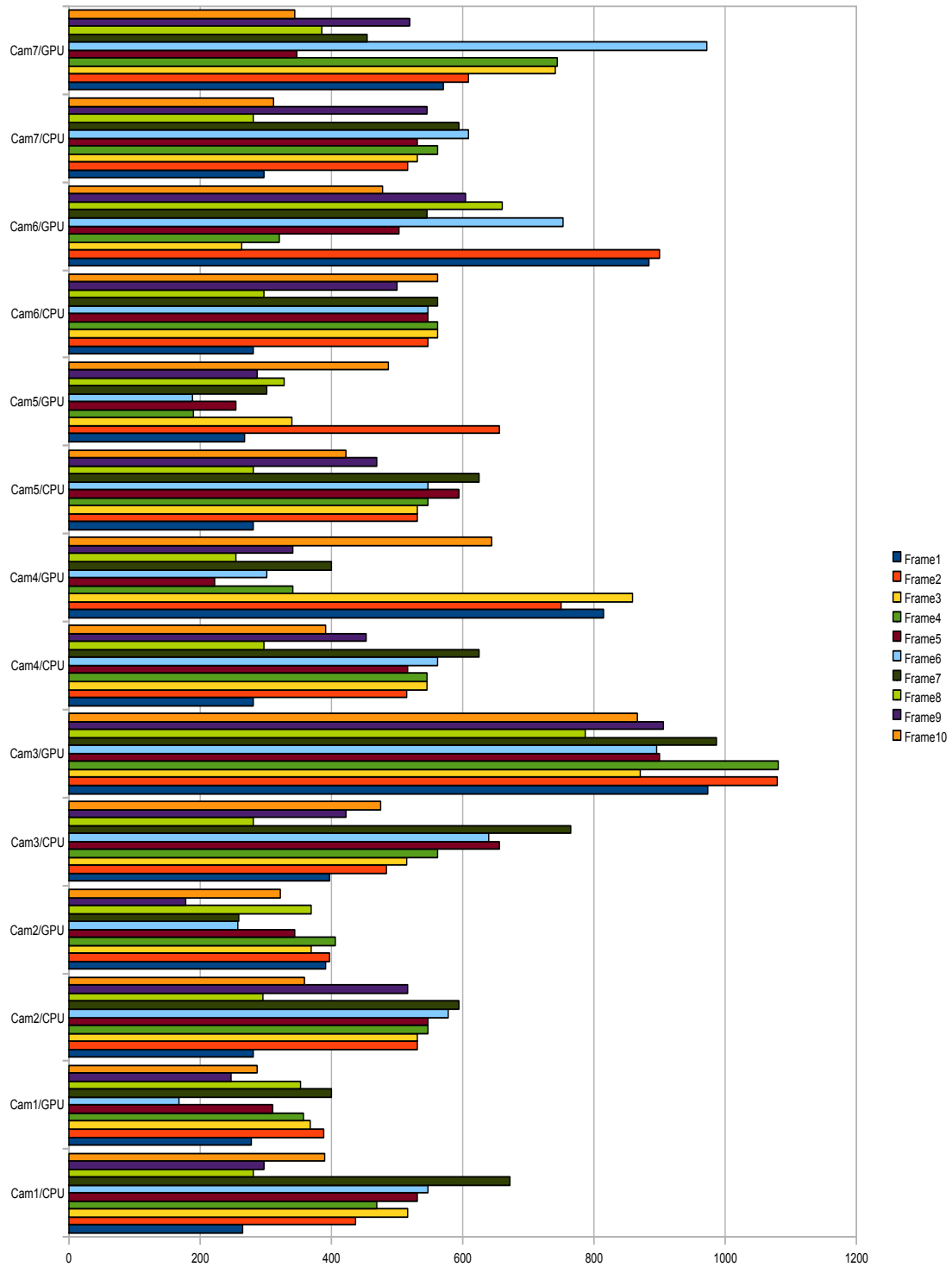


Abbildung 5.6: Zeitmessung des CPU- und GPU-Moduls im Release-Modus

5.3.3 Offene Fragen und ein kurzer Ausblick über weitere Optimierung

Im Vergleich mit der anderen Tests in diesem Kapitel, ist die Leistungssteigerung des integrierten Moduls nicht zufriedenstellend. Ein potentieller Grund ist, obwohl das Verfahren so entworfen wurde um es möglichst parallel ausführen zu können, dass es sehr viele Datenzugriffe für die Umformung der Bildsequenzen in Matrizen auf den Hauptspeicher bzw. den GPU-Speicher gibt. Im Release-Modus kann der GPU-Code nicht optimiert werden. Demzufolge bleibt die Frage nach der weiteren Optimierung des CUDA-Moduls offen.

Für die Weiterführung der Optimierung, kann man über folgende Faktoren nachdenken:

- Zugriffsmuster auf Globale- bzw. Shared-Speicher optimieren.
- Instruktionsdurchsatz erhöhen.
- Softwareumgebung-Optimierung
 - Konfiguration des NVCC-Kompilers
 - Konfiguration des Debugger-Profilers
- Hardwareumgebung-Optimierung
 - Modifikation der GPU
 - Modifikation der Bandbreite

Kapitel 6

Zusammenfassung

In dieser Arbeit werden einige Module mit der CUDA-Technologie entwickelt und anschließend evaluiert.

Im Verfahren der Vorder-/Hintergrundsegmentierung von Lars Diesselberg, sind die parallelen Berechnungen im Hintergrundmodell sehr aufwändig und relativ langsam. Um diesen Prozess zu beschleunigen, und die CPU bei den rechenintensiven Berechnungsaufgaben zu entlasten, werden die vorhandenen Algorithmen im Hintergrundmodell in CUDA-Code umgewandelt. Die GPU übernimmt die Berechnungsaufgaben. Die CPU verwaltet die Verteilung der CPU-/GPU-Speicherräume und die Datenübertragung zwischen Host und Device. Jedes Pixel des Quellbildes wird als ein Thread im Kernel auf der GPU parallel ausgeführt. Die Übertragung zwischen CPU- und GPU-Speicher ist ein aufwändiger Prozess. Um die Häufigkeit der Übertragungen zu minimieren, werden das Quellbild und die zugehörigen Parameter als Gleitkommazahl-Bilder einmal auf den GPU-Speicherraum hochgeladen. Danach werden die Bilder für jeden Kanal und jede Komponente berechnet und zusammengefasst. Schließlich wird das Zielbild zurück auf den CPU-Speicher kopiert.

Um die Geschwindigkeit der Übertragung zwischen CPU- und GPU-Speicher zu ermitteln, wird eine Geschwindigkeitsmessung durchgeführt. Die Zeitkurve zeigt, dass die Übertragungszeit mit steigender Bildgröße linear wächst. Die allgemeine Matrizenberechnung kann als ein typischer paralleler Prozess auf der GPU umgesetzt werden, wobei ebenfalls eine Geschwindigkeitsmessung durchgeführt wird. Diese zeigt, dass durch die Portierung auf CUDA die Berechnung um den Faktor 40 beschleunigt werden kann. Am Ende wird eine Evaluierung des integrierten CUDA-Moduls für Gauss-Mischverteilung implementiert. Die Zeitmessung hierfür zeigt, dass im Debug-Modus durch die GPU-Berechnung das Verfahren des Hintergrundmodells 6- bis 8-fach beschleunigt werden kann. Aber ist es enttäuschend, dass im Release-Modus das CUDA-Modul nicht optimiert werden kann. Die Ergebnisse sind stark abhängig von der Hardware und der Konfiguration der Softwareumgebung. Inwieweit jedoch der CUDA-Code optimiert werden kann bleibt offen. Außerdem wäre interessant zu untersuchen, ob die CUDA-Umsetzung für das Vordergrundmodell oder Schatten und Aufhellungen-Modell anwendbar ist.

Anhang A

Beispielanwendung

In diesem Anhang wird anhand eines Beispiels (Behandlung des **HumanEva**-Datensatzes) beschrieben, wie die Software, die bevor der Studienarbeit entwickelt wurde, verwendet werden kann. Das Programm wird mit einem Parameter aufgerufen, dem Namen einer Konfigurationsdatei. Diese enthält alle Einstellungen und Angaben, die für die Durchführung des Verfahrens nötig sind. Ein Beispiel für eine Konfiguration ist in A.1 zu finden.

Die Quelldaten müssen in einer bestimmten Verzeichnisstruktur vorliegen. Der Hauptordner enthält für jede Kamera ja ein Unterverzeichnis, in diesem Beispiel `cam1`, ..., `cam7`. In jedem Kameraverzeichnis befindet sich eine Datei `calibration.cal`, in der die Kalibrierungsdaten für die jeweilige Kamera gespeichert sind. Weiterhin gibt es zwei Unterordner, in diesem Beispiel mit `background` und `foreground` bezeichnet, in denen sich die Hintergrundbilder bzw. die Vordergrundbilder der Kamera befinden. Die segmentierten Bilder werden in Unterordner `shapes` gespeichert. In der Konfigurationsdatei kann der Name eines weiteren Unterordners angegeben werden, in dem die Ergebnisse gespeichert werden sollen. Ist dieser Ordner nicht vorhanden, wird er vom Programm angelegt.

Für andere Datensätze sind in der Konfigurationsdatei vor allem die Größe und Lage des Voxelraumes, sowie die Namen der Datenverzeichnisse anzupassen. Die anderen Parameter haben sich als generell gut geeignet herausgestellt und können daher oft unverändert übernommen werden.

Listing A.1: Beispielkonfiguration

```
640
480
200
90
100
-2000
0
0
20
5
50
2.0
640
10
0.5
0.05
0.5
0.05
1
0.66
1.5
1
128
192
0
0
1
1
foreground
background
shapes
cam1
cam2
cam3
cam4
cam5
cam6
cam7
```

Literaturverzeichnis

- [Patrick 08] Patrick Cardinal, Pierre Dumouchel, Gilles Boulianne and Michel Co-meau. „GPU Accelerated Acoustic Likelihood Computations“, Centre de Recherche Informatique de Montréal(CRIM), Montréal, Canada, 2008. Online erhältlich via URL: http://www.crim.ca/Publications/2008/documents/plein_texte/PAR_CarPals_Interspeech2008.pdf
- [Claus 08] Claus Lenz, Giorgio Panin and Alois Knoll. „A GPU-accelerated particle filter with pixel-level likelihood“, Robotics and Embedded Systems Lab, Computer Science Department, Technische Universität München, 2008. URL: <http://www6.in.tum.de/pub/Main/Publications/Lenz2008b.pdf>
- [Feldmann2009AFS] Lars Dießelberg, Tobias Feldmann and Annika Wörner. „Probabilistische 3D-Belegungsgitter als Feedback Zur adaptiven Vordergrung-/Hintergrundtrennung“, Karlsruher Institut für Technologie(KIT), 2009.
- [Bayestheorem] Dr. phil. Rudolf Sponsel. „Das Bayes’sche Theorem“, Internet Publikation für Allgemeine und Integrative Psychotherapie, 2002.
- [Lange 03] Sascha Lange. „Verfolgung von farblich markierten Objekten“, 2002. URL: <http://www-lehre.inf.uos.de/cg2/material/20021113/ausarbeitung.pdf>
- [Schlittgen2000] Rainer Schlittgen. „Einführung in die Statistik“, Oldenbourg Wissenschaftsverlag, 2000.
- [CUDAGuide] NVIDIA Corporation. „NVIDIA CUDA Programming Guide 2.3“, 2010.
- [Andreas06] Andreas Schätzle. „Grundprinzipien des Bayes’schen Lernens und Der naive Bayes-Klassifikator im Vergleich zum Maximum-Likelihood-Klassifikator“, University of Tübingen, 2006.
- [HornKorsche04] Hendrik Horn und Jörgen Kosche. „Segmentierung von Bilddaten“, 2004.
- [WikiCUDA] Wikipedia. „Compute Unified Device Architecture“. URL: <http://en.wikipedia.org/wiki/CUDA>
- [WikiGauss] Wikipedia. „Normalverteilung“. URL: <http://de.wikipedia.org/wiki/Normalverteilung>
- [WikiGPGPU] Wikipedia. „General Purpose Computation on Graphics Processing Unit“. URL: <http://de.wikipedia.org/wiki/GPGPU>

[WikiYUV] Wikipedia. „YUV-Farbmodell “. URL: <<http://de.wikipedia.org/wiki/YUV-Farbmodell>>

[Latsch08] Christian Latsch. „Personentracking über adaptive Hintergrund- und Differenzbildberechnung“, Universität Koblenz-Landau, 2008

